UNIT-III
INHERITANCE

INHERITANCE

DEFINITION: Inheritance is a process in which a new class known as derived class is created from another class called base class.

DEFINING A CLASS HIERARCHY

In C++, inheritance is supported by allowing one class to incorporate another class into its declaration. Inheritance allows a hierarchy of classes to be built, moving from most general to most specific. The process involves first defining a *base class*, which defines those qualities common to all objects to be derived from the base. The base class represents the most general description. The classes derived from the base are usually referred to as *derived classes*. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.

C++'s support of inheritance is both rich and flexible.

DEFINING THE BASE AND DERIVED CLASSES

Base Class: A class that is inherited is referred to as a base class.
Derived Class: The class that does the inheriting is called the *derived class*. Further, a derived class can be used as a base class for another derived class.

ACCESS TO THE BASE CLASS MEMBERS

When a class inherits another, the members of the base class become members of the derived class.

General Form of Class Inheritance:

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 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.

a. Public base class 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.

PROGRAM:

```
#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;
}
```

OUTPUT:

```
1 2
3
```

b. Private base class access specifier

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. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.

PROGRAM:

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

// This program won't compile.
class base
{
  int i, j;
public:
  void set (int a, int b)
  {
    i = a;
    j = b;
  }
  void show ()
  {
    cout << i << " " << j << "\n";
  }
};

// Public elements of base are private in derived.
class derived:private base
{
  int k;
public:
    derived (int x)
  {
    k = x;
  }
  void showk ()
  {
    cout << k << "\n";
  }
};

int main ()
{
  derived ob (3);
  ob.set (1, 2);              // error, can't access set()
  ob.show ();                          // error, can't access show()
  return 0;
}
```

OUTPUT:

```
main.cpp: In function ‘int main()’:
main.cpp:48:15: error: ‘void base::set(int, int)’ is inaccessible within this context
   ob.set (1, 2);  // error, can't access set()
               ^
main.cpp:18:8: note: declared here
   void set (int a, int b)
```

```
        ^~~
main.cpp:48:15: error: ‘base’ is not an accessible base of ‘derived’
   ob.set (1, 2);  // error, can't access set()
               ^
main.cpp:49:12: error: ‘void base::show()’ is inaccessible within this context
   ob.show ();   // error, can't access show()
            ^
main.cpp:23:8: note: declared here
   void show ()
        ^~~~
main.cpp:49:12: error: ‘base’ is not an accessible base of ‘derived’
   ob.show ();   // error, can't access show()
            ^
```

c. Protected base class access specifier

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, non-member elements of the program.

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.

A private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class.

PROGRAM:

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

// This program won't compile.
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;
}
```

OUTPUT:

```
2 3
6
```

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.

PROGRAM:

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

// This program won't compile.
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
  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;
}
```

OUTPUT:

```
120 10 12
```

DIFFERENT FORMS OF INHERITANCE

The following are the different types of inheritance,

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance
6. Multipath Inheritance

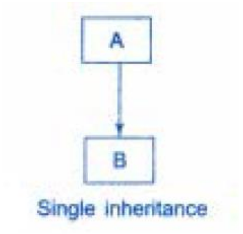
1. Single Inheritance

It is a process of creating new class called derived class from existing base class. The derived class inherits the member functions and variables of the existing base class.

General form:

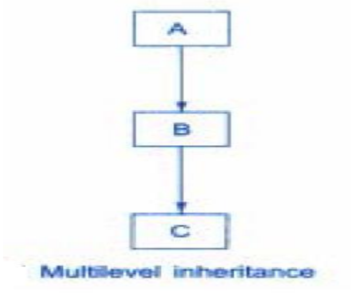
```
class derived-class-name : access base-class-name
{
        // body of class
};
```

PROGRAM: ABOVE 3 PROGRAMS



2. Multilevel Inheritance

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



PROGRAM:

```
#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;
}
```

OUTPUT:

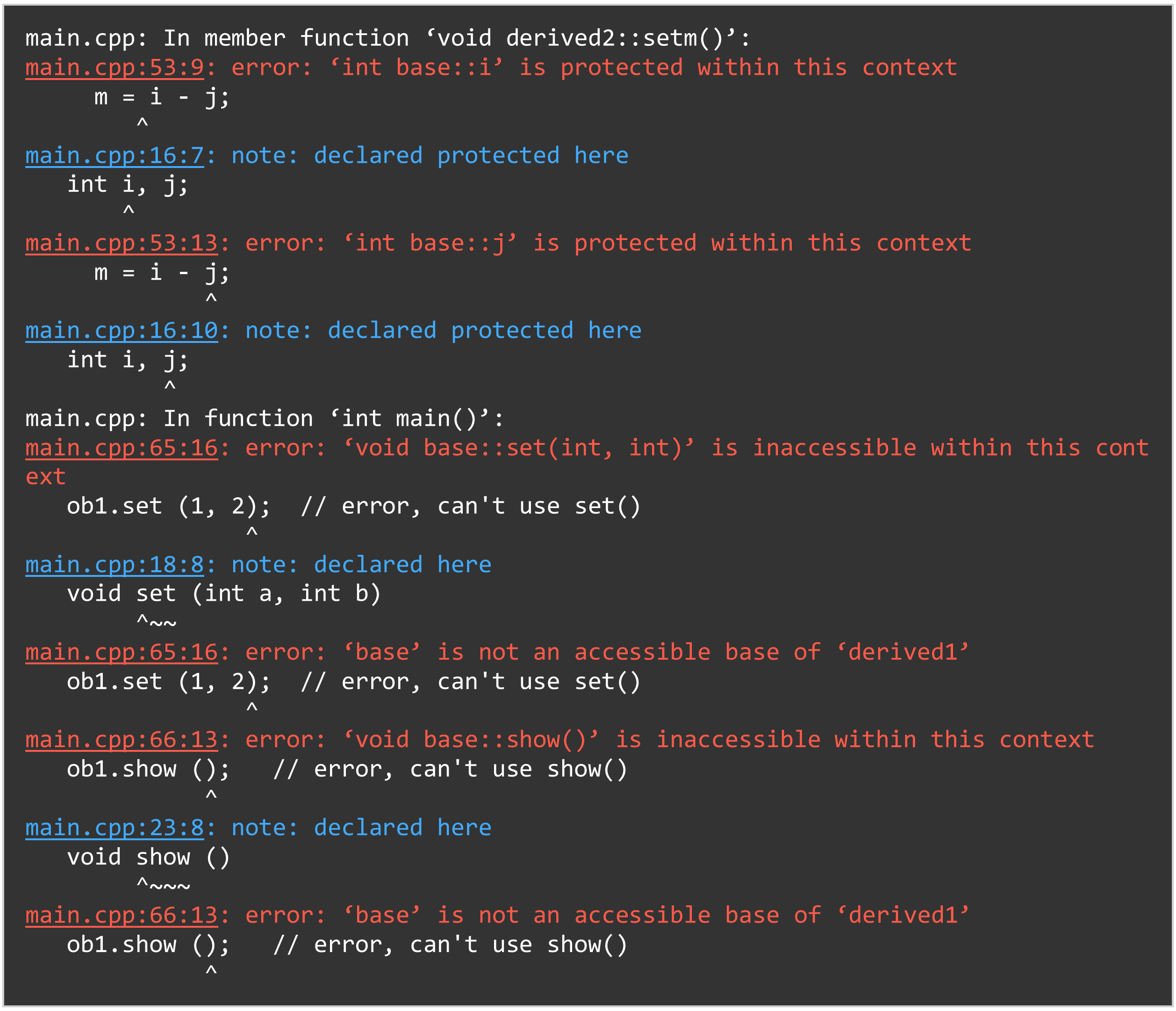
```
2 3
6
3 4
12
```

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**. (However, **i** and **j** would still be accessible by **derived1**.)

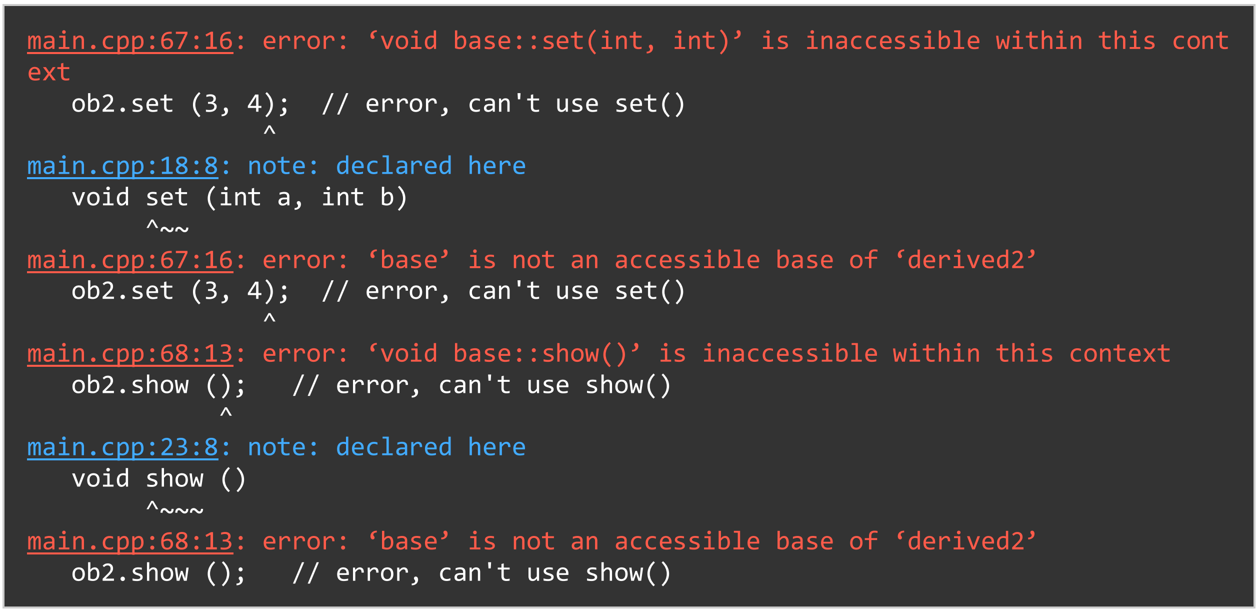
PROGRAM:

```
// This program won't compile.
#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";
  }
};

// Now, all elements of base are private in derived1.
class derived1:private base
{
  int k;
public:
// this is legal because i and j are private to derived1
  void setk ()
  {
    k = i * j;
  }                                  // OK
  void showk ()
  {
    cout << k << "\n";
  }
};

// Access to i, j, set(), and show() not inherited.
class derived2:public derived1
{
  int m;
public:
// illegal because i and j are private to derived1
  void setm ()
  {
   m = i - j;
  }                              // Error
  void showm ()
  {
   cout << m << "\n";
  }
};

int main ()
{
  derived1 ob1;
  derived2 ob2;
  ob1.set (1, 2);                // error, can't use set()
  ob1.show ();                   // error, can't use show()
  ob2.set (3, 4);                // error, can't use set()
  ob2.show ();                   // error, can't use show()
  return 0;
}
```

OUTPUT:



```
main.cpp: In member function ‘void derived2::setm()’:
main.cpp:53:9: error: ‘int base::i’ is protected within this context
     m = i - j;
         ^
main.cpp:16:7: note: declared protected here
   int i, j;
       ^
main.cpp:53:13: error: ‘int base::j’ is protected within this context
     m = i - j;
             ^
main.cpp:16:10: note: declared protected here
   int i, j;
          ^
main.cpp: In function ‘int main()’:
main.cpp:65:16: error: ‘void base::set(int, int)’ is inaccessible within this cont
ext
   ob1.set (1, 2);  // error, can't use set()
                ^
main.cpp:18:8: note: declared here
   void set (int a, int b)
        ^~~
main.cpp:65:16: error: ‘base’ is not an accessible base of ‘derived1’
   ob1.set (1, 2);  // error, can't use set()
                ^
main.cpp:66:13: error: ‘void base::show()’ is inaccessible within this context
   ob1.show ();   // error, can't use show()
             ^
main.cpp:23:8: note: declared here
   void show ()
        ^~~~
main.cpp:66:13: error: ‘base’ is not an accessible base of ‘derived1’
   ob1.show ();   // error, can't use show()
             ^
```



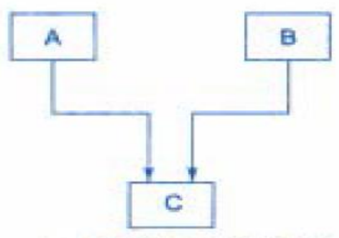
```
main.cpp:67:16: error: ‘void base::set(int, int)’ is inaccessible within this cont
ext
   ob2.set (3, 4);  // error, can't use set()
                ^
main.cpp:18:8: note: declared here
   void set (int a, int b)
        ^~~
main.cpp:67:16: error: ‘base’ is not an accessible base of ‘derived2’
   ob2.set (3, 4);  // error, can't use set()
                ^
main.cpp:68:13: error: ‘void base::show()’ is inaccessible within this context
   ob2.show ();   // error, can't use show()
             ^
main.cpp:23:8: note: declared here
   void show ()
        ^~~~
main.cpp:68:13: error: ‘base’ is not an accessible base of ‘derived2’
   ob2.show ();   // error, can't use show()
```

3. Multiple Inheritance

It is possible for a derived class to inherit two or more base classes. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new class.

General form:

class *derived-class-name : access base-class-name1, access base-class-name2*
{
    // body of class
};



Multiple inheritance

PROGRAM:

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

class base1
{
protected:
  int x;
public:
  void showx ()
  {
    cout << x << "\n";
  }
};
```

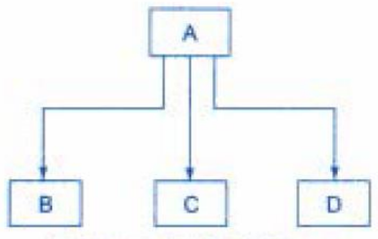
```
class base2
{
protected:
  int y;
public:
  void showy ()
  {
    cout << y << "\n";
  }
};

// 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;
}
```

OUTPUT:

```
10
20
```

4. Hierarchical Inheritance

In this more than one class are derived from a single base class. This supports hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level.



Hierarchical inheritance

PROGRAM:

```
#include<iostream>
```

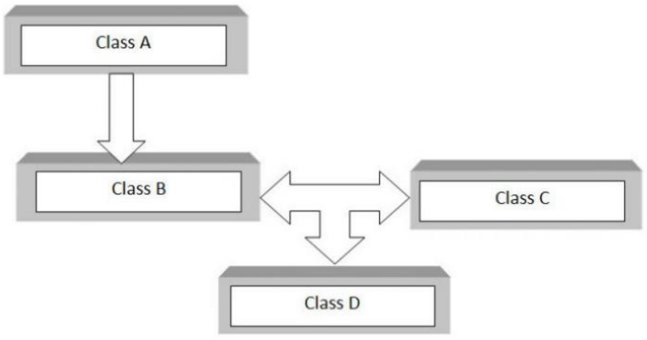
```
using namespace std;

class A                         //single base class
{
public:
  int x, y;
  void getdata ()
  {
    cout << "\nEnter value of x and y:\n";
    cin >> x >> y;
  }
};

class B:public A                //B is derived from class base
{
public:
  void product ()
  {
    cout << "\nProduct= " << x * y;
  }
};

class C:public A                //C is also derived from class base
{
public:
  void sum ()
  {
    cout << "\nSum= " << x + y;
  }
};

int main ()
{
  B obj1;                       //object of derived class B
  C obj2;                       //object of derived class C
  obj1.getdata ();
  obj1.product ();
  obj2.getdata ();
  obj2.sum ();
  return 0;
}                               //end of program
```

OUTPUT:

```
Enter value of x and y:
23 45
Product= 1035

Enter value of x and y:
34 56
Sum= 90
```

5. Hybrid Inheritance

It involves more than one form of any inheritance i.e. we apply two or more types of inheritance as one.



PROGRAM:

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

class A
{
public:
  int x;
};

class B:public A
{
public:
  B ()                              //constructor to initialize x in base class A
  {
    x = 10;
  }
};

class C
{
public:
  int y;
    C ()                       //constructor to initialize y
  {
    y = 4;
  }
};

class D:public B, public C      //D is derived from class B and class C
{
public:
  void sum ()
  {
    cout << "Sum= " << x + y;
  }
};
```

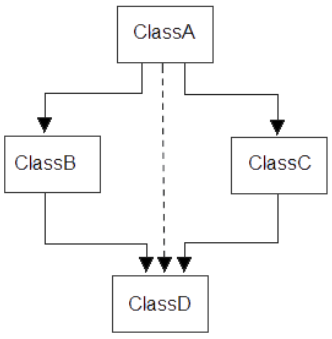
```
int
main ()
{
  D obj1;                         //object of derived class D
  obj1.sum ();
  return 0;
}                                 //end of program
```

OUTPUT:

```
Sum= 14
```

6. Multipath Inheritance

It is a derivation of a class from other derived classes, which are derived from the same base class. This type of inheritance involves other inheritance like multiple, multilevel, hierarchical etc.



PROGRAM:

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

class person
{
    public:
    char name[100];
    int code;
    void input()
    {
        cout<<"\nEnter the name of the person : ";
        cin>>name;
        cout<<endl<<"Enter the code of the person : ";
        cin>>code;
    }
    void display()
    {
        cout<<endl<<"Name of the person : "<<name;
        cout<<endl<<"Code of the person : "<<code;
    }
};
```

```
class account:virtual public person
{
   public:
   float pay;
   void getpay()
   {
      cout<<endl<<"Enter the pay : ";
      cin>>pay;

   }
   void display()
   {
      cout<<endl<<"Pay : "<<pay;
   }
};

class admin:virtual public person
{
   public:
   int experience;
   void getexp()
   {
      cout<<endl<<"Enter the experience : ";
      cin>>experience;

   }
   void display()
   {
      cout<<endl<<"Experience : "<<experience;
   }
};

class master:public account,public admin
{
   public:
   char n[100];
   void gettotal()
   {
      cout<<endl<<"Enter the company name : ";
      cin>>n;
   }
   void display()
   {
      cout<<endl<<"Company name : "<<n;
   }
};

int main ()
{
  master m1;
```

```
    m1.input();
    m1.getpay();
    m1.getexp();
    m1.gettotal();
    cout<<"Displaying Information";
    m1.person::display();
    m1.account::display();
    m1.admin::display();
    m1.display();
  return 0;
}                              //end of program
```

OUTPUT:

```
Enter the name of the person : asd
Enter the code of the person : 1234
Enter the pay : 20000
Enter the experience : 2
Enter the company name : cmr

Displaying Information
Name of the person : asd
Code of the person : 1234
Pay : 20000
Experience : 2
Company name : cmr
```

BASE AND DERIVED CLASS CONSTRUCTION, DESTRUCTORS

There are two important points relative to constructors and destructors when inheritance is involved.

1. When are base-class and derived-class constructors and destructors called?
2. How can parameters be passed to base-class constructors?

When Constructors and Destructors Are Executed

It is possible for a base class, a derived class, or both to contain constructors and/or destructors. 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.

When an object of a derived class is created, the base class' constructor 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.

Constructors are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Destructors be executed in reverse order of derivation. Because the base class underlies the derived class, the destruction of the base object implies the destruction of the derived object. Therefore, the derived destructor must be called before the object is fully destroyed.

PROGRAM:

```
#include <iostream>
```

```
using namespace std;

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

class derived1:public base
{
public:
  derived1 ()
  {
    cout << "Constructing derived1\n";
  }
   ~derived1 ()
  {
    cout << "Destructing derived1\n";
  }
};

class derived2:public derived1
{
public:
  derived2 ()
  {
    cout << "Constructing derived2\n";
  }
   ~derived2 ()
  {
    cout << "Destructing derived2\n";
  }
};

int main ()
{
  derived2 ob;
// construct and destruct ob
  return 0;
}                                    //end of program
```

OUTPUT:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes.

PROGRAM:

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

class base1
{
public:
  base1 ()
  {
    cout << "Constructing base1\n";
  }
   ~base1 ()
  {
    cout << "Destructing base1\n";
  }
};

class base2
{
public:
  base2 ()
  {
    cout << "Constructing base2\n";
  }
   ~base2 ()
  {
    cout << "Destructing base2\n";
  }
};

class derived:public base1, public base2
{
public:
  derived ()
  {
    cout << "Constructing derived\n";
  }
   ~derived ()
  {
    cout << "Destructing derived\n";
  }
};
```

```
int main ()
{
  derived ob;
// construct and destruct ob
  return 0;
}                                   //end of program
```

OUTPUT:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

Constructors are called in order of derivation, left to right, as specified in **derived**'s inheritance list. Destructors are called in reverse order, right to left.

PROGRAM:

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

class base1
{
public:
  base1 ()
  {
    cout << "Constructing base1\n";
  }
   ~base1 ()
  {
    cout << "Destructing base1\n";
  }
};

class base2
{
public:
  base2 ()
  {
    cout << "Constructing base2\n";
  }
   ~base2 ()
  {
    cout << "Destructing base2\n";
  }
};

class derived: public base2, public base1
```

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

int main ()
{
  derived ob;
// construct and destruct ob
  return 0;
}                                    //end of program
```

OUTPUT:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

Passing Parameters to Base-Class Constructors

When the derived class' constructor requires one or more parameters, you simply use the standard parameterized constructor syntax.

But to pass arguments to a constructor in a base class, we have to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors.

General form of expanded derived-class constructor declaration

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

base1 through *baseN* are the names of the base classes inherited by the derived class. A colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes.

PROGRAM:

```
#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;
}
```

OUTPUT:

```
Constructing base
Constructing derived
4 3
Destructing derived
```

Destructing base

The derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class.

PROGRAM:

```
#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 base1\n";
  }
};

class derived:public base1, public base2
{
  int j;
public:
    derived (int x, int y, int z):base1 (y), base2 (z)
  {
    j = x;
    cout << "Constructing derived\n";
  }
   ~derived ()
  {
    cout << "Destructing derived\n";
  }
  void show ()
  {
    cout << i << " " << j << " " << k << "\n";
  }
};

int main ()
{
  derived ob (3, 4, 5);
  ob.show ();                          // displays 4 3 5
  return 0;
}
```

OUTPUT:

```
Constructing base1
Constructing base2
Constructing derived
4 3 5
Destructing derived
Destructing base1
Destructing base1
```

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. In this situation, the arguments passed to the derived class are simply passed along to the base.

PROGRAM:

```
#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;
}
```

OUTPUT:

```
Constructing base1
Constructing base2
Constructing derived
3 4
```

```
Destructing derived
Destructing base2
Destructing base1
```

A derived class' constructor 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. Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. 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"; }
```

One final point to keep in mind when passing arguments to base-class constructors: The argument can consist of any expression valid at the time. This includes function calls and variables. This is in keeping with the fact that C++ allows dynamic initialization.

VIRTUAL BASE CLASS

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

PROGRAM:

```
// 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;
}
```

OUTPUT:

```
main.cpp: In function ‘int main()’:
main.cpp:40:6: error: request for member ‘i’ is ambiguous
   ob.i = 10;   // this is ambiguous, which i???
^
main.cpp:14:7: note: candidates are: int base::i
   int i;
       ^
main.cpp:14:7: note:                 int base::i
main.cpp:44:15: error: request for member ‘i’ is ambiguous
   ob.sum = ob.i + ob.j + ob.k;
               ^
main.cpp:14:7: note: candidates are: int base::i
   int i;
       ^
main.cpp:14:7: note:                 int base::i
main.cpp:46:14: error: request for member ‘i’ is ambiguous
   cout << ob.i << " ";
              ^
main.cpp:14:7: note: candidates are: int base::i
   int i;
       ^
main.cpp:14:7: note:                 int base::i
```

Both **derived1** and **derived2** inherit **base**. However, **derived3** inherits both **derived1** and **derived2**. This means that there are two copies 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.i**s!, the statement is inherently ambiguous.

Solutions:

There are two ways to remedy the preceding program.

1. Apply the scope resolution operator to i and manually select one i.

PROGRAM:

```
// This program uses explicit scope resolution to select i.
#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.derived1::i = 10; // scope resolved, use derived1's i
ob.j = 20;
ob.k = 30;
// scope resolved
ob.sum = ob.derived1::i + ob.j + ob.k;
// also resolved here
cout << ob.derived1::i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

OUTPUT:

```
10 20 30 60
```

This solution raises a deeper issue: What if only one copy of **base** is actually required? Preventing two copies from being included in **derived.**

2. Virtual Base Class

When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited. You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited.

PROGRAM:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;

class base
{
public:
  int i;
};

// 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;
};

/* derived3 inherits both derived1 and derived2. This time, there is only one copy of base
class. */
class derived3:public derived1, public derived2
{
public:
  int sum;
};

int main ()
{
  derived3 ob;
  ob.i = 10;                         // now unambiguous
  ob.j = 20;
  ob.k = 30;
// unambiguous
  ob.sum = ob.i + ob.j + ob.k;
// unambiguous
  cout << ob.i << " ";
  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;
  return 0;
}
```

OUTPUT:

```
10 20 30 60
```

Difference between a normal base class and a virtual base class

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 AND POLYMORPHISM

POLYMORPHISM

Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation.

Polymorphism refers to the ability to associate multiple meanings to one function name.

Polymorphism is supported by C++ both at compile time and at run time. Compile time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions

STATIC AND DYNAMIC BINDING

Early Binding or Static 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.

The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

Late Binding or Dynamic Binding

The opposite of early binding is *late binding*. 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. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time.

The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.

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. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*.

DYNAMIC BINDING THROUGH VIRTUAL FUNCTIONS

When accessed "normally," virtual functions behave just like any other type of class member function. Virtual functions behaviour when accessed via a pointer is what which makes them important and capable of supporting run-time polymorphism

A base-class pointer can be used to point to an object of any class derived from that base. 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. The same effect applies to base-class references.

PROGRAM:

```
#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;
}
```

OUTPUT:

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

Rules for Virtual Functions:

When virtual functions are created for implementing late binding, observe some basic rules that satisfy the compiler requirements.

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer points to any type of the derived object, the reverse is not true. i.e. we cannot use a pointer to a derived class to access an object of the base class type.
9. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.

VIRTUAL FUNCTION CALL MECHANISM

1. Normal manner

You can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved.

For example, assuming the preceding example, this is syntactically valid:

d2.vfunc(); // calls derived2's vfunc()

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc( )**.

2. Calling a Virtual Function through a Base Class Reference

Polymorphic nature of a virtual function is also available when called through a base-class reference. A reference is an implicit pointer. Thus, 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.

PROGRAM:

```
/* 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;
}
```

OUTPUT:

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

PURE VIRTUAL FUNCTIONS

Situations leading to Pure Virtual Functions

When a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, 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.

Definition: A *pure virtual function* is a virtual function that has no definition within the base class.

General form:

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

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

PROGRAM:

```
#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;
}
```

OUTPUT:

```
20
14
24
```

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.

PROGRAM:

```
// C++ Program to Illustrate Abstract Class
#include <iostream>
using namespace std;

class Abstract
{
  int i, j;
public:
    virtual void setData (int i = 0, int j = 0) = 0;
  virtual void printData () = 0;
};

class Derived:public Abstract
{
  int i, j;
public:
  Derived (int ii = 0, int jj = 0):i (ii), j (jj)
  {
    cout << "Creating object " << endl;
  }
  void setData (int ii = 0, int jj = 0)
  {
    i = ii;
    j = jj;
  }
  void printData ()
  {
    cout << "Derived::i = " << i << endl << "Derived::j = " << j << endl;
  }
};

int main ()
{
  // Cannot create an instance of Abstract Class
  // Abstract a;
  Derived d;

  cout << "Current data " << endl;
  d.printData ();
  d.setData (10, 20);
  cout << "New data " << endl;
  d.printData ();
}
```

OUTPUT:

```
Creating object
Current data
Derived::i = 0
Derived::j = 0
New data
Derived::i = 10
Derived::j = 20
```

VIRTUAL DESTRUCTORS

Inheritance also lends itself to *virtual methods*, where implementation is provided by any specific subclasses. However, once an inheritance hierarchy is created, with memory allocations occurring at each stage in the hierarchy, it is necessary to be very careful about how objects are destroyed so that any memory leaks are avoided. In order to achieve this, we make use of a *virtual destructor*.

In simple terms, a virtual destructor ensures that when derived subclasses go *out of scope* or are deleted the order of destruction of each class in a hierarchy is carried out correctly. If the destruction order of the class objects is incorrect, in can lead to what is known as a *memory leak*. This is when memory is allocated by the C++ program but is never deallocated upon program termination. This is undesirable behaviour as the operating system has no mechanism to regain the lost memory (because it does not have any references to its location!). Since memory is a finite resource, if this leak persists over continued program usage, eventually there will be no available RAM (random access memory) to carry out other programs.

For instance, consider a pointer to a base class (such as PayOff) being assigned to a derived class object address via a reference. If the object that the pointer is pointing to is deleted, and the destructor is not set to virtual, then the base class destructor will be called instead of the derived class destructor. This can lead to a memory leak. Consider the following code:

```
class Base
{
public:
 Base();
 ~Base();
};

class Derived : public Base {
private:
  double val;
public:
 Derived(const double& _val);
 ~Derived();
}

void do_something() {
 Base* p = new Derived;
 // Derived destructor not called!!
 delete p;
}
```

What is happening here? Firstly, we create a base class called Base and a subclass called Derived. The destructors are NOT set to virtual. In our do_something() function, a pointer p to a Base class is created and a reference to a new Derived class is assigned to it. This is legal as Derived *is a* Base.

However, when we delete p the compiler only knows to call Base's destructor as the pointer is pointing to a Base class. The destructor associated with Derived is not called and val is not deallocated.

A memory leak occurs!

Now consider the amended code below. The virtual keyword has been added to the destructors:

```
class Base {
public:
 Base();
 virtual ~Base();
};

class Derived : public Base {
private:
  double val;
public:
 Derived(const double& _val);
 virtual ~Derived();
}

void do_something() {
 Base* p = new Derived;
 // Derived destructor is called
 delete p;
}
```

What happens now? Once do_something() is called, delete is invoked on the pointer p. At code execution-time, the correct destructor is looked up in an object known as a *vtable*. Hence the destructor associated with Derived will be called prior to a further call to the destructor associated with Base. This is the behaviour we originally desired. val will be correctly deallocated.
No memory leak this time!