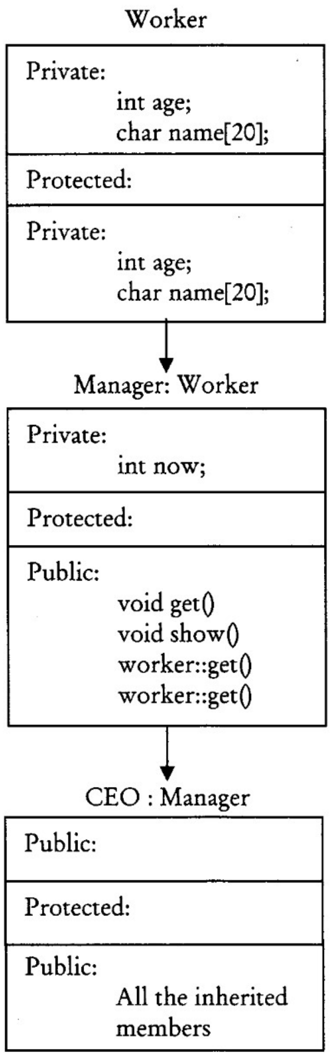
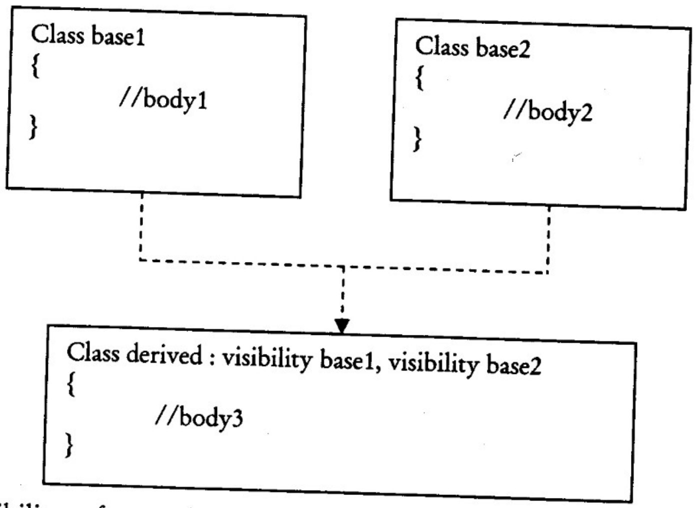


A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITENCE'. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like the child inheriting the physical feature of one parent and the and the intelligence of another. The syntax of the derived class is as follows:



```
Class base1
{
        //body1
}
```

```
Class base2
{
        //body2
}
```

```
Class derived : visibility base1, visibility base2
{
        //body3
}
```

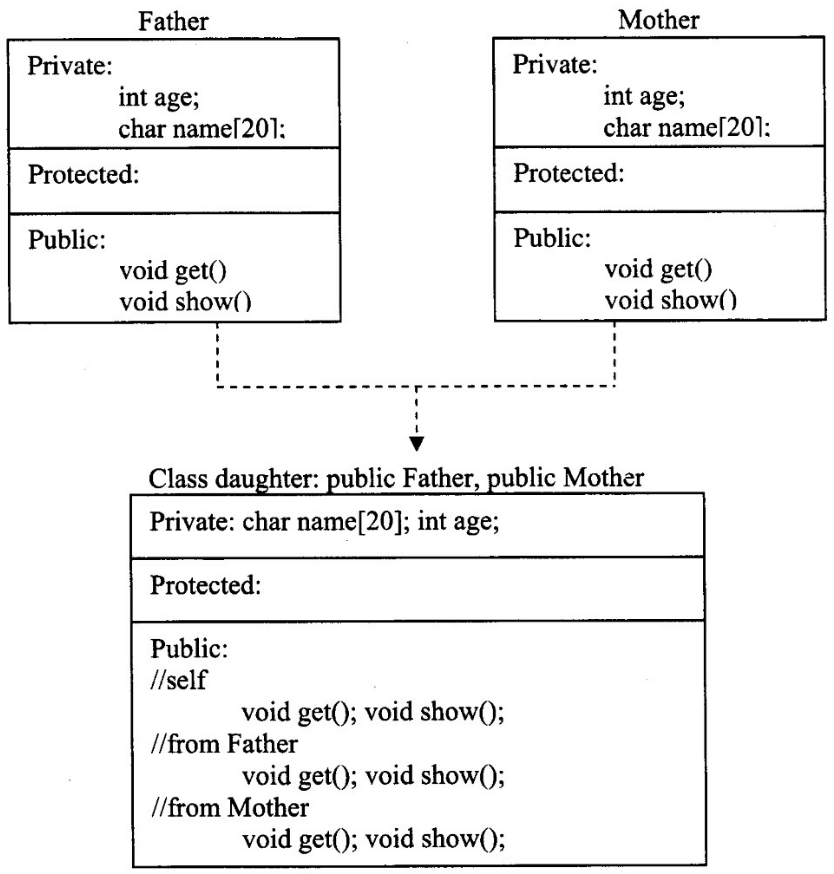
Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multiple inheritance.

```
    #include<iostream.h>
    #include<conio.h>
    class father                    //Declaration of base class1
    {
    int age;
    char name [20];
    public:
      void get ( );
      void show ( );
    };
    void father : : get ( )
    {
      cout << "your father name please";
      cin >> name;
      cout << "Enter the age";
      cin >> age;
    }
    void father : : show ( )
    {
cout<<"In my father's name is:"<<name<<"In my father's age is:<<age;
    }
    class mother                    //Declaration of base class 2
    {
```

```
    char name [20];
    int age;
    public:
      void get( )
      {
      cout << "mother's name please" << "In";
      cin >> name;
      cout << "mother's age please" << "in";
      cin >> age;
      }
      void show( )
      {
      cout << "In my mother's name is: "<<name;
      cout << "In my mother's age is: "<<age;
class daughter : public father, public mother //derived class: inheriting
{                                    //publicly
char name [20];          //the features of both the base class
    int std;
    public:
      void get ( );
      void show ( );
    };
    void daughter :: get ( )
    {
      father :: get ( );
      mother :: get ( );
      cout << "child's name: ";
      cin >> name;
      cout << "child's standard";
      cin >> std;
    }
    void daughter :: show ( )
    {
      father :: show ( );
      nfather :: show ( );
      cout << "In child's name is : "<<name;
      cout << "In child's standard: "<<std;
    }
```

```
main( )
{
clrscr( );
   daughter d1;
   d1.get  ( );
   d1.show  ( );
}
```

Diagramatic Representation of Multiple Inheritance is as follows:



7.7 CONTAINER CLASSES

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports get another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its member as shown below:

```
class A { };
class B { };
class C
{
   A a1; // creation of object of class A
   B b1; // creation of object of class B
};
```

All objects of C class will contain the objects a1 and b1. This kind of relationship is called containership or nesting. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other "ordinary" members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class C
{
   A a1; // creation of object of class A
   B b1; // creation of object of class B public:
   C (arglist): a1 (arglist 1), b1(arglist 2)
   {
          // constructor body
   }
};
```

arglist is the list of arguments that is to be supplied when a C object is defined. These parameters are used for initializing the members of C. arglist 1 is the argument list for the constructor of a and arglist 2 is the argument list for the constructor of b1. Remember a1 (arglist 1) and b1 (arglist 2) are function calls and therefore the arguments do not contain the data types. They are simply variables or constants.

for example,

```
( (int x, int y, float z) : a(x), (bx, z)
{
   //Assignment section
}
```

We can use as many member objects as are required in a class.

Check Your Progress

Fill in the blanks:

1. The mechanism of deriving a new class from an old one is called ....
2. Members of base class can be accessed using the ....
3. When a class inherits another, the members of the base class become members of the .... class.
4. A class can inherit the .... of two or more classes.

7.8 LET US SUM UP

Inheritance is the capability of one class to inherit properties from another class. It supports reusability of code and is able to simulate the transitive nature of real life objects. Inheritance has many forms: Single inheritance, multiple inheritance, hierarchical inheritance, multilevel inheritance and hybrid inheritance.

A subclass can derive itself publicity, privately or protectedly. The derived class constructor is responsible for invoking the base class constructor, the derived class can directly access only the public and protected members of the base class.

When a class inherits from more than one base class, this is called multiple inheritance. A class may contain objects of another class inside it. This situation is called nesting of objects and in such a situation, the contained objects are constructed first before constructing the objects of the enclosing class.

7.9 KEYWORDS

Abstract Class: A class serving only a base class for other classes and no objects of which are created.

Base Class: A class from which another class inherits (also called super class).

Containership: The relationship of two classes such that the objects of a class are enclosed within the other class.

Derived Class: A class inheriting properties from another class (also called sub-class).

Inheritance: Capability of one class to inherit properties from another class.

Inheritance Graph: The chain depicting relationship between a base class and derived class.

Visibility Mode: The public, private or protected specifier that controls the visibility and availability of a member in a class.

7.10 QUESTIONS FOR DISCUSSION

1. Define derived classes.
2. What is multi level inheritance?
3. Write a note on hierarchical inheritance.

4. Consider a situation where three kinds of inheritance are involved.
5. What is the difference between protected and private members?
6. What is the major use of multilevel inheritance?
7. Discuss a situation in which the private derivation will be more appropriate as compared to public derivation.
8. Write a C++ program to read and display information about employees and managers. Employee is a class that contains employee number, name, address and department. Manager class and a list of employees working under a manager.

Check Your Progress: Model Answer

1. inheritance.
2. visibility modes.
3. Derived
4. attributes

7.11 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo* C++, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with* C++, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference* C++, Tata Mc Graw Hill

LESSON 8

OVERLOADING

CONTENTS

8.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of function overloading
- Define operator overloading
- Describe the overloading of binary operators
- Identify and explain the overloading of unary operators

8.1 INTRODUCTION

C++ provides a rich collection of various operators. We have already seen the meaning and uses of many such operators in previous lesson. One special feature offered by C++ is operator overloading. This feature is necessary in a programming language supporting objects oriented features.

8.2 FUNCTION OVERLOADING

C++ allows you to define several different functions with the same name, provided their parameter list differs. Here's a small example of doing so.

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

```
void print (const char* string)
 cout << "print(\ "  " << string << " \" )" << endl;
void print(int i)
cout << "print ("<< i <<")" << endl;
int main (void)
print ("string printing");      // calls print (const char*)
print (5)                       // calls print(int)
return 0;
```

Compiling and Running Yields

Print ("string printing")

Print (5)

Handled right, this can severely reduce the member of names you have to remember. To do something similar in C, you'd have to give the function different names, like "print_int" and "print_string" of course, handled wrong, it can cause mess. Only overloaded functions when they actually do the same things- in this case, printing their parameters, had the functions been doing different things you would soon forget which of them did what. Function overloading is powerful, and it will be used throughout this course, but everything with power is dangerous, so be careful.

To differentiate between overloaded function, the parameter list is often included when mentioning their names. In the example above, we have the two functions "print (int)" and "print (const char*)", and not just two functions called "print".

Function over loading is a general concept in C++. To understand what is meant by it, consider an example first to illustrate the point.

```
void print (char *p)
cout << "Print a string" << p << "\n".
void print (int:)
cout << "Print an integer "<<p<< "\n";
void Dosomething ( )
char *p [ ] = "My code \0";
int mynumber = 33;
print (p); // call character Print
print (i); // call
```

It makes sense to use one function name for the same functionality to have a good readability of a program, even though different types have to be printed. Above, we have defined two functions print for different arguments and the compiler will decide which one is the right one to take in accordance with the arguments provided. On this level of our C++ knowledge this has only the meaning of good readable programs, but concept becomes essential when we introduce templates in section <mode 40. html>.

When we write function templates we do not know what type a variable will have. That will be specified at compile time. As long as we associate a name with a functionality, which is declared else where, we can write fully general programs on a high abstraction level. Here is an example of how we can use function overloading already is our Point class

```
Class Point
Private:
double x; // x coordinate
double y; // y coordinate;
Public:
Point ( ) {x = y =0;}; // constructor.
void set (double vx =0) {x = v x;};
Void set (point p) {
x = P.x; // the methods of Point have access to private data
y = P.y; };
```

We use the function name set twice. If we provide an argument of type double, we set the x value, otherwise, we copy the storage of the provided Point into the data section.

In the line set (double vx = 0) { x = vx; }; we used another interesting feature of C++, the default argument: If we use a point p as p.set (7), the x value is set to 7, while the line p.set ( ) would set x to zero.

Scope Rules for Function Overloading

The ability to redefine the building blocks of the language can be a blessing in that it can make your listing more intuitive and readable. It can also have the opposite effect, making it more obscure and hard to understand. Here are few guidelines.

Using Similar Meanings

Use overloaded operators to perform operations that are as similar as possible to those performed on basic data types.

Use Similar Syntax

Use overloaded operators in the same way they are used for basic types. For example, if **alpha** and **beta** are basic types, the assignment operator in the statement,

counter alpha + = beta;

set alpha to the sum of alpha and beta. Any overloaded version of this operator should do something analogous. It should probably do the same thing as

counter alpha = alpha + beta;

where the + is overloaded.

Show Restraint

If you have overloaded the + operator, anyone unfamiliar with your listing will need to do considerable research to find out what a statement like

counter a = b + c;

really means. If the number of overloaded operators grows too large, and if they are used in nonintuitive ways, then the whole point of using them is lost, and the listing becomes less readable instead of more.

Avoid Ambiguity

If you use both a one-argument constructor and a conversion function to perform the same conversion, how will the compiler know which conversion to use? The answer is that it won't. The compiler does not like to be placed in a situation where it doesn't know what to do, and it will signal an error. Avoid doing the same conversion in more than one way.

Not all Operators can be Overloaded

The following operators cannot be overloaded, the member access or dot operator (.), the scope resolution operator (: :), and the conditional operator (? :). And, the pointer-to-member operator (.*), which we have not yet encountered, cannot be overloaded.

8.3 OPERATOR OVERLOADING

Overloading an operator simply means attaching additional meaning and semantics to an operator. It enables an operator to exhibit more than one operations polymorphically, as illustrated below:

You know that additional operator (+) is essentially a numeric operator and therefore, requires two number operands and evaluates to a numeric value equal to the sum of the two operands. Evidently this cannot be used in adding two strings. We can extend the operation of addition operator to include string concatenation. It implies that the additional operator would work as follows:

"COM" + "PUTER"

should produce a single string

"COMPUTER"

This redefining the effect of an operator is called operator overloading. The original meaning and action of the operator however remains as it is.

An operator is overloaded

We will consider overloading a unary operator - minus (-) to enable it to be applicable on a set of numbers instead of a single number.

Function overloading allows different functions with different argument list having the same name. Similarly an operator can be redefined to perform additional tasks. Operator overloading is accomplished using a special function which can be a member function or friend function. The general syntax of operator overloading is:

<return_type>operator<operator_being_overloaded>(<argument list>);

operator is the keyword and is preceded by the return_type of the function.

To overload the addition operator (+) to concatenate two characters, the following declaration, which could be either member or friend function, would be needed:

char * operator + (char *s2);

Rules for Overloading Operators

To overload any operator, we need to understand the rules applicable Let us revise some of them which have already been explored...

Following are the operators that cannot be overloaded.

Operator	Purpose
.	Class member access operator
.*	Class member access operator
::	Scope Resolution Operator
?:	Conditional Operator
Size of	Size in bytes operator
#	Preprocessor Directive

Table 8.1: Operators that cannot be overloaded

Operator	Purpose
=	Assignment operator
0	Function call operator
0	Subscripting operator
->	Class member access operator

- Operators already predefined in the C++ compiler can be only overloaded. Operator cannot change operator templates that is for example the increment operator ++ is used only as unary operator. It cannot be used as binary operator.
- Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

  class integer
  ```
  {intx, y;
    public:
    int operator + ();
  }
    int integer: : operator + ( )
  {
    return (x-y);
  }
  ```
- Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).
- Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

Operator to Overload	Arguments passed to the Member Function	Argument passed to the Friend Function
Unary Operator	No	1
Binary Operator	1	2

8.4 OVERLOADING OF BINARY OPERATORS

Binary operators are operators which require two operands to perform the operation. When they are overloaded by means of member function, the function takes one argument, whereas it takes two arguments in case of friend function. This will be better understood by means of the following program.

The following program creates two objects of class integer and overloads the + operator to add two object values.

```
#include<iostream.h>
#include<como.h>
class integer
{
private :
int val;
public:
integer ();
integer(int one );
integer operator+ (integer objb);
void disp ();
};
integer :: integer ()
{
    val = 0;
}
integer:: integer (int one)
{
    val = one;
}
integer integer:: operator+ (integer objb)
{
    integer objsum;
    objsum.val = val+objb. val;
    return (objsum);
}
```

```
void integer:: disp ( )
{
    cout<< "value ="<< val<< endl;
}
void main ()
{
    integer obj1 (11);
    integer obj2 (22);
    integer objsum;
    objsum = objl +obj2;
    obj1.disp ();
    obj2.disp ( );
    objsum.disp ( );
    getch ( );
}
```

You should see the following output.

```
value = 11
value = 22
value = 33
```

Note that the operator overloading function is invoked by S3=S1+S2. We have passed only one argument to the function. The left hand object S1 invokes the function and S2 is actually the argument passed to the function.

The following program is the same as previous one. The only difference is that we use a friend function to find the sum of two objects. Note that an argument is passed to this friend function.

```
#include<iostream.h>
#include<conio.h>
class integer
{
private:
int val;
public:
integer ();
integer (in tone );
friend integer operator+ (integer obja,integer objb);
void disp ( );
};
integer:: integer ()
```

```
{
    val = 0;
}
integer:: integer (int one)
{
    val = one;
}
integer. operator+ (integer obja,integer objb)
{
    integer objsum;
    objsum.val = obja.val+objb.val;
    return (objsum);
}
Void integer :: disp ( )
{
    cout<< "value ="<< val <<endl;
}
void main ( )
{
    integer obj 1 (11);
    integer obj2 (22);
    integer objsum;
    objsum = obj 1 +obj2;
    obj1.disp ();
    obj2.disp ( );
    objsum.disp ( );
    getch ( );
}
```

You should see the following output.

```
value = 11
value = 22
value = 33
```

friend function being a non-member function does not belong to any class. This function is invoked like a normal function. Hence the two objects that are to be added have to be passed as arguments exclusively.

8.5 OVERLOADING OF UNARY OPERATORS

In case of unary operator overloaded using a member function no argument is passed to the function whereas in case of a friend function a single argument must be passed.

Following program overloads the unary operator to negate an object. The operator function defined outside the class negates the individual data members of the class integer.

```
#include<iostream.h>
#include<conio.h>
class integer
{
    int x,y,z;
    public:
    void getdata(int a, int b, int c);
    void disp(void);
    void operator- ();   // overload unary operator minus
};
void integer:: getdata (int a, int b, int c)
{
    x=a;
    y=b;
    z=c;
}
void integer::disp (void)
{
    cout << x << " ";
    cout<< y<<" ";
    cout<< z<< "\n";
}
void integer:: operator- ()   // Defining operator- ()
{
    x = -x;
    y =-y;
    z = -z;
}
main 0
{
    integer S;
    S.getdata(11,-21,-31);
    Cout<< "S: ";
    S.disp ( );
    -S;
    cout<< "s : ";
    S.disp ( );
    getch( );
}
```

You should see the following output.

```
S: 11   -21    -31
S:-11   21     31
```

Note the function written to overload the operator is a member function. Hence no argument is passed to the function. In the main() function, the statement -S invokes the operator function which negates individual data elements of the object S.

The same program can be rewritten using friend function. This is demonstrated in the following Program. In this program, we define operator function to perform unary subtraction using a friend function.

```
#incluae <iostream.h>
#include <conio.h>
class integer
{
    intx;
    int y;
    intz;
    public:
    void getdata(int a, int b, int c);
    void disp(void);
    friend void operator- (integer &s );   // overload unary minus
};
void integer::getdata (int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
void integer::disp (void)
{
    cout << x << " ";
    cout << y <<" ";
    cout << z << "\n";
}
void operator- (integer &s )          // Defining operator- ( )
{
    s.x = -S.x;
    s.y = -s.y;
    s.z = -S.z;
}
main 0
{
    integer S;
    S.getdata(11 ,-21 ,-31);
    Cout<< "S: ";
    S.disp ( );
    -S;                           //activates operator-( )
    cout<< "S : ";
    S.dis ( );
    getch ( );
}
```

You should see the following output.

```
S: 11 -21 -31
S : -11 21 31
```

Note how only one argument is passed to the friend function. The operator function declared as friend is not the property of the class. Hence when we define this friend function, we should pass the object of the class on which it operates.

Check Your Progress

Fill in the blanks:

1. Overloading an operator simply means attaching additional meaning and .... to an operator.
2. There are no operators for manipulating the ....
3. Binary operators are operators which require .... operands to perform the operation.
4. In case of unary operator overloaded using a member function no .... is passed to the function.

8.6 LET US SUM UP

In this lesson, we have seen how the normal C++ operators can be given new meanings when applied to user-defined data types. The keyword operator is used to overload an operator, and the resulting operator will adopt the meaning supplied by the programmer.

Closely related to operator overloading is the issue of type conversion. Some conversions take place between user defined types and basic types. Two approaches are used in such conversion: a one argument constructor changes a basic type to a user defined type, and a conversion operator converts a user-defined type to a basic type. When one user-defined type is converted to another, either approach can be used.

8.7 KEYWORD

Operator overloading: Attaching additional meaning and semantics to an operator. It enables to exhibit more than one operations polymorphically.

8.8 QUESTIONS FOR DISCUSSION

1. What is operator overloading?
2. How many arguments are required in the definition of an overloaded unary operator?
3. When used in prefix form, what does the overloaded + + operator do differently from what it does in postfix form?
4. Write the complete definition of an overloaded + + operator that works with the string class from the STRPLUS example and has the effect of changing its operand to uppercase. You can use the library function toupper ( ), which takes as its only argument the character to be changed, and returns the changed character.
5. Write a note on unary operators.
6. What are the various rules for overloading operators?

Check Your Progress: Model Answer

1. Semantics
2. Strings
3. Two
4. argument

8.9 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo* C++, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with* C++, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference* C++, Tata Mc Graw Hill

UNIT V

LESSON

9

POLYMORPHISM

CONTENTS

9.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of polymorphism
- Discuss polymorphism with pointers
- Describe the significance of virtual functions
- Identify and explain the late binding abstract classes
- Discuss the constructors and destructors under inheritance
- Explain the concept of virtual destructors
- Discuss the virtual base classes

9.1 INTRODUCTION

Polymorphism is an Anglicization of two Greek words - 'poly' means many, and 'moop' meaning shape. Polymorphism allows the program to use the exactly same function name with the exactly same argument in both a base class and its subclasses. This allows having a function that behaves in a different way depending upon the object class.

Polymorphism may be the most powerful aspect of object-oriented programming because the program can be written to call member functions without regard for what class the object belongs to. This can be done by defining the sub-class.

9.2 POLYMORPHISM

Polymorphism refers to the implicit ability of a function to have different meanings in different contexts. Consider the class hierarchy that contains Number and ComplexNumber. If a number is defined as a pointer to Number then a Number can be instantiated as either a Number or a ComplexNumber.

```
Eg:   Number *aNumber;
    aNumber = new Number (1);
    aNumber → output ( );
    delete  aNumber;
    aNumber → output ( );
    Delete aNumber;
```

In the first case, the output function in Number would be called and in the second case, it would be called again. This happens because a Number is a pointer to a Number and not a ComplexNumber. To solve this problem, both output functions must be declared as virtual. Then the compiler keeps track of which the actual functions associated with each object and calls the appropriate ones when needed.

9.3 POLYMORPHISM WITH POINTERS

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential.

We are going to start by rewriting our program about the rectangle and the triangle of the previous section taking into consideration this pointer compatibility property:

```
pointers to base class
#include <iostream>
using namespace std;
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
  };
class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

In function `main`, we create two pointers that point to objects of class `CPolygon` (`ppoly1` and `ppoly2`). Then we assign references to `rect` and `trgl` to these pointers, and because both are objects of classes derived from `CPolygon`, both are valid assignment operations.

The only limitation in using `*ppoly1` and `*ppoly2` instead of `rect` and `trgl` is that both `*ppoly1` and `*ppoly2` are of type `CPolygon*` and therefore we can only use these pointers to refer to the members that `CRectangle` and `CTriangle` inherit from `CPolygon`. For that reason when we call the `area()` members at the end of the program we have had to use directly the objects `rect` and `trgl` instead of the pointers `*ppoly1` and `*ppoly2`.

In order to use `area()` with the pointers to class `CPolygon`, this member should also have been declared in the class `CPolygon`, and not only in its derived classes, but the problem is that `CRectangle` and `CTriangle` implement different versions of `area`, therefore we cannot implement it in the base class.

9.4 VIRTUAL FUNCTIONS

Virtual functions, one of advanced features of OOP is one that does not really exist but it appears real in some parts of a program. This section deals with the polymorphic features which are incorporated using the virtual functions.

The general syntax of the virtual function declaration is:

```
class use_defined_name{
private:
public:
virtual return_type function_name1(arguments);
virtual return_type function_name2(arguments);
virtual return_type function_name3(arguments);
--------------
--------------
};
```

To make a member function virtual, the keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual precedes the return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

For example, the following declaration of the virtual function is valid.

```
class point {
intx;
inty;
public:
virtual int length ( );
virtual void display ( );
};
```

Remember that the keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.

For example

```
class point {
intx;
inty;
public:
virtual void display ();
};
virtual void point: : display () //error
{
```