Program 6.2 shows the use of overloaded constructors.

OVERLOADED CONSTRUCTORS

```
#include <iostream>

using namespace std;

class complex
{
      float x, y;
  public:
      complex(){ }                    // constructor no arg
      complex(float a) {x = y = a;}   // constructor-one arg
      complex(float real, float imag) // constructor-two args
      {x = real; y = imag;}

      friend complex sum(complex, complex);
      friend void show(complex);
};
complex sum(complex c1, complex c2)   // friend
{
      complex c3;
      c3.x = c1.x + c2.x;
      c3.y = c1.y + c2.y;
      return(c3);
}
void show(complex c)          // friend
{
      cout << c.x << " + j" << c.y << "\n";
}

int main()
{
      complex A(2.7, 3.5);            // define & initialize
      complex B(1.6);                 // define & initialize
      complex C;                      // define

      C = sum(A, B);                  // sum() is a friend
      cout << "A = "; show(A);        // show() is also friend
      cout << "B = "; show(B);
      cout << "C = "; show(C);

// Another way to give initial values (second method)

      complex P,Q,R;                  // define P, Q and R
```

(Contd)

```
        P = complex(2.5,3.9);              // initialize P
        Q = complex(1.6,2.5);              // initialize Q
        R = sum(P,Q);

        cout << "\n";
        cout << "P = "; show(P);
        cout << "Q = "; show(Q);
        cout << "R = "; show(R);

        return 0;
}
```

PROGRAM 6.2

The output of Program 6.2 would be:

```
A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1

P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j6.4
```

note

There are three constructors in the class **complex**. The first constructor, which takes no arguments, is used to create objects which are not initialized; the second, which takes one argument, is used to create objects and initialize them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

Let us look at the first constructor again.

```
complex(){ }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now?. As pointed out earlier, C++ compiler has an *implicit constructor* which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

6.5 Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor complex() can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument **imag** is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the **real** variable and 0.0 to **imag** (by default). However, the statement

```
complex C(2.0,3.0);
```

assigns 2.0 to **real** and 3.0 to **imag**. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor **A::A()** and the default argument constructor **A::A(int = 0)**. The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A a;
```

The ambiguity is whether to 'call' **A::A() or A::A(int = 0)**.

6.6 Dynamic Initialization of Objects

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Program 6.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

DYNAMIC INITIALIZATION OF CONSTRUCTORS

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
class Fixed_deposit
{
        long int P_amount;      // Principal amount
        int      Years;         // Period of investment
        float    Rate;          // Interest rate
        float    R_value;       // Return value of amount
  public:
        Fixed_deposit(){ }
        Fixed_deposit(long int p, int y, float r=0.12);
        Fixed_deposit(long int p, int y, int r);
        void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for(int i = 1; i <= y; i++)
            R_value = R_value * (1.0 + r);
}

Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;

        for(int i=1; i<=y; i++)
            R_value = R_value*(1.0+float(r)/100);
}

void Fixed_deposit :: display(void)
{
        cout << "\n"
            << "Principal Amount = " << P_amount << "\n"
            << "Return Value     = " << R_value  << "\n";
}
```

(Contd)

```
int main()
{
        Fixed_deposit FD1, FD2, FD3;  // deposits created

        long int p;                   // principal amount

  int     y;                    // investment period, years
  float   r;                    // interest rate, decimal form
  int     R;                    // interest rate, percent form

  cout << "Enter amount,period,interest rate(in percent)"<<"\n";
  cin >> p >> y >> R;
  FD1 = Fixed_deposit(p,y,R);

  cout << "Enter amount,period,interest rate(decimal form)" << "\n";
  cin >> p >> y >> r;
  FD2 = Fixed_deposit(p,y,r);

  cout << "Enter amount and period" << "\n";
  cin >> p >> y;
  FD3 = Fixed_deposit(p,y);

  cout << "\nDeposit 1";
  FD1.display();

  cout << "\nDeposit 2";
  FD2.display();

  cout << "\nDeposit 3";
  FD3.display();

  return 0;
}
```

PROGRAM 6.3

The output of Program 6.3 would be:

```
Enter amount,period,interest rate(in percent)
10000 3 18
Enter amount,period,interest rate(in decimal form)
10000 3 0.18
Enter amount and period
10000 3

Deposit 1
Principal Amount = 10000
Return Value     = 16430.3
```

```
Deposit 2
Principal Amount = 10000
Return Value     = 16430.3

Deposit 3
Principal Amount = 10000
Return Value     = 14049.3
```

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

note

Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms (1) and (3), and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for **r**.

6.7 Copy Constructor

We briefly mentioned about the copy constructor in Sec. 6.3. We used the copy constructor

```
integer(integer &i);
```

in Sec. 6.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

```
integer I2 = I1;
```

The process of initializing through a copy constructor is known as *copy initialization*. Remember, the statement

```
I2 = I1;
```

will not invoke the copy constructor. However, if **I1** and **I2** are objects, this statement is legal and simply assigns the values of **I1** to **I2**, member-by-member. This is the task of the overloaded assignment operator(=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in Program 6.4.

COPY CONSTRUCTOR

```
#include <iostream>

using namespace std;

class code
{
     int id;
  public:
     code(){ }                    // constructor
     code(int a) { id = a;}       // constructor again
     code(code & x)               // copy constructor

     {
            id = x.id;            // copy in the value
     }
     void display(void)
     {
            cout << id;
     }
};
int main()
{
     code A(100);  // object A is created and initialized
     code B(A);    // copy constructor called
     code C = A;   // copy constructor called again

     code D; // D is created, not initialized
     D = A;        // copy constructor not called

     cout << "\n id of A: "; A.display();
     cout << "\n id of B: "; B.display();
     cout << "\n id of C: "; C.display();
     cout << "\n id of D: "; D.display();

     return 0;
}
```

PROGRAM 6.4

The output of Program 6.4 is shown below

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

note

A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.

When no copy constructor is defined, the compiler supplies its own copy constructor.

6.8 Dynamic Constructors

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator. Program 6.5 shows the use of new, in constructors that are used to construct strings in objects.

CONSTRUCTORS WITH new

```
#include <iostream>
#include <string>

using namespace std;

class String
{
      char *name;
      int  length;
  public:
      String()          // constructor-1
      {
            length = 0;
            name = new char[length + 1];
      }

      String(char *s)   // constructor-2
      {
            length = strlen(s);
```

(Contd)

```
                name = new char[length + 1];      // one additional
                                                  // character for \0
                strcpy(name, s);
                }

                void display(void)
                {cout << name << "\n";}
                void join(String &a, String &b);
        };

        void String :: join(String &a, String &b)
        {
                length = a.length + b.length;
                delete name;
                name = new char[length+1];             // dynamic allocation

                strcpy(name, a.name);
                strcat(name, b.name);
        };

        int main()
        {
                char *first = "Joseph ";
                String name1(first), name2("Louis "),name3("Lagrange"),s1,s2;

                s1.join(name1, name2);
                s2.join(s1, name3);
                name1.display();
                name2.display();
                name3.display();
                s1.display();
                s2.display();

                return 0;
        }
```

PROGRAM 6.5

The output of Program 6.5 would be:

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

note

This Program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the **length** of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character '\0'.

The member function **join( )** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions **strcpy( )** and **strcat( )**. Note that in the function **join( )**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a**. The **main( )** function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```

6.9 Constructing Two-dimensional Arrays

We can construct matrix variables using the class type objects. The example in Program 6.6 illustrates how to construct a matrix of size m x n.

CONSTRUCTING MATRIX OBJECTS

```
#include <iostream>

using namespace std;

class matrix
{
        int **p; // pointer to matrix
        int d1,d2;      // dimensions
   public:
        matrix(int x, int y);
        void get_element(int i, int j, int value)
        {p[i][j]=value;}
        int & put_element(int i, int j)
        {return p[i][j];}
};
matrix :: matrix(int x, int y)
{
        d1 = x;
        d2 = y;
        p = new int *[d1];              // creates an array pointer
        for(int i = 0; i < d1; i++)
```

(Contd)

```
            p[i] = new int[d2]; // creates space for each row
}

int main()
{
        int m, n;

        cout << "Enter size of matrix: ";
        cin  >> m >> n;
   matrix A(m,n); // matrix object A constructed

   cout << "Enter matrix elements row by row \n";
   int i, j, value;

   for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
        {
               cin >> value;
               A.get_element(i,j,value);
        }
   cout << "\n";
   cout << A.put_element(1,2);

   return 0;
};
```

PROGRAM 6.6

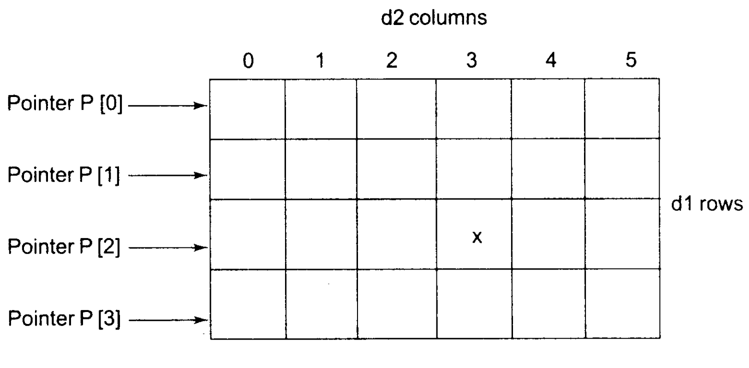
The output of a sample run of Program 6.6 is as follows.

```
Enter size of matrix: 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22

17
```

17 is the value of the element (1,2).

The constructor first creates a vector pointer to an **int** of size **d1**. Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i]**. Thus, space for the elements of a d1 × d2 matrix is allocated from free store as shown above.



x represents the element P[2] [3]

6.10 const Objects

We may create and use constant objects using **const** keyword before object declaration. For example, we may create X as a constant object of the class **matrix** as follows:

```
const matrix X(m,n); // object X is constant
```

Any attempt to modify the values of **m** and **n** will generate compile-time error. Further, a constant object can call only **const** member functions. As we know, a **const** member is a function prototype or function definition where the keyword const appears after the function's signature.

Whenever **const** objects try to invoke non-**const** member functions, the compiler generates errors.

6.11 Destructors

A *destructor*, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer(){ }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory. For example, the destructor for the **matrix** class discussed above may be defined as follows:

```
matrix :: ~matrix()
{
    for(int i=0; i<dl; i++)
    delete p[i];
    delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been invoked implicitly by the compiler.

IMPLEMENTATION OF DESTRUCTORS

```
#include <iostream>

using namespace std;

int count = 0;

class alpha
{
  public:
    alpha()
    {
        count++;
        cout << "\nNo.of object created " << count;
    }

    ~alpha()
    {
        cout << "\nNo.of object destroyed " << count;
        count--;
    }
};

int main()
{
    cout << "\n\nENTER MAIN\n";

    alpha A1, A2, A3, A4;
    {
        cout << "\n\nENTER BLOCK1\n";
        alpha A5;
    }

    {
        cout << "\n\nENTER BLOCK2\n";
        alpha  A6;
    }
    cout << "\n\nRE-ENTER MAIN\n";

    return 0;
}
```

PROGRAM 6.7

The output of a sample run of Program 6.7 is shown below:

```
ENTER MAIN

No.of object created 1
No.of object created 2
No.of object created 3
No.of object created 4

ENTER BLOCK1

No.of object created 5
No.of object destroyed 5

ENTER BLOCK2

No.of object created 5
No.of object destroyed 5

RE-ENTER MAIN

No.of object destroyed 4
No.of object destroyed 3
No.of object destroyed 2
No.of object destroyed 1
```

note

As the objects are created and destroyed, they increase and decrease the count. Notice that after the first group of objects is created, **A5** is created, and then destroyed, **A6** is created, and then destroyed. Finally, the rest of the objects are also destroyed. When the closing brace of a scope is encountered, the destructors for each object in the scope are called. Note that the objects are destroyed in the reverse order of creation.

SUMMARY

- C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects.
- A constructor has the same name as that of a class.
- Constructors are normally used to initialize variables and to allocate memory.
- Similar to normal functions, constructors may be overloaded.

⇔ When an object is created and initialized at the same time, a copy constructor gets called.

⇔ We may make an object **const** if it does not modify any of its data values.

⇔ C++ also provides another member function called the destructor that destroys the objects when they are no longer required.

Key Terms

- automatic initialization
- **Const**
- Constructor
- constructor overloading
- copy constructor
- copy initialization
- default argument
- default constructor
- **Delete**
- Destructor
- dynamic construction
- dynamic initialization
- explicit call
- implicit call
- implicit constructor
- initialization
- **new**
- parameterized constructor
- reference
- shorthand method
- **strcat()**
- **strcpy()**
- **strlen()**
- **virtual**

Review Questions

6.1 *What is a constructor? Is it mandatory to use constructors in a class?*

6.2 *How do we invoke a constructor function?*

6.3 *List some of the special properties of the constructor functions.*

6.4 *What is a parameterized constructor?*

6.5 *Can we have more than one constructors in a class? If yes, explain the need for such a situation.*

6.6 *What do you mean by dynamic initialization of objects? Why do we need to do this?*

6.7 *How is dynamic initialization of objects achieved?*

6.8 *Distinguish between the following two statements:*

```
time T2(T1);
time T2 = T1;
```

T1 and T2 are objects of **time** class.

6.9 *Describe the importance of destructors.*

6.10 *State whether the following statements are TRUE or FALSE.*

(a) *Constructors, like other member functions, can be declared anywhere in the class.*

(b) *Constructors do not return any values.*

(c) *A constructor that accepts no parameter is known as the default constructor.*

(d) *A class should have at least one constructor.*

(e) *Destructors never take any argument.*

Debugging Exercises

6.1 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
      int length;
      int width;
public:
      Room(int l, int w=0):
           width(w),
           length(l)
      {
      }
};
void main()
{
      Room objRoom1;
      Room objRoom2(12, 8);
}
```

6.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
      int length;
      int width;
public:
		Room()
		{
			length = 0;
			width = 0;
		}
		Room(int value=8)
		{
			length = width = 8;
		}
		void display()
		{
			cout << length << ' ' << width;
		}
};

void main()
{
		Room objRoom1;
		objRoom1.display();
}
```

6.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
		int width;
		int height;
		static int copyConsCount;
public:
		void Room()
		{
			width = 12;
			height = 8;
		}

		Room(Room& r)
		{
			width = r.width;
			height = r.height;
```

```
                copyConsCount++;
        }

        void dispCopyConsCount()
        {
                cout << copyConsCount;
        }
};

int Room::copyConsCount = 0;

void main()
{
        Room objRoom1;
        Room objRoom2(objRoom1);
        Room objRoom3 = objRoom1;
        Room objRoom4;
        objRoom4 = objRoom3;

        objRoom4.dispCopyConsCount();
}
```

6.4 Identify the error in the following program.

```
#include <iostream.h>

class Room
{
        int width;
        int height;
        static int copyConsCount;
public:
        Room()
        {
                width = 12;
                height = 8;
        }

        Room(Room& r)
        {
```

```
                        width = r.width;
                        height = r.height;
                        copyConsCount++;
                }

                void disCopyConsCount()
                {
                        cout << copyConsCount;
                }
        };

        int Room::copyConsCount = 0;

        void main()
        {
                Room objRooml;
                Room objRoom2 (objRooml);
                Room objRoom3 = objRooml;
                Room objRoom4;
                objRoom4 = objRoom3;

                objRoom4.dispCopyConsCount();
        }
```

Programming Exercises

6.1 *Design constructors for the classes designed in Programming Exercises 5.1 through 5.5 of Chapter 5.*

6.2 *Define a class* ***String*** *that could work as a user-defined string type. Include constructors that will enable us to create an uninitialized string*

```
String sl; // string with length 0
```

and also to initialize an object with a string constant at the time of creation like

```
     String s2("Well done!");
```

Include a function that adds two strings to make a third string. Note that the statement

```
     s2 = sl;
```

will be perfectly reasonable expression to copy one string to another.

Write a complete program to test your class to see that it does the following tasks:

(a) *Creates uninitialized string objects.*

(b) *Creates objects with string constants.*

(c) *Concatenates two strings properly.*

(d) *Displays a desired string object.*

6.3 *A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message "Required copies not in stock" is displayed.*

Design a system using a class called ***books*** *with suitable member functions and constructors. Use* ***new*** *operator in constructors to allocate memory space required.*

6.4 *Improve the system design in Exercise 6.3 to incorporate the following features:*

(a) *The price of the books should be updated as and when required. Use a private member function to implement this.*

(b) *The stock value of each book should be automatically updated as soon as a transaction is completed.*

(c) *The number of successful and unsuccessful transactions should be recorded for the purpose of statistical analysis. Use* **static** *data members to keep count of transactions.*

6.5 *Modify the program of Exercise 6.4 to demonstrate the use of pointers to access the members.*

7

Operator Overloading and Type Conversions

Key Concepts

- Overloading
- Operator functions
- Overloading unary operators
- String manipulations
- Basic to class type
- Class to class type
- Operator overloading
- Overloading binary operators
- Using friends for overloading
- Type conversions
- Class to basic type
- Overloading rules

7.1 Introduction

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as *operator overloading.*

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can almost create a new language of our own by the creative use of the function and operator overloading techniques. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operators (., .*).
- Scope resolution operator (::).
- Size operator **(sizeof)**.
- Conditional operator (?:).

The excluded operators are very few when compared to the large number of operators which qualify for the operator overloading definition.

Although the *semantics* of an operator can be extended, we cannot change its *syntax*, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

7.2 Defining Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called *operator function*, which describes the task. The general form of an operator function is:

```
return type classname :: operator op(arglist)
{
        Function body          // task defined
}
```

where *return type* is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword **operator**. **operator** *op* is the function name.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with **friend** functions. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes as follows:

```
vector operator+(vector);                    // vector addition
vector operator-();                          // unary minus
friend vector operator+(vector,vector);      // vector addition
friend vector operator-(vector);             // unary minus
vector operator-(vector &a);                 // subtraction
int operator==(vector);                      // comparison
friend int operator==(vector,vector)         // comparison
```

vector is a data type of **class** and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics)

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operaion.
2. Declare the operator function **operator** *op*() in the public part of the class. It may be either a member function or a **friend** function.
3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

```
op x or x op
```

for unary operators and

```
x op y
```

for binary operators. *op* x (or x *op*) would be interpreted as

```
operator op (x)
```

for **friend** functions. Similarly, the expression x op y would be interpreted as either

```
x.operator op (y)
```

in case of member functions, or

```
operator op (x,y)
```

in case of **friend** functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

7.3 Overloading Unary Operators

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.

Program 7.1 shows how the unary minus operator is overloaded.

[illegible]

```
#include <iostream>

using namespace std;

class space
{
        int x;
        int y;
        int z;
  public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator-();     // overload unary minus
};
void space :: getdata(int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space :: display(void)
{
        cout << x << " ";
        cout << y << " " ;
        cout << z << "\n";
}
void space :: operator-()
{
        x = -x;
        y = -y;
        z = -z;
}

int main()
{
        space S;
        S.getdata(10, -20, 30);
```

(Contd)

```
        cout << "S : ";
        S.display();

        -S;                    // activates operator-() function

        cout << "S : ";
        S.display();

        return 0;
    }
```

PROGRAM 7.1

The Program 7.1 produces the following output:

```
S : 10 -20 30
S : -10 20 -30
```

note

The function **operator** – ( ) takes no argument. Then, what does this operator function do?. It changes the sign of data members of the object **S**. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function **operator**–() does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend  void operator-(space &s);          // declaration
void operator-(space &s)                   // definition
  {
      s.x = -s.x;
      s.y = -s.y;
      s.z = -s.z;
  }
```

note

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-(). Therefore, the changes made inside the operator function will not reflect in the called object.

7.4 Overloading Binary Operators

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum(A, B);          // functional notation.
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;              // arithmetic notation
```

by overloading the + operator using an operator+() function. The Program7.2 illustrates how this is accomplished.

OVERLOADING + OPERATOR

```
#include <iostream>

using namespace std;

class complex
{
        float x;                              // real part
        float y;                              // imaginary part
  public:
        complex(){ }                          // constructor 1
        complex(float real, float imag)       // constructor 2
        { x = real; y = imag; }
        complex operator+(complex);
        void display(void);
};

complex complex :: operator+(complex c)
{
        complex temp;                 // temporary
        temp.x = x + c.x;             // these are
        temp.y = y + c.y;             // float additions
        return(temp);
}

void complex :: display(void)
{
        cout << x << " + j" << y << "\n";
```

(Contd)

```
}

int main()
{
        complex C1, C2, C3;          // invokes constructor 1
        C1 = complex(2.5, 3.5);      // invokes constructor 2
        C2 = complex(1.6, 2.7);
        C3 = C1 + C2;

        cout << "C1 = "; C1.display();
        cout << "C2 = "; C2.display();
        cout << "C3 = "; C3.display();

        return 0;
}
```

PROGRAM 7.2

The output of Program 7.2 would be:

```
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2
```

note

Let us have a close look at the function **operator+( )** and see how the operator overloading is implemented.

```
complex complex :: operator+(complex c)
{
      complex temp;
      temp.x = x + c.x;
      temp.y = y + c.y;
      return(temp);
}
```

We should note the following features of this function:

1. It receives only one **complex** type argument explicitly.
2. It returns a **complex** type value.
3. It is a member function of **complex**.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
C3 = C1 + C2;                // invokes operator+() function
```

where **A** and **B** are objects of the same class. This will work for a member function but the statement

```
A = 2 + B; (or A = 2 * B)
```

will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However **friend** function allows both approaches. How?

It may be recalled that an object need not be used to invoke a **friend** function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the *left-hand* operand and an object as the *right-hand* operand. Program 7.3 illustrates this, using scalar *multiplication* of a vector. It also shows how to overload the input and output operators >> and <<.

OVERLOADING OPERATORS USING FRIENDS

```
#include <iostream.h>

  const size = 3;

  class vector
  {
        int v[size];
   public:
        vector();                    // constructs null vector
        vector(int *x);              // constructs vector from array
        friend vector operator *(int a, vector b);       // friend 1
        friend vector operator *(vector b, int a);       // friend 2
        friend istream & operator >> (istream &, vector &);
        friend ostream & operator << (ostream &, vector &);
  };

  vector :: vector()
  {
        for(int i=0; i<size; i++)
              v[i] = 0;
  }

  vector :: vector(int *x)
  {
        for(int i=0; i<size; i++)
              v[i] = x[i];
  }
```

(Contd)

```
vector operator *(int a, vector b)
{
        vector c;

        for(int i=0; i < size; i++)
                c.v[i] = a * b.v[i];
        return c;
}

        vector operator *(vector b, int a)
{
        vector c;

        for(int i=0; i<size; i++)
                c.v[i] = b.v[i] * a;
        return c;
}

istream & operator >> (istream &din, vector &b)

{
        for(int i=0; i<size; i++)
                din >> b.v[i];
        return(din);
}
ostream & operator <<  (ostream &dout, vector &b)
{
        dout << "(" << b.v [0];

        for(int i=1; i<size; i++)
                dout << ", " << b.v[i];
        dout << ")";
        return(dout);
}

int x[size] = {2,4,6};

int main()
{
        vector m;               // invokes constructor 1
        vector n = x;           // invokes constructor 2

        cout << "Enter elements of vector m " << "\n";
        cin >> m;               // invokes operator>>() function
```

(Contd)

```
        cout << "\n";
        cout << "m = " << m << "\n";          // invokes operator <<()

        vector p, q;

        p = 2 * m;            // invokes friend 1
        q = n * 2;            // invokes friend 2

        cout << "\n";
        cout << "p = " << p << "\n";          // invokes operator<<()
        cout << "q = " << q << "\n";

        return 0;
}
```

PROGRAM 7.3

Shown below is the output of Program 7.3:

```
Enter elements of vector m
5 10 15

m = (5, 10, 15)
p = (10, 20, 30)
q = (4, 8, 12)
```

The program overloads the operator * two times, thus overloading the operator function operator*() itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scalar multiplication such as

```
p = 2 * m;            // equivalent to p = operator*(2,m);
q = n * 2;            // equivalent to q = operator*(n,2);
```

The program and its output are largely self-explanatory. The first constructor

```
vector();
```

constructs a vector whose elements are all zero. Thus

```
vector m;
```

creates a vector m and initializes all its elements to 0. The second constructor

```
vector(int &x);
```

creates a vector and copies the elements pointed to by the pointer argument x into it. Therefore, the statements

```
int x[3] = {2, 4, 6};
vector n = x;
```

create n as a vector with components 2, 4, and 6.

note

We have used vector variables like **m** and **n** in input and output statements just like simple variables. This has been made possible by overloading the operators >> and << using the functions:

```
friend istream & operator>>(istream &, vector &);
friend ostream & operator<<(ostream &, vector &);
```

istream and **ostream** are classes defined in the **iostream.h** file which has been included in the program.

7.6 Manipulation of Strings Using Operators

ANSI C implements strings using character arrays, pointers and string functions. There are no operators for manipulating the strings. One of the main drawbacks of string manipulations in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

Although these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. (Recently, ANSI C++ committee has added a new class called **string** to the C++ class library that supports all kinds of string manipulations. String manipulations using the **string** class are discussed in Chapter 15.

For example, we shall be able to use statements like

```
string3 = string1 + string2;
if(string1 >= string2) string = string1;
```

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use *new* to allocate memory for each string and a pointer variable to point to the string array. Thus we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

```
class string
{
        char *p;            // pointer to string
        int len;            // length of string
    public:
        .....               // member functions
        .....               // to initialize and
        .....               // manipulate strings
    };
```

We shall consider an example to illustrate the application of overloaded operators to strings. The example shown in Program 7.4 overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

MATHEMATICAL OPERATIONS ON STRINGS

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

class string
{
       char *p;
       int len;
public:
       string() {len = 0; p = 0;}          // create null string
       string(const char * s);             // create string from arrays
       string(const string & s);           // copy constructor
       ~ string(){delete p;}               // destructor

       // + operator
       friend string operator+(const string &s, const string &t);

       // <= operator
       friend int operator<=(const string &s, const string &t);
       friend void show(const string s);
};
string :: string(const char *s)
{
       len = strlen(s);
       p   = new char[len+1];
       strcpy(p,s);
}

string :: string(const string & s)
{
       len = s.len;
       p   = new char[len+1];
```

(Contd)

```
        strcpy(p,s.p);
}

// overloading + operator
string operator+(const string &s, const string &t)
{
        string temp;
        temp.len = s.len + t.len;
        temp.p = new char[temp.len+1];
        strcpy(temp.p,s.p);
        strcat(temp.p,t.p);
        return(temp);
}
// overloading <= operator
int operator<=(const string &s, const string &t)
{
        int m = strlen(s.p);
        int n = strlen(t.p);

        if(m <= n) return(1);

        else return(0);
}
void show(const string s)
{
        cout << s.p;
}

int main()
{
        string s1 = "New ";
        string s2 = "York";
        string s3 = "Delhi";
        string t1,t2,t3;
        t1 = s1;
        t2 = s2;
        t3 = s1+s3;

        cout << "\nt1 = "; show(t1);
        cout << "\nt2 = "; show(t2);
        cout << "\n";
        cout << "\nt3 = "; show(t3);
        cout << "\n\n";
```

(Contd)

```
        if(tl <= t3)
        {
              show(tl);
              cout << " smaller than ";
              show(t3);
              cout << "\n";
        }
        else
        {
              show(t3);
              cout << " smaller than ";
              show(tl);
              cout << "\n";
        }

        return 0;
}
```

PROGRAM 7.4

The following is the output of Program 7.4

```
t1 = New
t2 = York

t3 = New Delhi

New smaller than New Delhi
```

7.7 Rules for Overloading Operators

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded. (See Table 7.1.)
6. We cannot use **friend** functions to overload certain operators. (See Table 7.2.) However, member functions can be used to overload them.

7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, –, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Table 7.1 *Operators that cannot be overloaded*

Sizeof	Size of operator
.	Membership operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	Conditional operator

Table 7.2 *Where a friend cannot be used*

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

7.8 Type Conversions

We know that when constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

```
int m;
float x = 3.14159;
m = x;
```

convert **x** to an integer before its value is assigned to **m**. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

What happens when they are user-defined data types?

Consider the following statement that adds two objects and then assigns the result to a third object.

```
    v3 = v1 + v2;            // v1, v2 and v3 are class type objects
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore, design the conversion routines by ourselves, if such operations are required.

Three types of situations might arise in the data conversion between uncompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail.

Basic to Class Type

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an **int** type array. Similarly, we used another constructor to build a string type object from a **char*** type variable. These are all examples where constructors perform a *defacto* type conversion from the argument's type to the constructor's class type.

Consider the following constructor:

```
    string :: string(char *a)
    {
        length = strlen(a);
        P = new char[length+1];
        strcpy(P,a);
    }
```

This constructor builds a **string** type object from a **char*** type variable **a**. The variables **length** and **p** are data members of the class **string**. Once this constructor has been defined in the string class, it can be used for conversion from **char*** type to string type. Example:

```
string s1, s2;
char* name1 = "IBM PC";
char* name2 = "Apple Computers";
s1 = string(name1);
s2 = name2;
```

The statement

```
s1 = string(name1);
```

first converts **name1** from **char*** type to **string** type and then assigns the string type values to the object **s1**. The statement

```
s2 = name2;
```

also does the same job by invoking the constructor implicitly.

Let us consider another example of converting an **int** type to a **class** type.

```
class time
{
    int hrs;
     int mins;
  public:
     ....
     ....
     time(int t)              // constructor
     {
         hours = t/60;        // t in minutes
         mins  = t%60;
     }
};
```

The following conversion statements can be used in a function:

```
time T1;                      // object T1 created
int duration = 85;
T1 = duration;                // int to class type
```

After this conversion, the **hrs** member of **T1** will contain a value of 1 and **mins** member a value of 25, denoting 1 hours and 25 minutes.

note

The constructors used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a **class** object. Therefore, we can also accomplish this conversion using an overloaded = operator.

Class to Basic Type

The constructors did a fine job in type conversion from a basic to class type. What about the conversion from a class to basic type? The constructor functions do not support this operation. Luckily, C++ allows us to define an overloaded *casting operator* that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function, usually referred to as a *conversion function*, is:

```
    operator typename()
    {
          .....
          ..... (Function statements)
          .....
    }
```

This function converts a class type data to *typename*. For example, the **operator double()** converts a class object to type **double**, the **operator int()** converts a class type object to type int, and so on.

Consider the following conversion function:

```
vector :: operator double()
{
  double sum = 0;
  for(int i=0; i<size; i++)
        sum = sum + v[i] * v[i];
  return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. Recall that the magnitude of a vector is given by the square root of the sum of the squares of its components. The operator **double()** can be used as follows:

```
double length = double(V1);
    or
double length = V1;
```

where **V1** is an object of type **vector**. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Since it is a member function, it is invoked by the object and, therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from string to **char*** as follows:

```
string :: operator char*()
{
        return(p);
}
```

One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situations where we would like to convert one class type data to another class type.

Example:

```
objX = objY;       // objects of different types
```

objX is an object of class **X** and **objY** is an object of class **Y**. The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from **class Y** to **class X, Y** is known as the *source* class and **X** is known as the *destination* class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do we decide which form to use? It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

We know that the casting operator function

```
operator typename()
```

converts the class object *of which it is a member* to *typename*. The *typename* may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, *typename* refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the *argument's type* to the class type of *which it is a member*. This implies that the argument belongs to the *source* class and is passed to the *destination* class for conversion. This makes it necessary that the conversion constructor be placed in the destination class. Figure 7.2 illustrates these two approaches.

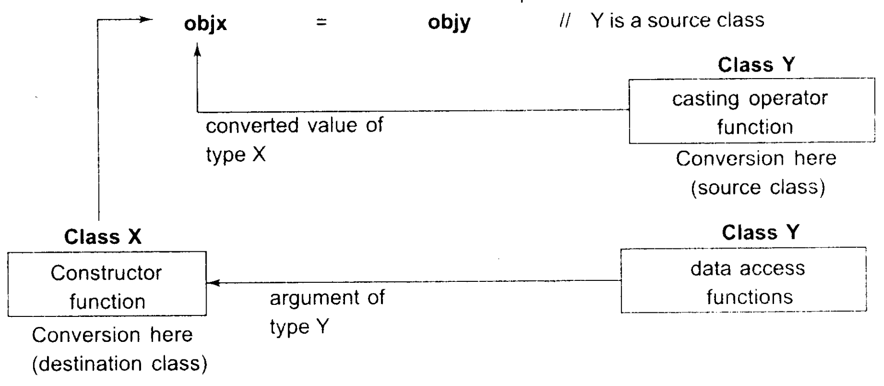

objx = objy // Y is a source class
Class Y
casting operator function
converted value of type X
Conversion here (source class)
Class X
Constructor function
Class Y
data access functions
argument of type Y
Conversion here (destination class)

Fig. 7.2 ⇔ *Conversion between object*

Table 7.3 provides a summary of all the three conversions. It shows that the conversion from a class to any other type (or any other class) should make use of a casting operator in the source class. On the other hand, to perform the conversion from any other type/class to a class type, a constructor should be used in the destination class.

Table 7.3 *Type conversions*

Conversion required	***Conversion takes place in***	
	Source class	***Destination class***
Basic → class	Not applicable	Constructor
Class → basic	Casting operator	Not applicable
Class → class	Casting operator	Constructor

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

A Data Conversion Example

Let us consider an example of an inventory of products in store. One way of recording the details of the products is to record their code number, total items in the stock and the cost of each item. Another approach is to just specify the item code and the value of the item in the stock. The example shown in Program 7.5 uses two classes and shows how to convert data of one type to another.

DATA CONVERSIONS

```
    #include <iostream>

    using namespace std;

    class invent2               // destination class declared

    class invent1               // source class
    {
        int code;               // item code
        int items;              // no. of items
        float price;            // cost of each item
  public:
    invent1(int a, int b, float c)
    {
        code =  a;
        items = b;
        price = c;
    }
     void putdata()
    {
              cout << "Code:  " << code  << "\n";
              cout << "Items: " << items << "\n";
              cout << "Value: " << price << "\n";
    }
    int getcode() {return code;}
    int getitems() {return items;}
    float getprice() {return price;}
    operator float() {return(items * price);}

    /* operator invent2()       // invent1 to invent2
    {
        invent2 temp;
        temp.code = code;
        temp.value = price * items;
        return temp;
    } */
  };    // End of source class
```

(Contd)

```
class invent2          // destination class
{
       int code;
       float value;
  public:
       invent2()
       {
              code = 0; value = 0;
       }
       invent2(int x, float y)     // constructor for
                                   // initialization
       {
              code = x;
              value = y;
       }
       void putdata()
       {
              cout << "Code:  " << code  << "\n";
              cout << "Value: " << value << "\n\n";
       }
       invent2(invent1 p)          // constructor for conversion
       {
              code = p.getcode();
              value =p.getitems() * p.getprice();
       }
};     // End of destination class

int main()
{
       invent1 s1(100,5,140.0);
       invent2 d1;
       float total_value;

       /* invent1 To float  */
       total_value = s1;

       /* invent1 To invent2 */
       d1 = s1;

       cout << "Product details - invent1 type" << "\n";
       s1.putdata();

       cout << "\nStock value" << "\n";
       cout << "Value = " << total_value << "\n\n";

       cout << "Product details-invent2 type" << "\n";
       d1.putdata();

       return 0;
}
```

PROGRAM 7.5

Following is the output of Program 7.5:

```
Product details-inventl type
Code:  100
Items: 5
Value: 140
Stock value
Value = 700
Product details-invent2 type
Code:  100
Value: 700
```

note

We have used the conversion function

```
operator float( )
```

in the class **inventl** to convert the **inventl** type data to a **float**. The constructor

```
invent2 (inventl)
```

is used in the class **invent2** to convert the **inventl** type data to the **invent2** type data.

Remember that we can also use the casting operator function

```
operator invent2()
```

in the class inventl to convert **inventl** type to **invent2** type. However, it is important that we do not use both the constructor and the casting operator for the same type conversion, since this introduces an ambiguity as to how the conversion should be performed.

SUMMARY

⇔ Operator overloading is one of the important features of C++ language. It is called compile time polymorphism.

⇔ Using overloading feature we can add two user defined data types such as objects, with the same syntax, just as basic data types.

⇔ We can overload almost all the C++ operators except the following:

- class member access operators(., .*)
- scope resolution operator (::)

- size operator(sizeof)
- conditional operator(?:)

⇔ Operator overloading is done with the help of a special function, called operator function, which describes the special task to an operator.

⇔ There are certain restrictions and limitations in overloading operators. Operator functions must either be member functions (non-static) or friend functions. The overloading operator must have at least one operand that is of user-defined type.

⇔ The compiler does not support automatic type conversions for the user defined data types. We can use casting operator functions to achieve this.

⇔ The casting operator function should satisfy the following conditions:
- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Key Terms

- arithmetic notation
- binary operators
- casting
- casting operator
- constructor
- conversion function
- destination class
- **friend**
- **friend** function
- functional notation
- manipulating strings
- operator
- operator function
- operator overloading
- scalar multiplication
- semantics
- **sizeof**
- source class
- syntax
- temporary object
- type conversion
- unary operators

Review Questions

7.1 *What is operator overloading?*

7.2 *Why is it necessary to overload an operator?*

7.3 *What is an operator function? Describe the syntax of an operator function.*

7.4 *How many arguments are required in the definition of an overloaded unary operator?*

7.5 *A class alpha has a constructor as follows:*

alpha(int a, double b);

Can we use this constructor to convert types?

7.6 *What is a conversion function How is it created Explain its syntax.*

7.7 *A friend function cannot be used to overload the assignment operator =. Explain why?*

7.8 *When is a friend function compulsory? Give an example.*

7.9 *We have two classes X and Y. If **a** is an object of X and b is an object of **Y** and we want to say **a** = **b**; What type of conversion routine should be used and where?*

7.10 *State whether the following statements are TRUE or FALSE.*

(a) *Using the operator overloading concept, we can change the meaning of an operator.*

(b) *Operator overloading works when applied to class objects only.*

(c) *Friend functions cannot be used to overload operators.*

(d) *When using an overloaded binary operator, the left operand is implicitly passed to the member function.*

(e) *The overloaded operator must have at least one operand that is user-defined type.*

(f) *Operator functions never return a value.*

(g) *Through operator overloading, a class type data can be converted to a basic type data.*

(h) *A constructor can be used to convert a basic type to a class type data.*

Debugging Exercises

7.1 Identify the error in the following program.

```
#include <iostream.h>
class Space
{
    int mCount;
public:
    Space()
    {
        mCount = 0;
    }

    Space operator ++()
    {
        mCount++;
```

```
            return Space(mCount);
        }
};

void main()
{
     Space objSpace;
     objSpace++;
}
```

7.2 Identify the error in the following program.

```
#include <iostream.h>
enum WeekDays
{
     mSunday,
     mMonday,
     mTuesday,
     mWednesday,
     mThursday,
     mFriday,
     mSaturday
};
bool op==(WeekDays& w1, WeekDays& w2)
{
     if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else if(w1== mSunday && w2 == mSunday)
          return 1;
     else
          return 0;
}
void main()
{
     WeekDays w1 = mSunday, w2 = mSunday;
     if(w1==w2)
          cout << "Same day";
     else
          cout << "Different day";
}
```

7.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
     float mWidth;
     float mLength;
public:
     Room()
     {
     }
     Room(float w, float h)
          :mWidth(w), mLength(h)
     {
     }
     operator float()
     {
          return (float)mWidth * mLength;
     }

     float getWidth()
     {
     }

     float getLength()
     {
          return mLength;
     }
};

void main()
```

```
{
    Room objRoom1(2.5, 2.5);
    float fTotalArea;
    fTotalArea = objRoom1;
    cout << fTotalArea;
}
```

Programming Exercises

NOTE: *For all the exercises that follow, build a demonstration program to test your code.*

7.1 *Create a class **FLOAT** that contains one float data member. Overload all the four arithmetic operators so that they operate on the objects of **FLOAT**.*

7.2 *Design a class **Polar** which describes a point in the plane using polar coordinates **radius** and **angle**. A point in polar coordinates is shown in Fig. 7.3.*

Use the overloaded + operator to add two objects of Polar.

Note that we cannot add polar values of two points directly. This requires first the conversion of points into rectangular co-ordinates, then adding the corresponding rectangular co-ordinates and finally converting the result back into polar co-ordinates. You need to use the following trigonometric formulae:

```
x = r * cos(a);
y = r * sin(a);
a = atan(y/x);     // arc tangent
r = sqrt(x*x + y*y);
```

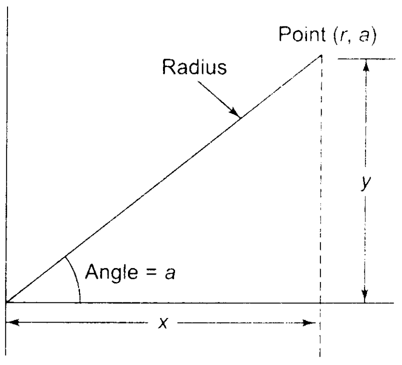


Fig. 7.3 ⇔ *Polar coordinates of a point*

7.3 *Create a class **MAT** of size **m x n**. Define all possible matrix operations for **MAT** type objects.*

7.4 *Define a class String. Use overloaded == operator to compare two strings.*

7.5 *Define two classes **Polar** and **Rectangle** to represent points in the polar and rectangle systems. Use conversion routines to convert from one system to the other.*