```
class Goods: public HeavyMotor
{
   public:
      void GetData()
      {
        HeavyMotor::GetData();
      }
      void DisplayData()
      {
        HeavyMotor::DisplayData();
      }
};
void main()
{
   GearMotor vehi1;
   Passenger vehi2;
   // read vehicle data
   cout << "Enter Data for Gear Motor Vehicle ..." << endl;
   vehi1.GetData();
   cout << "Enter Data for Passenger Motor Vehicle ..." << endl;
   vehi2.GetData();
   // display vehicle data
   cout << "Data of Gear Motor Vehicle ..." << endl;
   vehi1.DisplayData();
   cout << "Data of Passenger Motor Vehicle ..." << endl;
   vehi2.DisplayData();
}
```

Run

```
Enter Data for Gear Motor Vehicle ...
Name of the Vehicle ? Maruti-Car
Wheels ? 4
Speed Limit ? 4
No. of Gears ? 5
Enter Data for Passenger Motor Vehicle ...
Name of the Vehicle ? KSRTC-BUS
Wheels ? 4
Load Carrying Capacity ? 60
Permit Type ? National
Maximum Seats ? 45
Maximum Standing ? 60
Data of Gear Motor Vehicle ...
Name of the Vehicle : Maruti-Car
Wheels : 4
Speed Limit : 4
Gears: 5
Data of Passenger Motor Vehicle ...
Name of the Vehicle : KSRTC-BUS
Wheels : 4
Load Carrying Capacity : 60
```

```
Permit: National
Maximum Seats: 45
Maximum Standing: 15
```

14.14 Multipath Inheritance and Virtual Base Classes

The form of inheritance which derives a new class by multiple inheritance of base classes, which are derived earlier from the same base class, is known as *multipath inheritance*. It involves more than one form of inheritance namely multilevel, multiple, and hierarchical as shown in Figure 14.18. The `child` class is derived from the base classes `parent1` and `parent2` (multiple inheritance), which themselves have a common base class `grandparent` (hierarchical inheritance). The child inherits the properties of the `grandparent` class (multilevel inheritance) via two separate paths as shown by the broken line. The classes `parent1` and `parent2` are referred to as direct base classes, whereas `grandparent` is referred to as the indirect base class.

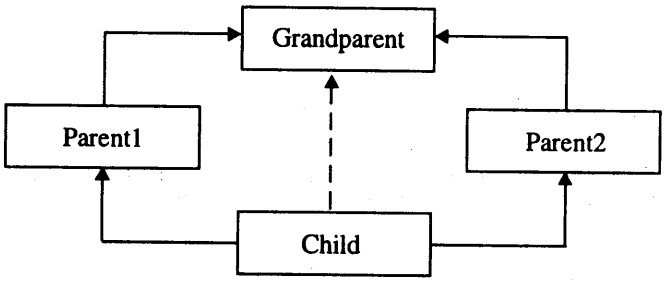

Figure 14.18: Multipath inheritance

Multipath inheritance can pose some problems in compilation. The public and protected members of `grandparent` are inherited into the `child` class twice, first, via `parent1` class and then via `parent2` class. Therefore, the `child` class would have duplicate sets of members of the `grandparent` which leads to ambiguity during compilation and it should be avoided.

C++ supports another important concept called *virtual base classes* to handle ambiguity caused due to the multipath inheritance. It is achieved by making the common base class as a virtual base class while declaring the direct or intermediate classes as shown below:

```
class A
{
   public:
      void func()
      {
         // body of function
      }
};
class B1: public virtual A
{
   // body of class B1
};
```

```
class B2: public virtual A
{
   // body of class B2
};

class D: public B1, public B2
{
   // body of class D
};
```

Consider the statement

```
objd.func();
```

where `objd` is the object of class `D` and invokes the `func()` defined in the class `A`. If the keyword `virtual` does not exist in the declaration of classes `B1` and `B2`, a call to `func()` leads to the following compilation error:

```
Error: Member is ambiguous: 'A::func' and 'A::func'
```

C++ takes necessary care to see that only one copy of the class is inherited, when a class is inherited as `virtual` irrespective of the number of paths that exist between the virtual base class and the derived class. The keywords `virtual` and `public` or `protected` may be used in any order.

Consider the processing of the result of student depicted in the Figure 14.19. In this case, the `result` class is derived from the classes `InternalExam` and `ExternalExam`, which are derived classes of the common class `student`. The program `int_ext.cpp` implements the concepts of virtual classes.

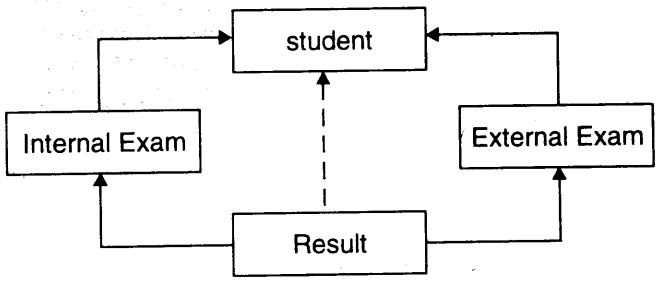

Figure 14.19: Virtual base classes

```
// int_ext.cpp: Student result based on internal and external marks
//          Multipath Inheritance with virtual classes
#include <iostream.h>
const int MAX_LEN = 25;    // maximum length of name
class student
{
   protected:
      int RollNo;        // student roll number in a class
      char branch[20];   // branch or subject student is studying
   public:
      void ReadStudentData()
      {
         cout << "Roll Number ? ";
```

```
        cin >> RollNo;
        cout << "Branch Studying ? ";
        cin >> branch;
     }
     void DisplayStudentData()
     {
        cout << "Roll Number: " << RollNo << endl;
        cout << "Branch: " << branch << endl;
     }
};
class InternalExam: virtual public student
{
   protected:
      int Sub1Marks;
      int Sub2Marks;
   public:
      void ReadData()
      {
         cout << "Marks Scored in Subject 1 < Max:100> ? ";
         cin >> Sub1Marks;
         cout << "Marks Scored in Subject 2 < Max:100> ? ";
         cin >> Sub2Marks;
      }
      void DisplayData()
      {
         cout<<"Internal Marks Scored in Subject 1: "<<Sub1Marks << endl;
         cout<<"Internal Marks Scored in Subject 2: "<<Sub2Marks << endl;
         cout<<"Internal Total Marks Scored: "<<InternalTotalMarks()<<endl;
      }
      int InternalTotalMarks()
      {
         return Sub1Marks + Sub2Marks;
      }
};
class ExternalExam: virtual public student
{
   protected:
      int Sub1Marks;
      int Sub2Marks;
   public:
      void ReadData()
      {
         cout << "Marks Scored in Subject 1 < Max:100> ? ";
         cin >> Sub1Marks;
         cout << "Marks Scored in Subject 2 < Max:100> ? ";
         cin >> Sub2Marks;
      }
      void DisplayData()
      {
         cout<<"External Marks Scored in Subject 1: "<<Sub1Marks << endl;
```

```
        cout<<"External Marks Scored in Subject 2: "<<Sub2Marks << endl;
        cout<<"External Total Marks Scored: "<<ExternalTotalMarks()<<endl;
      }
     int ExternalTotalMarks()
      {
        return Sub1Marks + Sub2Marks;
      }
};
class result: public InternalExam, public ExternalExam
{
  private:
     int total;
  public:
     int TotalMarks()
      {
        return InternalTotalMarks() + ExternalTotalMarks();
      }
};
void main()
{
  result student1;
  cout << "Enter data for Student1 ..." << endl;
  student1.ReadStudentData();  // virtual resolves ambiguity
  cout << "Enter Internal Marks ..." << endl;
  student1.InternalExam::ReadData();
  cout << "Enter External Marks ..." << endl;
  student1.ExternalExam::ReadData();
  cout << "Student details ..." << endl;
  student1.DisplayStudentData();  // virtual resolves ambiguity
  student1.InternalExam::DisplayData();
  student1.ExternalExam::DisplayData();
  cout << "Total Marks = " << student1.TotalMarks();
}
```

Run

```
Enter data for Student1 ...
Roll Number ? 9
Branch Studying ? Computer-Technology
Enter Internal Marks ...
Marks Scored in Subject 1 < Max:100> ? 80
Marks Scored in Subject 2 < Max:100> ? 85
Enter External Marks ...
Marks Scored in Subject 1 < Max:100> ? 89
Marks Scored in Subject 2 < Max:100> ? 90
Student details ...
Roll Number: 9
Branch: Computer-Technology
Internal Marks Scored in Subject 1: 80
Internal Marks Scored in Subject 2: 85
Internal Total Marks Scored: 165
External Marks Scored in Subject 1: 89
```

```
External Marks Scored in Subject 2: 90
External Total Marks Scored: 179
Total Marks = 344
```

Another typical example of virtual classes having their derived classes invoking their base class's constructors is through the initialization section. The program `vir.cpp` has classes A, B, C, and D representing multi-path inheritance.

```
// vir.cpp: virtual classes with data members initialization
#include <iostream.h>
class A
{
   protected:
      int x;
   public:
      A()
      {  x = -1; }
      A( int i )
      {  x = i; }
      int geta()
      {  return x; }
};
class B: virtual public A
{
   protected:
      int y;
   public:
      B( int i, int k) : A(i)
      {  y = k;  };
      int getb()
      {  return y;  }
      void show()
      {
         cout << x << " " << geta() << " " << getb();
      }
};
class C:virtual public A
{
   protected:
      int z;
   public:
      C(int i,int k) : A( i )
      {  z = k;  };
      int getc()
      {  return z;  }
      void show()
      {
         cout << x << " " << geta() << " " << getc();
      }
};
```

```
class D: public B,public C
{
   public:
      // invoke A() and then B(i,j) and C(i,j)
      D( int i, int j ) : B(i,j), C(i,j) {}
      void show()
      {
        cout << x << " " << geta() << " " << getb();
        cout << " " << getc() << " " << getc();
      }
};
void main()
{
   D d1( 3, 5 );
   cout << endl << "Object d1 contents: ";
   d1.show();
   B b1( 7, 9 );
   cout << endl << "Object b1 contents: ";
   b1.show();
   C c1( 11, 13 );
   cout << endl << "Object c1 contents: ";
   c1 show();
}
```

Run

```
Object d1 contents: -1 -1 5 5 5
Object b1 contents: 7 7 9
Object c1 contents: 11 11 13
```

In `main()`, the statement

```
B b1( 7, 9 );
```

invokes the constructor of the class `B`

```
B( int i, int k) : A(i)
```

which calls the single argument constructor of the class `A` and then it executes. Similarly, the statement

```
C c1( 11, 13 );
```

invokes first the single argument constructor of the class `B` and then it executes. The first statement in the `main()` function

```
D d1( 3, 5 );
```

is supposed to invoke the constructor

```
D( int i, int j ) : B(i,j), C(i,j) {}
```

which in turn invokes the constructors of the `B` and `C` classes and is expected to produce the results:

```
Object d1 contents: 3 3 5 5 5
```

assuming that the constructor `A(i)` is invoked, but this has not happened.

According to the inheritance principle, first, the super base class must be instantiated and then followed by the lower level class, finally the one whose object has to be created (No grand child without grand father). When an object of the class `D` has to be created, first the constructor of the class `A` is to be invoked. The default no-argument constructor `A()` is invoked instead of the one-argument

constructor. Even if it invokes the one-argument constructor, either through the

```
B(int i,int k) : A( i )
```

or through the

```
C(int i,int k) : A( i )
```

it leads to confusion; there are two calls to the constructor which is illegal. It is similar to arguing *father is created before the grandfather*, which is neither true in real life nor in C++. Therefore, C++ selects, the no-argument constructor to avoid all these issues. If the constructor of `D` specification is changed to,

```
D( int i, int j ) : A(i), B(i,j), C(i,j) {}
```

It produces the result as expected; the one-argument constructor of the super class `A` is explicitly specified in the initialization section..

14.15 Hybrid Inheritance

There are many situations where more than one form of inheritance is used in designing the class. For example, consider the case of processing the student results as discussed in the program, `exam.cpp` in multilevel inheritance. Suppose the weightage for a sport is also taken into consideration for finalizing the results. The weightage for sports is stored in a separate class called `sports`. The new inheritance relationships between various classes would be as shown in Figure 14.20, which indicate both multilevel and multiple inheritance.

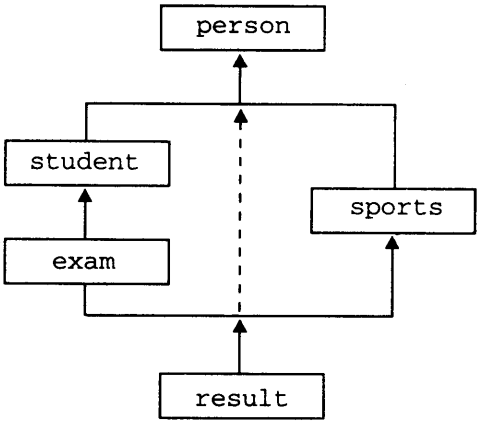

Figure 14.20: Hybrid (multilevel, multipath) inheritance

The inheritance relation shown in Figure 14.20 is modeled in the program `sports.cpp`. It consists of five classes namely `person`, `student`, `exam`, `sports`, and `result`. The class `exam` is derived by multilevel inheritance. The derivation of the class `result` from the classes `exam` and `sports` exhibits multipath inheritance. Therefore, it has properties of the class `person` indirectly through two paths: from the `exam` class and `sport` class.

```
// sports.cpp: Models student grading based on exam score and sports
#include <iostream.h>
const int MAX_LEN = 25;     // maximum length of name
```

```
class person
{
   private:                  // Note: cannot be referred by derived class
      char name[ MAX_LEN ];      // person name
      char sex;                  // person sex, M - male, F - female
      int age;                   // person age
   public:
      void ReadPerson()
      {
         cout << "Name ? ";
         cin >> name;
         cout << "Sex ? ";
         cin >> sex;
         cout << "Age ? ";
         cin >> age;
      }
      void DisplayPerson()
      {
         cout << "Name: " << name << endl;
         cout << "Sex : " << sex << endl;
         cout << "Age : " << age << endl;
      }
};
class sports: public virtual person    // note: virtual class
{
   private:
      char name[MAX_LEN];  // name of game
      int score;           // score awarded for result declaration
   protected:
      void ReadData()
      {
         cout << "Game Played ? ";
         cin >> name;
         cout << "Game Score ? ";
         cin >> score;
      }
      void DisplayData()
      {
         cout << "Sports Played: " << name << endl;
         cout << "Game Score: " << score << endl;
      }
      int SportsScore()
      {
         return score;
      }
};
class student : public virtual person     // note: virtual class
{
   private:
      int RollNo;       // student roll number in a class
```

```
      char branch[20];  // branch or subject student is studying
   public:
      void ReadData()
      {
         cout << "Roll Number ? ";
         cin >> RollNo;
         cout << "Branch Studying ? ";
         cin >> branch;
      }
      void DisplayData()
      {
         cout << "Roll Number: " << RollNo << endl;
         cout << "Branch: " << branch << endl;
      }
};
class exam: public student
{
   protected:
      int Sub1Marks;
      int Sub2Marks;
   public:
      void ReadData()
      {
         cout << "Marks Scored in Subject 1 < Max:100> ? ";
         cin >> Sub1Marks;
         cout << "Marks Scored in Subject 2 < Max:100> ? ";
         cin >> Sub2Marks;
      }
      void DisplayData()
      {
        student::DisplayData(); // uses DisplayData() of student class
         cout << "Marks Scored in Subject 1: " << Sub1Marks << endl;
         cout << "Marks Scored in Subject 2: " << Sub2Marks << endl;
         cout << "Total Marks Scored: " << TotalMarks() << endl;
      }
      int TotalMarks()
      {
         return Sub1Marks + Sub2Marks;
      }
};
class result: public exam, public sports
{
   private:
      int total;
   public:
      void ReadData()
      {
         ReadPerson();        // access person class member
         student::ReadData();
```

```
        exam::ReadData();     // uses ReadData() of exam class
        sports::ReadData();
    }
    void DisplayData()
    {
        DisplayPerson();          // access person class member
        student::DisplayData();
        exam::DisplayData();
        sports::DisplayData();
       cout<<"Overall Performance,(exam+sports): "<<Percentage()<<" %";
    }
    int Percentage()
    {
        return (exam::TotalMarks() + SportsScore())/3;
    }
};
void main()
{
   result student;
   cout << "Enter data for Student ..." << endl;
   student.ReadData();
   cout << "Student details ..." << endl;
   student.DisplayData();
}
```

Run

```
Enter data for Student ...
Name ? Rajkumar
Sex ? M
Age ? 24
Roll Number ? 9
Branch Studying ? Computer-Technology
Marks Scored in Subject 1 < Max:100> ? 92
Marks Scored in Subject 2 < Max:100> ? 88
Sports Played ? Cricket
Game Score ? 85
Student details ...
Name: Rajkumar
Sex : M
Age : 24
Roll Number: 9
Branch: Computer-Technology
Marks Scored in Subject 1: 92
Marks Scored in Subject 2: 88
Total Marks Scored: 180
Sports Played: Cricket
Game Score: 85
Overall Performance, (exam+sports): 88 %
```

14.16 Object Composition–Delegation

Most of us understand concepts such as objects, interfaces, classes, and inheritance. The challenge lies in applying them to build flexible and reusable software. The two most common techniques for reusing functionality in object-oriented systems are *class inheritance* and *object composition*. As explained, inheritance is a mechanism of building a new class by deriving certain properties from other classes. In inheritance, if the class D is derived from the class B, it is said that *D is a kind of B*; the class D has all the properties of B in addition to the features of its own.

A commonly recurring situation is one where objects are used as data members in a class. The use of objects in a class as data members is referred to as *object composition*. Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or composing objects to support more powerful functionality. This new approach takes a view that an object can be a collection of many other objects and the relationship is called a *has-a* relationship or *containership*. In OOP, the *has-a* relationship occurs when an object of one class is contained in another class as a data member. In other words, a class can contain objects of other classes as its members (see Figure 14.21).

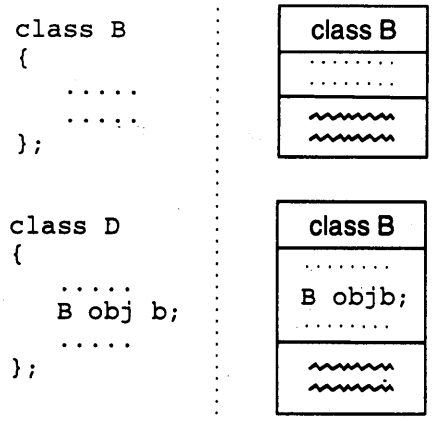

Figure 14.21: Object composition

In the case of inheritance (*kind of* relationship), the constructors of base class are first invoked before the constructor of the derived class. Whereas, in the case of *has-a* relationship, the constructor of the class D is invoked first and then the object of B is created. The concept of creating the member objects first using respective member constructors and then the other ordinary members can also be accomplished in *has-a* relationship by using an *initialization-list* in the constructor of the nested class. Consider the following class declarations

```
class B
{
   // body of a class
};
class D
{
    B ObjectB;      // b is a object of class B
    public:
      D( arg-list ) : ObjectB( arg-list1 );
};
```

where `arg-list` is the list of arguments to be supplied during the creation of objects of the class `D`. These parameters are used in initializing the members of class `D`. The `arg-list1` is used to initialize the members of the class `B`. In this case, first, the constructor of the class `B` is executed and then the constructor of the class `D`. The program `nesting.cpp` demonstrates the method of invoking a constructor of another object in a class.

```
// nesting.cpp: Nested class constructor invocation
#include <iostream.h>
class B
{
   public:
      int num;
      B()              // no argument constructor
      { num = 0; }
      B( int a )
      {
         cout << "Constructor B( int a ) is invoked" << endl;
         num = a;
      }
};
class D
{
   int data1;
   B objb;          // object of another class
   public:
      D( int a ): objb( a )   // invokes the constructor of 'objb'
      {
         data1 = a;
      }
      void output()
      {
        cout << "Data in Object of Class D = " << data1 << endl;
        cout<<"Data in Member object of class B in class D = "<<objb.num;
      }
};
void main()
{
   D objd( 10 );
   objd.output();
}
```

Run

```
Constructor B( int a ) is invoked
Data in Object of Class D = 10
Data in Member object of class B in class D = 10
```

Delegation

Delegation is a way of making object composition as powerful as inheritance for reuse. In delegation, two objects are involved in handling a request: a receiving object delegates operations to its `delegate`. This is analogous to subclasses deferring requests to parent classes. In certain situations, inheritance and containership relationships can serve the same purpose. It is illustrated by the follow-

ing code:

```
class publication    // base class1
{
   // body of the publication class
};
class sales          // base class2
{
   // body of the sales class
};
```

The book class can be derived from the publication and sales classes using inheritance relationship as follows:

```
class book: public publication, public sales
{
   // body of the book class
};
```

The above functionality can also be achieved by composing objects of the classes publication and sales into the class book as follows:

```
class book
{
   ....
   publication pub; // composition of object of the class publication
   sales market;    // composition of object of the class sales
   ....
};
```

The book class contains instances of the classes publication and sales. The book class delegates its publication and sales issues to instances of the publication and sales classes (see Figure 14.22). Delegation shows that inheritance can be replaced with object composition as a mechanism for code reuse. The program publish2.cpp models the delegation shown in Figure 14.21.

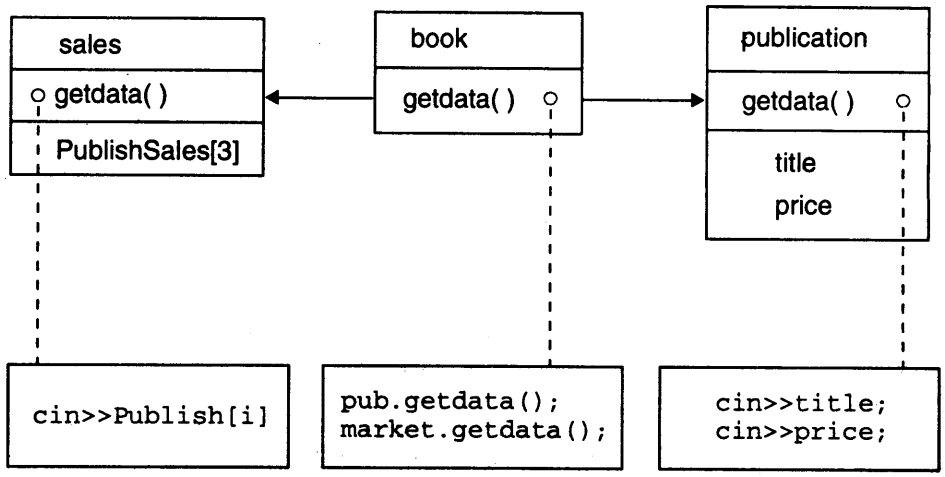

Figure 14.22: Delegation in publication class

The classes publication and sales have the same declaration as in inheritance relation but they are used in a different way by the book class. Although containership is an alternative to inheritance and offers the functionality of inheritance, it does not provide flexibility of ownership. Inheritance relationship is simpler to implement and offers a clearer conceptual framework.

```
// publish2.cpp: Publication, Sales details, Objects inside a class
#include <iostream.h>
class publication    // base class, appears as abstract class
{
   private:
      char title[40];    // name of the publication work
      float price;       // price of a publication
   public:
      void getdata()
      {
         cout << "\tEnter Title: ";
         cin >> title;
         cout << "\tEnter Price: ";
         cin >> price;
      }
      void display()
      {
         cout << "\tTitle = " << title << endl;
         cout << "\tPrice = " << price << endl;
      }
};
class sales    // base class
{
   private:
      float PublishSales[3];  //sales of publication for the last 3 months
   public:
      void getdata();
      void display();
};
void sales::getdata()
{
   int i;
   for( i = 0; i < 3; i++ )
   {
      cout << "\tEnter Sales of " << i+1 << " Month: ";
      cin >> PublishSales[i];
   }
}
void sales::display()
{
   int i;
   int TotalSales = 0;
   for( i = 0; i < 3; i++ )
   {
      cout<<"\tSales of " << i+1 <<" Month = " << PublishSales[i] << endl;
```

```
      TotalSales += PublishSales[i];
    }
    cout << "\tTotal Sales = " << TotalSales << endl;
}
class book
{
   private:
      int pages;  // number of pages in a book
   public:
      publication pub;
      sales market;
      void getdata() // overloaded function
      {
         pub.getdata();
         cout << "\tEnter Number of Pages: ";
         cin >> pages;
         market.getdata();
      }
      void display()
      {
         pub.display();
         cout << "\tNumber of Pages = " << pages << endl;
         market.display();
      }
};
void main()
{
   book book1;
   cout << "Enter Book Publication Data ..." << endl;
   book1.getdata();
   cout << "Book Publication Data ..." << endl;
   book1.display();
}
```

Run

```
Enter Book Publication Data ...
   Enter Title: Microprocessor-x86-Programming
   Enter Price: 180
   Enter Number of Pages: 750
   Enter Sales of 1 Month: 1000
   Enter Sales of 2 Month: 500
   Enter Sales of 3 Month: 800
Book Publication Data ...
   Title = Microprocessor-x86-Programming
   Price = 180
   Number of Pages = 705
   Sales of 1 Month = 1000
   Sales of 2 Month = 500
   Sales of 3 Month = 800
```

14.17 When to Use Inheritance ?

The following principles have to be followed to promote the use of inheritance in programming, which leads to code reuse, ease of code maintenance and extension:

- The most common use of inheritance and subclassing is for specialization, which is the most obvious and direct use of the *is-a* rule. If two abstract concepts A and B are being considered, and the sentence *A is a B* makes sense, then it is probably correct in making A as a subclass of B. Examples, *Car is a Vehicle, Triangle is a Shape,* etc.
- Another frequent use of inheritance is to guarantee that classes maintain a certain common interface; that is, they implement the same methods. The parent class can be a combination of implemented operations and operations that are to be implemented in the child classes. Often, there is no interface change between the supertype and subtype - the child implements the behavior described instead of its parent class. This feature has much significance with pure virtual function and will be discussed in the chapter *Virtual Functions*.
- Using generalization technique, a subclass extends the behavior of the superclass to create a more general kind of object. This is often applicable when one is building on a base of existing classes that should not, or cannot be modified.
- While subclassing for generalization modifies or expands on the existing functionality of a class, subclassing for extension adds totally new abilities. Subclassing for extension can be distinguished from subclassing for generalization in derivation. Generalization must override at least one method from the parent, and the functionality is tied to that of the parent whereas extension simply adds new methods to those of the parent, and functionality is less strongly tied to the existing parent methods.
- In subclassing for limitation, the behavior of the subclass is more restricted than the behavior of the superclass. Like subclassing for generalization, subclassing for limitation occurs most frequently when a programmer is building on a base of existing classes that should not or cannot be modified.
- Subclassing for variance is useful when two or more classes have similar implementations, but there does not seem to be any hierarchical relationship between the concepts represented by the classes. Often, however, a better alternative is to factor out the common code into an abstract class, and derive the classes from these common ancestors.
- Subclassing by combination occurs when a subclass represents a combined feature from two or more parent classes.

14.18 Benefits of Inheritance

There are many important benefits that can be derived from the proper use of inheritance. They are code reuse, ease of code maintenance and extension, and reduction in the time to market. The following situations explain benefits of inheritance:

- When inherited from another class, the code that provides a behavior required in the derived class need not have to be rewritten. Benefits of reusable code include increased reliability and a decreased maintenance cost because of sharing of the code by all its users.
- Code sharing can occur at several levels. For example, at a higher level, many users or projects can use the same class. These are referred to as software components. At the lower level, code can be shared by two or more classes within a project.
- When multiple classes inherit from the same superclass, it guarantees that the behavior they inherit will be the same in all cases.

- Inheritance permits the construction of reusable software components. Already, several such libraries are commercially available and many more are expected to be available in the near future.
- When a software system can be constructed largely out of reusable components, development time can be concentrated on understanding the portion of a new system. Thus, software systems can be generated more quickly and easily by rapid prototyping.

14.19 Cost of Inheritance

Inspite of many benefits of inheritance, it incurs compiler overhead. In inheritance relationship, there are certain members in the base class that are not at all used, however data space is allocated to them. This necessitates the need for specialized inheritance, which is complex to develop. The following are some of the perceived costs of inheritance:

- Inherited methods, which must be prepared to deal with arbitrary subclasses, are often slower than specialized codes.
- The use of any software library frequently imposes a size penalty over the use of systems specially constructed for a specific project. Although this expense may in some cases be substantial, it is also true that as memory cost decreases, the size of programs is becoming less important.
- Message passing by its very nature is a more costly operation than the invocation of simple procedures. The increased cost is however marginal and is often much lower in statically bound languages like C++. Therefore, the increased cost must be weighed against the benefits of the object oriented techniques.
- Although object oriented programming is often touted as a solution to the problem of software complexity, overuse or improper use of inheritance can simply replace one form of complexity with another.

Review Questions

14.1 What is inheritance ? Explain the need of inheritance with suitable examples.

14.2 What are the differences between the access specifiers private and protected ?

14.3 What are base and derived classes ? Create a base class called Stack and derived class called MyStack. Write a program to use these classes for manipulating objects.

14.4 Explain the syntax for declaring the derived class. Draw access privilege diagram for members of a base and derived class.

14.5 What are the differences between a C++ `struct` and C++ `class` in terms of encapsulation and inheritance ?

14.6 What are the different forms of inheritance supported by C++ ? Explain them with an example.

14.7 What is a class hierarchy ? Explain how inheritance helps in building class hierarchies.

14.8 Can base class, access members of a derived class ? Give reasons.

14.9 What is visibility mode? What are the different inheritance visibility modes supported by C++

14.10 What are the differences between inheriting a class with public and private visibility mode ?

14.11 Declare two classes named `Window` and `Door`. Derive a new class called `House` from those two classes. The `Window` and `Door` bases classes must have attributes which reflects happy home. All classes must have interface functions such as overloaded stream operator functions for reading and displaying attributes. Write an interactive program to model the above relation.

14.12 State with reasons whether the following statements are TRUE or FALSE:
(a) Both base and derived classes need not have constructors.
(b) Only base class cannot have constructors.
(c) Only derived class can have constructors.
(d) No-argument constructor of the base class is invoked when a derived class is instantiated.
(e) When a derived class is instantiated only the derived class constructors are invoked.
(f) Derived class members cannot access private members of a base class.
(g) When a derived class is instantiated, memory is allocated to all data members of both the base and derived classes.
(h) If a base class does not has no-argument constructor and has parameterized constructors, it must be explicitly invoked from a derived class.
(i) Constructors are invoked starting from the top base class to derived class order.
(j) Destructors are invoked starting from the top base class to derived class order.
(k) Destructors are invoked in the reverse order of constructors.
(l) Base class constructors can be explicitly invoked from the derived class.

14.13 Explain how base class member functions can be invoked in a derived class if the derived class also has a member function with the same name.

14.14 What are virtual classes ? Explain the need for virtual classes while building class hierarchy.

14.15 What are abstract classes ? Explain the role of abstract class while building a class hierarchy.

14.16 Consider an example of declaring the examination result. Design three classes: `Student`, `Exam`, and `Result`. The `Student` class has data members such as those representing roll number, name, etc. Create the class `Exam` by inheriting the `Student` class. The `Exam` class adds data members representing the marks scored in six subjects. Derive the `Result` from the `Exam` class and it has its own data members such as `total_marks`. Write an interactive program to model this relationship. What type of inheritance this model belongs to ?

14.17 A new scheme for evaluation of students performance is formulated that gives also weightage for sports. Extend the inheritance relation discussed in the above program (14.16) such that the `Result` class also inherits properties of `Sports` class. Note that the `Sports` class is a derived class of the `Student` class. Write a program to model this relationship such that members of the Students class are not inherited twice. What type of inheritance this model belongs to ?

14.18 What is containership or delegation ? How does it differ from inheritance ?

14.19 It is required to find out the cost of constructing a house. Create a base class called `House`. There are two classes called `Door` and `Window` available. The `House` class has members which provide information related to the area of construction, door, windows details, etc. It delegates responsibility of computing the cost of doors and windows construction to `Door` and `Window` classes respectively. In C++, this can be achieved by having instances of the classes `Door` and `Window` in the `House` class. Write an interactive program to model the above relationship.

14.20 Write an interactive program to create a graphic class hierarchy. Create an abstract base class called `Figure` and derive two classes `Close` and `Open` from that. Declare two more classes called `Polygon` and `Ellipse` using the `Close` class. Create derived classes `Line` and `Polyline` from the `Open` class. Define three objects (triangle, rectangle, and pentagon) of the class `Polygon`. All classes must have appropriate member functions including constructors and destructors.

14.21 Discuss cost and benefits of inheritance emphasizing ease of design, code reusability, overhead, etc.

15

Virtual Functions

15.1 Introduction

Polymorphism in biology means the ability of an organism to assume a variety of forms. In C++, it indicates the form of a member function that can be changed at runtime. Such member functions are called *virtual functions* and the corresponding class is called *polymorphic class*. The objects of the polymorphic class, addressed by pointers, change at runtime and respond differently for the same message. Such a mechanism requires postponement of binding of a function call to the member function (declared as `virtual`) until runtime.

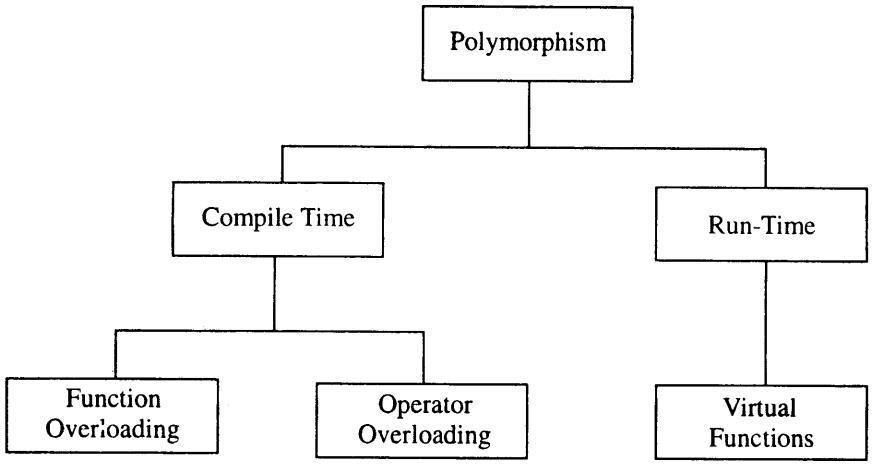

Figure 15.1: Types of polymorphism in C++

It has been observed that, *function overloading* and *operator overloading* features of C++ have allowed to realize polymorphism. Yet, there is another mechanism to implement polymorphism in C++; through *dynamic binding*. Figure 15.1, illustrates the taxonomy of polymorphism in C++. *Function overloading* is realized by invoking a suitable function whose *signature* matches with the arguments specified in the function call statement. *Operator overloading* is realized by allowing operators to operate on the user defined data-types with the same interface as that of the standard data types. In both cases, the compiler is aware of the complete information regarding the type and number of operands. Hence, it is possible for the compiler to select a suitable function at the compile-time.

15.2 Need for Virtual Functions

When objects of different classes in a class hierarchy, react to the same message in their own unique ways, they are said to exhibit polymorphic behavior. The program `parent1.cpp` illustrates the need of such polymorphic behavior. It has the base class `Father` and the derived class `Son` and has a member function (called `show`) with the same name and prototype. Note that, in C++ a pointer to the base class can be used to point to its derived class objects.

```
// parent1.cpp: invoking derived class member through base class pointer
#include <iostream.h>
#include <string.h>
class Father
{
   char name[20];     // father name
   public:
      Father( char *fname )
      {
         strcpy( name, fname );  // fname contains Father's name
      }
      void show()     // show() in base class
      {
         cout << "Father name: " << name << endl;
      }
};
class Son: public Father
{
   char name[20];     // son name
   public:
    // two-argument constructor; invokes one-argument constructor of Father
      Son( char *sname, char *fname ): Father( fname )
      {
         strcpy( name, sname );  // sname contains son's name
      }
      void show()     // show() in derived class
      {
         cout << "Son name: " << name << endl;
      }
};
void main()
{
   Father *fp; // pointer to the Father class's objects;
   Father f1( "Eshwarappa" );
   fp = &f1;   // fp points to Father class object
   fp->show(); // display father show() function
   Son s1( "Rajkumar", "Eshwarappa" );
   fp = &s1;   // valid assignment
   fp->show(); // guess what is the output ? Father or Son!
```

Run

```
Father name: Eshwarappa
Father name: Eshwarappa
```

In `main()`, the statement

```
Father *fp; // pointer to the Father class's objects;
```

defines a pointer variable `fp`. The statement

```
fp = &f1;   // fp points to Father class object
```

assigns the address of the object `f1` of the class `Father` to `fp`. After this, when statement such as

```
fp->show(); // display father show() function
```

is executed, the member function defined in the class `Father` is invoked. In C++ base class pointer is fully type-compatible with its derived class pointers and hence, the statement such as

```
fp = &s1;   // valid assignment
```

is valid. It assigns the address of the object `s1` of the class `Son` to `fp`. After this, when statement

```
fp->show(); // guess what is the output ? Father or Son!
```

is executed, (it is interesting to note that) it still invokes the member function `show()` defined in the class `Father`!

There must be a provision to use the member function `show()` to display the state of objects of both the `Father` and `Son` classes using the same interface. This decision cannot be taken by the compiler, since the prototype is identical in both the cases.

In C++, a function call can be bound to the actual function either at compile time or at runtime. Resolving a *function call* at compile time is known as *compile-time* or *early* or *static binding* whereas, resolving a function call at runtime is known as *runtime* or *late* or *dynamic binding*. Runtime polymorphism allows to postpone the decision of selecting the suitable member functions until runtime. In C++, this is achieved by using *virtual functions*.

Virtual functions allow programmers to declare functions in a base class, which can be defined in each derived class. A pointer to an object of a base class can also point to the objects of its derived classes. In this case, a member function to be invoked depends on the the class's object to which the pointer is pointing. When a call to any object is made using the same interface (irrespective of object to which pointer variable is pointing), the function relevant to that object will be selected at run time. The program `parent2.cpp` illustrates the effect of virtual functions on an overloaded function in a class hierarchy.

```
// parent1.cpp: base class pointer and virtual function
#include <iostream.h>
#include <string.h>
class Father
{
   char name[20];     // father name
   public:
      Father( char *fname )
      {
         strcpy( name, fname );  // fname contains Father's name
      }
```

```
    virtual void show()    // show() in base class declared as virtual
    {
       cout << "Father name: " << name << endl;
    }
};
class Son: public Father
{
   char name[20];    // son name
   public:
    // two-argument constructor; invokes one-argument constructor of Father
     Son( char *sname, char *fname ): Father( fname )
     {
        strcpy( name, sname );  // sname contains son's name
     }
     void show()    // show() in derived class
     {
        cout << "Son name: " << name << endl;
     }
};
void main()
{
   Father *fp; // pointer to the Father class's objects.
   Father f1( "Eshwarappa" );
   fp = &f1;   // fp points to Father class object
   fp->show(); // display father show() function
   Son s1( "Rajkumar", "Eshwarappa" );
   fp = &s1;   // valid assignment
   fp->show(); // guess what is the output ? Father or Son!
}
```

Run

```
Father name: Eshwarappa
Son name: Rajkumar
```

It is interesting to note that the output generated by the above program is as expected. (What is interesting about the above program when compared to the earlier `parent1.cpp` ?) The only difference is, the member function `show()` defined in the class `Father` has the following declarator:

```
virtual void show()    // show() in base class declared as virtual
```

It indicates that the member function `show()` is virtual and binding of a call to this function must be postponed until runtime. Hence, the last statement in `main()`,

```
fp->show(); // guess what is the output ? Father or Son!
```

invokes the member function defined in the class `Son`!; during the execution of this statement, the system notices that, `show()` is a *virtual function* in base class and hence, it decides to invoke the member function defined in the derived class (instead of the base class) if the base class pointer is pointing to the derived class object.

The knowledge of pointers to base class and derived classes is essential to understand and to explore full potential of virtual functions. Hence, a detailed discussion on how the above program is able to work as expected and syntax of virtual functions is postponed to later section.

15.3 Pointer to Derived Class Objects

The concept of derived classes specifies hierarchical relationship between various objects and expresses the commonality between them. The properties common to different classes are placed at the top of the hierarchy, which becomes the base class, and all other classes are derived from this base class. A derived class is often said to inherit properties of the base class, and so, this relationship is known as the *inheritance relationship*.

Pointers can be used with the objects of base classes or derived classes. Pointer to objects of a base class are *type-compatible* with pointers to objects of a derived class, thus allowing a single pointer variable to be used as pointer to objects of a base class and its derived classes. For instance, in the above declaration having the classes `Parent` and `Child`, a pointer declared as a pointer to `Parent` objects can also be used as a pointer to `Child` objects. C++ makes polymorphism possible through a rule that one should memorize: *a base class pointer may address an object of its own class or an object of any class derived from the base class.* (See Figure 15.2.)

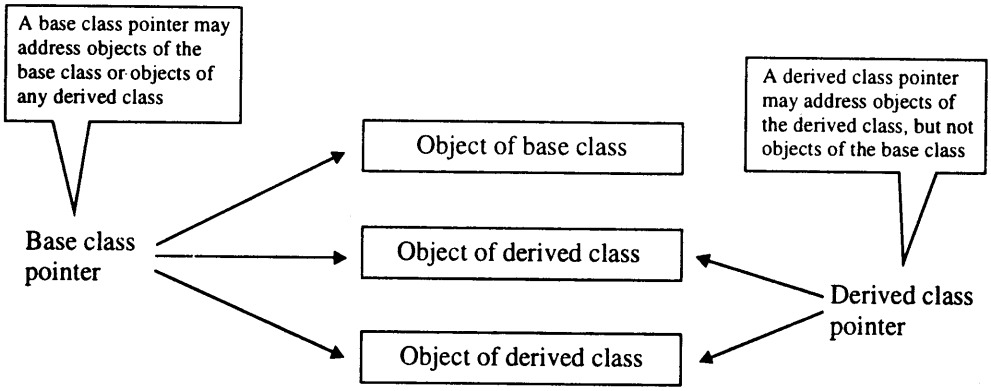

Figure 15.2: A base class pointer may address a derived class object

Consider the following definitions to illustrate type compatibility of pointers.

```
Father *basep;  // pointer to Parent class
Father f;       // base class object
Son s;          // derived class object
```

The statement

```
basep = &f;
```

assigns the address of the object `f` of the class `Father` to the pointer variable `basep`. The statement

```
basep = &s;
```

assigns the address of the object `s` of the class `Son` to the pointer variable `basep`. Such an assignment is perfectly valid in C++, since the pointer to an object of a base class is fully *type-compatible* with pointer to objects of its derived classes.

The use of a pointer to the objects of a base class with the objects of its derived class raises a new problem. It does not allow access even to public members of a derived class. That is, it allows access only to those members inherited from the base class but not to the members which are defined in the derived class. Even in case, any member of the `Son` class has the same name as one of the members of

the `Father` class, reference to it using the base class pointer `basep` will always access the base class member and not the derived class member. The program `family1.cpp` illustrates the use of the base pointer with the derived objects.

```
// family1.cpp: pointer to base class and derived class objects
#include <iostream.h>
class Father
{
   protected:
      int f_age;
   public:
      Father( int n )
      {
         f_age = n;
      }
      int GetAge(void)
      {
         return f_age;
      }
};
// Son inherits all the properties of father
class Son : public Father
{
   protected:
      int s_age;
   public:
      Son( int n, int m ):Father(n)
      {
         s_age = m;
      }
      int GetAge(void)
      {
         return s_age;
      }
      void son_func()
      {
         cout << "son's own function";
      }
};
void main()
{
   Father *basep;
   basep = new Father(45);    // pointer to father
   cout << "basep points to base object..." << endl;
   cout << "Father's Age: ";
   cout << basep->GetAge() << endl;  // calls Father::GetAge
   delete basep;
   // accessing derived object
   basep = new Son(45, 20);   // pointer to son
   cout << "basep points to derived object..." << endl;
```

```
    cout << "Son's Age: ";
    cout << basep->GetAge() << endl;  // calls Father::GetAge()
    cout << "By typecasting, ((Son*) basep)..." << endl;
    cout << "Son's Age: ";
    cout << ((Son*) basep)->GetAge() << endl;    // calls Son::GetAge()
    delete basep;
    // accessing with derived object pointer
    Son son1( 45, 20 );
    Son *derivedp = &son1;
    cout << "accessing through derived class pointer..." << endl;
    cout << "Son's Age: ";
    cout << derivedp->GetAge();
}
```

Run

```
basep points to base object...
Father's Age: 45
basep points to derived object...
Son's Age: 45
By typecasting, ((Son*) basep)...
Son's Age: 20
accessing through derived class pointer...
Son's Age: 20
```

The expression, `basep->GetAge()` in the statement,

```
cout << basep->GetAge() << endl;
```

invokes `GetAge()` defined in the `Father` class; `basep` holds the address of the `Father` class object. Even when the pointer `basep` is made to point to the derived object, it invokes the function defined in the `Father` class. However, the typecasted expression

```
((Son*) basep)->GetAge()
```

invokes the `GetAge()` defined in the derived class `Son` since the pointer is explicitly typecasted. In the above program, the use of the statement

```
basep->son_func();   // error: not member of Father
```

generates a compilation error since, `son_func()` is not a member of the `Father` class or it is not within the scope of the `Father` class. However, when typecasted as

```
((Son *)basep)->son_func(); // OK
```

it will not generate any errors and will invoke the function defined in the `Son` class. (See Figure 15.3.)

The rule, *a base class pointer may address an object of its own class or an object of any class derived from the base class* is a one-way route. In other words, a pointer to a derived class object cannot address an object of the base class. If a pointer to a derived class is allowed to address the base class object, the compiler will expect members of the derived class to be in the base class also (which is not possible). (See Figure 15.4.) A pointer to the derived class can be used as a pointer to other classes which are derived from it. In general , *a pointer to a class at a particular level can be used as a pointer to objects of classes which are below that level in the class hierarchy.* Any attempt to override this rule is treated as an error.

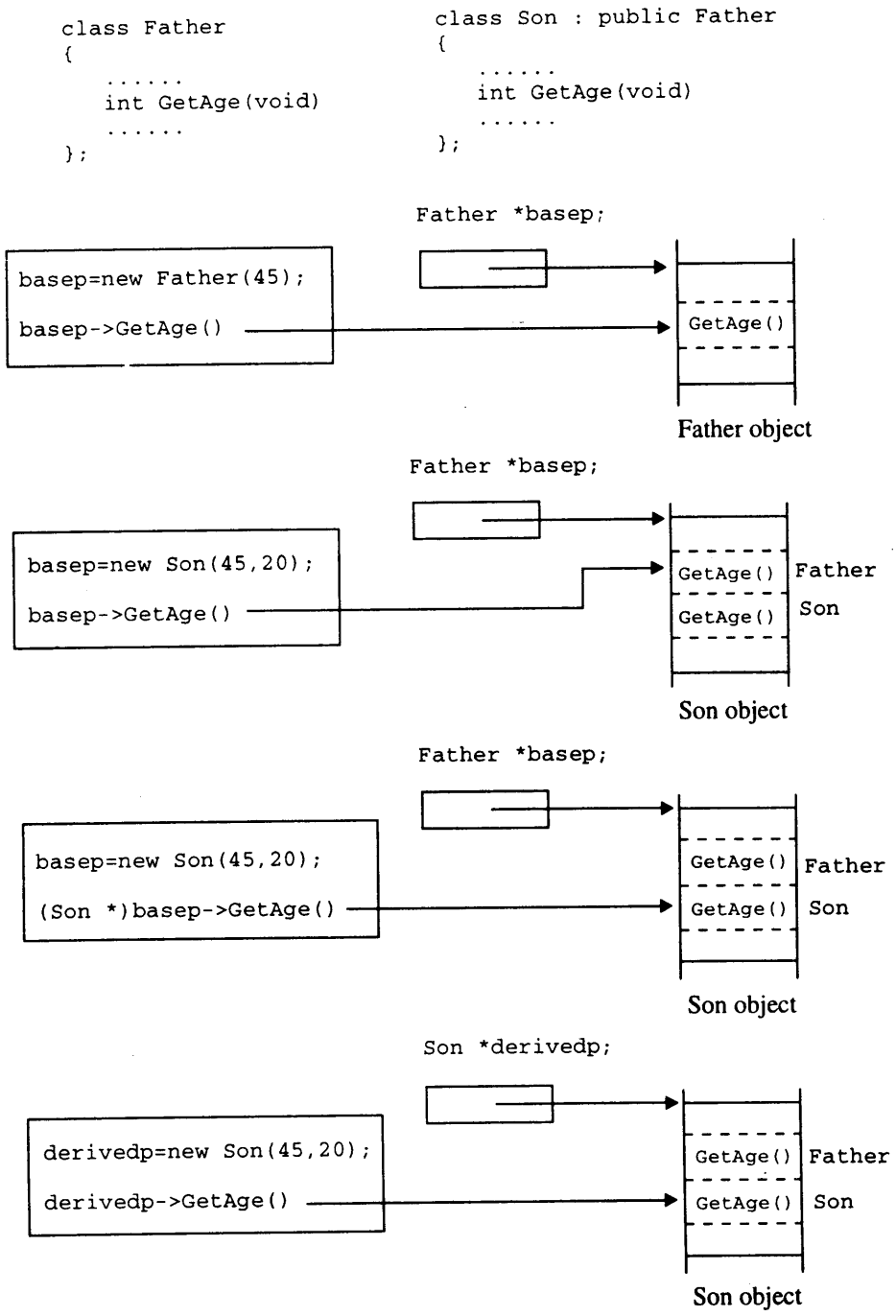

Figure 15.3: A base pointer accessing derived objects

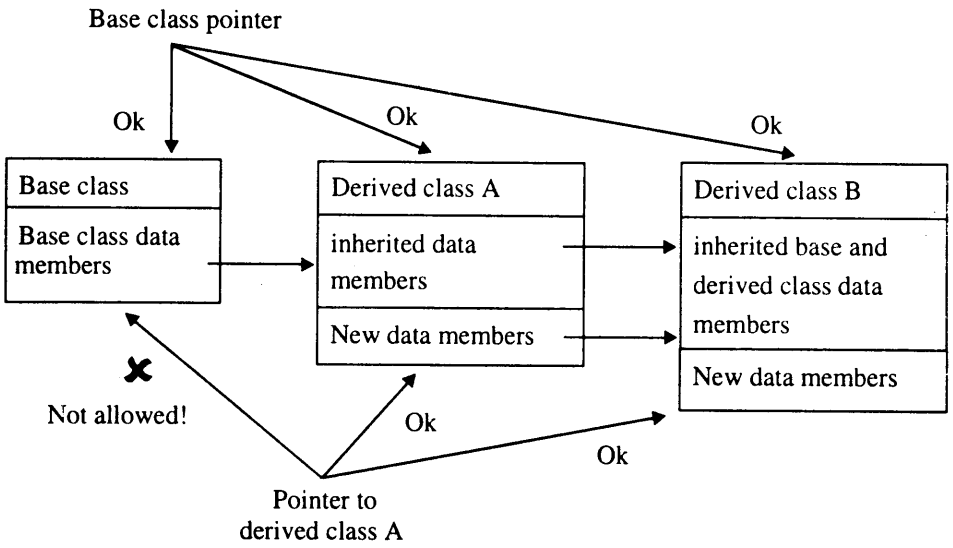

Figure 15.4: A base class pointer can address data members inherited by a derived class

15.4 Definition of Virtual Functions

C++ provides a solution to invoke the exact version of the member function, which has to be decided at runtime using virtual functions. They are the means by which functions of the base class can be *overridden* by the functions of the derived class. The keyword `virtual` provides a mechanism for defining the virtual functions. When declaring the base class member functions, the keyword `virtual` is used with those functions, which are to be bound dynamically. The syntax of defining a virtual function in a class is shown in Figure 15.5.

```
class MyClass
{
   public:          keyword
   ......
   ......
   virtual ReturnType FunctionName(arguments)
   {
      ......
      ......
   }
   ......
};
```

Figure 15.5: Syntax of virtual function

Virtual functions *should be defined in the public section of a class* to realize its full potential benefits. When such a declaration is made, it allows to decide which function to be used at runtime, based on the type of object, pointed to by the base pointer, rather than the type of the pointer. The program `family2.cpp` illustrates the use of base pointer to point to different objects for executing different implementations of the virtual functions.

```
// family2.cpp: Binding pointer to base class's object to base or derived
// objects at runtime and invoking respective members if they are virtual
#include <iostream.h>
class Father
{
   protected:
      int f_age;
   public:
      Father( int n )
      {
         f_age = n;
      }
      virtual int GetAge(void)
      {
         return f_age;
      }
};
// Son inherits all the properties of father
class Son : public Father
{
   protected:
      int s_age;
   public:
      Son( int n, int m ):Father(n)
      {
         s_age = m;
      }
      int GetAge(void)
      {
         return s_age;
      }
};
void main()
{
   Father *basep;
   // points to Father's object
   basep = new Father(45);    // pointer to father
   cout << "Father's Age: ";
   cout << basep->GetAge() << endl;  // calls Father::GetAge
   delete basep;
   // points to Son's object
   basep = new Son(45, 20);   // pointer to son
   cout << "Son's Age: ";
   cout << basep->GetAge() << endl;  // calls Son::GetAge()
   delete basep;
}
```

Run

```
Father's Age: 45
```

```
Son's Age: 20
```

The statement in the base class `Father`

```
virtual int GetAge(void)
```

indicates that an invocation of the `GetAge()` through the pointer to an object must be resolved at runtime based on *to which class's object the pointer is pointing*. A pointer to objects of the base class can be made to point to its derived class objects. Figure 15.6 illustrates the use of virtual functions in invoking functions at runtime.

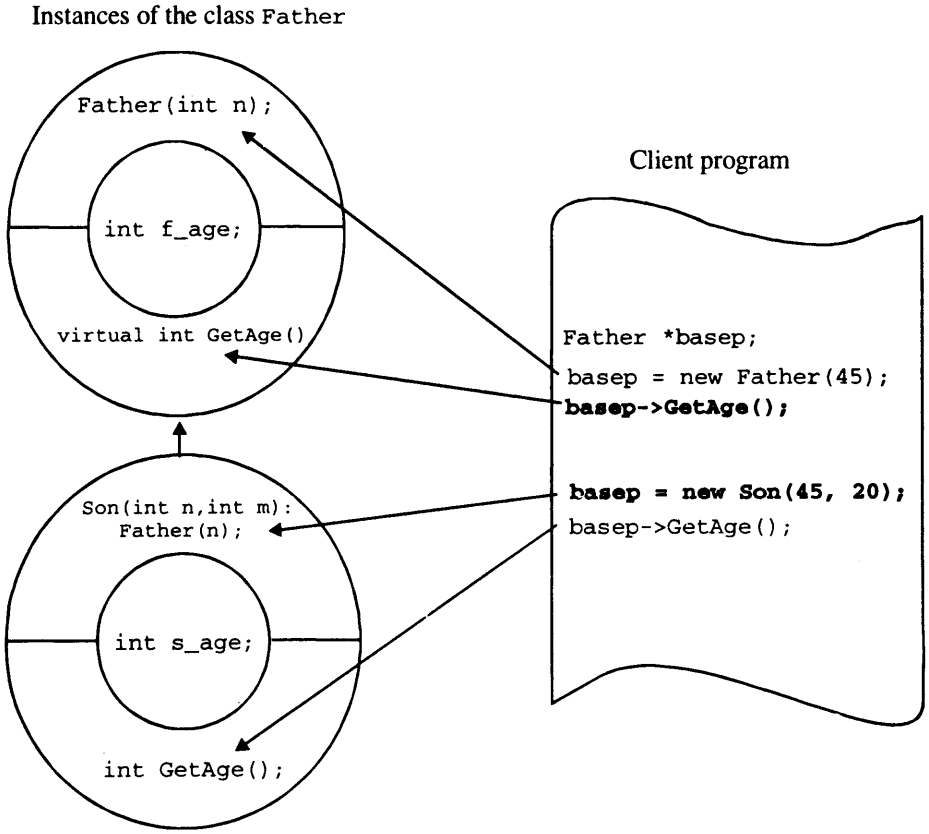

Figure 15.6: Virtual functions and dynamic binding (base pointer accessing derived objects)

In `main()`, the statement

```
Father *basep;
```

creates a pointer variable to the object of the base class `Father`, and the statement

```
basep = new Father(45);     // pointer to Father
```

creates an object of the class `Father` dynamically and assigns the pointer to the variable `basep`. The statement

```
cout << basep->GetAge() << endl;  // calls father::GetAge
```

invokes the member function `GetAge()` of the `Father` class. The statement

```
basep = new Son(45, 20);   // pointer to son
```

creates an object of the class `Son` dynamically and assigns its address to the pointer variable `basep`. The statement

```
cout << basep->GetAge() << endl;  // calls Son::GetAge
```

invokes the member function `GetAge()` of the `Son` class. If a call to a non-virtual function is made in this case, it invokes the member function of the class `Father` instead of the class `Son`. Note that the same pointer is able to invoke base or derived class's member function depending on which class's object the pointer is addressing.

It is important to note that, *virtual functions have to be accessed through a pointer to the base class*. However, they can be accessed through objects instead of pointers. It is to be remembered that runtime polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. Note that, when a function is defined as virtual in the base class, and the same function is redefined in the derived class, then that function is *virtual by default*. Only class member functions can be declared as virtual functions. Regular functions and friend functions do not qualify as virtual functions.

15.5 Array of Pointers to Base Class Objects

A key property associated with polymorphism is late or dynamic binding, which ensures that if an operation with more than one implementation (method) is called on a *polymorphic entity*, then the appropriate version is selected on the basis of its *dynamic type* (and is called *runtime dispatch).* In C++ runtime dispatch is only available for operations declared as virtual in the superclass. The process of runtime dispatch of *a function call request* is illustrated in Figure 15.7. The code which requests runtime dispatcher holds pointers to objects of different classes of the same class hierarchy. One of the simplest methods of implementation is to create an array of pointers (or pointers to pointers or linked list or any other data structure suitable for holding pointers to objects) as a *pointer store house* and invoke functions dynamically by scanning over them.

In Figure 15.7, it can be observed that, the class `graphics` has the function `draw()`, which plots the points and each of the derived classes, `line`, `triangle`, `rectangle`, and `circle` have their own `draw()` function, which plots the corresponding entities on the screen. In the absence of virtual functions, all the outputs would be picture of *points* because all the calls refer to the function `draw()` of the base class. However, with *virtual functions*, the same segment of program code generates different outputs by invoking the member function of the corresponding object.

The program `draw.cpp` illustrates a practical usage of virtual functions and models the problem described above. It uses an array of pointers to objects for storing pointer to objects of different derived classes of the base class `graphics`. The common interface function in all the classes is `draw()`, which is declared as virtual in the base class and defined as a normal function in all the other derived classes.

```
// draw.cpp: graphic class hierarchy with virtual functions
#include <iostream.h>
class graphics
{
   public:
      virtual void draw()  // virtual draw function in base class
```

```
      {
         cout << "point" << endl;
      }
};
class line: public graphics
{
   public:
      void draw()
      {
         cout << "line" << endl;
      }
};
class triangle: public graphics
{
   public:
      void draw()
      {
         cout << "triangle" << endl;
      }
};
class rectangle: public graphics
{
   public:
      void draw()
      {
         cout << "rectangle" << endl;
      }
};
class circle: public graphics
{
   public:
      void draw()
      {
         cout << "circle" << endl;
      }
};
void main()
{
   graphics point_obj;
   line line_obj;
   triangle tri_obj;
   rectangle rect_obj;
   circle circle_obj;
   graphics *basep[] =
   {
      &point_obj, &line_obj,
      &tri_obj, &rect_obj, &circle_obj
   };
   cout << "Following figures are drawn with basep[i]->draw()..."<< endl;
   for( int i = 0; i < 5; i++ )
      basep[i]->draw();
}
```

Run

```
Following figures are drawn with basep[i]->draw()...
point
line
triangle
rectangle
circle
```

In `main()`, the statement

```
for( int i = 0; i < 5; i++ )
  basep[i]->draw();
```

invokes a different `draw()` version based on the object to which the current pointer `basep[i]` is pointing. (See Figure 15.7.)

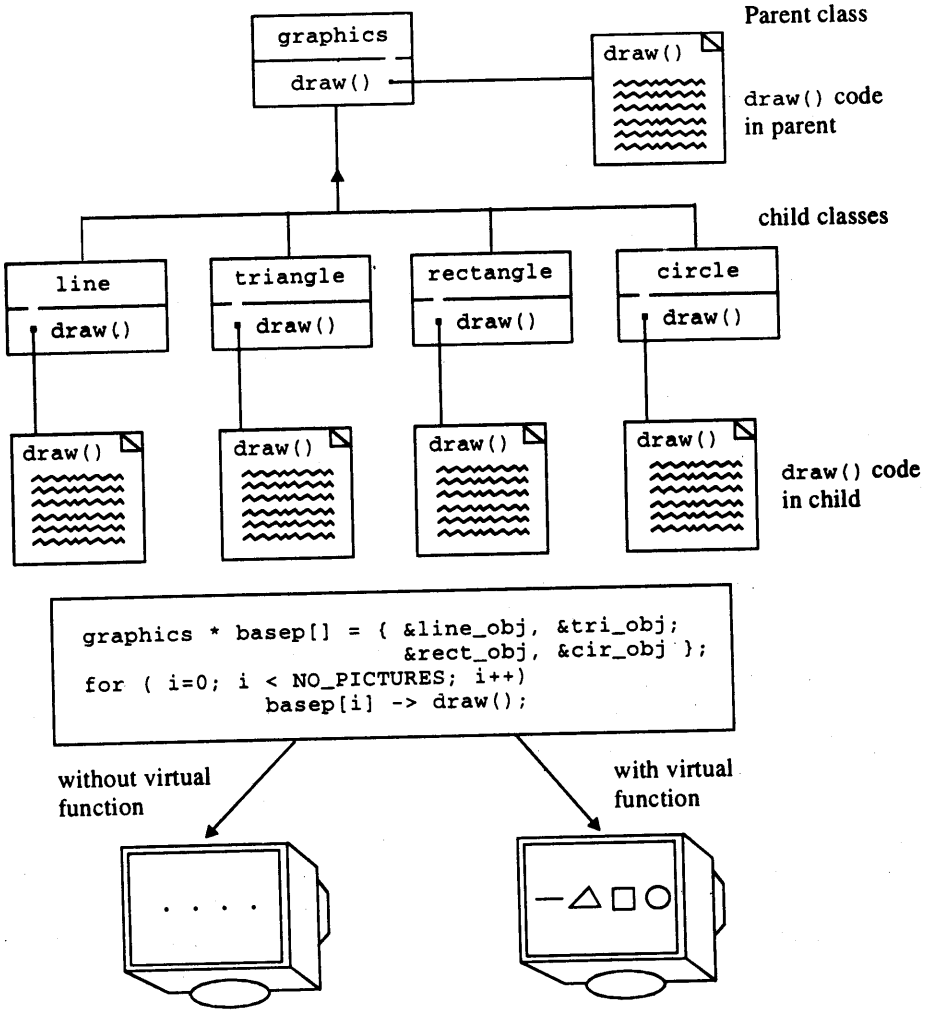

Figure 15.7: Compile time and runtime binding of functions

15.6 Pure Virtual Functions

Virtual functions defined inside the base class normally serve as a framework for future design of the class hierarchy; these functions can be *overridden* by the methods in the derived classes. In most of the cases, these virtual functions are defined with a *null-body*; it has no definition. Such functions in the base class are similar to *do-nothing* or *dummy* functions and in C++, they are called *pure virtual* functions. The syntax of defining pure virtual functions is shown in Figure 15.8. Pure virtual function is declared as a virtual function with its declaration followed by = *0*.

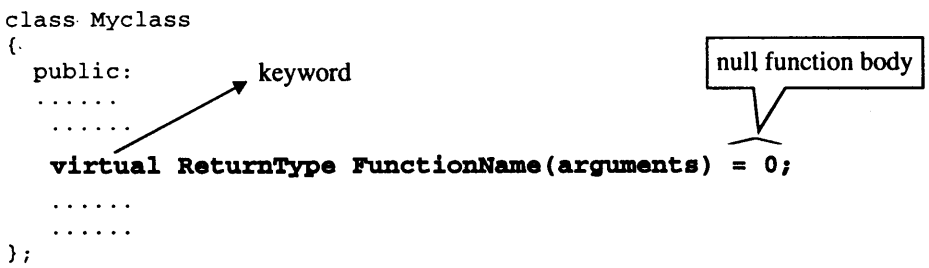

Figure 15.8: Syntax of pure virtual function

A pure virtual function declared in a base class has no implementation as far as the base class is concerned. The classes derived from a base class having a pure virtual function have to define such a function or redeclare it as a pure virtual function. It must be noted that, a class containing pure virtual functions cannot be used to define any objects of its own and hence such classes are called *pure abstract classes* or simply *abstract classes*. Whereas all other classes without pure virtual functions and which are instantiated are called as *concrete classes*.

A pure virtual function is an unfinished placeholder that the derived class is expected to complete. The following are the properties of pure virtual functions:

- A pure virtual function has no implementation in the base class hence, a class with pure virtual function cannot be instantiated.
- It acts as an empty bucket (virtual function is a partially filled bucket) that the derived class is supposed to fill.
- A pure virtual member function can be invoked by its derived class.

The concept of abstract class (a class with pure virtual function) is necessary in order to understand pure virtual functions and it is illustrated in the program `pure.cpp`. Note that a class with one or more pure virtual functions cannot be instantiated.

```
// pure.cpp: pure virtual function with abstract class
#include <iostream.h>
class AbsPerson
{
   public:
      virtual void Service1 (int n);     // normal virtual member function
      virtual void Service2 (int n) = 0; // Pure virtual member function
};
```

```
void AbsPerson::Service1(int n)
{
   Service2(n);
}
class Person : public AbsPerson
{
   public:
      void Service2(int n);
};
void Person::Service2(int n)
{
   cout << "The number of Years of service: " << (58 - n) << endl;
}
void main()
{
   Person Father, Son;
   Father.Service1(50);
   Son.Service2(20);
}
```

Run

```
The number of Years of service: 8
The number of Years of service: 38
```

In `main()`, the statement

```
Father.Service1(50);
```

invokes the virtual function `Service1()` defined in the class `AbsPerson` and this in turn invokes `Service2()`. The `Service2()` of the class `Person` is invoked instead of `AbsPerson`; it is declared as a pure virtual function.

15.7 Abstract Classes

Abstract classes (classes with atleast one virtual function) can be used as a framework upon which new classes can be built to provide new functionality. A *framework* is a combination of class libraries (set of cooperative classes) with predefined flow of control. It can be a set of reusable abstract classes and the programmer can extend them. For instance, abstract classes can be easily tuned to develop graphical editors for different domains like artistic drawing, music composition, and mechanical CAD. Abstract classes with virtual functions can be used as an aid to debugging. Suppose, it is required to build a project consisting of a number of classes, possibly using a large number of programmers. It is necessary to make sure that every class in the project has a common debugging interface. A good approach is to create an abstract class from which all other classes in the project will be inherited. Since any new classes in the project must inherit from the base class, programmers are not free to create a different interface. Therefore, it can be guaranteed that all the classes in the project will respond to the same debugging commands.

The implementation of such a software system is illustrated by creating a header file containing an abstract debugger class with abstract functions. The header file `debug.h` is an example of an abstract base class for debugging. (The program `pure.cpp` has the pure abstract class `AbsPerson`.)

```
// debug.h: Abstract class for debugging
#include <iostream.h>
class debuggable
{
   public:
      virtual void dump()
      {
        cout<< "debuggable error:no dump() defined for this class"<<endl;
      }
};
```

If someone derives a new class from the class `debuggable` and does not redefine `dump()`, it warns when the user tries to dump any object of that new class, because the base class version of `dump()` will be used. A few classes derived from the class `debuggable` are listed in the program `dbgtest.cpp`, for testing the `debuggable` class.

```
// dbgtest.cpp: testing of debuggable class
#include "debug.h"
class X: public debuggable
{
   int a, b, c;
   public:
      X( int aa = 0, int bb = 0, int cc = 0 )
      {
         a = aa; b = bb; c = cc;
      }
      // other implementation of dump
      void dump()
      {
         cout << "a=" << a << " b=" << b << " c=" << c << endl;
      }
};
class Y: public debuggable
{
   int i, j, k;
   public:
      Y( int ii = 0, int jj = 0, int kk = 0 )
      {
         i = ii; j = jj; k = kk;
      }
      // other implementation of dump
      void dump()
      {
         cout << "i=" << i << " j=" << j << " k=" << k << endl;
      }
};
class Z: public debuggable
{
   int p, q, r;
   public:
      Z( int pp = 0, int qq = 0, int rr = 0 )
```

```
      {
         p = pp; q = qq; r = rr;
      }
};
void main()
{
   X x( 1, 2, 3 );
   Y y( 2, 4, 5 );
   Z z;
   x.dump();
   y.dump();
   z.dump();
   // you can treat x, y, and z as members of the class debuggable
   debuggable *dbg[3];
   dbg[0] = &x;
   dbg[1] = &y;
   dbg[2] = &z;
   cout<< "Dumping through passing the same message to all objects...\n";
   for( int i = 0; i < 3; i++ )
      dbg[i]->dump();
}
```

Run

```
a=1 b=2 c=3
i=2 j=4 k=5
debuggable error: no dump() defined for this class
Dumping through passing the same message to all objects...
a=1 b=2 c=3
i=2 j=4 k=5
debuggable error: no dump() defined for this class
```

In `main()`, the statements

```
x.dump();
y.dump();
```

invoke their own implementation of `dump()` whereas, the statement

```
z.dump();
```

executes the virtual function `dump()` defined in the base class since it does not have an implementation of `dump()` in its own class. The statement which is in the scope of the `for` loop

```
dbg[i]->dump();
```

passes the same messages to all the objects, which are instances of the class derived from the class `debuggable`. All of them respond in different ways to the same message. If they do not have any response-function of their own, they respond through their parent function (in this the object `z` responds by invoking the `dump()` defined in the parent class `debuggable`). Thus, any object in the system can be dumped or can add the object's address to the list of debuggable pointers and call `dump()` as a member of the object. Hence, it is said that "`switch` *statements are to C what* `virtual` *functions are to C++.*"

An abstract class becomes very powerful when it is integrated into a system and changes are required for the interface. Imagine how difficult this would have been in a conventional language. First,

```
    basep->show();
    delete basep;
    // points to Son's object
    basep = new Son("Eshwarappa", "Rajkumar");   // pointer to son
    cout << "basep points to derived object..." << endl;
    basep->show();
    delete basep;
}
```

Run

```
basep points to base object...
Father's Name: Eshwarappa
~Father() is invoked
basep points to derived object...
Father's Name: Eshwarappa
Son's Name: Rajkumar
~Son() is invoked
~Father() is invoked
```

In `main()`, the variable `basep` is a pointer to the base class `Father`. The statement

```
basep = new Son("Eshwarappa", "Rajkumar");   // pointer to son
```

creates dynamic object of the class `Son` by allocating memory required for its data members also. It is important that memory allocated to object and its data members has to be released explicitly when the object pointed to by `basep` goes out of scope.

In the normal case, when the destructor of the base class is not a virtual function, the statement

```
delete basep;
```

would have deleted only the first string through the base class destructor, but in this case it also deletes the string, `Eshwarappa` through the derived class destructor. The base class destructor is declared as `virtual` and `basep` actually addresses the `Son`'s object and hence, the destructors in the `Son`'s class as well as the `Father`'s class are invoked. Note that while constructing an object, *the constructors are invoked from the top of a hierarchy (top most base class) upto the current class and while destroying an object, destructors are invoked from the current class to the top most base class in the hierarchy*. For instance, in the above program, the statement

```
basep = new Son("Eshwarappa", "Rajkumar");   // pointer to son
```

invokes the constructor of the class `Father` first and then the constructor of the class Son. The statement

```
delete basep;
```

having `basep` pointing to the dynamically created instance of the class `Son`, invokes destructor of the class `Son` first and the destructor of the class `Father` (unlike in the natural world, in C++ son dies first before his father; however there are exceptions).

Virtual destructor is used in the following situations:

- A virtual destructor is used when one class needs to delete object of a derived class that are addressed by the base-pointers and invoke a base class destructor to release resources allocated to it.
- Destructors of a base class should be declared as virtual functions. When a delete operation is performed on an object by a pointer or reference, the program will first call the object destructor instead of the destructor associated with the pointer or reference type.

15.9 How is Dynamic Binding Achieved ?

To perform dynamic binding of a member function in C++, the function is declared as virtual. Any function in a class can be declared as virtual. When functions are declared as virtual, the compiler adds a data member *secretly* to the class. This data member is referred to as a virtual pointer (VPTR). Virtual Table (VTBL) contains pointers to all the functions that have been declared as virtual in a class, or any other classes that are inherited. The program `vptrsize.cpp` shows evidence of the secret existence of VPTR.

```
// vptrsize.cpp: using sizeof operator to detect existence of VPTR
#include <iostream.h>
class nonvirtual
{
   int x;
   public:
      void func()
      {}
};
class withvirtual
{
   int x;
   public:
      virtual void func()
      {}
};
void main()
{
   cout << "sizeof( nonvirtual ) = " << sizeof( nonvirtual ) << endl;
   cout << "sizeof( withvirtual ) = " << sizeof( withvirtual );
}
```

Run

```
sizeof( nonvirtual ) = 2
sizeof( withvirtual ) = 4
```

Whenever a call to a virtual function is made in the C++ program, the compiler generates code to treat VPTR as the starting address of an array of pointers to functions. The function call code simply indexes into this array and calls the function located at the *indexed addresses*. The binding of the function call always requires this dynamic indexing activity; it always happens at runtime. That is, if a call to a virtual function is made, while treating the object in question, as a member of its base class, the correct derived class function will be called. It is illustrated in the program `shapes.cpp`.

```
// shapes.cpp: inheritance and virtual functions
#include <iostream.h>
class description
{
   protected:  // so derived class have access
      char *information;
   public:
      description( char *info ):information( info )
      {}
```

```
      virtual void show()
      {
         cout << information << endl;
      }
};
class sphere: public description
{
   float radius;
   public:
      sphere( char *info, float rad ):description(info), radius( rad )
      {}
      void show()
      {
         cout << information;
         cout << " Radius = " << radius << endl;
      }
};
class cube: public description
{
   float edge_length;
   public:
      cube(char *info, float edg_len):description(info),edge_length(edg_len )
      {}
      void show()
      {
        cout << information;
        cout << " Edge Length = " << edge_length << endl;
      }
};
sphere small_ball( "mine", 1.0 ),
      beach_ball( "plane", 24.0 ),
      plan_toid( "moon", 1e24 );
cube crystal( "carbon", 1e-24 ),
      ice( "party", 1.0 ),
      box( "card board", 16.0 );
description *shapes[] =
{
      &small_ball,
      &beach_ball,
      &plan_toid,
      &crystal,
      &ice,
      &box
};
void main()
{
      small_ball.show();
      beach_ball.show();
      plan_toid.show();
      crystal.show();
      ice.show();
```

```
   box.show();
   // put all description in the list
   cout << "Dynamic Invocation of show()..." << endl;
   for( int i = 0; i < sizeof( shapes )/sizeof( shapes[0] ); i++ )
      shapes[i]->show();
}
```

Run

```
mine Radius = 1
plane Radius = 24
moon Radius = 1e+24
carbon Edge Length = 1e-24
party Edge Length = 1
card board Edge Length = 16
Dynamic Invocation of show()...
mine Radius = 1
plane Radius = 24
moon Radius = 1e+24
carbon Edge Length = 1e-24
party Edge Length = 1
card board Edge Length = 16
```

From the output, it can be observed that virtual functions are essential for creating objects with the same interface and similar functionality but with different implementations. A debatable issue is "Why is the programmer given the option to make a function virtual and why not just let the compiler create all functions as virtual ?" C++ allows the programmer to decide whether to declare function as virtual or non-virtual. This design decision has been made to favor runtime efficiency. A virtual function requires an extra dereference to be made when it is invoked. The language defaults are in favor of maximum efficiency, which is accomplished through static binding. Thus, the programmer is forced to be aware of the difference between early and late binding, and to know when to apply late binding. Several other object-oriented languages, such as Smalltalk and Java, always use *late binding*.

Virtual Functions Trade-Offs

C++ stores the addresses of the virtual member functions in the internal table. When C++ statements call these member functions, the correct address is fetched from the internal table; this process consumes some time. Hence, the use of virtual functions reduces the program's performance to a certain extent but at the same time offers greater flexibility.

15.10 Rules for Virtual Functions

The following rules hold good with respect to virtual functions:

- When a virtual function in a base class is created, there must be a definition of the virtual function in the base class even if base class version of the function is never actually called. However pure virtual functions are exceptions.
- They cannot be static members.
- They can be a friend function to another class.
- They are accessed using object pointers.

- A base pointer can serve as a pointer to a derived object since it is type-compatible whereas a derived object pointer variable cannot serve as a pointer to base objects.
- Its prototype in a base class and derived class must be identical for the virtual function to work properly.
- The class cannot have virtual constructors, but can contain virtual destructor. In fact, virtual destructors are essential to the solutions of some problems. It is also possible to have *virtual operator overloading*.
- More importantly, to realize the potential benefits of virtual functions supporting runtime polymorphism, they should be declared in the *public* section of a class.

Review Questions

15.1 Describe different methods of realizing polymorphism in C++.

15.2 Justify the need for virtual functions in C++.

15.3 Why C++ supports type compatibles pointers unlike C ?

15.4 State which of the following statements are TRUE or FALSE. Give reasons.
(a) In C++, pointers to `int` data type can be used to point to float types.
(b) Pointer to base class can point to an object of any class.
(c) Pointer to a class at the top of the class hierarchy can point to any class objects in that hierarchy.
(d) Virtual functions allows to invoke different function with the same statement.
(e) The sizeof a class having virtual function is the same as that without virtual functions.
(f) A class with virtual function can be instantiated.
(g) A class with pure virtual function can be instantiated.
(h) A class with pure virtual functions is created by designers whereas, derived classes are created by programmers.
(i) Specification of a virtual function in the base class and its derived class must be same.
(j) Pure virtual functions postpone implementation of a member function to its derived class.

15.5 Create a vehicle class hierarchy with top most base having the following specification:

```
class vehicle
{
   int reg_no;
   int cost;
   public:
      virtual void start() = 0;
      virtual void stop();
      virtual void show();
      ....
};
```

Write a complete program having derived classes such as heavy, lightweight vehicle, etc.

15.6 What is runtime dispatching ? Explain how C++ handles runtime dispatching.

15.7 What are pure virtual functions ? How do they differ from normal virtual functions ?

15.8 What are abstract classes ? Write a program having `student` as an abstract class and create many derived classes such as Engineering, Science, Medical, etc., from the `student` class. Create their objects and process them.

15.9 What are virtual destructors ? How do they differ from normal destructors ? Can constructors be declared as virtual constructors ? Give reasons.

15.10 Explain how dynamic binding is achieved by the C++ compilers. What is the size of the following classes:

```
class X
{
   int x;
   public:
       void read();
};
class Y
{
   int a;
   public:
       virtual void read();
};
class Z
{
   int a;
   public:
       virtual void read();
       virtual void show();
};
```

15.11 What are the rules that need to be kept in mind in deciding virtual functions ?

15.12 Correct the errors in the following program and include missing components:

```
class ABC
{
   int a;
   public:
};
void main()
{
   ABC.a1;
   a1.read();
   a1.show();
   ABC a2 = 10;
   a2.show();
}
```

15.13 Consider an example of book shop which sells books and video tapes. These two classes are inherited from the base class called `media`. The `media` class has command data members such as `title` and `publication`. The `book` class has data members for storing number of pages in a book and the `tape` class has the playing time in a tape. Each class will have member functions such as `read()` and `show()`. In the base class, these members have to be defined as virtual functions. Write a program which models the class hierarchy for book shop and processes objects of these classes using pointers to the base class.

16

Generic Programming with Templates

16.1 Introduction

A significant benefit of object-oriented programming is reusability of code which eliminates redundant coding. An important feature of C++ called *templates* strengthens this benefit of OOP and provides great flexibility to the language. Templates support *generic programming*, which allows to develop reusable software components such as functions, classes, etc., supporting different data types in a single framework. For instance, functions such as sort, search, swap, etc., which support various data types can be developed.

A template in C++ allows the construction of a family of template functions and classes to perform the same operation on different data types. The templates declared for functions are called *function templates* and those declared for classes are called *class templates*. They perform appropriate operations depending on the data type of the parameters passed to them.

A C++ function/class is normally designed to handle a specific data type. Often, their functionality makes sense conceptually with other data types. Considering a class/function as a framework around a data-type and supporting various operations on that data type, makes sense to isolate the data type altogether from the function/class. It allows a single template to deal with a *generic data type* `T`.

16.2 Function Templates

There are several functions of considerable importance which have to be used frequently with different data types. The limitation of such functions is that they operate only on a particular data type. It can be overcome by defining that function as a *function template* or *generic function*. A function template specifies how an individual function can be constructed. The program `mswap.cpp` illustrates the need for function templates. It consists of multiple `swap` functions for swapping different values of different data types.

```
// mswap.cpp: Multiple swap functions
#include <iostream.h>
void swap( char & x, char & y )
{
   char t;  // temporary variable used in swapping
   t = x;
   x = y;
   y = t;
}
void swap( int & x, int  & y ) // by reference
{
   int t;   // temporary variable used in swapping
```

```
   t = x;
   x = y;
   y = t;
}
void swap( float & x, float & y ) // by reference
{
   float t; // temporary variable used in swapping
   t = x;
   x = y;
   y = t;
}
void main()
{
   char ch1, ch2;
   cout << "Enter two Characters <ch1, ch2>: ";
   cin >> ch1 >> ch2;
   swap( ch1, ch2 ); // compiler invokes swap( char &a, char &b );
   cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
   int a, b;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   swap( a, b );  // compiler invokes swap( int &a, int &b );
   cout << "On swapping <a, b>: " << a << " " << b << endl;
   float c, d;
   cout << "Enter two floats <c, d>: ";
   cin >> c >> d;
   swap( c, d );  // compiler invokes swap( float &a, float &b );
   cout << "On swapping <c, d>: " << c << " " << d;
}
```

Run

```
Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5
```

The above program has three swap functions

```
void swap( char & x, char & y );
void swap( int & x, int & y );
void swap( float & x, float & y );
```

whose logic of swapping is same and differs only in terms of data-type. Such functions can be declared as a single function template without redefining them for each and every data type. The C++ template feature enables substitution of a single piece of code for all these overloaded functions with a single template function as follows:

```
template <class T>
void swap( T & x, T & y ) // by reference
{
   T t;  // template type temporary variable used in swapping
```

```
    t = x;
    x = y;
    y = t;
}
```

Such functions are known as *function templates.* When swap operation is requested on operands of any data type, the compiler creates a function internally without the user intervention and invokes the same.

Syntax of Function Template

A function template is prefixed with the keyword `template` and a list of template type arguments. These template-type arguments are called *generic data types*, since their exact representation (memory requirement and data representation) is not known in the declaration of the function template. It is known only at the point of a call to a function template. The syntax of declaring the function template is shown in Figure 16.1.

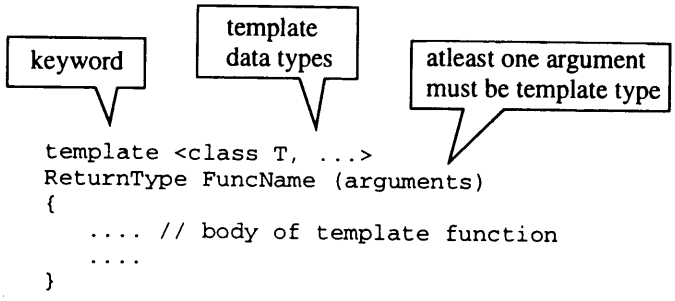

Figure 16.1: Syntax of function template

The syntax of a function template is similar to normal function except that it uses variables whose data types are not known until a call to it is made. A call to a template function is similar to that of a normal function and the parameters can be of any data-type. When the compiler encounters a call to such functions, it identifies the data type of the parameters and creates a function internally and makes a call to it. The internally created function is unknown to the user. The program `gswap.cpp` makes use of templates and avoids the overhead of rewriting functions having body of the same pattern, but operating on different data types.

```
// gswap.cpp: generic function for swapping
#include <iostream.h>
template <class T>
void swap( T &x, T & y ) // by reference
{
   T t;  // template type temporary variable used in swapping
    t = x;
    x = y;
    y = t;
}
void main()
{
   char ch1, ch2;
   cout << "Enter two Characters <ch1, ch2>: ";
```

```
    cin >> ch1 >> ch2;
    swap( ch1, ch2 ); // compiler creates and calls swap( char &x, char &y );
    cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( a, b );  // compiler creates and calls swap( int &x, int &y );
    cout << "On swapping <a, b>: " << a << " " << b << endl;
    float c, d;
    cout << "Enter two floats <c, d>: ";
    cin >> c >> d;
    swap( c, d );  // compiler creates and calls swap( float &x, float &y );
    cout << "On swapping <c, d>: " << c << " " << d;
}
```

Run

```
Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5
```

In `main()`, the statement

```
swap( ch1, ch2 );
```

invokes the `swap` function with `char` type variables. When it is encountered by the compiler, it internally creates a function of type,

```
swap( char &x, char &y );
```

The compiler automatically identifies the data type of the arguments passed to the template function and creates a new function and makes an appropriate call. The process of handling the template functions by the compiler is totally invisible to the user. Similarly, the compiler converts the following calls

```
swap( a, b );  // compiler creates swap( int &x, int &y );
swap( c, d );  // compiler creates swap( float &x, float &y );
```

into equivalent functions and calls them based on their parameter data types. Theoretically speaking, all the data types share the same template function swap. However, the complier has created three swap functions operating on `char`, `int`, and `float`.

Invocation of Function Template

The example of the function template for finding the maximum of two data items is given below:

```
template <class T>
T max( T a, T b )
{
   if( a > b )
       return a;
   else
       return b;
}
```

The function template is invoked in the same manner as a normal function as follows:

```
x = max( y, z );
```

However, it is processed differently by the compiler. The compiler creates a new function using its template and makes a call to it. A function generated internally from a function template is called *template function.* Template arguments are not specified explicitly while calling a function template. The program max1.cpp demonstrates the method of declaring a function template and its usage.

```
// max1.cpp: finding maximum of two data items using function template
#include <iostream.h>
template <class T>
T max( T a, T b )
{
   if( a > b )
      return a;
   else
      return b;
}
void main()
{
   // max with character data types
   char ch, ch1, ch2;
   cout << "Enter two characters <ch1, ch2>: ";
   cin >> ch1 >> ch2;
   ch = max( ch1, ch2 );
   cout << "max( ch1, ch2 ): " << ch << endl;
   // max with integer data types
   int a, b, c;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   c = max( a, b );
   cout << "max( a, b): " << c << endl;
   // max with floating data types
   float f1, f2, f3;
   cout << "Enter two floats <f1, f2>: ";
   cin >> f1 >> f2;
   f3 = max( f1, f2 );
   cout << "max( f1, f2): " << f3;
}
```

Run

```
Enter two characters <ch1, ch2>: A B
max( ch1, ch2 ): B
Enter two integers <a, b>: 20 10
max( a, b): 20
Enter two floats <f1, f2>: 20.5 30.9
max( f1, f2): 30.9
```

In the above program, the compiler creates as many `max()` functions as the number of calls to the function template `max()`. Once, an internal function is created for a particular data type, all future invocation to the function template with that data type will refer to it. For instance, the statement

```
c = max( a, b );      // a, b, and c are integers
```