Unit 7 Inheritance

Structure:

7.1 Introduction

In the previous unit you studied about operator overloading and type conversions. In this unit you are going to study the concept of inheritance. You will learn about types of inheritance and multiple inheritances. The method by which new classes called 'derived classes' are created from the exiting classes called 'base classes' is known as 'inheritance'. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class. For example, you can create a base class called 'employee', and define the derived classes as manager, supervisor, accountant etc. All of these derived classes have all the features of their base class (employee), and yet differ because they have some more specific features added.

Objectives:

After studying this unit you should be able to:

- explain the concept of inheritance
- discuss base and derived classes
- describe public, private and protected inheritance
- discuss function overriding
- discuss the different types of inheritance
- explain multiple inheritances
- explain constructors in derived classes

7.2 Inheritance in C++

Inheritance is a very powerful feature of object-oriented programming. It allows reuse of code without modifying the original code. It also provides flexibility to the programmer to make modifications to the program without altering the original code which saves debugging and programming time and effort.

Inheritance feature enables distribution of class libraries which allows the programmer to use the standard code developed by some other company. Inheritance is the process of creating new classes known as derived classes from the existing or base class. The features of the base class or parent class are said to be inherited by the derived class or child class. The child class has all the functionality of the parent class, and has additional functionalities of its own.

As you know, the class data members are declared in the private section of class definition. Member function of the derived class cannot access the private data members of the base class. So the data members of the base class should be declared as 'protected' since they will be inherited by another class. The protected access specifier makes the data members inaccessible outside that class. But the protected data members of the base class can be accessed by the 'members and friends' function of that base class, and members and friends of any class derived from the base class. Derived class member functions can refer to public and protected member functions of the base class. The accessibility of base class members is also dependent on the type of inheritance, private or public. The public inheritance is commonly used which is shown in the program inheritance.cpp discussed next. The types of inheritance are discussed in the sections follow.

Advantages of Inheritance:

Reusability:

Inheritance helps the code to be reused in many situations. The base class is defined, and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

Saves Time and Effort:

The above concept of reusability achieved by inheritance saves the programmer's time and effort. Since the main code written can be reused in various situations as needed, it leads to increase in program structure which, in turn, results in greater reliability.

7.3 Public, Private and Protected Inheritance

A derived class can be defined as follows:

```
class derived_classname: access specifier baseclassname
{
 Members of derived class
};
```

The access specifer can be public, private or protected.

Access specifier *public:*

When we derive a class using public base class access specifier, all inherited public members of the base class will become public members of the derived class. All inherited protected members of the base class will become protected members of the derived class. All inherited private members will remain private to the base class, and cannot be accessed by the member functions of the derived class.

Access specifier *protected*:

When you derive a class using a protected base class access specifier, then all the inherited public and protected members of the base class will become protected members of the derived class. And all the private members of the base class cannot be accessed by the member functions of the derived class.

Access specifier *private:*

When you derive a class using a private base class access specifier, all inherited public and protected members of the base class will become private members of the derived class. All the private members of the base class cannot be accessed by the member functions of the base class.

The following program implements the inheritance. The class manager is inherited from the class employee.

```
//inheritance.cpp
# include<iostream.h>
```

```
# include<string.h>
# include<conio.h>
class employee
{
protected:
 int empno;
char ename[25];
public:
 employee()
 { empno=0;
 strcpy(ename,"");
}
employee(int n, char ch[25])
{ empno=n;
strcpy(ename,ch);
}
void display()
{cout<<"Emp Code:"<<empno;
cout<<"Name:"<<ename;
}
};
class manager: public employee
 {
protected:
float basic;
float hra;
public:
manager():employee()
{ basic=0.0; hra=0.0;}
manager(int n, char ch[25],float i, float j): employee(n,ch)
{ basic=i; hra=j;}
void display()
{ employee ::display();
cout<<"Basic"<<basic<<endl;
cout<<"HRA"<<hra<<endl;
}
};
```

```
void main()
{
clrscr();
employee e1(106,"amit");
manager m1(205,"pawan",40000.00,5000.00);
e1.display();
m1.display();
getch();
}
```

As you can see, in the above program, there is a class employee having data member's empcode and ename. These are declared as protected data members so that they can be inherited by the derived class. The employee class contains two constructors and a member function named 'display'.

We have derived the manager class form the class employee using the following statement:

class manager: public employee

Here public is a keyword which specifies that the class is derived publically from the class employee. You will study the types of inheritance in the next section. The derived manager class will have inherited data members i.e. ename and empcode of the base class employee. It also contains its own data members i.e. basic and hra.

The derived manager class also contains the constructors and a member function named as 'display'. In the class manager also, we have defined constructors and a member function display. While initializing the data members of the manager class, the respective members of the parent class will also have to be initialized. The constructors of the parent class are invoked by the following statements: manager(): employee()

manager(int n,char ch[25],float i, float j): employee(n,ch)

You can also assign default values to the members.

While displaying the contents of the manager class, respective members of the parent class are also displayed. The display function of the base class is invoked in 'the display function' of the derived class. This is done by the following statement:

employee::display()

As you can see, the class name, followed by the scope resolution operator, is used to refer the function name belong to the class. When both the base class and the derived class have a function with the same name, the object.functionname statement with the base class object will always access function defined in the base class as the base class does not know anything about the derived classes. If the object is a derived class object, then the function defined in the derived class will be invoked. This feature is known as function overriding. If the function is not defined in the derived class, then the object will access the function in the parent class.

Self Assessment Questions

1. The derived class object can access ____ members of the parent class.
2. The derived class member functions can access _____ members of the parent class.
3. Inheritance allows _____ of the program code.

7.4 Types of Inheritance

It is possible that a class inherits its properties from more than one class or from more than one level. On this basis the inheritance is classified as follows:

1. Single Inheritance
2. Multiple Inheritance
3. Multi-level Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance

As you already know, inheritance is a process of deriving a new class from an existing base class. When only one class is derived from a single base class, then the inheritance is known as single inheritance.

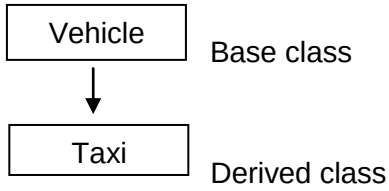


Figure 7.1: Single inheritance

In the above figure 7.1 vehicle is the base class and the taxi is the derived class, which describes the vehicle of a special type.

2. Multiple Inheritance

The derived class can also have multiple parents, which is known as multiple inheritance. Here the child - or the derived class - has two or more parent classes as depicted in figure 7.2. The child class inherits all the properties of all its parents. Multiple inheritance is implemented in a similar way as single inheritance except that both the parent names have to be specified while defining the class.

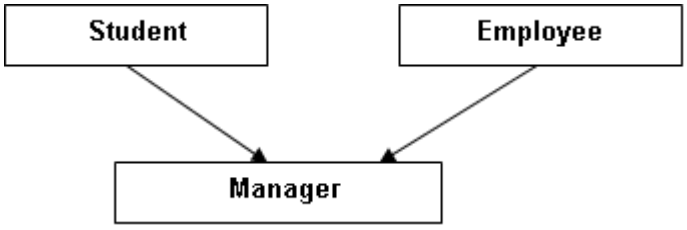


Figure 7.2: Multiple Inheritance

3. Multi-level Inheritance

When a class is derived from the derived class it is known as multilevel inheritance. In such case, the grandchild class inherits all the properties of the child and the parent classes as shown in figure 7.3.

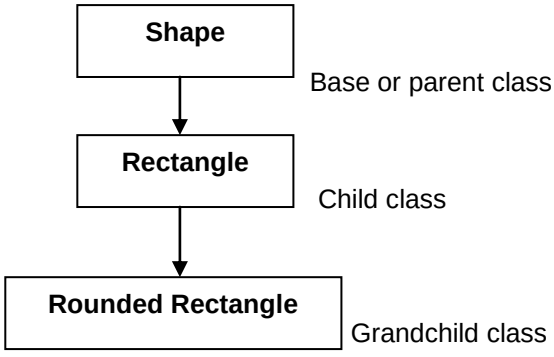


Figure 7.3: Multi-level Inheritance

4. Hierarchical Inheritance

Many programming problems are cast into hierarchy when certain features of one level are shared by many others below that level. The example of the

hierarchical classification of students in university is shown in figure 7.4 below.

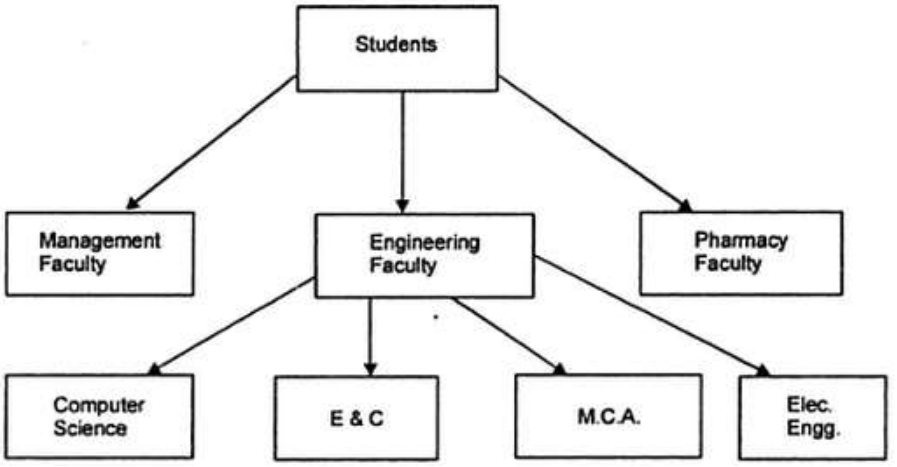


Figure 7.4: Hierarchical Inheritance example

5. Hybrid Inheritance

If we apply more than one type of inheritance to design a problem, then it is known as hybrid inheritance. Here, a new class can be created from multiple and multi-level classes, or from the multiple and hybrid classes. The figure 7.5 shows an example of hybrid inheritance using multiple and multi-level inheritance.

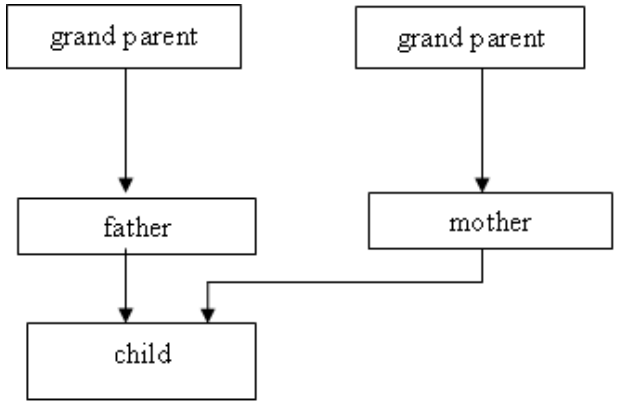


Figure 7.5: Example of hybrid inheritance

As you can see in the above figure, the child may inherit features from the grandparents, father and mother. Here inheriting the features of grandparent

father and grandparent mother is called multi-level inheritance, and inheriting the features of father and mother is called multiple inheritance.

You can see in the above example that the manager class is derived from both the student and the employee class. This is useful in instances where you want to create an employee whose educational qualifications need not be stored in the separate class.

Self Assessment questions

4. When the derived class has more than one parent, it is known as _____.
5. In _____ inheritance, objects of derived class cannot access public member functions of the base class.
6. A class can be derived further from the derived class. (True/ False)

7.5 Function Overriding

If a class is inherited in the derived class, and if one of the functions of the base class is again defined in the derived class, then that function is said to be overridden and this procedure is known as function overriding.

Requirements of function overriding

- Inheritance should be present**.**
- The function which is overridden should have the same definition in both the derived and base class, i.e. the function should have the same name, return type and parameter list.

The figure 7.6 shows an example of function overriding.

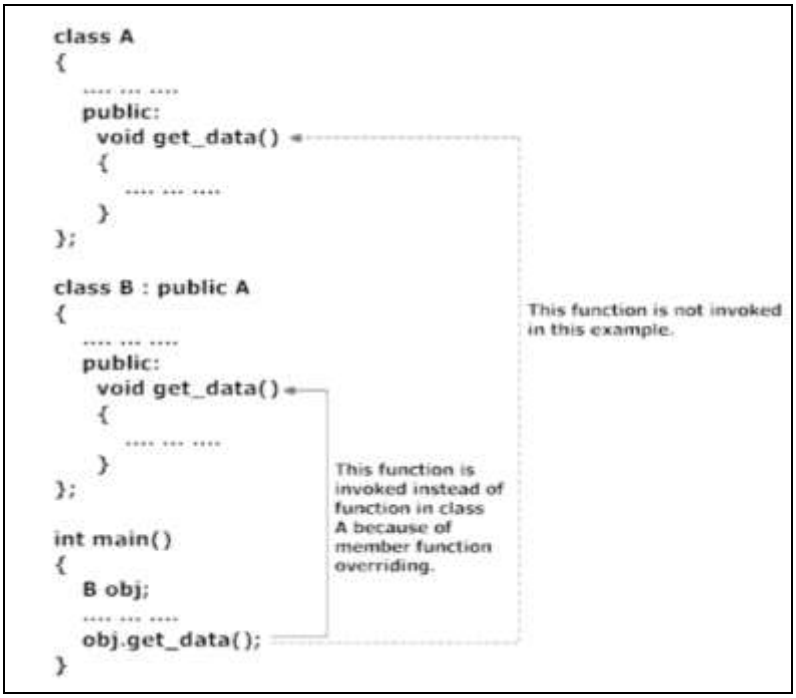


Figure 7.6: Function overriding in C++

Accessing the Overridden Function in Base Class from Derived Class

If you want to access base class' overridden function in derived class, then use base class name, and then the scope resolution operator (:), and finally, the name of the overridden function. For example, as shown in figure 7.6, if you want to access get_data() function of base class from derived class, then you should use the statement:

A::get_data(); // calls get_data() function of class A.

Example of function overriding

```
class A
{
 public:
  void show()
   {
     cout << “base class”;
  }
```

```
};
class B: public A
{
   public:
   void show ()
   {
     cout<< " derived class";
  }
};
void main()
{
  A obj1;                  //object of base class
  B obj2;                  //object of derived class
  obj1.show();      // calls the function show defined in base class A
  obj2.show();             // calls the function show defined in derived class B
}
```

Self Assessment Questions

7. _____ is the process in which a class is inherited in the derived class and the one of the functions of the base class is again defined in the derived class.
8. The function which is overridden can have different definitions in both the derived and the base classes. (True/False)

7.6 Multiple Inheritance

You have already studied the concept of multiple inheritance in the previous section. In this section we will study this concept in detail along with the ambiguities that arise in this type of inheritance. Multiple Inheritance is the process of inheriting a class from more than one parent class. This would be required in several instances where you would like to have the functionalities of several classes. This is also extensively used in class libraries. To derive from more than one base class, you separate each base class by commas in the class designation as shown below:

```
class Derived : [virtual] [access_type] base1, [virtual] [access_type] base2,……. [virtual] [access_type] baseN
 {
   ……
};
```

Here base1, base2, baseN are direct bases of *Derived,* and each of them should have a distinct name.

An access_type can be private, public or protected, and follows the same access rules as single inheritance.

The keyword 'virtual' is optional, and specifies a sharable base.

Member functions or data that have the same name in base1, base2 or baseN are potential ambiguities.

Here are several examples of multiple inheritance declarations:

class A : public B, public C { ….. };

Class A derives publically from base classes B and C like single inheritance.

class D: public E, private F, public G{…..};

Class D derives publically from E and G and privately from F. This derivation makes D a subtype pf E and G and not a subtype of F. C

class X: Y, Z {….};

Class X derives privately from both y and Z by default as no access specifier is mentioned.

class M: virtual public N, virtual public p {….};

Here N and P are virtual bases of M. which we will discuss in unit 8.

Let us implement a program where there are two classes namely 'student' and 'employee'. We shall derive a class manager from the above two classes and see how member functions and constructors are implemented in multiple inheritance:

```
//multiple.cpp
# include<iostream.h>
# include<string.h>
# include<conio.h>
class student
{protected:
char qual[6];                    // highest degree earned
int percent;                     // percentage score in the last degree
public:
student()
```

```
{strcpy(qual, “”); percent=0;}
student(char ch[6], int p)
{strcpy(qual, ch); percent=p;}
void display()
{cout<<endl<<”Qualification”<<qual;
cout<<endl<<”Score”<<percent;
}} ;
class employee {
protected:
int empno;
char ename[25];
public:
employee()
{empno=0;
strcpy(ename,"");
}
employee(int n, char ch[25])
{empno=n;
strcpy(ename,ch);
}
void display()
{cout<<endl <<"Emp Code:"<<empno;
cout<<endl <<"Name:"<<ename;
}
};
class manager: public employee, public student
 {
protected:
float basic;
float hra;
public:
manager():employee(),student()
{basic=0.0; hra=0.0;}
manager(int n,char ch[25], char ch1[6], int p, float i, float j): employee(n,ch),
student(ch1,p)
{basic=i; hra=j;}
void display()
```

```
{ employee::display();
  student::display();
cout<<endl <<"Basic"<<basic;
cout<<endl <<"HRA"<<hra;
}
};
void main()
{
clrscr();
manager m1(205, “pawan”, “MBA”, 80, 40000.00, 5000.00);
m1.display();
getch();
}
```

As you can see in the above program, both the parent class and constructors are called by the constructors of derived class. This is because every object of the derived class has its own copy of parent data members. Therefore, their initializations too is required. The parent class member functions are invoked using the scope resolution operator as shown in the display function of the manager class.

The output of the above program will be:
Emp Code:205
Name:pawan
Qualification MBA
Score 80
Basic 40000
HRA 5000

Ambiguity in multiple Inheritance

There are several types of ambiguities that might arise in the process of implementation of multiple inheritance. Let us suppose that there are two parent classes P and Q. And class R is the derived class of P and Q. Suppose that there is a function ‘func1()’ defined in both the parent classes P and Q, but this func1() has not been defined in the child class. When the child class object (obj) tries to access the function func1() through a statement obj.func1(), there is compiler error. The reason is that this statement is ambiguous for the compiler as it will not be able to find out

which parent's f1() function is called. The ambiguity can be resolved by prefixing the parent class name followed by scope resolution operator before the function name. The following statement would resolve the ambiguity.

obj.P::func1();

This statement tells the compiler to use the function func1() of parent class P. Another solution would be to introduce a dummy function func1() in Class R, and invoke the applicable parent functions.

Another common ambiguity that arises is in the case of diamond inheritance. The situation when both parent classes are derived from a single parent is known as diamond inheritance. The situation is shown in figure 7.7.

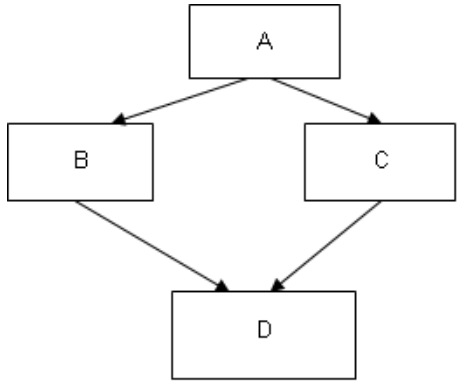


Figure 7.7: Diamond Inheritance

```
class X
{protected:
int a;};
class Y: public parent
{ };
class Z: public parent
{ };
class P: public X, public Y
{ public:
 int f1()
  return a;              //ambiguous
};
```

Let us consider a situation where there is a parent class X having a protected data member a. Two child classes X and Z are derived publically from parent class X.

Class P is derived publically from base classes X and Y. There arises an ambiguity when grandchild P tries to access the data member of parent class X. The ambiguity arises when classes Y and Z are derived from X: each inherits a copy of X called sub-object, and inherits an own copy of the parent data. The ambiguity arises for the grandchild in resolving which copy of the child class subobject is to be accessed.

The solution for this is virtual base class. In this, the common base class is made virtual base class while declaring direct or intermediate base class.

For example:

By making classes Y and Z as virtual classes, the two classes will share a common subobject, and will result in resolving the ambiguity as shown below.

```
class X
{protected:
int a;};
class Y: virtual public X
{};
class Z: virtual public X
{};
class P: public Y, public Z
{ public:
 int f1()
  return a;                  //only one copy of the parent
};
```

Self Assessment Questions

9. _____ is the process of inheriting a class from more than one parent class.
10. In multiple inheritance, when a class is derived from more than one base class, each base class is separated by commas in the class definition. (True/False)
11. The situation where both parent classes are derived from a single parent is known as _____.

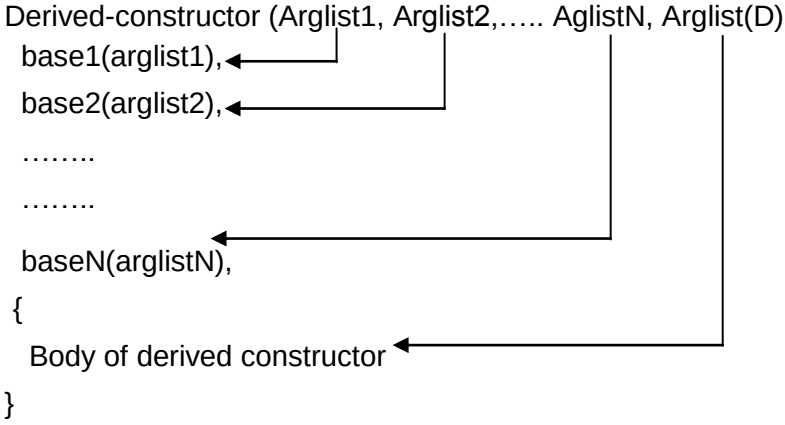
7.7 Constructors in Derived Classes

You already know that constructors play an important role in initializing the objects. One thing you should note is that, as long as no base class object takes any argument, the derived class need not have a constructor function. But there is a constructor in any base class with one or more arguments. Then it is compulsory for the derived class to have a constructor and pass the arguments to the base class constructors. In inheritance, we create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When the constructors are present in both the derived and the base classes, the base class constructor is executed first, and then the constructor in the derived class is executed.

In the case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multi-level inheritance, the constructors will be executed in the order of inheritance.

Since it is the responsibility of the derived classes to provide initial values to its base classes, when a derived class object is declared, we supply initial values that are required by all the classes together. For such situations C++ has a special argument passing mechanism.

In these situations, the constructor of the derived class receives the entire list of values as its arguments, and passes them on to the base class in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructors. The general form of defining a derived constructor is as follows:



The header line of the derived-constructor function contains two parts separated by colon(:). The declaration of the arguments is provided by the first part which is passed to the derived constructor. The second part lists the function calls to the base constructors.

base1(arglist1), base2(arglist2)…. Are function calls to the base constructors base1(), base2(),…. And the arglist1, arglist2….etc. denotes actual parameters that are passed to the constructor of the base class. Arglist1 to Arglist N are the argument declarations for the base constructors base1 to base N. ArglistD provides the parameters that are necessary to initialize the members of the derived class.

For example:

```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),      //call to constructor A
B(b1, b2)       //call to constructor B
 {
   d=d1;        //executes its own body
}
```

Here A(a1, a2) invokes the base constructor A(), and B(b1, b2) invokes another base constructor B(). The constructor D() supplies the values for these four arguments. D() may be invoked as follows:

```
…….
D objD(5, 12, 2.5, 7.54, 30)
…….
```

These values are assigned to various parameters by the constructor D() as follows:

```
5 ->  a1
12 -> a2
2.5 -> b1
7.54 -> b2
30 ->   d1
```

The constructors of the virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed.

The following program demonstrates how the constructors are implemented when the classes are inherited.

```
#include<iostream.h>
class alpha
{
  int x;
 public:
 alpha(int i)
 {
   x=i;
   cout << “” alpha initialized”;
}
void show_x(void)
{
 cout << “x=” << x \n”;
}
};
class beta
{
   float y;
   public:
    beta(float j)
    {
      y=j;
      cout<< “beta initialized”;
   }
void show_y(void)
  {
   cout << “y=” << y << “\n”;
  }
};
class gamma (int a, float b, int c, int d): alpha(a), beta(b)
{
 m=c;
  n=d;
 cout << “gamma initialized\n”;
}
```

```
void show_mn(void)
 {
   cout<< “m = “ << m << “\n” << “n=” << n << “\n”;
}
};
void main()
 {
    gamma g(5, 10, 75, 20, 30)
     cout << “\n”;
    g.show_x();
    g.show_y();
    g.show_mn();
}
```

The output of the program will be:

beta initialized
alpha initialized
gamma initialized
x=5
y=10.5
m=20
n=30

Self Assessment Questions

12. In the case of _____ the base classes are constructed in the order in which they appear in the declaration of the derived class.
13. The constructors of the virtual base classes are invoked before any non-virtual base classes. (True/False)

7.8 Summary

- Inheritance allows creating a class known as derived class from a class known as base class, and inheriting all the properties of the parent class allows programs to be reused without rewriting entire code.
- The members that can be inherited have to be declared using protected access specifier. There can be several levels of inheritance and the derived class can be inherited from multiple parents as well.
- Inheritance helps the code to be reused in many situations and this concept of reusability saves the programmer’s time and effort.

- A derived class can be defined as follows:
  class derived_classname: access specifier baseclassname
  {
  Menmbers of derived class
  };
  Inheritance can be public, private or protected, depending on the access specifier in the above declaration.
- Inheritance is classified as follows: Single Inheritance, Multiple Inheritance, Multi-level Inheritance, Hierarchical Inheritance and Hybrid Inheritance.
- Multiple Inheritance is the process of inheriting a class from more than one parent class.
- If there is a constructor in any base class with one or more arguments, then it is compulsory for the derived class to have a constructor, and pass the arguments to the base class constructors.
- In the case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance.

7.9 Terminal Questions

1. Write the output of the following program

```
# include  <iostream.h>
# include<conio.h>
class A {
public:
A() {cout<<"Null Constructor for A" <<endl;}
};
class B: public A {
public:
B()  {cout<<"Null Constructor for B" <<endl;}
B(int x) {cout<< "Int constructor for B"<<endl;}
};
class C: public B {
public:
C() {cout<<"Null Constructor for C" <<endl;}
```

```
C(int x) : B(x){cout<< "Int constructor for C"<<endl;}
};
void main()
{
clrscr();
A a;
B b;
B bobj(5);
C c;
C cobj(1);
getch();
}
```

2. Write the output of the following program

```
# include  <iostream.h>
# include<conio.h>
class A {
public:
A() {cout<<"Null Constructor for A" <<endl;}
~A() {cout<< "Destructor for A"<<endl;}
};
class B: public A {
public:
B()  {cout<<"Null Constructor for B" <<endl;}
~B() {cout<< "Destructor for B"<<endl;}
};
class C: public B {
public:
C() {cout<<"Null Constructor for C" <<endl;}
~C() {cout<< "Destructor for C"<<endl;}
};
void main()
{
C c;
getch();
}
```

3. Explain public, private and protected inheritance with examples.
4. List and explain different types of inheritance.
5. Describe the ambiguities of multiple inheritance.
6. Explain function overriding with the help of an example.
7. Explain how to use constructors in derived classes.

7.10 Answers

Self Assessment Questions

1. public
2. public and protected
3. reusability
4. Multiple Inheritance
5. Private Inheritance
6. True
7. Function overriding
8. False
9. Multiple inheritance
10. True
11. Diamond Inheritance
12. Multiple Inheritance
13. True

Terminal Questions

1. Output is:
   Null Constructor for A
   Null Constructor for A
   Null Constructor for B
   Null Constructor for A
   Int Constructor for B
   Null Constructor for A
   Null Constructor for B
   Null Constructor for C
   Null Constructor for A
   Int Constructor for B
   Int Constructor for C

2. **Output is**
   Null Constructor for A
   Null Constructor for B
   Null Constructor for C
   Destructor for C
   Destructor for B
   Destructor for A

3. A derived class can be defined as follows:
   class derived_classname: access specifier baseclassname
   {
   Members of derived class
   };
   The access specifer can be public, private or protected. Depending on the access specifier, the inheritance can be public, private or protected. (Refer section 7.3 for more details).

4. The different types of inheritance are: Single Inheritance, Multiple Inheritance, Multi-level Inheritance, Hierarchical Inheritance, and Hybrid Inheritance. (Refer section 7.4 for more details).

5. There are several types of ambiguity that might arise during the implementation of multiple inheritance. Let us suppose that there are two parent classes P and Q. And class R is the derived class of P and Q. Suppose further that there is a function func1() defined in both the parent classes P and Q, but func1() has not been defined in the child class. When the child class object (obj) tries to access the function func1() through a statement obj.func1(), there is a compiler error. The reason is that this statement is ambiguous for the compiler, as it will not be able to find out which parent's f1() function is called. Another common ambiguity that arises is in the case of diamond inheritance. (Refer section 7.6 for more details).

6. If a class is inherited in the derived class, and one of the functions of the base class is again defined in the derived class, then that function is said to be overridden, and this procedure is known as function overriding. (Refer section 7.5 for more details).

7. If there is a constructor in any base class with one or more arguments, then it is compulsory for the derived class to have a constructor and

pass the arguments to the base class constructors. In inheritance, we create objects using the derived class. When the constructors are present in both the derived and the base class, the base class constructor is executed first and then the constructor in the derived class is executed. (Refer section 7.7 for more details).

References:

- *Object-Oriented C++ Programming, First edition,* by Hirday Narayan Yadav. Firewall Media.
- *Interfacing with C++: Programming Real-World Applications*, by Jayantha Katupitiya, Kim Bentley. Springer Science & Business Media.
- *Object-oriented Programming with C++ - Sixth Edition*, by E Balagurusamy. Tata McGraw-Hill Education.
- http://www.programiz.com/