Object Oriented Programming
14CS55

UNIT - 3

Inheritance: is one of the cornerstones of OOP because it allows the creation of hierarchical classifications.

Base-Class Access Control

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

```
class derived-class-name : access base-class-name
{
        // body of class
};
```

- The access status of the base-class members inside the derived class is determined by access
- The base-class access specifier must be either **public, private, or protected**.
- If no access specifier is present, the access specifier is ***private by default*** if the derived class is a class. If the derived class is a struct, then public is the default in the absence of an explicit access specifier

- When the access specifier for a base class is public, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class.
- In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class

Example :
```
#include <iostream>
using namespace std;
class base
{
        int i, j;
        public:
                void set(int a, int b)
                {
                        i=a;
                        j=b;
                 }
                void show()
                 {
                        cout << i << " " << j << "\n";
                 }
};

class derived : public base
{
        int k;
        public:
                derived(int x)
                {
                        k=x;
                 }
                void showk()
                {
                         cout << k << "\n";
                }
};

int main()
{
        derived ob(3);
        ob.set(1, 2); // access member of base
        ob.show(); // access member of base
        ob.showk(); // uses member of derived class
        return 0;
}
```

When the base class is inherited by using the private access specifier, all public and protected members of the base class become private members of the derived class

On change of the above program with private access specifier will make a compiler error when object of derived class try to access set() function of base class as its now a private member of the drived class hence it cannot be accessed outside the class.

Inheritance and protected Members

The protected keyword is included in C++ to provide greater flexibility in the

- When a member of a class is declared as protected, that member is not accessible by other, nonmember elements of the program

- With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class
- The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

Example :

```
#include <iostream>
using namespace std;
class base
        {
                protected:
                        int i, j; // private to base, but accessible by derived
                public:
                        void set(int a, int b)
                        {
                                i=a;
                                j=b;
                         }
                        void show()
                        {
                                cout << i << " " << j << "\n";
                        }
};

class derived : public base
                {
                        int k;
                        public:
                                // derived may access base's i and j
                                void setk()
                                {
                                        k=i*j;
                                 }
                                void showk()
                                {
                                        cout << k << "\n";
                                }
                };
int main()
{
        derived ob;
        ob.set(2, 3); // OK, known to derived
        ob.show(); // OK, known to derived
        ob.setk();
        ob.showk();
        return 0;
}
```

When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class

Example :

```
#include <iostream>
using namespace std;
class base
{
        protected:
                int i, j;
        public:
                void set(int a, int b)
                {
                         i=a;
                         j=b;
                 }
                void show()
                {
                         cout << i << " " << j << "\n";
                }
};
// i and j inherited as protected.
class derived1 : public base
                {
                        int k;
                        public:
                                void setk()
                                {
                                        k = i*j;
                                } // legal
                                void showk()
                                {
                                        cout << k << "\n";
                                 }
                };
// i and j inherited indirectly through derived1.
class derived2 : public derived1
                {
                        int m;
                        public:
```

```
                                        void setm()
                                        {
                                                   m = i-j;
                                         } // legal
                                        void showm()
                                         {
                                                   cout << m << "\n";
                                         }
          };
int main()
{
          derived1 ob1;
          derived2 ob2;
          ob1.set(2, 3);
          ob1.show();
          ob1.setk();
          ob1.showk();
          ob2.set(3, 4);
          ob2.show();
          ob2.setk();
          ob2.setm();
          ob2.showk();
          ob2.showm();
          return 0;
}
```

If, however, base were inherited as private, then all members of base would become private members of derived1, which means that they would not be accessible by derived2

Protected Base-Class Inheritance

It is possible to inherit a base class as protected. When this is done, all public and protected members of the base class become protected members of the derived class.

```
#include <iostream>
using namespace std;
class base
{
          protected:
                    int i, j; // private to base, but accessible by derived
          public:
                    void setij(int a, int b)
                              {
                                         i=a;
                                         j=b;
```

```
                        }
                void showij() { cout << i << " " << j << "\n"; }
};
// Inherit base as protected.
class derived : protected base
                {
                        int k;
                        public:
                                // derived may access base's i and j and setij().
                                void setk()
                                 {
                                        setij(10, 12);
                                         k = i*j;
                                 }
                                // may access showij() here
                                void showall()
                                {
                                        cout << k << " ";
                                        showij();
                                }
                };
int main()
{
        derived ob;
        ob.setij(2, 3); // illegal, setij() is protected member of derived i.e cant be accessed outside
                        class(just like private)
        ob.setk(); // OK, public member of derived
        ob.showall(); // OK, public member of derived
        ob.showij(); // illegal, showij() is protected member of derived
        return 0;
}
```

Inheriting Multiple Base Classes

It is possible for a derived class to inherit two or more base classes.

Example

```
// An example of multiple base classes.
#include <iostream>
using namespace std;
class base1
{
protected:
        int x;
public:
        void showx()
        {
                cout << x << "\n";
        }
};

class base2
{
protected:
        int y;
public:
        void showy()
        {
                cout << y << "\n";
        }
};

// Inherit multiple base classes.
class derived: public base1, public base2
{
public:
        void set(int i, int j)
         {
                x=i;
                y=j;
         }
};

int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

Constructors, Destructors, and Inheritance

There are two major questions that arise relative to constructors and destructors when inheritance is involved.

- First, when are base-class and derived-class constructor and destructor functions called?
- Second, how can parameters be passed to base-class constructor functions?

When Constructor and Destructor Functions Are Executed?

It is possible for a base class, a derived class, or both to contain constructor and/or destructor functions. It is important to understand the order in which these functions are executed when an object of a derived class comes into existence and when it goes out of existence

```
#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 ob; // do nothing but construct and destruct ob
return 0;
}
```

Output :

```
Constructing base
Constructing derived
```

```
Destructing derived
Destructing base
```

- When an object of a derived class is created, if the base class contains a constructor, it will be called first, followed by the derived class' constructor.
- When a derived object is destroyed, its destructor is called first, followed by the base class' destructor, if it exists
- In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order
- The same general rule applies in situations involving multiple base classes

Passing Parameters to Base-Class Constructors

However, how do you pass arguments to a constructor in a base class?
The answer is to use an *expanded form of the derived class's* constructor declaration that passes along arguments to one or more base-class constructors

```
derived-constructor(arg-list) : base1(arg-list),
                                base2(arg-list),
                                       // ...
                                baseN(arg-list)
{
// body of derived constructor
}
```

- Here, base1 through baseN are the names of the base classes inherited by the derived class.
- Notice that a colon separates the derived class' constructor declaration from the base-class specifications,
- The base-class specifications are separated from each other by commas

```
#include <iostream>
using namespace std;
class base
{
protected:
        int i;
public:
        base(int x)
        {
                i=x;
                cout << "Constructing base\n";
        }
        ~base()
```

```
	{
		cout << "Destructing base\n";
	}
};

class derived: public base
 {
int j;
public:
	// derived uses x; y is passed along to base.
	derived(int x, int y): base(y)
	 {
		j=x; cout << "Constructing derived\n";
	 }
~derived()
	{
		cout << "Destructing derived\n";
	}
void show()
	{
		cout << i << " " << j << "\n";
	}
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

- Here, derived's constructor is declared as taking two parameters, x and y.
- However, derived( ) uses only x; y is passed along to base( ).
- In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class
- It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class requires it

Example :

```
#include <iostream>
using namespace std;
class base1
{
protected:
        int i;
public:
        base1(int x)
        {
                i=x; cout << "Constructing base1\n";
        }
        ~base1()
        {
                cout << "Destructing base1\n";
        }
};
class base2
{
protected:
        int k;
public:
        base2(int x)
        {
                k=x; cout << "Constructing base2\n";
        }
~base2()
        {
                cout << "Destructing base2\n";
        }
};
class derived: public base1, public base2
{
public:
        /* Derived constructor uses no parameter, but still must be declared as taking them to pass
        them along to base classes. */
        derived(int x, int y): base1(x), base2(y)
        {
                cout << "Constructing derived\n";
        }
        ~derived()
        {
                cout << "Destructing derived\n";
        }
        void show()
        {
                cout << i << " " << k << "\n";
        }
};
int main()
{
derived ob(3, 4);
```

```
ob.show(); // displays 3 4
return 0;
}
```

A derived class' constructor function is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class

For example, this fragment is perfectly valid:

```
class derived: public base
{
int j;
public:
        // derived uses both x and y and then passes them to base.
        derived(int x, int y): base(x, y)
        {
                j = x*y; cout << "Constructing derived\n";
         }
}
```

Granting Access

When a base class is inherited as private, all public and protected members of that class become private members of the derived class. However, in certain circumstances, you may want to restore one or more inherited members to their original access specification

In Standard C++, you have two ways to accomplish this.

- First, you can use a **using** statement, which is the preferred way
- The second way to restore an inherited member's access specification is to employ an access declaration within the derived class like below

base-class::member;

Example :

```
class base
{
public:
        int j; // public in base
};
// Inherit base as private.
class derived: private base
{
public:
        // here is access declaration
        base::j; // make j public again
.
.
};
```

However, you cannot use an access declaration to raise or lower a member's access status

For example, a member declared as private in a base class cannot be made public by a derived class.

```
#include <iostream>
using namespace std;
class base
{
int i; // private to base
public:
        int j, k;
        void seti(int x)
        {
                 i = x;
         }
        int geti()
        {
                return i;
        }
};
// Inherit base as private.
class derived: private base
{
public:
        /* The next three statements override base's inheritance as private and restore j, seti(), and geti()
        to public access. */

        base::j; // make j public again - but not k
        base::seti; // make seti() public
        base::geti; // make geti() public
        // base::i; // illegal, you cannot elevate access i.e you can not change from private to public
        int a; // public
};
int main()
{
        derived ob;
        //ob.i = 10; // illegal because i is private in derived
        ob.j = 20; // legal because j is made public in derived
        //ob.k = 30; // illegal because k is private in derived
        ob.a = 40; // legal because a is public in derived
        ob.seti(10);
        cout << ob.geti() << " " << ob.j << " " << ob.a;
        return 0;
}
```

Virtual Base Classes

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited.

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;
class base
{
public:
        int i;
};
// derived1 inherits base.
class derived1 : public base
{
public:
        int j;
};
// derived2 inherits base.
class derived2 : public base
{
public:
        int k;
};
/* derived3 inherits both derived1 and derived2. This means that there are two copies of base in
derived3! */
class derived3 : public derived1, public derived2
{
public:
        int sum;
};
int main()
{
derived3 ob;
ob.i = 10; // this is ambiguous, which i???
ob.j = 20;
ob.k = 30;
// i ambiguous here, too
ob.sum = ob.i + ob.j + ob.k;
// also ambiguous, which i?
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As the comments in the program indicate, both derived1 and derived2 inherit base.
However, derived3 inherits both derived1 and derived2. This means that there are twocopies of base present in an object of type derived3. Therefore, in an expression like

```
ob.i = 10;
```

which i is being referred to, the one in derived1 or the one in derived2? Because there are two copies of base present in object ob, there are two ob.is! As you can see, the statement is inherently ambiguous.

There are two ways to remedy the preceding program.

- The first is to apply the scope resolution operator to i and manually select one i

  ob.derived1::i = 10; // scope resolved, use derived1's i

- Second by declaring the base class as virtual when it is inherited

```
// derived1 inherits base as virtual.
class derived1 : virtual public base
 {
        public:
                int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base
 {
        public:
        int k;
};
```

The only difference between a normal base class and a virtual one is what occurs when an object inherits the base more than once. If virtual base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.

Virtual Functions

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword virtual.

When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs.

When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon the type of object pointed to by the pointer. And this determination is made at run time. Thus, when different objects are pointed to, different versions of the virtual function are executed

```
#include <iostream>
using namespace std;
class base
{
public:
virtual void vfunc()
{
        cout << "This is base's vfunc().\n";
}
};

class derived1 : public base
{
public:
void vfunc()
 {
        cout << "This is derived1's vfunc().\n";
}
};

class derived2 : public base
{
public:
void vfunc()
{
cout << "This is derived2's vfunc().\n";
}
};

int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

Restrictions on Virtual functions

- **The term overloading is not applied to virtual function** redefinition because several differences exist. Perhaps the most important is that the prototype for a redefined virtual function must match exactly the prototype specified in the base class. This differs from overloading a normal function, in which return types and the number and type of parameters may differ

- However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost
- virtual functions **must be nonstatic members of the classes** of which they are part. They cannot be friends.
- Constructor functions **cannot be virtual**, but destructor functions can.

Calling a Virtual Function Through a Base Class Reference

- In the preceding example, a virtual function was called through a base-class pointer, but the polymorphic nature of a virtual function is also available when called through a base-class reference a base-class reference can be used to refer to an object of the base class or any object derived from that base. When a virtual function is called through a base-class reference, the version of the function executed is determined by the object being referred to at the time of the call.

- The most common situation in which a virtual function is invoked through a base class reference is when the reference is a function parameter.

```
/* Here, a base class reference is used to access a virtual function. */
#include <iostream>
using namespace std;
class base
 {
```

```
public:
virtual void vfunc()
{
        cout << "This is base's vfunc().\n";
}
};

class derived1 : public base
{
public:
void vfunc()
{
        cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base
 {
public:
void vfunc()
{
        cout << "This is derived2's vfunc().\n";
}
};

// Use a base class reference parameter.
void f(base &r)
{
        r.vfunc();
}
int main()
{
base b;
derived1 d1;
derived2 d2;
f(b); // pass a base object to f()
f(d1); // pass a derived1 object to f()
f(d2); // pass a derived2 object to f()
return 0;
}
```

The Virtual Attribute Is Inherited

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden

Example :

```
#include <iostream>
using namespace std;
class base
{
public:
virtual void vfunc()
{
	cout << "This is base's vfunc().\n";
}
};
class derived1 : public base
{
public:
void vfunc()
{
	cout << "This is derived1's vfunc().\n";
}
};
/* derived2 inherits virtual function vfunc()
from derived1. */
class derived2 : public derived1
{
public:
// vfunc() is still virtual
void vfunc()
{
	cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

As expected, the preceding program displays this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

Virtual Functions Are Hierarchical

When a function is declared as virtual by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used

```
#include <iostream>
using namespace std;
class base
{
public:
virtual void vfunc()
{
        cout << "This is base's vfunc().\n";
}
};
class derived1 : public base
{
public:
void vfunc()
{
        cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base
{
public:
// vfunc() not overridden by derived2, base's is used
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // use base's vfunc()
return 0;
}
```

The program produces this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used. For example, in the following program, derived2 is derived from derived1, which is derived from base. However, derived2 does not override vfunc( ). This means that, relative to derived2, the closest version of vfunc( ) is in derived1. Therefore, it is derived1::vfunc( ) that is used when an object of derived2 attempts to call vfunc( ).

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public derived1 {
public:
/* vfunc() not overridden by derived2.
In this case, since derived2 is derived from
derived1, derived1's vfunc() is used.
*/
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // use derived1's vfunc()
return 0;
}
```

The program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived1's vfunc().
```

Pure Virtual Functions

A pure virtual function is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

virtual type func-name(parameter-list) = 0;

In many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

```
#include <iostream>
using namespace std;
class number
{
protected:
int val;
public:
void setval(int i) { val = i; }
// show() is a pure virtual function
virtual void show() = 0;
};
class hextype : public number {
public:
void show() {
cout << hex << val << "\n";
}
};
class dectype : public number {
```

```
public:
void show() {
cout << val << "\n";
}
};
class octtype : public number {
public:
void show() {
cout << oct << val << "\n";
}
};
int main()
{
dectype d;
hextype h;
octtype o;
d.setval(20);
d.show(); // displays 20 - decimal
h.setval(20);
h.show(); // displays 14 – hexadecimal
o.setval(20);
o.show(); // displays 24 - octal
return 0;
}
```

Abstract Classes

A class that contains at least one pure virtual function is said to be abstract. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), ***no objects of an abstract class may be created***. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes.

Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function

Using Virtual Functions

One of the central aspects of object-oriented programming is the principle of "one interface, multiple methods." This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations

in the base class you create and define everything you can that relates to the general case. The derived class fills in the specific details.

```
// Virtual function practical example.
#include <iostream>
using namespace std;
class convert {
protected:
double val1; // initial value
double val2; // converted value
public:
convert(double i) {
val1 = i;
}
double getconv() { return val2; }
double getinit() { return val1; }
virtual void compute() = 0;
};
// Liters to gallons.
class l_to_g : public convert {
public:
l_to_g(double i) : convert(i) { }
void compute() {
val2 = val1 / 3.7854;
}
};
// Fahrenheit to Celsius
class f_to_c : public convert {
public:
f_to_c(double i) : convert(i) { }
void compute() {
val2 = (val1-32) / 1.8;
}
};
int main()
{
convert *p;
// pointer to base class
l_to_g lgob(4);
f_to_c fcob(70);
// use virtual function mechanism to convert
p = &lgob;
cout << p->getinit() << " liters is ";
p->compute();
cout << p->getconv() << " gallons\n"; // l_to_g
p = &fcob;
cout << p->getinit() << " in Fahrenheit is ";
p->compute();
cout << p->getconv() << " Celsius\n"; // f_to_c
return 0;
}
```

Early vs. Late Binding

- ***Early binding*** refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators.

  Main advantage to early binding is efficiency so the function calls are very fast.

- The opposite of early binding is ***late binding***. As it relates to C++, late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer

  late binding can make for somewhat slower execution times.