CHAPTER 7

POLYMORPHISM

7.1 Introduction

The word polymorphism means many forms (poly – many, morph – shapes). In C++, polymorphism is achieved through function overloading and operator overloading. The term overloading means a name having two or more distinct meanings. Thus an '**overloaded function'** refers to a function having more than one distinct meaning. Function overloading is one of the facets of C++ that supports object oriented programming.

7.2 Function overloading

> The ability of the function to process the message or data in more than one form is called as function overloading.

Consider the situation wherein a programmer desires to have the following functions

```
area_circle()       // to calculate the area of a circle
area_triangle()     // to calculate the area of a triangle
area_rectangle()  // to calculate the area of a rectangle
```

The above three different prototype to compute area, for different shapes can be rewritten using a single function header in the following manner

```
float area ( float radius);
float area ( float half, float base, float height );
float area ( float length , float breadth);
```

Now look at the ease in invoking the function area(…) for any of the three shapes as shown in Program - 7.1

```
// Program - 7.1
// to demonstrate the polymorphism - function overloading

#include<iostream.h>
#include<conio.h>

float area ( float radius )
{       cout << "\nCircle …";
      return ( 22/7 * radius * radius );
}
float area (float half, float base, float height)
{       cout << "\nTriangle ..";
      return (half* base*height);
}


float area ( float length, float breadth )
{        cout << "\nRectangle …";
      return ( length *breadth ) ;
}

void main()
{
      clrscr();
      float r,b,h;
      int choice = 0 ;
      do
      {
            clrscr();
            cout << "\n Area Menu ";
            cout << "\n 1. Circle ... ";
            cout << "\n 2. Traingle ...";
            cout << "\n 3. Rectangle ... ";
            cout <<"\n 4. Exit ... ";
            cin>> choice;
            switch(choice)
            {
                  case 1 :
                        cout << "\n Enter radius ... ";
                        cin>>r;
                        cout<<"\n The area of circle is ... "
                           << area(r);
                        getch();
                        break;
                  case 2:
                        cout<< "\n Enter base, height ... ";
                        cin>>b>>h;
                        cout<<"\n The area of a triangle is .. "
                             << area (0.5, b, h);
                        getch();
                        break;
                  case 3:
                       cout<< "\n Enter length, breadth.. ";
                        cin>>h>>b;
                        cout<<"\n The area of a rectangle is ... "
                             << area(h,b);
                       getch();
                       break;
             }
            }while (choice <=3);
}
```

Have you noticed the function prototypes for all the 3 functions? The prototypes are:

```
float area ( float radius );
float area (float half, float base, float height)
float area ( float length, float breadth );
```

How do you think each function 'area' definition is differing from one and another ? Yes, each function prototype differs by their number of arguments. The first prototype had one argument, second one 3 arguments and the third one had 2 arguments. In the example we have dealt , all the three functions has float type arguments. It need not necessarily be this way. Arguments for each prototype can be of different data type . Secondly the number of arguments for each function prototype may also differ. The following prototypes for function overloading is invalid. Can you tell why is it so ?

Function Prototype	Invalid prototype

void fun(int x);
void fun(char ch);
void fun(int y);
void fun(double d);
	void fun(**int x**);
	void fun(**int y**);
	
	Both the prototypes have same number and type of arguments. Hence it is invalid.

How are functions invoked in function overloading?

The compiler adopts BEST MATCH strategy. As per this strategy, the compiler will

- ✓ Look for the exact match of a function prototype with that of a function call statement

- ✓ In case an exact match is not available, it looks for the next nearest match. That is, the compiler will promote integral data promotions and then match the call statement with function prototype.

For example, in the above example (program –1) we have **float area(float radius)** with area(r) where the parameter **'r'** should be of float type. In case, the variable 'r' is of integer type, then as per integral promotions integer constant/variable can be mapped to char, or float or double. So, by this strategy the area(r) will be mapped to area(float radius).

Integral promotions are purely compiler oriented. By and large integral promotions are as follows:

- ✓ char data type can be converted to integer/float/double
- ✓ int data type can be converted to char/double/float
- ✓ float data type to integer/double/char
- ✓ double data type to float or integer

Now based on the following call statements to area() of Program – 7.1 can you tell as to what will be the output?

Function call statement	Output displayed
area(5.0)	
area(0.5,4.0,6.0)	
area(3.0,4.5)	

Rules for function overloading

1) Each overloaded function must differ either by the number of its formal parameters or their data types
2) The return type of overloaded functions may or may not be the same data type
3) The default arguments of overloaded functions are not considered by the C++ compiler as part of the parameter list
4) Do not use the same function name for two unrelated functions

Improper declarations leading to conflict in a function call statement is shown below.

```
void fun ( char a, int times)
{
      for (int i=1; i<=times;i++)
                  cout<<a;
}
void fun( char a= '*', int times )
{
      for(int i=1;i<=times;i++)
                  cout<<a;
}
void fun( int times)
{
      for(int i=1; i<=times ;i++)
                  cout<<'@';
}
void main()
{
      fun ( '+', 60);
      fun(60);
}
```

When the above program is compiled, two errors will be flagged:

- ✓ Conflict between fun(char a, int times) and fun(char a='*', int times)

- ✓ Conflict between fun(char a='*', int times) and fun (int times)

The call statement fun('+', 60) can be matched with fun (char a, int times) and fun (char a='*', int times)

The call statement fun(60) can be matched with fun (char a='*', int times) and fun (int times)

Overload a function with the help of different function definitions having a unique parameter list. That is, the parameter list differ either by number or types.

7.3 Operator Overloading

The term operator overloading, refers to giving additional functionality to the normal C++ operators like +,++,-,—,+=,-=,*.<,>. The statement sum = num1 + num2 would be interpreted as a statement meant to perform addition of numbers(integer/float/double) and store the result in the variable sum. Now look at the following statement:

name = first_name + last_name;

where the variables name, first_name and last_name are all character arrays.Can one achieve concatenation of character arrays using '+' operator in C++? The compiler would throw an error stating that '+' operator cannot handle concatenation of strings. The user is forced to use **strcat()** function to concatenate strings. Won't it be a lot easier if one is permitted to use '+' operator on strings as used for number data type? The functionality of '+' operator can be extended to strings through **operator overloading**.

Look at the following example:

```
// Program -7.2 – OPERATOR OVERLOADING
# include <iostream.h>
# include <conio.h>
# include <string.h>

class strings
{
   char s[10];
   public :
   strings()
    {
          s[0] = '\0';
    }

   strings(char *c)
    {
         strcpy(s,c);
    }

   char * operator+(strings x1)
    {
         char *temp;
         strcpy(temp,s);
         strcat(temp,x1.s);
         return temp;
    }
};
```

```
void main()
{
clrscr();
strings s1("test"),s2(" run\0");
char *concatstr ;
concatstr = s1 + s2;
cout << "\nConcatenated string ..."
     << concatstr;
getch();
}
```

The statement **concatstr = s1 + s2** merges two strings, as the operator '+' is given additional function through the member function:

char * operator + (strings x1)

The member function **char * operator + (strings x1)** takes x1 as the argument. It may be viewed as:

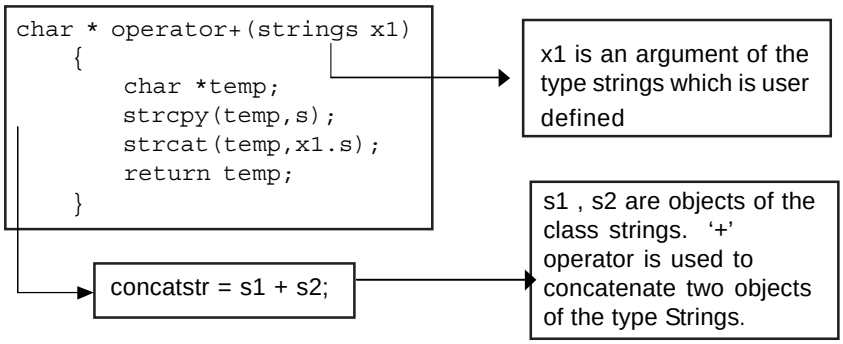

Fig. 7.1 demonstrates the association of variables and their values.

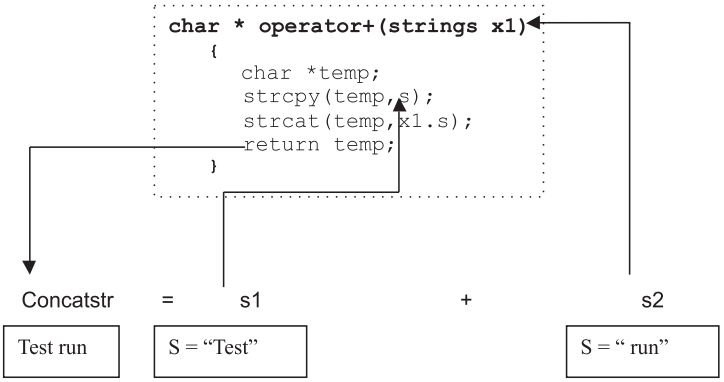

Fig. 7.1 Association of Variables and Values

Operator overloading provides:

- ✓ New function definitions for basic C++ operators like +, *, -, ++, - -, >, <, += and the like. One cannot overload C++ specific operators like membership operator (.), scope resolution operator (::), sizeof operator and conditional operator.

- ✓ The overloaded function definitions are permitted for user defined data type.

- ✓ Operator functions must be either member functions or friend functions. (Friend functions is beyond the scope of this book)

- ✓ The new definition that is provided to an operator does not overrule the original definition of the operator. For example, in the above program – OPERATOR OVERLOADING the '+' operator has been used to merge two strings. In the same program one can also perform addition of numbers in the usual way. The compiler applies user defined definition based on the style of call statement. That is the statements cout << 5 + 10 will display the result as 15 (original definition

of '+' is applied), where as concatstr = s1 + s2 will invoke the member function **char * operator + (strings s1)** as the operands provided for the '+' operator are s1 and s2 which are the objects of the class strings.

The process of overloading involves:

- ✓ Create a class that defines the data type that is to be used in the overloading operations
- ✓ Declare the operator function **operator ()** in the public part of the class.
- ✓ Define the operator function to implement the required operations.

The following examples demonstrate the ease of using operators with user defined data types – objects.

Program – 7.3 demonstrates as to how one can negate the data members of a class using the operator – (minus)

```
// Program - 7.3
# include <iostream.h>
# include <conio.h>

class negative
{
   int i;
   public :
   void accept()
   {
       cout << "\nEnter a number ...";
         cin >> i;
   }
   void display()
   {
         cout << "\nNumber ..."<<i;
   }
```

```
  void operator-()
   {
        i = -i;
   }
};

void main()
{
      clrscr();
      negative n1,n2;
      n2.accept();
      -n2;
      n2.display();
      getch();
}
```

The function void **operator –()** simply negates the data members of the class as one would do with a normal variable as follows: sum = **-** num1;

Look at the following program and answer the questions:

```
// Program – 7.4
# include <iostream.h>
# include <conio.h>

class distance
{
   int feet,inches;
   public :
   void distance_assign(int f, int i)
   {
        feet = f;
        inches = i;
   }

   void display()
   {
       cout << "\nFeet   : " << feet
              << "\tInches : " << inches;
   }

   distance operator+(distance d2)
   {
        distance d3;
        d3.feet = feet + d2.feet;
        d3.inches = (inches + d2.inches) % 12;
        d3.feet += (inches + d2.inches)/12;
        return d3;
   }
};

void main()
{
    clrscr();
    distance dist_1,dist_2;
    dist_1.distance_assign(12,11)
    dist_2.distance_assign(24,1);
    distance dist_3 = dist_1 + dist_2;
    dist_1.display();
    dist_2.display();
    dist_3.display();
    getch();
}
```

1. Identify the operator that is overloaded.
2. Write out the prototype of the overloaded member function.
3. What types of operands are used for the overloaded operator?
4. Write out the statement that invokes the overloaded member function.

Program-7.5 demonstrates the overloaded functions of += and -=

```
//Program-7.5
//  operator  overloading

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

class  library_book
{
   char  name[25];
   int  code,stock;

   public  :

   void  book_assign(char  n[15],int  c,int  s)
   {
         strcpy(name,n);
         code  =  c;
         stock  =  s;
   }

   void  display()
   {
          cout  <<  "\n   Book  details  ....";
          cout  <<  "\n   1.  Name        ....."  <<  name;
          cout  <<  "\n   2.  Code        ....."  <<  code;
          cout  <<  "\n   3.  Stock       ....."  <<  stock;
   }

   void  operator  +=(int  x)
   {
        stock  +=  x;
   }

   void  operator  -=(int  x)
   {
         stock  -=  x;
   }
};
```

```
class library_cdrom
{
 char name[25];
 int code,stock;

 public :

 void cdrom_assign(char n[15],int c,int s)
 {
            strcpy(name,n);
            code = c;
            stock = s;
 }

 void display()
 {
             cout << "\n  CD ROM details ....";
             cout << "\n  1. Name     ....." << name;
             cout << "\n  2. Code     ....." << code;
             cout << "\n  3. Stock    ....." << stock;
 }

 void operator +=(int x)
 {
           stock += x;
 }

 void operator -=(int x)
 {
            stock -= x;
 }
};


void main()
{
  library_book book;
  library_cdrom cdrom;

  book.book_assign("Half Blood Prince",101,55);
  cdrom.cdrom_assign("Know your Basics",201,50);

   char choice,borrow;

do
  {
            cout << "\nBook,cdrom,exit<b/c/e> ...";
              cin >> choice;
              if (choice != 'e')
                {
                    cout << "\nBorrow/Return <b/r> ...";
                    cin >> borrow;
                }
```

```
switch (choice)
        {
  case 'b' :
        switch (borrow)
        {
          case 'b' : book += 1;break;
          case 'r' : book -= 1;break;
        }
        book.display();
        break;
  case 'c' :
        switch (borrow)
        {
          case 'b' : cdrom += 1;break;
          case 'r' : cdrom -= 1;break;
        }
        cdrom.display();
        break;

  case 'e' : cout << "\nTerminating ..";
                    break;
 }
 } while (choice != 'e');
 getch();
}
```

Have you noticed the ease with which the objects stock is incremented or decremented in a standard style by using the operators **+= / -=**

```
book   += 1;
book   -= 1;
cdrom  += 1;
cdrom   -= 1;
```

The mechanism of giving **special meaning to an operator** is called as **operator overloading.**

Rules for overloading operators:

There are certain restrictions and limitations in overloading operators**.** They are:

- ✓ Only existing operators can be overloaded. New operators cannot be created.
- ✓ The overloaded operator must have at least one operand of user defined type.
- ✓ The basic definition of an operator cannot be replaced or in other words one cannot redefine the function of an operator. One can give additional functions to an operator
- ✓ Overloaded operators behave in the same way as the basic operators in terms of their operands.
- ✓ When binary operators are overloaded, the left hand object must be an object of the relevant class
- ✓ Binary operators overloaded through a member function take one explicit argument.

Exercises

I. Write a program that uses function overloading to do the following tasks

   a. find the maximum of two numbers (integers)

   b. find the maximum of three numbers (integers)

   SOLUTION: function protype – max (int , int) and max(int , int, int)

II. Write function definitions using function overloading to

   a. increment the value of a variable of type float

   b. increment the value of a variable of type char

   SOLUTION: function prototype – float incr (float) , char incr (char)

III. Write a program in C++ to do the following tasks using function overloading

   a. compute x^y where x and y are both integers

   b. compute x^y where x and y are both float

   SOLUTION: function prototype – int power(int,int),float power(float,float);

IV. What is the advantage of operator overloading?

V. List out the steps involved to define an overloaded operator.

VI. List out the operators that cannot be overloaded.

VII. Write a program to add two objects of the class **complex_numbers .** A complex number has two data members – real part and imaginary part. Complete the following definition and also write a main() function to perform addition of the complex_numbers objects c1 and c2 .

```
Class complex_numbers
{
   float x;
   float y;
   public :
   void assign_data(float real, float imaginary);
   void display_data();
   complex_numbers operator +(complex_numbers n1);
}
```