Sooner or later the OOPL community will have to realize [that] if you cannot draw a precise picture of the pattern, it doesn't exist.

Edward Yourdon, 1992

Do not multiply objects without necessity.

W. Occam

5.1 Introduction

Object oriented programming extends abstract data types to allow for type/subtype relationships. This is achieved through a mechanism referred to as inheritance. It is a process of creating new classes, called derived classes, from existing or base classes. The derived class inherits all the capabilities of the base class but can add new features and refinements of its own. By adding these refinements the base class does not change. Rather than implementing shared characteristics, a class can inherit selected data members and member functions of other classes. In C++ inheritance is implemented through the mechanism of class derivation. Class inheritance allows the members of one class to be used as if they were members of a second class.

Inheritance has several advantages to offer. The most important among these is that it permits code reusability. Once a base class is written and debugged, it need not be touched again, but at the same time it can be adapted to work in different situations. Reusing existing code saves time and money, and increases a program's reliability. This code reusability is of great help in the case of distributing class libraries. A programmer can use a class created by person or company (like Microsoft Foundation Classes (MFC) or JAVA classes etc.), and, without modifying it can derive other classes from it that are suited to particular programming situations.

If we say that class B publicly inherits from class A, we are telling the C++ compiler that every object of type B is also an object of type A, but not vice-versa. We are saying that anything that is true of an object A is also true of an object B, but not vice-versa. We are saying that A represents a more general concept than B, that B represents a more specialized concept than A. We are asserting that anywhere an object of type A can be used; an object of type B can be used as well, because every object of B is an object of type A. On the other hand, if you need an object of type B, an object of type A will not do, as every B is A, but not vice-versa. Hence the single most important rule in object oriented programming with C++ is this - public inheritance means "is a".

In addition to "is a" concept we will look at another concept called "has a" or "is implemented in terms of" concept. For example, we have a class person that is defined in terms of three other classes - string, address and phone_number.

```
class string   { };
class address  { };
class phone_number   { };
class person
{
   string name;
   address home;
   phone_number home, office;
};
```

In the previous example, the person class demonstrates “has a” relationship. A person object has a name, an address, home and office phone numbers. We can’t say a person is a name or a person is an address or a person is a home or office phone number.

To understand inheritance, we need to make a distinction between relationship between objects and relationship between classes.

Objects are tied together with “has a” (ownership) links. For example,
Jay has a manager named Ashok
That Ford has a specific V-8 engine.

Objects are tied to classes with “is a” (membership) links
Ashok is a manager
That Ford’s engine is a V-8 engine

In inheritance, classes are tied to other, more general classes with is a links
Every supervisor is an employee
Every V-8 engine is an engine

Inheritance lets us define relationships explicitly that we might otherwise implement with implicit relationships among groups of linked objects.

Another benefit of inheritance is software organization and understandability. By having objects organized according to classes, each object in a class inherits characteristics from parent objects. This makes the job of documenting, understanding, and benefiting from previous generations of software easier, because the functionality of the software has incrementally grown as more objects are created. Objects at the end of long inheritance chain can be very specialized and powerful.

In structural language like C, one is only able to copy code and change it but it hasn't worked well. The solution here is, instead of creating new classes from scratch, you use the existing classes that someone has already built, tested and debugged. There are two ways to accomplish this.

1. Composition and
2. Inheritance

Composition means you compose your new class with the objects of existing classes but without soiling existing code. Here you just use the functionality of the code and not redefine its form. Composition is also called containment.

5.1 Code Listing

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

class parent
{
  int num1, num2;

   protected:
      int getNum1()  {  return num1; }

   public:
```

```
    parent(int n1 = 0, int n2 = 0) {
      num1 = n1;
      num2 = n2;
    }

    int getNum2() {  return num2; }
    int mathOp(char type = '+') {
      cout << num1 << " " << num2 << endl;
      if(type == '*')
        return num1 * num2;
      return num1 + num2;
    }
};

class child : public parent
{
  int num3;
  public:
    child(int n1, int n2, int n3):parent(n1, n2) { num3 = n3; }
};

void main()
{
   parent pobj(5,15);
   cout << pobj.mathOp() << endl;
   child cobj(10, 20, 30);

   cout << cobj.getNum2() << endl;
   cout << cobj.mathOp('*') << endl;
   // What happens if the next two lines of code are uncommented?
   // cout << "Private data is " << obj.num1 << endl;
   // cout << "Protected function is " << obj.getNum1() << endl;
}
```

5.2 Shape class example

In designing a graphics system, we might decide to build all our graphics objects on top of a base object called a shape. Everything common to all these objects will be defined in our shape class (The object's center, the object's color and a pointer to the next object in the linked list). In addition, all our objects have a common interface like functions to draw and erase an object, a function to move the object to a new position and a function to rotate the object in place. Refer to Fig. 5.1 for the shape class design.

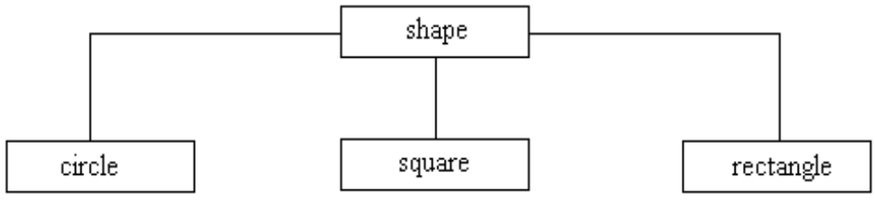

Fig. 5.1 Shape class design

We will first define two classes, point and color that will be used in the shape class.

5.2 Code Listing

```
#include <iostream.h>
#include <string.h>
```

```
class point
{
   int x, y;
   public:
      point()
      {
         cout << "\n default constructor of point called";
      }

      point(int i, int j)
      {
         cout << "\n two argument constructor of point called";
         x = i;
         y = j;
      }
      friend ostream & operator << (ostream &, const point &);
};

ostream & operator << (ostream & o, const point & p)
{
   return o << p.x << " : " << p.y;
}

class color
{
   char *c;
   public:
      color()
      {
         cout << "\n default constructor of color called";
      }

      color(char *x)
      {
         cout << "\n one argument constructor of color called";
         c = new char[strlen(x) + 1];
         strcpy(c, x);
      }
};
```

Our shape class is as follows

```
class shape
{
   shape *next;

   protected:
      point center;
      color col;

   public:
      shape()
      {
         cout << "\n default  constructor of shape called";
      }

      shape(point p)
      {
         center = p;
         cout << "\n one argument shape constructor called";
      }

      shape(point p, color c1)
      {
         center = p;
         col = c1;
         cout << "\n two argument shape constructor called";
      }
```

```
      void move(point to)
      {
         erase();
         center = to;
         draw();
         cout << "\n shape move called";
      }

      virtual void erase()
      {
         cout << "\n Shape erase called";
      };

      virtual void draw()
      {
         cout << "\n Shape draw called";
      };

      virtual void rotate(int)
      {
         cout << "\n Shape rotate called";
      };

      virtual ~shape()
      {
         cout << "\n Shape destructor called";
      }
};
```

With inheritance, we add a new protected region to our class definition.

Private members (both data and functions) are accessible only to those declared in the definition of the base class and not available to derived classes.

Protected designation means “make these items visible to this class and to classes that derive from this class”. When preceding a list of class members, the protected keyword specifies that those members are accessible only from member functions and friends of the class and its derived classes. This applies to all members declared up to the next access specifier or the end of the class.

Protected data members and functions are fully visible to derived classes, but otherwise remain private to others. It cannot be accessed from functions outside these classes such as main(), hence the protected access specifier.

When preceding the name of a base class, the protected keyword specifies that the public and protected members of the base class are protected members of the derived class.

Default access of members in a class is private. Default access of members in a structure or union is public. Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

Public members are accessible to everyone. All public base class members are accessible inside the scope of the derived class. Our different shapes would then inherit from the base class.

Code Listing continued

```
class circle : public shape
{
   protected:
      int radius;
```

```
   public:
      circle(point p, int r) : shape(p)
      {
         radius = r;
         cout << "\n two argument constructor of circle called";
      }

      circle(point p, int r, color c1) : shape(p, c1)
      {
         radius = r;
         cout << "\n three argument constructor of circle called";
      }

      void erase()
      {
         cout <<"\n circle erase called";
      }

      void draw()
      {
         cout <<"\n circle draw called";
      }

      void rotate(int x)
      {
         cout << "\n circle rotate called";
      }

      ~circle()
      {
         cout << "\n circle destructor called";
      }
};

class square : public shape
{
   protected:
      int side;

   public:
      square(point p, int s) : shape(p)
      {
         side = s;
         cout << "\n two argument constructor of square called";
      }

      square(point p, int s, color c1) : shape(p, c1)
      {
         side = s;
         cout << "\n three argument constructor of square called";
      }

      void erase()
      {
         cout << "\n square erase called";
      }

      void draw()
      {
         cout << "\n square draw called";
      }

      void rotate(int x)
      {
         cout << "\n square rotate called";
      }

      ~square()
```

```
      {
         cout << "\n square destructor called";
      }
};
```

Refer to Fig. 5.2 for the structure of a circle.

next
center
col
radius

Fig. 5.2 Structure of a circle

The next pointer is accessible only to member functions defined as part of class shape, center and col are accessible to members of shape and circle, and radius is accessible to members of class circle. A main program to test the above concept is shown below.

5.2 Code Listing (main function)

```
#include "prog52.h"
void main()
{
   point p(10, 20), q(30, 50);

   shape *sp = new circle(p, 10);
   sp->draw();
   sp->move(q);

   delete sp;

   sp = new square(p, 5);
   sp->draw();
   sp->move(q);

   delete sp;
}
```

In the above program, we have assigned the address of a derived class object to the base class pointer through the statement

shape *sp = new circle(p, 10);

The above statement doesn't result in an error, although we are assigning an object of a derived class to a base class, since the compiler relaxes the type checking. The rule is that pointers to objects of a derived class are type compatible with pointers to objects of the base class. When we execute the statement,

sp->draw();

which draw does it invoke - the shape draw or circle draw? The function of the base class (shape) is called if the virtual keyword in class shape is removed. This is because the compiler ignores the contents of the pointer **sp** and chooses the member function that matches the type of the pointer. Since, the pointer type matches the base class, the **draw()** of the base class is called. Same thing happens when we call draw() for the second time. Choosing functions in the above way, during compilation, is called early, or static binding. Sometimes we want this, but it doesn't solve the problem, where we want to access objects of different classes using the same statement. How it can be solved is explained in the next topic.

5.3 Polymorphism and Virtual functions

This idea of polymorphism requires the ability to determine the class of an instance and the member function to execute at runtime. In C++, we achieve polymorphism by declaring member functions as virtual. Virtual functions are not necessary for every C++ program but are nevertheless important. A virtual function is a function that does not really exist but still appears real to some parts of a program. Suppose we have a number of objects of different classes but we want to put all of them on a list and perform a particular operation on them using the same function call. This can be achieved using virtual functions. This is an important example of polymorphism, which means giving different meanings to the same thing.

A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

In our shape class, we declared erase(), rotate(), draw() and the destructor as virtual. Once we change the function to virtual, the previous function call executes differently, depending on the contents of **sp**. The rule here is that the compiler selects the function based on the contents of the pointer **sp**, and not based on the type of the pointer. How does the compiler know which function to compile, when it doesn't know which class the contents of **sp** may contain? In fact the compiler doesn't know what to do, so it defers the decision until the program is running. At runtime, when it knows what class is pointed to by **sp**, the appropriate version of **draw()** is called. This is called late binding or dynamic binding. Late binding has more overheads, but provides increased power and flexibility.

When a virtual function is created within an object, the object must keep track of that function. Many compilers build a virtual function table, called a v-table. A v-table is maintained for each type, and each object of that type keeps a virtual table pointer (called a vptr) that point to that table. Each object's vptr points to the v-table that, in turn, has a pointer to each of the virtual functions. When the shape part of circle is created, the vptr is initialized to point to the correct part of the v-table as shown in Fig. 5.3.

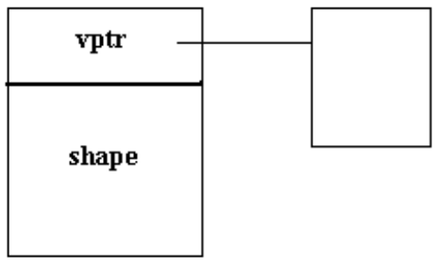

Fig. 5.3

When the circle constructor is called, creating the circle part of this object, the vptr is adjusted to point to the virtual function overrides (if any) in the circle object, as shown in Fig. 5.4.

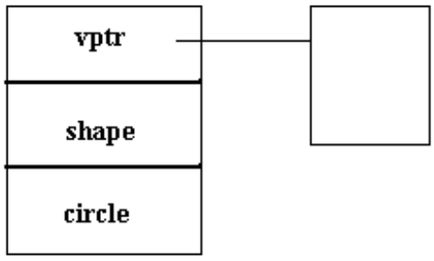

Fig. 5.4

If a member function is not declared virtual, the choice is always made at compile time. This choice is based on the type of the variable, and not its contents. If a member function is declared virtual, the choice is always made at run time. This choice is based on the contents of the variable, and not its type.

5.3.1 Member function calling

For a call to a member function to be resolved, the class of the object must be identified. If the class can be determined at compile time, a simple function call is generated with no extra overhead. If the class isn't known and the function was declared as virtual, then the choice of the function call is made at runtime.

Classes inherit all public and protected functions from their parent except constructors. By default, C++ provides a member wise copy constructor shape(shape &) and an assignment operator, which will work if the class does not involve pointer data members. This will not work for our class, as it involves a data member that is a pointer. We need to override.
If a class defines a function with the same name as one in parent, the child class function takes precedence.

```
void circle :: move(point to)
{
   cout << "We are moving a circle " << endl;
   shape :: move();
}

void main()
{
   point p(10, 20), q(30, 40);
   circle c(p, 10);
   c.move(q);        // calls circle :: move()
}
```

Rules for virtual functions

1. The virtual functions must be members of some class.

2. They cannot be static members

3. They are accessed by using object pointers.

4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.

6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.

7. We cannot have virtual constructors, but we can have virtual destructors.

8. While a base pointer can point to any type of the derived object, the reverse is not true. That is, we cannot use a pointer to a derived class to access an object of the base type.

9. When a base pointer points to a derived class, incrementing or decrementing it will not make it point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.

10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

5.3.2 Constructors and Destructors

Circle or square objects are shape objects. When a circle or a square is created, it's base constructor is called first, creating a shape. The circle or square constructor is then called, completing the construction of the circle or square object. In essence, the constructor for a base class is called prior to the derived class. By default, the constructor with a void argument list is called. Base class initialization can be performed during class initialization by writing the base class name, followed by the parameters excepted by the base class.

```
circle :: circle(point p, int rad, color c) : shape(p, c)
{
   radius = rad;
}
```

The argument received by the parent need not be those specified by the child

```
circle :: circle(point p, int rad, color c) : shape(p, "blue")
{
   radius = rad;
}
```

When an object is deleted or leaves scope, its destructors are invoked in the order from child to parent. This ordering gives us a chance to clean up add on structures before the base class's structures are reclaimed.

```
shape :: ~shape()
{
   cout << "shape destructor called" << endl;
}

circle :: ~circle()
{
   cout << "circle destructor called" << endl;
}

void main()
{
   point p(10, 20);
   circle c(p, 5);
```

```
}
```

The output is as follows

```
   circle destructor called
   shape destructor called
```

Normally, deletion through a pointer to a parent class will not cause all the destructors to be invoked.

```
void main()
{
   point p(10, 20);
   shape *sp = new circle(p, 5);
   delete sp;
}
```

The output is as follows

```
   shape destructor called
```

The cause of the problem is the call to delete. As you know, when a pointer is deleted, two things happen. First, one or more destructors are called. Second, memory is reclaimed. In the above situation, we have initialized a shape pointer to point to a circle, but this later forced the compiler to face a vexing question: when the shape pointer is deleted, which destructors should be called? You and I know that the pointer really points to a circle object, but how is the compiler supposed to really know that? The fact is that it doesn't, so it only calls the destructor for the base class - shape in this example.

We solve this problem by declaring the parent's destructor as virtual (refer to the shape class definition). The rule is "If any of the functions in a class is virtual, the destructor should be virtual as well". In other words, by declaring the destructor virtual in the base class, you tell the compiler that it must examine the object being deleted to see where to start calling destructors.

5.4 Polymorphism and Overloading

Overloading

- Can be used with any data type
- Can be used with any function not explicitly declared extern "C"
- Always selects the function at compile time
- Can select the function based upon any combination of arguments
- Can find a match by type casting of arguments

Polymorphism

- Can be used with class/struct objects
- Can be used only with member functions declared to be virtual
- May defer the selection of the function until runtime
- Selects the function based upon its first argument
- Must match the class of the argument, possibly via inheritance

5.5 Abstract Base Class (ABC) and Interface

There are situations in which you will want to define a super class that declares the structure of abstraction without providing the complete implementation of every method. It only defines a generalized form that will be shared by all subclasses, allowing each subclass to implement the

details in their own context. Such a class exists only to be inherited and cannot be instantiated. It should contain at least one method which is not implemented (only method prototype exists and no implementation). An abstract class contains at least one pure virtual function. Specify a virtual function as pure by placing = 0 at the end of its declaration. You don't have to supply a definition for a pure virtual function.

Any class extending it is required to provide a meaningful implementation of this "abstract" method, failing which the derived class would also be an abstract class. There can exist any number of abstract methods along with fully defined methods in an abstract class. The subclass may then choose to define all the abstract methods of the super class. If it doesn't do so, it is also an abstract class and cannot be instantiated. It just exists to be inherited further. Sometimes we have a class that is meant only for the purpose of inheritance. Our shape is one such example. We can specify that a class is an abstract base class by defining one or more member functions to be zero.

```
class shape
{
   // Rest of the code as before
   virtual void erase() = 0;
   virtual void draw() = 0;
   virtual void rotate(int) = 0;
};
```

Declaring a pure virtual function in a class signals two things to clients of the class

1. We cannot create an instance of such a class, either directly or by casting, however pointers and references to ABC objects are valid.
2. Any class that is derived from an ABC must override all the pure virtual methods it inherits.

Interfaces are similar to abstract classes. The difference is that you do not provide the implementation for any methods in an interface. Thus an interface is just a collection of method prototypes, which tell what must be done and not how it must be done.

In C++ there is no separate keyword “interface” like in Java. How we create an interface in C++, is to make all the class methods as pure virtual. One or more classes now can "implement" this interface and provide the "implementation" for all the methods prototyped in the interface. An interface just exists to be implemented by a class and any class that implements an interface must implement all the methods prototyped in the interface, if it doesn’t implement all the interface methods the class is an abstract class.

5.3 Code Listing

```
# include <iostream.h>
# include <string.h>

// Abstract Base class
class Employee
{
   protected:
      char name[30];
      double basic, netSal;

   public:
      Employee() { }
```

```
      Employee(char * s, double d)
      {
         strcpy(name, s);
         basic = d;
      }
      virtual void calSal() = 0;

      virtual ~Employee ()
      {
         cout << "destructor of employee called" << endl;
      }
      virtual void Show();
};

void Employee :: Show()
{
   cout << "Employee show" << endl;
   cout << "Name     :" << name << endl;
   cout << "Net Salary  :" << netSal << endl;
}

class Manager : public Employee
{
   protected:
      int type;

   public:
      void calSal();
      void Show();

      Manager(char * s, double d, int i) : Employee(s, d)
      {
         type = i;
      }

      ~Manager ()
      {
         cout << "destructor of Manager called" << endl;
      }
};

void Manager :: Show()
{
   cout << "Manager show" << endl;
   Employee :: Show();
   cout << "type     :" << type << endl;
}

void Manager :: calSal()
{
   cout << "calsal of manager called" << endl;
   double hra = (5.0/100.0) * basic;       // assuming hra is 5% of basic
   netSal = basic + hra;
}

void main()
{
   Employee *e = new Manager("Ravi", 10000, 1);
   e->calSal();
   e->Show();
   delete e;
   // What happens if the following statement is uncommented?
   // Employee e;
}
```

5.6 Private v/s Public inheritance

C++ treats public inheritance as an isa relationship by showing that, given a hierarchy in which a class student publicly inherits from a class person, the compiler implicitly converts student pointers to person pointers when it is necessary for a call to succeed. For example consider,

```
class person { ….. };
class student : private person { ….. };
void dance (const person & p) ;        // anyone can dance
void study (const student & s);        // only students study
person p;
student s;

dance (p);                             // fine, p is a person
dance (s);                             // error, a student isn't a person
```

Private inheritance does not mean "is a". If the inheritance relationship between the classes is private, the compiler will not convert a pointer to a derived class object (such as student) into a pointer to a base class object (such as person). The members inherited from private base class become private members of the derived class, even if they were protected or public in the base class.

If you make a class B privately inherit from a class A, you do so because you are interested in taking advantage of some of the code that has already been written for class A, not because there is any deep conceptual relationship between objects of type A and objects of type B. As such, private inheritance is purely an implementation technique. Private inheritance means that implementation only should be inherited; interface should be ignored. If B privately inherits from A, it means that B objects are implemented in terms of A objects, nothing more.

Inheritance Summary

1. A derived class inherits all the capabilities of the base class, but can add new features of its own. By making these additions the base class remains unchanged.

2. Protected members behave just like private members until a new class is derived from a base class that has protected members.

3. If a base class has private members these members are not accessible to the derived class. Protected members are public to derived classes but private to the rest of the program.

4. A derived class can specify that a base class is public or private by using the following notation in the definition of the derived classes.

   ```
   class c : public b
   class a : private b
   ```

   The public access specifier means that the protected members of the base class are protected members of the derived class and the public members of the base class are public members of the derived class. The private access specifier means that the protected and public members of the base class are private members of the derived class. Default access specifier is private.

5. When you execute the object of a derived class, the compiler executes the constructor function of the base class followed by the constructor function of the derived class. The parameter list for the derived class's constructor function may be different from that of the base class's constructor function. Therefore the constructor function for the derived class

must tell the compiler what values to use as arguments to the constructor function for the base class.

6. When a base and a derived class have public member functions with the same name and parameter list types, the function in the derived class gets a priority when the function is called as a member of the derived class object.

7. A program can declare objects of both the base and derived classes. The two objects are independent of one another.

8. A pure virtual means that function interface only is inherited.

9. A simple virtual function means that function interface plus a default implementation is inherited.

10. A nonvirtual function means that function interface plus a mandatory implementation is inherited.

5.8 Exercise

Theory

1. Explain the concept of "is a" and "has a" relationship.
2. What are the different forms of inheritance? Give an example for each.
3. Describe the syntax of single inheritance in C++.
4. What is an abstract base class? Why do we define one? What restrictions apply?
5. What is a virtual function and a virtual table? How is it used to implement polymorphism?
6. In what order are class constructors and destructors called when a derived class object is created and deleted. Why.
7. What does the "protected" keyword in the class definition mean?
8. What is the difference between "public" derivation and "private" derivation?
9. What is a virtual destructor? Why do we define one?
10. What is containership? How does it differ from inheritance? Describe how an object of a class that contains objects of other classes is created?

Problems

1. Define a class account with name and acct_no as data and operations such as getdata() and putdata(). From the account class derive classes savings and checking which have the following features: interest_rate and min_balance. The final output should look like this.
   Name:
   Account Number:
   Interest Rate:
   Minimum balance:

2. Define class base as follows
   class base
   {

```
    public:
        virtual self()
        {
            cout << "base" << endl;
        }
};
```

Derive 2 classes from base and for each define self() to write out the name of the class. Create objects of these classes and call self() for them. Assign the address of the derived classes and call self() through these pointers.

3. A company markets books and audio cassettes. Create a class publication that stores the title (string) and price (double) of a publication. Derive 2 classes from this class-book, which adds a page count (int) and tape, which adds a playing time in minutes (double). Write a program to read and output the data.