Chapter 7

Inheritance

In the last two chapters, we have discussed the key OO issues of Encapsulation, Abstraction and Operator Overloading. In this and the following chapter, we shall concentrate on two more key issues of inheritance and polymorphism. Well, the main advantage of OO language is the extent to which it supports reuse and inheritance plays a crucial role in supporting this feature as a part of the language set-up. Inheritance aims to make use of existing class declarations and definitions as a part of new software development wherever possible and thereby minimize the coding effort and time. Inheritance promotes the feature of resuability and is similar in concept to the syntax of standard library functions supported in languages such as C. Here, instead of function level resuability, class level reuse is supported. This chapter primarily concentrates on inheritance discussing the various issues involved, types of inheritance and their associated syntaxes in C++.

7.1 Reusability and Its Benefits

Software engineering observations over the years have been that program development time and code maintenance are lot more easier when software is created or built using existing code. In other words, reusability brings down coding effort or program development time, since programmers do not develop code from the scratch. Rather they make use of classes that are already available [with or without modifications] and their interfaces [functions] to solve the problem on hand [effectively coding]. Since codes that have been already created are used, the program or coding time is brought down (in fact drastically).

Another hidden advantage of reusability is that it facilitates easier maintenance. How? Codes that are reused or distributed have been tested and tried. The chances that such codes are susceptible to errors is very minimal. Thus, user-created programs, which reuse such code, is less-prone to errors or bugs and thus is lot more easier to maintain. Why on earth should we discuss reusability, all of a sudden? Well, inheritance is a key form of reusability. Inheritance in reality is that trait wherein entities inherit (derive/extract/imbibe) properties from other entities.

7.2 Parent–Child Relationship

A child inherits certain properties from the parent. Whether it is for the good or not, there is a separate class of diseases that are referred by medics as Hereditary. Biologists observe that a child imbibes almost 50% of its characteristics from its parent. All engineering or

for that matter any concept existing today is an imitation or an evolution of some natural phenomenon. In programming (OO) terminologies, inheritance is that concept wherein new classes are created from existing classes by extracting the features [attributes/data members] and behaviours [member functions] of existing classes.

Well, by rule of nature, the creator is referred to as the parent and the entity created by the parent is referred to as the child. Similarly, in OO language the existing class is commonly referred to as the parent or base class and the new class is referred to as the child or the derived class. To get into actual programming terminologies, inheritance helps the programmer in creating new classes by absorbing [inheriting] the data members and member functions of existing classes. Thus, the effort involved in the creation of the new class is brought down.

On the lighter side of it, again rule of nature is that the hierarchy does not stop with a child. As years go by the child becomes the parent. In programming terminologies a derived class can itself act as a parent or base class for another class. Nothing but to say that a derived class may be derived further. Programming requirements could be that a class derives or inherits its features from more than one class or there are more than one parent or base class for a derived class. Well, we do not get into correlation with rule of nature, at least here.

But of course, in terms of the properties or characteristics of a child, a grand parent also has some effect. In programming terminologies the case, when a class is created by inheriting features from a single class, is referred to as single inheritance. When a class derives its features from more than one class, it is referred to as multiple inheritance. In this chapter, we shall at length elaborate on both forms of inheritance. Thus, whether it is single or multiple inheritance, a derived class extracts features [data member or member functions] from the base class. Again by nature's law a child may have his or her own unique traits.

7.3 C++ Notion of Inheritance

Derived class if required can have additional data members or member functions, in addition to what it has inherited from the base class. This is to say that derived classes will be larger than that of base classes. Again nature's law, 'whether you like or not you inherit your parents' features'. To put it in simple terms, derived classes will always be of size greater than or equal to that of the base classes. The minimal size of the derived class is when it does not have any unique [additional features], i.e. data members or member functions. In terms of size, most probably derived classes will always be larger than base classes.

It is also common in programming circle to refer base class as superclass and derived class as subclass. Well, in terms of the hierarchy this sounds good. But in terms of the class size, it seems contradicting. It is in fact derived classes that are larger in size than base classes that should be termed as superclasses and base classes as subclasses. Well, we will justify the naming in a short while. The previous discussions on Data Encapsulation or Abstraction, among the three access specifiers 'protected' specifier ensures that such members are visible to derived classes/their objects. In C++ there are three kinds of inheritance associated with the access specifiers, namely private, protected and public inheritance.

In this section, we shall discuss these forms of inheritance. Irrespective of the inheritance type, the final accessibility is as well dependent on the base class member access specifier. With public inheritance, a public base class member becomes public in the derived class also.

Thus, it can be accessed by member functions, friend functions and non-member functions. A protected base class member becomes protected in derived class. Such a member can be accessed directly by member functions and friend functions. A private base class member becomes private in the derived class. The various possibilities are clearly explained in Table 7.1.

Table 7.1: Inheritance types/access specifier

Base class member access specifier	*Type of inheritance*		
	Public	*Protected*	*Private*
Public	Public in derived class (can be asscessed directly by non-static member functions, non-member functions and friend functions)	Protected in derived class (can be accessed directed by all non-static member functions and friend functions)	Private in derived class (can be accessed directly by all non-static member functions and friend functions)
Protected	Protected in derived class (can be accessed directly by all non-static member functions and friend functions)	Protected in derived class (can be accessed directed by all non-static member functions and friend functions)	Private in derived class (can be accessed directly by all non-static member functions and friend functions)
Private	Hidden in derived class (can be accessed by non-static member functions and friend functions but only via the public or protected member functions of the base class)	Hidden in derived class (can be accessed by non-static member functions and friend functions but only via the public or protected member functions of the base class)	Hidden in derived class (can be accessed by non-static member functions and friend functions but only via the public or protected member functions of the base class)

The syntax for specifying the three modes of inheritance discussed above is as follows:

```
Class DerivedClassName :  mode BaseClassName
{
derived class attributes (data members)
derived class properties (member functions)
}
```

For example if `A` is the base class and `B` is a class derived from `A`, and it is in the protected mode that `B` is derived from `A`, the syntax is as follows:

```
Class B : protected A
{
..
..
..
}
```

Inheritance can always be interpreted as (in the form of) a tree like hierarchical structure. The base class exists in a hierarchical relationship with derived classes. A good example for inheritance hierarchy could be: Let us assume the student entity in a university. A typical university or college has students of various types such as Research Student, Postgraduate Student and Graduate Student. Thus, these three types of students could be possible derivations from a base class called Student.

Private members of a base class cannot be accessed by derived classes. And also a derived class does not inherit friend functions of the base class. A base class's public members are accessible throughout the program. However, a base class's private members are accessible only by member functions and friends of the base class. A class's protected members can be accessed by members or friends of base and derived classes. Derived class members can refer by their names. An object of a derived class can be treated as an object of a base class object, but the vice versa treatment causes syntax error. This shall be explained in detail in the next section. Now let us look into a programming example for inheritance. We shall consider the student hierarchy in the following example, code for which is shown in Figure 7.1.

```
"student.h"
class Student
{
protected:
char * fname;
char * lname;
friend ostream & operator << (ostream &, const Student &);
public: Student (char * ="", char * ="");
~Student( );
};
"StudentFunctions.cpp"
#include <iostream.h>
#include "Student.h"
Student :: Student (char * first, char * last)
{
fname = new char [strlen(first) + 1];
strcpy(fname,first);
lname = new char [strlen(last) + 1];
strcpy(lname,last);
}
ostream & operator << (ostream & out, const Student & stud)
{
out << stud.fname <<"\t"<<stud.lname<<"\n";
return out;
}
"Graduate.h"
#include<iostream.h> #include "Student.h" class Graduate : public Student
{
friend ostream & operator << (ostream &, const Graduate &);
public :Graduate (char * = " ", char * = " ", char * =" ");
```

Figure 7.1: Continued

```
Protected :
Char * degree;
};
Graduate :: Graduate (char * fna, char *lna, char * dn) : Student(fna,lna)
{
degree = char new [strlen(dn)+1];
strcpy(degree,dn);
}
ostream & operator << (ostream & out, const Graduate & grad)
{
out << "First Name is \t" << static cast <Student>(grad);
out << "Degree is \t" << grad.degree;
return 0;
}
"main.cpp"
#include<iostream.h>
#include "Graduate.h"
#include "Student.h"
int main ()
{
Student * sptr = 0,S("Karthik","Krishnan");
Graduate *gptr =0,G("Anusha","Srinivasan","B.E");
cout << "Student S's details "<< S <<"\n";
cout << "G's details" <<G<<"\n";
//Trying to view a Graduate student as a Student
sptr = &G; cout << * gptr;
sptr = &S;
gptr = static cast<Graduate *>(sptr);
cout << * gptr;
return 0;
}
```

Figure 7.1: C++ code to illustrate the feature of inheritance.

7.3.1 Pointer Casting

In the above example, we have employed casting operators to convert base class pointer to derived class type and vice versa. These are referred to as casting operations. The earlier discussion on inheritance referred to base class as superclass and derived class as subclass. Even before we get into the above programming example, we shall dwelve on the casting operations. Well, all derived class objects can be treated as base class objects and it is syntactically and semantically valid to do so. It is on this basis that base class is referred to as superclass and derived class as subclass [based on the count of base class objects being more as a result of derived class objects being treated as be class objects].

The process of assigning derived class pointers to base class pointers is referred to as up-casting a pointer and the process of converting a base class pointer to a derived class pointer is referred to as down-casting a pointer. Down casting must be performed very carefully to ensure that the pointer points to an object of the correct type. Now getting back to the programming example, we consider the student hierarchy. We have a base class called Student whose members fname, lname are protected to enable access to derived classes of it. The constructor of the base class allocates memory dynamically for the fname, lname members

and then constructs the object suitably. The destructor deallocates the memory associated with fname, lname members. We have also overloaded the stream insertion (<<) operator to redirect a student's object at one shot.

The class Graduate which represents Graduate students has an explicit member, namely `char * degree` to identify the degree programme. The derived class constructor accepts arguments including the members that would be derived from the base class, namely fname and lname. Thus, the derived class constructor accepts all three arguments, two for the construction of derived class members inherited from the base class and one for its explicit data member. Within the definition of the derived class constructor, we invoke the base class constructor by its name and passing the respective arguments using the member initializer syntax : operator. There is a valid reasoning behind invoking the base class constructor within the derived class constructor.

Whenever derived class objects are created, even before the derived class's explicit data members are constructed, the base class members that are inherited in the derived class should be constructed first. It is with this reasoning that whenever derived class objects are created, automatically the base class constructor is called to construct the "from the base class" derived members of the derived class and then proceed with the construction of explicit derived class members. If the derived class constructor does not invoke the base class constructor using the member initializer syntax, the compiler automatically invokes the base class's default constructor whenever objects of the derived class are created.

A default base constructor not available in such cases causes syntax error. The overloaded stream insertion operator (<<) of the derived class makes use of the overloaded stream insertion operator of the base class student to redirect the "from the base class" derived members of the derived class. For the duration when we wish to treat the derived class objects as a base class object, which is syntactically and semantically valid, we use the `static _cast` [casting at compile time] operator to convert the derived class object (G) to the base class (Student) type.

After this temporary conversion the overloaded << operator of the class Student is invoked to redirect the "from the base class" derived members of the derived class. The derived class's explicit member (degree) is redirected separately (as an individual statement).

Note that the base class members are made protected to enable the derived class to refer them directly by their names. If the base class members were private instead of protected as we have here, then they cannot be accessed directly by the derived class, but can be referred to in the derived classes using the base class public member functions.

7.4 Class Graduate

Public Student instructs the compiler that the class Graduate is a publicly derived class from Student base class. The public and protected members of the base class are inherited as public and protected into the derived class Graduate. The drive routine creates `sptr` (student pointer) or pointer to student class and then initializes the address of student object S to this base class pointer. A similar derived class pointer and object pair is created. The overloaded output operator is applied with each instance of base and derived classes `S` and `G`. Well, this exercise is typically to understand the pointer manipulations of upcasting and downcasting efficiently. First, the base class pointer `(sptr)` is assigned the address of the derived class

object G, then redirection statement on line displays the student (base class) portion of the Graduate (derived Class).

Well, this pointer manipulation is allowed since a derived class object can always be treated as a base class object. The base class pointer (sptr) sees only the base class members of the derived class object. This casting or conversion is syntactically and semantically valid. Then the base class pointer is converted again to a derived class pointer type and this conversion is assigned to a derived class pointer.

```
[gptr = static_cast < Graduate *> (sptr)].
```

This is again valid since the derived class pointer is pointing to the address of a derived class object.

The validity is checked by dereferencing the contents pointed to by the derived class pointer

```
[cout << *gptr;]
```

Well, this casting of a base class pointer type to a derived class pointer type must be performed carefully, else may cause serious run-time errors. Such an operation is allowed syntactically but then semantically it is meaningless. In the above case, the derived class pointer is actually pointing to a derived class objects. Hence there were no run-time or serious problems to be considered.

However, if the objects type do not match, that is a derived class pointer assigned the address of a base class object without any casting, then referring to explicit derived class members [degree] via such a pointer might cause serious logical errors, since there is no such member existing at that point of program execution. This is illustrated in the above example by the cout statement at line. It displays only the base class portion of the derived class object and the explicit derived class member is defaulted to null.

It is up to the programmer to be careful while performing such manipulations, since calling or referring to data members or member functions non-existent can even crash the program in execution. In certain cases whatever values happens to be in the memory (location of the derived class pointer to the base class object). In all, such non-existent member references can prove very dangerous.

The reasoning behind not allowing private members of a base class directly accessible by derived classes is to prevent the data encapsulation being violated since such access might allow further derivations [classes derived from derived class] to access private data and would amount to data encapsulation being violated throughout the hierarchy. Well, you have a way out; go in for protected specifiers in such cases where the base class members should be directly accessible to derived class. The above example can be extended with another possible class Research Student, a derivation from Graduate class with its possible exclusive data member being fellowship amount. This is left as an exercise for the reader.

7.5 Order of Constructor/Destructor Calls with Inheritance

In the earlier discussions on encapsulation, we have seen one example to understand the order in which constructors and destructors get invoked. Well, in this section, we would like to extend those concepts and understand the constructor/destructor calls in an inheritance (hierarchy)

involved program scenario. All those concepts explained in the earlier section are valid here as well. The only new issue here would be to resolve the ordering of base class constructors and destructor calls when a derived class object is created or instantiated.

When a derived class object is created, since a derived class inherits its base class members first and then has its own or explicit derived class members, as we already mentioned it is first base class constructor(s) that should be called and so it is and then the derived class constructor [to construct the explicit derived class members]. As is the principle that most recently constructed objects are destructed first, the destructor for the derived class object and then the base class object is invoked.

The base class constructors and assignment operators are not inherited by derived class, but still derived classes can call constructors and overloaded assignment operators of base classes. If the derived class constructor is not defined by the programmer, when a derived class object is created, the compiler which invokes the default constructor of derived class in turn invokes the base class's default constructor even before the construction of the derived class's explicit data members.

In the case of multiple inheritance when a class derives its properties from more than one base class, the order in which the base class constructors are called depends on the inheritance order specification and not on the order in which the base class constructors are specified in the member initializer syntax of the derived class constructor. That is, let us assume a class `C` is publicly derived from both class `A` and `B`, whose syntax is as follows:

```
Class C: public class A, public class B
{
;
;
}
```

Let us assume that the derived class constructor definition [with no emphasis on the number of arguments] looks like `C() : B(),A()`.

When an object of class C is created it is the ordering specified in the inheritance specification `[A, B]` that determines the order of the base constructors call and not the ordering of the member initializer syntax. That is to say that first the constructor of the base class `A` is called and then the constructor for the base class `B` is called, and not in the order of `B`'s constructor followed by `A`'s as is specified in the member initializer syntax. Whatever be the case, destructors always get invoked in the reverse order of their construction.

7.6 Inheritance and Composition Issues

Another issue to be resolved with inheritance-based OO programming is that a base class or derived class can have objects of other class as its members or a base or derived class can compose objects of other classes as its members. This feature, which is referred to as composition is differentiated from inheritance by the fact that composition signifies a 'has a' relationship, while inheritance signifies a 'is a' relationship. This naming is based on the fact that in composition an enclosing class composes or has objects of other class as its members. In inheritance, a derived class object is an instance of or can be treated as a base class object.

When composition and inheritance are exploited together in an OO program, that is both base class and derived class contain objects of other classes, when an object of a derived class is created, the constructor for the base class member objects (base class constructor), and then the constructors of derived class member objects (derived class constructors) are invoked.

Member objects are constructed again based on the order in which they are declared in the enclosing or composing class and not on the ordering in which the member initializer syntax of the composing class's constructor specification. Destructors are called in the reverse order of constructor calls. Getting back to our earlier programming example on student hierarchy, let us assume we have the following driver function to drive the classes Student and Graduate, code for which is shown in Figure 7.2.

```
#include "Student.h"
#include "Graduate.h"
int main ( )
{
Student S1("Sai","Siddarth");
Graduate G1("Kripa","Shankar","B.S");
{
Student S2("Shri","Vastsan");
}
Graduate G2("Ganesh","Ramani","B.E");
return 0;
}
```

Figure 7.2: Order of constructor/destructor calls with inheritance.

The above statements shown in Table 7.2 clearly indicate the order in which base class and derived class constructors/destructors are invoked when a derived class object is created.

Table 7.2: Output for code in Figure 7.2

The order of constructor and destructor calls in the above example is as follows:
Base Class Constructor for S1 [Sai Siddarth]
Base Class constructor for G1 [Kripa Shankar]
Derived Class constructor for G1 [BS]
Base Class Constructor for S2 [Shri Vatsan]
Derived Class Destructor for S2 [Shri Vatsan]
Base Class Constructor for G2 [Ganesh Ramani]
Derived Class Constructor for G2 [B.E]
Derived Class Destructor for G2[B.E]
Base Class Destructor for G2[Ganesh Ramani]
Derived Class Destructor for G1[BS]
Base Class Destructor for G1[Kripa Shankar]
Base Class Destructor for S1[Sai Siddarth]

7.7 Base Class Members Redefinition

A derived class inherits both data members as well as member functions from the base class. The derived class can have its own explicit data members or member functions. In certain occasions, the derived class may decide to redefine member functions that have been inherited by the derived class. That is member functions in the derived and base class have the same name or same function identifier.

The derived class may or may not redefine member functions that are inherited from the base class. This concept of redefining member functions of the base class in the derived class is referred to as overriding of base class member functions in the derived class. Function overriding is a key concept in inheritance. We shall explore function overriding in the following sections. The first simple question to be resolved is how does function overriding differ from function overloading.

Well, in function overloading, the functions have to necessarily differ in their signatures. It is based on the signature that an overloaded function call gets resolved. In the case of function overriding, there is no such syntactic requirement that the base class member function and its derived class version (overridden function) differ in their signatures. In fact, most probably the signatures of the base class and the overriding derived class version have the same signature. The correct function gets called based on the object type [base class or derived class].

When an overridden function is mentioned by name with a derived class object, it is the derived class version that is invoked automatically. If one requires the base class version of the member function within the derived class, the scope resolution operator preceded by the base class name and followed by the member function name is used to access the base class version from the derived class. Normally in function overriding, the derived class version of an overridden function makes use of the base class version to achieve the tasks related to the "from the base class" derived members of the derived class.

It is only the additional tasks required for the explicit derived class members that a new code is specified. This prevents code redundancy and promotes easier code maintenance. Our earlier Student–Graduate example will now be modified by excluding the overloaded stream insertion (`<<`) operator. Instead, we will define a function called `display()` in the base class student [that displays the first name and last name] and the `display()` function is overridden in the derived class.

The overridden version invokes the base class version of `display()` to redirect the `fname`, `lname` [base class members] of the derived class object [Graduate]. The overridden version in the derived class redirects the explicit data member, namely degree. The modified member function definitions are as shown in Figure 7.3.

The class structures are quite similar to the earlier discussion on student hierarchy. Base class member function display redirects the first name and last name of the Student object. The derived class Graduate overrides the display function inherited from the Student class. The overridden version invokes the base class version to redirect the first name and last name members inherited from the Student class and then redirects the explicit derived class member (degree) using a separate `cout` statement.

The driver function test drives the base and derived classes and invokes the display with respect to each instance to verify that the correct version gets invoked. The reasoning behind allowing derived class objects to be treated as base class objects is that derived class has members corresponding to or required for the base class members. However, base class objects

```
"Student.h"
Class Student
{
protected :
char * fname;
char * lname;
public : Student (char * ="", char * ="");
~Student ( );
void display ( ) const;
};
"Student.cpp"
#include <iostream.h>
#include "Student.h"
Student :: Student (char * fn, char * ln)
{
fname = new char [strlen (fn) + 1];
strcpy(fname,fn);
lname = new char [strlen (ln) + 1];
strcpy(lname,ln);
}
Student :: ~Student ( )
{
delete [] fname;
delete [] lname;
}
void Student :: display( ) const
{
cout <<"First name is \t" << Q fname <<"\n";
cout <<"Last name is \t" << lname <<"\n";
}
"Graduate.h"
#include "Student.h"
Class Graduate : public Student
{
protected :
char * degree;
Graduate ( char * = , char * = , char * = );
~Graduate ( );
void display ( ) const;
};
Graduate :: Graduate (char * fnam, char *lnam, char * deg)
: Student(fnam,lnam)
{
degree = new char [strlen(deg) + 1];
strcpy(degree,deg);
}
Graduate :: ~Graduate ( )
{
delete [] degree;
```

Figure 7.3: Continued

```
}
void Graduate :: display ( ) const
{
Student :: display ( );
cout << "Degree is \t" << degree <<"\n";
}
"main.cpp"
#include<iostream.h>
#include"Graduate.h"
#include"Student.h"
int main ( )
{
Student S1("Siddarth", "Jonnathan");
Graduate G1("Sudharshan", "Rangarajan", "M.Tech");
S1.display ( ); // Base class Version display
G1.display ( ); // Derived class version of display
return 0;
}
```

Output:

First Name is Siddarth
Last Name is Jonnathan
First Name is Sudarshan
Last Name is Rangarajan
Degree is M.Tech

Figure 7.3: C++ code for student class based on inheritance.

cannot be treated as derived class objects since the explicit derived class members would then be undefined. In simple terms, base class object = derived class object is allowed (syntactically and semantically valid), but derived class object = base class object is not allowed (syntactically valid, but not semantically!).

The reason is that in the second assignment the additional explicit derived class members will not have valid values. The earlier assignment statement is allowed since the derived class object has the required base class members for the assignment operation to proceed. In other words, in the first case the derived class object on the RHS of the assignment statement should be compatible or of similar type to the LHS base class object. Thus, this case wherein a derived class object is to be converted or treated as a base class object is allowed while the vice versa operation is dangerous, though it may be syntactically allowed, but definitely dangerous from a semantic perspective. Such manipulations require intermediate casting operations as well. In the following section, we shall elaborate on multiple inheritance and focus on virtual base class, to resolve one of the key issues involved in multiple inheritance.

7.8 Multiple Inheritance

It is a hierarchy set-up in which a class derives its properties from more than one base class. Well, the issues pertaining to the constructor/destructor calls has already been explained. Let us consider the following inheritance hierarchy where the class `C` is derived from class `A` and class `B`. Let us go through the definitions of these classes which are given in Figure 7.4.

```
"Example.cpp"
#include <iostream.h>
Class A
{
public :
A (int a)
{
val = a;
}
~A( )
{
cout <<"Destructor \n";
}
protected :
int val;
void print ( ) const
{
cout <<"Value is"<<val<<" \n";
};
Class B
{
public:
B (char c)
{
id = c;
}
~B( )
{
cout << "Destructor \n";
}
protected:
char id;
void print ( ) const
{
cout <<"Value is"<< id <<"\n";
};
class C : public A, public B
{
public :
C(int=0, char='',double=0.0);
~C ( );
friend ostream & operator << (ostream &, const C &);
private : double iden;
};
C :: C( int i, char j, double k) : A(i), B(j)
{
iden = k;
}
C :: ~ C( )
{
```

Figure 7.4: Continued

```
cout <<"Destructor of C \n";
}
ostream & operator <<( ostream & out, const C & obj)
{
out << "Integer value is "<< obj.val <<"\n";
out << "Character value is"<< obj.id <<"\n";
out << "Double value is "<< obj.iden <<"\n";
return out;
}
int main ( )
{
A o1(50), * aptr = 0;
B o2('B'), * bptr = 0;
C o3(10,'C',1.1);
o1.print();
o2.print();
cout << o3;
aptr = &o3;
aptr→print();
bptr = &o3;
bptr→print();
return 0;
}
```

Output:
Value is 50
Value is B
Integer Value is 10
Character Value is C
Double Value is 1.1
Value is 10
Value is C

Figure 7.4: C++ code for multiple inheritance.

In the case of multiple inheritance a derived class object can be treated as an object of either of the base classes as has been done above. Class `A` is identified by an integer identifier, `B` is identified by a character identifier, and `C` which inherits from `A` and `B` has its explicit member of a double identifier. Corresponding constructors and destructors are provided to suitably initialize the data members. Note that the constructor of derived class `C` accepts arguments inclusive of the base class(es) `A` and `B`. This being an example to understand the concept of multiple inheritance not much member functions are required here.

The `print` function of base classes `A` and `B` redirect their respective data members. Note that we have overloaded the output operator for the derived class `C`. However, an alternative solution would be to override the `print` function of class `C` within which the respective base class versions be invoked to redirect the inherited base class members. We will defer such programming at least till we resolve the need for virtual base classes, the topic of interest in our next section. The `driver` function creates normal instances of each of the three classes `(o1,o2,o3)` and two pointer instances, one `(aptr)` to base class `A` and the other `(bptr)` to base class `B`.

To understand the concept that a derived class object (inherited from more than one base class) can be treated as an instance of either of the classes, the driver function assigns the address of o3 to aptr trying to treat derived class object o3 as an instance of base class A (line). Then bptr is assigned the address of o3 to treat the derived class object as an instance of base class B. In each of the above cases, the print function is invoked to display the object identifier. aptr→print() displays the value 10 (integer identifier of 03), a property inherited from base class A. bptr→print() displays the value C (character identifier of o3), a property inherited from base class B.

As should be clear member function invokes of print() on each of the two normal instances of the base class o1 and o2 displays the state of o1 and o2. State of o3 is displayed making use of the overloaded redirection operator. This is our normal syntax of invoking a member function on a object is in no way different in interpretation with respect to inheritance. Well, with the above discussion, readers should be clear of the fact a derived class object that inherits properties from more than one base class, can be treated as an instance of either of the base classes. Having explored multiple inheritance in terms of feature inheritance, we are now in a position to focus on the issue of virtual base class, one of the key aspects of multiple inheritance scenario.

7.9 Virtual Base Class

The virtual base class declaration is allowed to overcome one of the key duplication issues involved with multiple inheritance. We shall go through the following example to understand the duplication issue involved with multiple inheritance. Let us assume we have the following class structure shown in Table 7.3 available. Well, in this section, we are more interested in understanding the need for virtual base class, a concept and hence we will not be particular about the syntax behind each of these declarations, which should be clear by now and can be very easily incorporated.

Table 7.3: Virtual inheritance situation

Class A → Data Member I
Class B (derived from A) →Explicit member J
Class C (derived from A)→Explicit member K
Class D (derived from B,C)→Explicit member L

The information within the parentheses clearly identifies the parent class in each of the derivations and the information on the right of the arrow identifies the respective derived class's explicit member. Well, what is the issue at all? That should be the intriguing question disturbing all readers who have understand the previous discussions on inheritance and multiple inheritance. In fact, the issue should have become clear by now! But, still let us for a while interpret the structure of each of the above classes in terms of data members.

Class A is existing as a normal class (concrete class) and has one data member named I. Class B, a derivation from Class A has two data members, one its explicit data member (J) and the other being the inherited I from class A. Class C, also a derivation from class A has two data members, one its explicit data member (K) and the other being the inherited I from class A. The issue with multiple inheritance is as follows. We have a class D, that is a derivation from two classes, namely B and C. D in all will have five data members, one

its explicit data member (L), two inherited from class B(I and J), the last two inherited from class C (I and K). Now the issue should be clear. D in terms of the final class structure has two data members named I and references to I via an instance of D would be irresolvable or the compiler is in a fix to differentiate which I it is. Yes, virtual base class is the solution for such inheritance scenarios. To prevent such ambiguous references, the virtual keyword is used to declare what are called as virtual base classes. In this case, classes which will serve as base classes for further (possibly) multiple inheritance situation are derived in the virtual mode. The virtual keyword precedes the public, protected or private mode of derivation specification. That is, in the above programming example, the derivations should be as shown in Figure 7.5 (of course to avoid the duplication effect). The dots indicate the data member(s)/member function(s) that can possibly belong to each of the above classes. Thus, virtual base class overcomes the ambiguous (duplicated members) scenario by maintaining only one copy of members in the derived class that inherits properties from more than one base class, each of which is a derivation from another common base class.

```
Class B: virtual public A
{
...
}
Class C :virtual public A
{
...
}
Class D: public B, public A
{
...
}
```

Figure 7.5: Abstract base class declaration.

Review Questions

1. Discuss the benefits offered by inheritance feature of C++.
2. Define the terms base and derived classes.
3. Appreciate the access specifiers of members in classes derived from a base class.
4. Differentiate single from multiple inheritance. What is the issue involved with multiple inheritance?
5. Appreciate the usage of casting operations with inheritance programming situations.
6. Develop a university package in C++ with options to display details of Student, Faculty and Staff. A few of the other operations to be supported include display of CGPA for Students, display of salary details for Faculty and Staff. Use inheritance feature of C++.
7. Differentiate inheritance from class composition.
8. Differentiate an abstract base from a virtual base class.
9. Appreciate the benefits of inheritance in terms of long-term software development.
10. Discuss constructors and destructors with respect to derived classes and highlight the order of construction and destruction with a programming example.