5

Classes and Objects

Key Concepts

- Using structures
- Creating a class
- Defining member functions
- Creating objects
- Using objects
- Inline member functions
- Nested member functions
- Private member functions
- Arrays as class members
- Storage of objects
- Static data members
- Static member functions
- Using arrays of objects
- Passing objects as parameters
- Making functions friendly to classes
- Functions returning objects
- const member functions
- Pointers to members
- Using dereferencing operators
- Local classes

5.1 Introduction

The most important feature of C++ is the "class". Its significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an

extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type. We shall discuss, in this chapter, the concept of class by first reviewing the traditional structures found in C and then the ways in which classes can be designed, implemented and applied.

5.2 C Structures Revisited

We know that one of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user-defined data type with a *template* that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
      char   name[20];
      int    roll_number;
      float total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as *structure members* or *elements*. The identifier student, which is referred to as *structure name* or *structure tag*, can be used to create variables of type student. Example:

```
struct student A;   // C declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the *dot* or *period operator* as follows:

```
strcpy(A.name, "John");
A.roll_number = 999;
A.total_marks = 595.5;
Final_total = A.total_marks + 5;
```

Structures can have arrays, pointers or structures as members.

Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

Copyrighted material

```
struct complex
{
      float x;
      float y;
};
      struct complex c1, c2, c3;
```

The complex numbers c1, c2, and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit *data hiding*. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

Extensions to Structures

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. *Inheritance*, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

```
student A;    // C++ declaration
```

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.

note

The only difference between a structure and a class in C++ is that, by default, the members of a class are *private*, while, by default, the members of a structure are *public*.

Copyrighted material

5.3 Specifying a Class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
        private:
             variable declarations;
             function declarations;
        public:
             variable declarations;
             function declaration;
};
```

The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members*. They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are *private* and which of them are *public*. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are **private**. If both the labels are missing, then, by default, all the members are **private**. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as *data members* and the functions are known as *member functions*. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. This is illustrated in Fig. 5.1. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

Copyrighted material

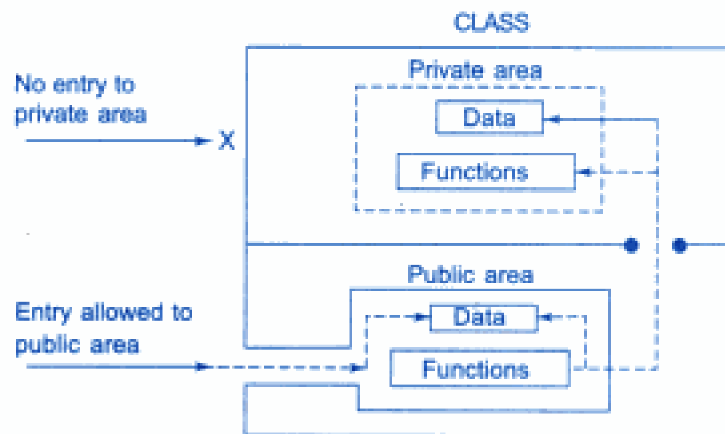


Fig. 5.1 ⇔ *Data hiding in classes*

A Simple Class Example

A typical class declaration would look like:

```
class item
{
        int number;                    // variables declaration
        float cost;                    // private by default
    public:
        void getdata(int a, float b);  // functions declaration
        void putdata(void);            // using prototype
    };// ends with semicolon
```

We usually give a class some meaningful name, such as **item**. This name now becomes a new type identifier that can be used to declare *instances* of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function **getdata()** can be used to assign values to the member variables number and cost, and **putdata()** for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item**. Note that the functions are declared, not defined. Actual function definitions will appear later in the program. The data members are usually declared as **private** and the member functions as **public**. Figure 5.2 shows two different notations used by the OOP analysts to represent a class.

Creating Objects

Remember that the declaration of **item** as shown above does not define any objects of **item** but only specifies *what* they will contain. Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

Copyrighted material

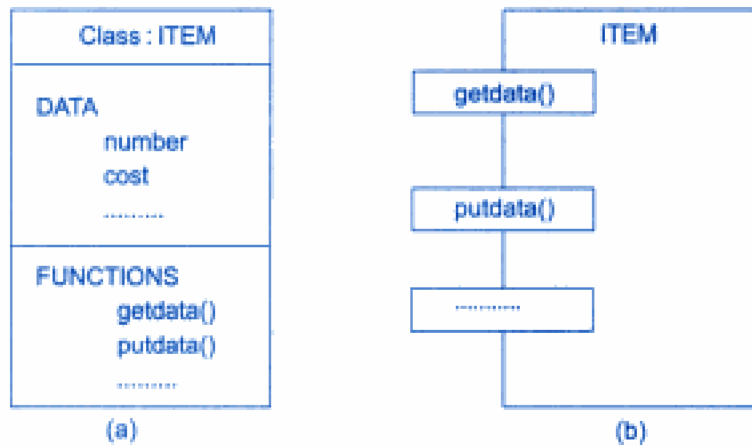


Fig. 5.2 ⇔ *Representation of a class*

```
item x;        // memory for x is created
```

creates a variable **x** of type **item**. In C++, the class variables are known as *objects*. Therefore, **x** is called an object of type **item**. We may also declare more than one object in one statement. Example:

```
item x, y, z;
```

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure, provides only a *template* and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class item
{
        .....
        .....
        .....
} x,y,z;
```

would create the objects **x, y** and **z** of type **item**. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that access **number** and **cost** directly. The following is the format for calling a member function:

Copyrighted material

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata()** function. The assignments occur in the actual function. Please refer Sec. 5.4 for further details.

Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although **x** is an object of the type **item** to which **number** belongs, the number (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

```
x.putdata();
```

sends a message to the object **x** requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
        int x;
        int y;
    public:
        int z;
};
        .....
        .....
xyz p;
p.x = 0;            // error, x is private
p.z = 10            // OK, z is public
        .....
        .....
```

Copyrighted material

note

The use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

5.4 Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the *ANSI prototype* form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which **class** the function belongs to. The general form of a member function definition is:

```
return-type class-name :: function-name (argument declaration)
  {
          Function body
  }
```

The membership label class-name :: tells the compiler that the function *function-name* belongs to the class *class-name*. That is, the scope of the function is restricted to the *class-name* specified in the header line. The symbol :: is called the *scope resolution* operator.

For instance, consider the member functions **getdata()** and **putdata()** as discussed above. They may be coded as follows:

```
void item :: getdata(int a, float b)
{
     number = a;
     cost = b;
}
```

Copyrighted material

```
void item :: putdata(void)
{
     cout << "Number :" << number << "\n";
     cout << "Cost   :" << cost   << "\n";
}
```

Since these functions do not return any value, their return-type is void. Function arguments are declared using the ANSI prototype.

The member functions have some special characteristics that are often used in the program development. These characteristics are :

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so. (However, an exception to this rule is a *friend* function discussed later.)
- A member function can call another member function directly, without using the dot operator.

Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
    int number;
    float cost;
  public:
    void getdata(int a, float b);      // declaration
         // inline function
    void putdata(void)                 // definition inside the class
    {
         cout << number  <<  "\n";
         cout << cost    <<  "\n";
    }
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an **inline** function are also applicable here. Normally, only small functions are defined inside the class definition.

5.5 A C++ Program with Class

All the details discussed so far are implemented in Program 5.1.

Copyrighted material

CLASS IMPLEMENTATION

```
#include <iostream>

using namespace std;

class item
{
        int number;     // private by default
        float cost;     // private by default
    public:
        void getdata(int a, float b);        // prototype declaration,
                                             // to be defined
        // Function defined inside class
        void putdata(void)
        {
                cout << "number :" << number << "\n";
                cout << "cost   :" << cost   << "\n";
        }
};
//............ Member Function Definition ................
void item :: getdata(int a, float b)     // use membership label
{
        number = a;     // private variables
        cost = b;       // directly used
}
//.................... Main Program .....................

int main()
{
        item x;// create object x

        cout <<  "\nobject x "  <<  "\n";

        x.getdata(100, 299.95);              // call member function
        x.putdata();                         // call member function

        item y;                              // create another object

        cout <<  "\nobject y"  << "\n";

        y.getdata(200, 175.50);
        y.putdata();

        return 0;
  }
```

PROGRAM 5.1

Copyrighted material

This program features the class **item**. This class contains two private variables and two public functions. The member function **getdata()** which has been defined outside the class supplies values to both the variables. Note the use of statements such as

```
number = a;
```

in the function definition of **getdata()**. This shows that the member functions can have direct access to private data items.

The member function **putdata()** has been defined inside the class and therefore behaves like an **inline** function. This function displays the values of the private variables **number** and **cost**.

The program creates two objects, x and y in two different statements. This can be combined in one statement.

```
item x, y;        // creates a list of objects
```

Here is the output of Program 5.1:

```
object x
number :100
cost   :299.95

object y
number :200
cost   :175.5
```

For the sake of illustration we have shown one member function as **inline** and the other as an 'external' member function. Both can be defined as **inline** or external functions.

5.6 Making an Outside Function Inline

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by just using the qualifier **inline** in the header line of function definition. Example:

```
class item
{
        .....
        .....
   public:
        void getdata(int a, float b);          // declaration
};
```

Copyrighted material

```
inline void item :: getdata(int a, float b)  // definition
{
        number = a;
        cost = b;
}
```

5.7 Nesting of Member Functions

We just discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as *nesting of member functions*. Program 5.2 illustrates this feature.

NESTING OF MEMBER FUNCTIONS

```
#include <iostream>

using namespace std;

class set
{
        int m, n;
    public:
        void input(void);
        void display(void);
        int largest(void);
};

int set :: largest(void)
{
        if(m >= n)
                return(m);
        else
                return(n);
}

void set :: input(void)
{
        cout << "Input values of m and n" << "\n";
        cin >> m >> n;
}

void set :: display(void)
{
```

(Contd)

Copyrighted material

```
        cout << "Largest value = "
             << largest() << "\n";        // calling member function
}

int main()
{
        set A;
        A.input();
        A.display();

        return 0;
}
```

PROGRAM 5.2

The output of Program 5.2 would be:

```
Input values of m and n
25 18
Largest value = 25
```

5.8 Private Member Functions

Although it is normal practice to place all the data items in a private section and all the functions in public, some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
        int m;
        void read(void);        // private member function
    public:
        void update(void);
        void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();        // won't work; objects cannot access
                  // private members
```

Copyrighted material

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void sample :: update(void)
{
      read();    // simple call; no object used
}
```

5.9 Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;      // provides value for array size

class array
{
      int a[size];      // 'a' is int type array
  public:
      void setval(void);
      void display(void);
};
```

The array variable **a[ ]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[ ]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Program 5.3 shows how these operations are implemented using a class with arrays as data members.

PROCESSING SHOPPING LIST

```
#include <iostream>

using namespace std;

const m=50;

class ITEMS
```

(Contd)

Copyrighted material

```
{
        int itemCode[m];
        float itemPrice[m];
        int count;
  public:
        void CNT(void){count = 0;}          // initializes count to 0
        void getitem(void);
        void displaySum(void);
        void remove(void);
        void displayItems(void);
};
//==========================================================
void ITEMS :: getitem(void)                 // assign values to data
                                            // members of item
{
        cout << "Enter item code :";
        cin >> itemCode[count];

        cout << "Enter item cost :";
        cin >> itemPrice[count];
        count++;
}
void ITEMS :: displaySum(void)              // display total value of
                                            // all items
{
        float sum = 0;
for(int i=0; i<count; i++)
                sum = sum + itemPrice[i];

        cout << "\nTotal value :" << sum << "\n";
}
void ITEMS :: remove(void)                  // delete a specified item
{
        int a;
        cout << "Enter item code :";
        cin >> a;

        for(int i=0; i<count; i++)
                if(itemCode[i] == a)
                        itemPrice[i] = 0;
}

void ITEMS :: displayItems(void)            // displaying items
{
```

(Contd)

Copyrighted material

```
        cout << "\nCode  Price\n";

        for(int i=0; i<count; i++)
        {
                cout <<"\n" << itemCode[i];
                cout <<"    " << itemPrice[i];
        }
        cout << "\n";
}
//==============================================================

int main()
{
        ITEMS order;
        order.CNT();
        int x;
        do              // do....while loop
        {
                cout << "\nYou can do the following;"
                     << "Enter appropriate number \n";
                cout << "\n1 : Add an item ";
                cout << "\n2 : Display total value";
                cout << "\n3 : Delete an item";
                cout << "\n4 : Display all items";
                cout << "\n5 : Quit";
                cout << "\n\nWhat is your option?";

                cin >> x;

                switch(x)
                {
                        case 1 : order.getitem(); break;
                        case 2 : order.displaySum(); break;
                        case 3 : order.remove(); break;
                        case 4 : order.displayItems(); break;
                        case 5 : break;
                        default : cout << "Error in input; try again\n";
                }

        } while(x != 5);                        // do...while ends

        return 0;
    }
```

PROGRAM 5.3

Copyrighted material

The output of Program 5.3 would be:

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :111
Enter item cost :100

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :222
Enter item cost :200

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :333
Enter item cost :300

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?2
Total value :600
```

(Contd)

Copyrighted material

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?3
Enter item code :222

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?4
Code      Price
111             100
222             0
333             300

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?5
```

note

The program uses two arrays, namely **itemCode[ ]** to hold the code number of items and **itemPrice[ ]** to hold the prices. A third data member **count** is used to keep a record of items in the list. The program uses a total of four functions to implement the operations to be performed on the list. The statement

```
const int m = 50;
```

defines the size of the array members.

The first function **CNT( )** simply sets the variable **count** to zero. The second function **getitem()** gets the item code and the item price interactively and assigns them to the array members **itemCode[count]** and **itemPriee[count]**. Note that inside this function **count**

Copyrighted material

is incremented after the assignment operation is over. The function **displaySum( )** first evaluates the total value of the order and then prints the value. The fourth function **remove()** deletes a given item from the list. It uses the item code to locate it in the list and sets the price to zero indicating that the item is not 'active' in the list. Lastly, the function **displayItems( )** displays all the items in the list.

The program implements all the tasks using a menu-based user interface.

5.10 Memory Allocation for Objects

We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects. This is shown in Fig. 5.3.

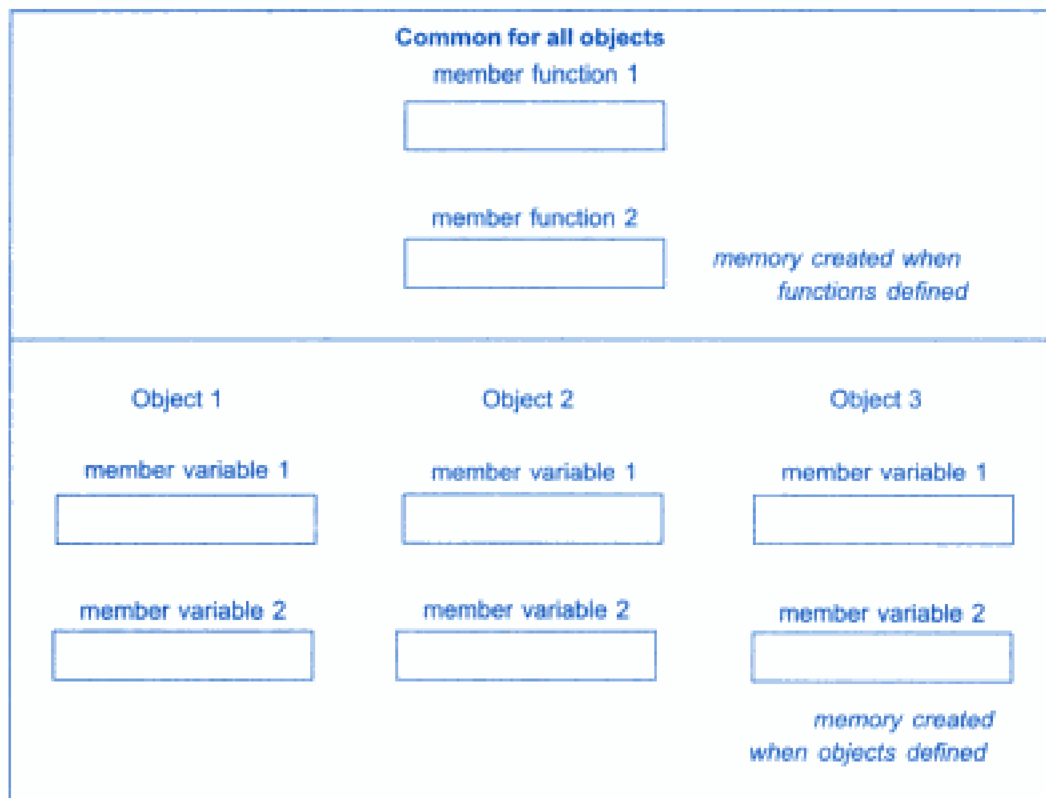


Fig. 5.3 ⇔ *Object of memory*

Copyrighted material

5.11 Static Data Members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. Program 5.4 illustrates the use of a static data member.

STATIC CLASS MEMBER

```
#include <iostream>

using namespace std;

class item
{
          static int count;
          int number;
  public:
          void getdata(int a)
          {
                 number = a;
                 count ++;
          }
          void getcount(void)
          {
                 cout << "count: ";
                 cout << count  << "\n";
          }
};

int item :: count;

int main()
{
```

(Contd)

Copyrighted material

```
        item a, b, c;           // count is initialized to zero
        a.getcount();           // display count
        b.getcount();
        c.getcount();

        a.getdata(100);         // getting data into object a
        b.getdata(200);         // getting data into object b
        c.getdata(300);         // getting data into object c

        cout << "After reading data" << "\n";

        a.getcount();           // display count
        b.getcount();
        c.getcount();
        return 0;
}
```

PROGRAM 5.4

The output of the Program 5.4 would be:

```
count: 0
count: 0
count: 0
After reading data
count: 3
count: 3
count: 3
```

note

Notice the following statement in the program:

```
int item :: count;      // definition of static data member
```

Note that the type and scope of each **static** member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as *class variables*.

The **static** variable **count** is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed. Figure 5.4 shows how a static variable is used by the objects.

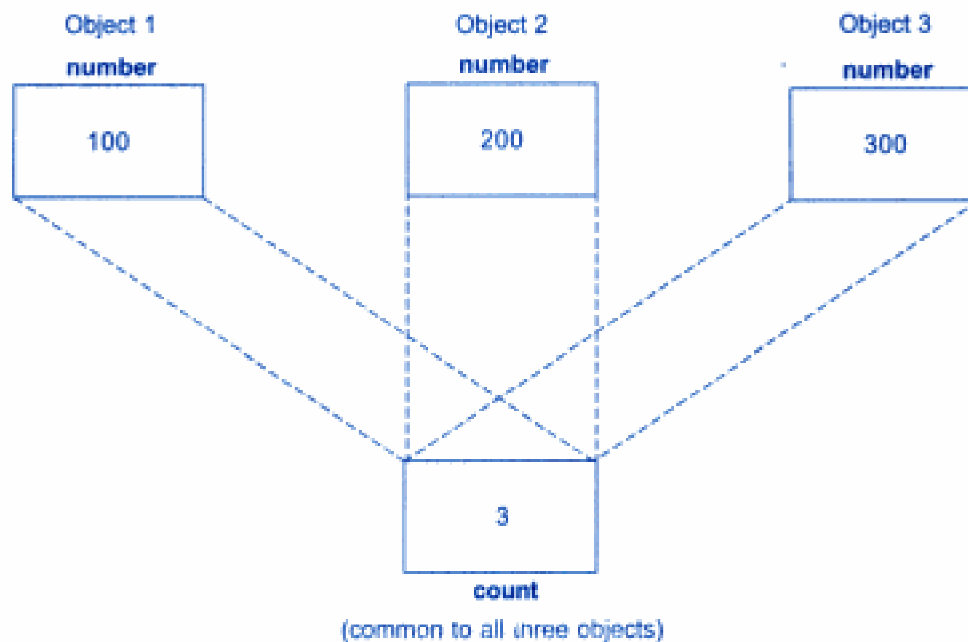


Fig. 5.4 ⇔ *Sharing of a static data member*

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

```
int item :: count = 10;
```

5.12 Static Member Functions

Like **static** member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

- A **static** function can have access to only other static members (functions or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of its objects) as follows:

```
class-name :: function-name;
```

Program 5.5 illustrates the implementation of these characteristics. The **static** function **showcount()** displays the number of objects created till that moment. A count of number of objects created is maintained by the **static** variable count.

The function **showcode()** displays the code number of each object.

Copyrighted material

STATIC MEMBER FUNCTION

```
#include <iostream>

using namespace std;

class test
{
        int code;
        static int count;              // static member variable
public:
        void setcode(void)
        {
        code = ++count;
        }
        void showcode(void)
        {
        cout << "object number: " << code << "\n";
        }
        static void showcount(void)    // static member function
        {
        cout << "count: " << count << "\n";
        }
};

int test :: count;
int main()
{
        test t1, t2;

        t1.setcode();
        t2.setcode();

        test :: showcount();    // accessing static function

        test t3;
        t3.setcode();

        test :: showcount();

        t1.showcode();
        t2.showcode();
        t3.showcode();

        return 0;
}
```

PROGRAM 5.5

Copyrighted material

Output of Program 5.5:

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```

note

Note that the statement

```
code = ++count;
```

is executed whenever **setcode( )** function is invoked and the current value of **count** is assigned to **code**. Since each object has its own copy of **code**, the value contained in **code** represents a unique number of its object.

Remember, the following function definition will not work:

```
static void showcount()
{
      cout << code;      // code is not static
}
```

5.13 Arrays of Objects

We know that an array can be of any data type including **struct**. Similarly, we can also have arrays of variables that are of the type **class**. Such variables are called *arrays of objects*. Consider the following class definition:

```
class employee
{
          char name[30];
          float age;
     public:
          void getdata(void);
          void putdata(void);
};
```

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager[3];          // array of manager
employee foreman[15];         // array of foreman
employee worker[75];          // array of worker
```

Copyrighted material

The array **manager** contains three objects(managers), namely, **manager[0], manager[1]** and **manager[2]**, of type **employee** class. Similarly, the **foreman** array contains 15 objects (foremen) and the **worker** array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array-accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata();
```

will display the data of the ith element of the array **manager**. That is, this statement requests the object **manager[i]** to invoke the member function **putdata()**.

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array manager is represented in Fig. 5.5. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

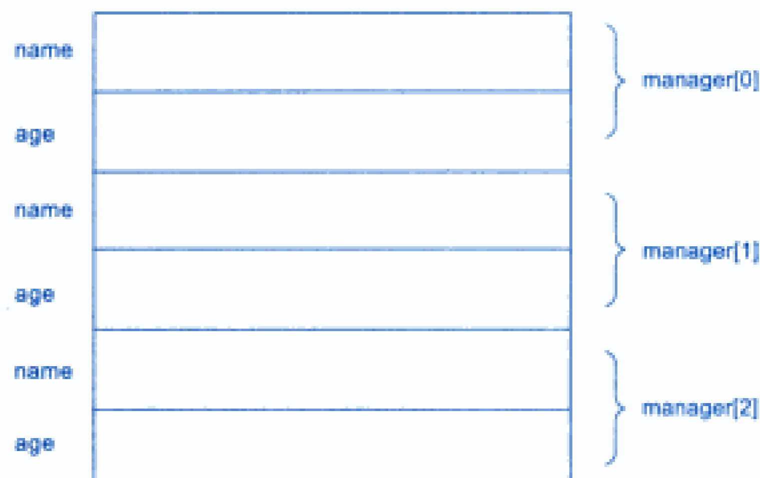


Fig. 5.5 ⇔ *Storage of data items of an object array*

Program 5.6 illustrates the use of object arrays.

ARRAYS OF OBJECTS

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

(Contd)

Copyrighted material

```
{
        char  name[30];      // string as class member
        float age;
  public:
        void getdata(void);
        void putdata(void);
};
void employee :: getdata(void)
{
        cout << "Enter name: ";
        cin  >> name;
        cout << "Enter age:  ";
        cin  >> age;
}
void employee :: putdata(void)
{
        cout << "Name: " << name << "\n";
        cout << "Age:  " << age  << "\n";
}
const int size=3;
int main()
{
        employee manager[size];
        for(int i=0; i<size; i++)
        {
                cout << "\nDetails of manager" << i+1 << "\n";
                manager[i].getdata();
        }
        cout <<  "\n";
        for(i=0; i<size; i++)
        {
                cout << "\nManager" << i+1 << "\n";
                manager[i].putdata();
        }
        return 0;
 }
```

PROGRAM 5.6

This being an interactive program, the input data and the program output are shown below:

```
Interactive input
    Details of manager1
    Enter name: xxx
    Enter age:  45
```

Copyrighted material

```
Details of manager2
Enter name: yyy
Enter age:  37

Details of manager3
Enter name: zzz
Enter age:  50
```

Program output

```
Manager1
Name: xxx
Age:  45

Manager2
Name: yyy
Age:  37

Manager3
Name: zzz
Age:  50
```

5.14 Objects as Function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called *pass-by-value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Program 5.7 illustrates the use of objects as function arguments. It performs the addition of time in the hour and minutes format.

Copyrighted material

OBJECTS AS ARGUMENTS

```
#include <iostream>

using namespace std;

class time
{
        int  hours;
        int  minutes;
   public:
        void gettime(int h, int m)
        {  hours = h; minutes = m; }
        void puttime(void)
        {
              cout << hours << " hours and ";
              cout << minutes << " minutes " << "\n";
        }
        void sum(time, time);  // declaration with objects as arguments
};
void time :: sum(time t1, time t2)          // t1, t2 are objects
{
        minutes = t1.minutes + t2.minutes;
        hours = minutes/60;
        minutes = minutes%60;
        hours = hours + t1.hours + t2.hours;
}
int main()
{
        time T1, T2, T3;

        T1.gettime(2,45);    // get T1
        T2.gettime(3,30);    // get T2

        T3.sum(T1,T2);// T3=T1+T2

        cout << "T1 = "; T1.puttime();      // display T1
        cout << "T2 = "; T2.puttime();      // display T2
        cout << "T3 = "; T3.puttime();      // display T3

        return 0;
}
```

PROGRAM 5.7

Copyrighted material

The output of Program 5.7 would be:

```
T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes
```

note

Since the member function **sum( )** is invoked by the object **T3**, with the objects **T1** and **T2** as arguments, it can directly access the hours and minutes variables of **T3**. But, the members of **T1** and **T2** can be accessed only by using the dot operator (like **T1.hours** and **T1.minutes**). Therefore, inside the function sum(), the variables **hours** and **minutes** refer to **T3**, **T1.hours** and **T1.minutes** refer to **T1**, and **T2.hours** and **T2.minutes** refer to **T2**.

Figure 5.6 illustrates how the members are accessed inside the function **sum()**.

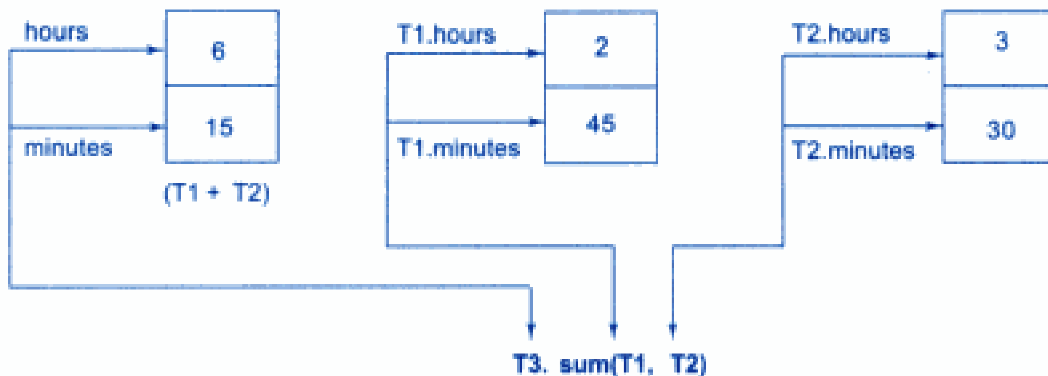


Fig. 5.6 ⇔ *Accessing members of objects within a called function*

An object can also be passed as an argument to a non-member function. However, such functions can have access to the **public member** functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

5.15 Friendly Functions

We have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, **manager** and **scientist**, have been defined. We would like to use a function **income_tax()** to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

Copyrighted material

To make an outside function "friendly" to a class, we have to simply declare this function as a **friend** of the class as shown below:

```
class ABC
{
        .....
        .....
   public:
        .....
        .....
        friend void xyz(void);   // declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope operator ::. The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

The friend functions are often used in operator overloading which will be discussed later.

Program 5.8 illustrates the use of a friend function.

FRIEND FUNCTION

```
#include <iostream>

using namespace std;

class sample
```

(Contd)

Copyrighted material

```
{
        int a;
        int b;
    public:
        void setvalue() {a=25; b=40; }
        friend float mean(sample s);
};
float mean(sample s)
{
        return float(s.a + s.b)/2.0;
}

int main()
{
        sample X;       // object X
        X.setvalue();
        cout << "Mean value = " << mean(X) << "\n";

        return 0;
}
```

PROGRAM 5.8

The output of Program 5.8 would be:

```
Mean value = 32.5
```

note

The friend function accesses the class variables **a** and **b** by using the dot operator and the object passed to it. The function call **mean(X)** passes the object **X** by value to the friend function.

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class X
{
        .....
        .....
        int fun1();         // member function of X
        .....
};

class Y
{
```

Copyrighted material

```
        .....
        .....
        friend int X :: fun1();         // fun1() of X
                                        // is friend of Y
        .....
};
```

The function **fun1()** is a member of **class X** and a **friend** of class **Y**.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class Z
{
        .....
        friend class X;     // all member functions of X are
                            // friends to Z
};
```

Program 5.9 demonstrates how friend functions work as a bridge between the classes.

A FUNCTION FRIENDLY TO TWO CLASSES

```
#include <iostream>

using namespace std;

class ABC;      // Forward declaration
//--------------------------------------------------------//
class XYZ
{
        int x;
  public:
        void setvalue(int i) {x = i;}
        friend void max(XYZ, ABC);
};
//-------------------------------------------------------//
class ABC
{
        int a;
  public:
        void setvalue(int i) {a = i;}
        friend void max(XYZ, ABC);
};
```

(Contd)

Copyrighted material

```
//----------------------------------------------------//
void max(XYZ m, ABC n)      // Definition of friend
{
        if(m.x >= n.a)
                cout << m.x;
        else
                cout << n.a;
}
//----------------------------------------------------//
int main()
{
        ABC abc;
        abc.setvalue(10);
        XYZ xyz;
        xyz.setvalue(20);
        max(xyz, abc);

        return 0;
}
```

PROGRAM 5.9

The output of Program 5.9 would be:

```
20
```

note

The function max() has arguments from both **XYZ** and **ABC**. When the function max() is declared as a friend in **XYZ** for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as

```
class ABC;
```

This is known as 'forward' declaration.

As pointed out earlier, a friend function can be called by reference. In this case, local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

This method can be used to alter the values of the private members of a class. Remember, altering the values of private members is against the basic principles of data hiding. It should be used only when absolutely necessary.

Program 5.10 shows how to use a common friend function to exchange the private values of two classes. The function is called by reference.

SWAPPING PRIVATE DATA OF CLASSES

```
#include <iostream>

using namespace std;

class class_2;

class class_1
{
        int value1;
  public:
        void indata(int a) {value1 = a;}
        void display(void) {cout << value1 << "\n";}
        friend void exchange(class_1 &, class_2 &);
};

class class_2
{
        int value2;
  public:
        void indata(int a) {value2 = a;}
        void display(void) {cout << value2 << "\n";}
        friend void exchange(class_1 &, class_2 &);
};

void exchange(class_1 & x, class_2 & y)
{
        int temp = x.value1;
        x.value1 = y.value2;
        y.value2 = temp;
}

int main()
{
        class_1 C1;
        class_2 C2;

        C1.indata(100);
        C2.indata(200);

        cout << "Values before exchange" << "\n";
        C1.display();
        C2.display();
```

(Contd)

Copyrighted material

```
        exchange(C1, C2);        // swapping

        cout << "Values after exchange " << "\n";
        C1.display();
        C2.display();

        return 0;
}
```

PROGRAM 5.10

The objects x and y are aliases of **C1** and **C2** respectively. The statements

```
int temp = x.value1
x.value1 = y.value2;
y.value2 = temp;
```

directly modify the values of **value1** and **value2** declared in **class_1** and **class_2**.

Here is the output of Program 5.10:

```
Values before exchange
100
200
Values after exchange
200
100
```

5.16 Returning Objects

A function cannot only receive objects as arguments but also can return them. The example in Program 5.11 illustrates how an object can be created (within a function) and returned to another function

RETURNING OBJECTS

```
#include <iostream>

using namespace std;

class complex                   // x + iy form
{
          float x;                    // real part
          float y;                    // imaginary part
     public:
          void input(float real, float imag)
          { x = real; y = imag; }
```

(Contd)

Copyrighted material

```
        friend complex sum(complex, complex);

        void show(complex);
};

complex sum(complex c1, complex c2)
{
        complex c3;              // objects c3 is created
        c3.x = c1.x + c2.x;
        c3.y = c1.y + c2.y;
        return(c3);              // returns object c3
}

void complex :: show(complex c)
{
        cout << c.x << " + j" << c.y << "\n";
}

int main()
{
        complex A, B, C;

        A.input(3.1, 5.65);
        B.input(2.75, 1.2);


        C = sum(A, B);         // C = A + B

        cout << "A = "; A.show(A);
        cout << "B = "; B.show(B);
        cout << "C = "; C.show(C);

        return 0;
}
```

PROGRAM 5.11

Upon execution, Program 5.11 would generate the following output:

```
A = 3.1 + j5.65
B = 2.75 + j1.2
C = 5.85 + j6.85
```

The program adds two complex numbers **A** and **B** to produce a third complex number **C** and displays all the three numbers.

Copyrighted material

5.17 const Member Functions

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows:

```
void mul(int, int) const;
double get_balance() const;
```

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

5.18 Pointers to Members

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name. A class member pointer can be declared using the operator ::* with the class name. For example, given the class

```
class A
{
  private:
     int m;
  public:
     void show();
};
```

We can define a pointer to the member m as follows:

```
int A::* ip = &A :: m;
```

The **ip** pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase **A::*** means "pointer-to-member of **A** class". The phrase **&A::m** means the "address of the m member of A class".

Remember, the following statement is not valid:

```
int *ip = &m;    // won't work
```

This is because **m** is not simply an **int** type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

Copyrighted material

The pointer **ip** can now be used to access the member **m** inside member functions (or friend functions). Let us assume that **a** is an object of **A** declared in a member function. We can access **m** using the pointer **ip** as follows:

```
cout << a.*ip;   // display
cout << a.m;     // same as above
```

Now, look at the following code:

```
ap = &a;                // ap is pointer to object a
cout << ap -> *ip;      // display m
cout << ap -> m; // same as above
```

The *dereferencing operator* ->* is used to access a member when we use pointers to both the object and the member. The *dereferencing* operator.* is used when the object itself is used with the member pointer. Note that ***ip** is used like a member name.

We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the **main** as shown below :

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10)
```

The precedence of () is higher than that of .* and ->*, so the parentheses are necessary.

Program 5.12 illustrates the use of dereferencing operators to access the class members.

DEREFERENCING OPERATORS

```
#include <iostream>

using namespace std;

class M
{
        int x;
        int y;
  public:
        void set_xy(int a, int b)
        {
              x = a;
              y = b;
        }
        friend int sum(M m);
```

(Contd)

Copyrighted material

```
};
int sum(M m)
{
        int M ::* px = &M :: x;
        int M ::* py = &M :: y;
        M *pm = &m;
        int S = m.*px + pm->*py;
        return S;
}

int main()
{
        M n;
        void (M :: *pf)(int,int) = &M :: set_xy;
        (n.*pf)(10,20);
        cout << "SUM = " << sum(n) << "\n";

        M *op = &n;
        (op->*pf)(30,40);
        cout << "SUM = " << sum(n) << "\n";

        return 0;
}
```

PROGRAM 5.12

The output of Program 5.12 would be:

```
sum = 30
sum = 70
```

5.19 Local Classes

Classes can be defined and used inside a function or a block. Such classes are called local classes. Examples:

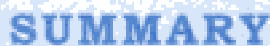
```
void test(int a)          // function
{
      .....
      .....
      class student       // local class
      {
            .....
            .....         // class definition
```

Copyrighted material

```
        .....
    };
    .....
    .....
    student s1(a);          // create student object
    .....                   // use student object
}
```

Local classes can use global variables (declared above the function) and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope operator (::).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.



SUMMARY

- A class is an extension to the structure data type. A class can have both variables and functions as members.
- By default, members of the class are private whereas that of structure are public.
- Only the member functions can have access to the private data members and private functions. However the public members can be accessed from outside the class.
- In C++, the class variables are called objects. With objects we can access the public members of a class using a dot operator.
- We can define the member functions inside or outside the class. The difference between a member function and a normal function is that a member function uses a membership 'identity' label in the header to indicate the class to which it belongs.
- The memory space for the objects is allocated when they are declared. Space for member variables is allocated separately for each object, but no separate space is allocated for member functions.
- A data member of a class can be declared as a **static** and is normally used to maintain values common to the entire class.
- The static member variables must be defined outside the class.
- A static member function can have access to the static members declared in the same class and can be called using the class name.
- C++ allows us to have arrays of objects.

Copyrighted material

- We may use objects as function arguments.
- A function declared as a **friend** is not in the scope of the class to which it has been declared as friend. It has full access to the private members of the class.
- A function can also return an object.
- If a member function does not alter any data in the class, then we may declare it as a **const** member function. The keyword **const** is appended to the function prototype.
- It is also possible to define and use a class inside a function. Such a class is called a local class.

Key Terms

- abstract data type
- arrays of objects
- **class**
- class declaration
- class members
- class variables
- **const** member functions
- data hiding
- data members
- dereferencing operator
- dot operator
- elements
- encapsulation
- **friend** functions
- inheritance
- **inline** functions
- local class
- member functions
- nesting of member functions
- objects
- pass-by-reference
- pass-by-value
- period operator
- **private**
- prototype
- **public**
- scope operator
- scope resolution
- static data members
- static member functions
- static variables
- **struct**
- structure
- structure members
- structure name
- structure tag
- template

Review Questions

5.1 *How do structures in C and C++ differ?*

5.2 *What is a class? How does it accomplish data hiding?*

5.3 *How does a C++ structure differ from a C++ class?*

5.4 *What are objects? How are they created?*

5.5 *How is a member function of a class defined?*

5.6 *Can we use the same function name for a member function of a class and an outside function in the same program file? If yes, how are they distinguished? If no, give reasons.*

5.7 *Describe the mechanism of accessing data members and member functions in the following cases:*

   (a) *Inside the **main** program.*

   (b) *Inside a member function of the same class.*

   (c) *Inside a member function of another class.*

5.8 *When do we declare a member of a class **static**?*

5.9 *What is a friend function? What are the merits and demerits of using friend functions?*

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

   (a) *Data items in a class must always be private.*

   (b) *A function designed as private is accessible only to member functions of that class.*

   (c) *A function designed as public can be accessed like any other ordinary functions.*

   (d) *Member functions defined inside a class specifier become inline functions by default.*

   (e) *Classes can bring together all aspects of an entity in one place.*

   (f) *Class members are public by default.*

   (g) *Friend functions have access to only public members of a class.*

   (h) *An entire class can be made a friend of another class.*

   (i) *Functions cannot return class objects.*

   (j) *Data members can be initialized inside class specifier.*

Debugging Exercises

5.1 Identify the error in the following program.

```
#include <iostream.h>
struct Room
{
        int width;
        int length;
```

Copyrighted material

```
        void setValue(int w, int l)
        {
            width = w;
            length = l;
        }
};
void main()
{
        Room objRoom;
        objRoom.setValue(12, 1,4);
}
```

5.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int width, height;
        void setValue(int w, int h)
        {
            width = w;
            height = h;
        }
};
void main()
{
        Room objRoom;
        objRoom.width = 12;
}
```

5.3 Identify the error in the following program.

```
#include <iostream.h>
class Item
{
private:
        static int count;
public:
        Item()
        {
```

Copyrighted material

```
        count++;
    }
    int getCount()
    {
        return count;
    }
    int* getCountAddress()
    {
        return count;
        }
    };
    int Item::count = 0;

    void main()
    {
        Item objItem1;
        Item objItem2;

        cout << objItem1.getCount() << ' ';
        cout << objItem2.getCount() << ' ';

        cout << objItem1.getCountAddress() << ' ';
        cout << objItem2.getCountAddress() << ' ';
}
```

5.4 Identify the error in the following program.

```
#include <iostream.h>
class staticFunction
{
        static int count;
public:
        static void setCount()
        {
            count++;
        }
        void displayCount()
        {
            cout << count;
```

Copyrighted material

```
        }
};
int staticFunction::count = 10;
void main()
{
        staticFunction obj1;
        obj1.setCount(5);
        staticFunction::setCount();
        obj1.displayCount();
}
```

5.5 Identify the error in the following program.

```
#include <iostream.h>
class Length
{
        int feet;
        float inches;
public:
        Length()
        {
            feet = 5;
            inches = 6.0;
        }
        Length(int f, float in)
        {
            feet = f;
            inches=in;
        }
        Length addLength(Length l)
        {
            l.inches += this->inches;
            l.feet += this->feet;
            if(l.inches>12)
            {
                l.inches-=12;
                l.feet++;
            }
            return l;
```

Copyrighted material

```
        }
        int getFeet()
        {
            return feet;
        }
        float getInches()
        {
            return inches;
        }
};
void main()
{
        Length objLength1;
        Length objLength1(5, 6.5);
        objLength1 = objLength1.addLength(objLength2);
        cout << objLength1.getFeet() << ' ';
        cout << objLength1.getInches() << ' ';
}
```

5.6 Identify the error in the following program.

```
#include <iostream.h>
class Room;
void Area()
{
        int width, height;
        class Room
        {
            int width, height;
            public:
            void setValue(int w, int h)
            {
                width = w;
                height = h;
            }
            void displayValues()
            {
                cout << (float)width << ' ' << (float)height;
```

Copyrighted material

```
            }
        };
        Room objRoom1;
        objRoom1.setValue(12, 8);
        objRoom1.displayValues();
}

void main()
{
        Area();
        Room objRoom2;
}
```

Programming Exercises

5.1 *Define a class to represent a bank account. Include the following members:*

Data members

1. *Name of the depositor*
2. *Account number*
3. *Type of account*
4. *Balance amount in the account*

Member functions

1. *To assign initial values*
2. *To deposit an amount*
3. *To withdraw an amount after checking the balance*
4. *To display name and balance*

Write a main program to test the program.

5.2 *Write a class to represent a vector (a series of float values). Include member functions to perform the following tasks:*

(a) *To create the vector*

(b) *To modify the value of a given element*

(c) *To multiply by a scalar value*

(d) *To display the vector in the form (10, 20, 30, ...)*

Write a program to test your class.

5.3 *Modify the class and the program of Exercise 5.1 for handling 10 customers.*

5.4 *Modify the class and program of Exercise 5.2 such that the program would be able to add two vectors and display the resultant vector. (Note that we can pass objects as function arguments.)*

Copyrighted material

5.5 *Create two classes **DM** and **DB** which store the value of distances. **DM** stores distances in metres and centimetres and **DB** in feet and inches. Write a program that can read values for the class objects and add one object of **DM** with another object of **DB**.*

*Use a friend function to carry out the addition operation. The object that stores the results may be a **DM** object or **DB** object, depending on the units in which the results are required.*

The display should be in the format of feet and inches or metres and centimetres depending on the object on display.

Copyrighted material

6

Constructors and Destructors

Key Concepts

- Constructing objects
- Constructors
- Constructor overloading
- Default argument constructor
- Copy constructor
- Constructing matrix objects
- Automatic initialization
- Parameterized constructors
- Default constructor
- Dynamic initialization
- Dynamic constructor
- Destructors

6.1 Introduction

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **putdata()** and **setvalue()** to provide initial values to the private member variables. For example, the following statement

```
A.input();
```

invokes the member function **input()**, which assigns the initial values to the data items of object **A**. Similarly, the statement

```
x.getdata(100,299.95);
```

passes the initial values as arguments to the function **getdata()**, where these values are assigned to the private variables of object **x**. All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Copyrighted material

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as **class**, that behave very similar to the built-in types. This means that we should be able to initialize a **class** type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

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. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.

6.2 Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
        int m, n;
    public:
        integer(void);          // constructor declared
        .....
        .....
};
integer :: integer(void)        // constructor defined
{
      m = 0; n = 0;
}
```

Copyrighted material

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1;            // object int1 created
```

not only creates the object **int1** of type **integer** but also initializes its data members **m** and **n** to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the *default constructor*. The default constructor for **class A is A::A()**. If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```
A a;
```

invokes the default constructor of the compiler to create the object **a**.

The constructor functions have some special characteristics. These are :

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**. (Meaning of virtual will be discussed later in Chapter 9.)
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.

6.3 Parameterized Constructors

The constructor **integer()**, defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called *parameterized constructors*.

Copyrighted material

The constructor **integer()** may be modified to take arguments as shown below:

```
class integer
{
        int m, n;
   public:
        integer(int x, int y);   // parameterized constructor
        .....
        .....
};
integer :: integer(int x, int y)
{
        m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);                    // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program 6.1 demonstrates the passing of arguments to the constructor functions.

Copyrighted material

CLASS WITH CONSTRUCTORS

```
#include <iostream>

using namespace std;

class integer
{
        int m, n;
    public:
        integer(int, int);              // constructor declared

        void display(void)
        {
             cout << " m = " << m  << "\n";
             cout << " n = " << n  << "\n";
        }
};

integer :: integer(int x, int y)        // constructor defined
{
        m = x;  n = y;
}

int main()
{
     integer int1(0,100);               // constructor called implicitly

     integer int2 = integer(25, 75);    // constructor called explicitly

     cout << "\nOBJECT1" << "\n";
     int1.display();

     cout << "\nOBJECT2" << "\n";
     int2.display();

     return 0;
}
```

PROGRAM 6.1

Program 6.1 displays the following output:

```
OBJECT1
m = 0
n = 100
```

Copyrighted material

```
OBJECT2
m = 25
n = 75
```

The constructor functions can also be defined as **inline** functions. Example:

```
class integer
{
      int m, n;
   public:
      integer(int x, int y)    // Inline constructor
      {
             m = x; y = n;
      }
      .....
      .....
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
       .....
       .....
    public:
       A(A);
};
```

is illegal.

However, a constructor can accept a *reference* to its own class as a parameter. Thus, the statement

```
Class A
{
      .....
      .....
   public:
      A(A&);
};
```

is valid. In such cases, the constructor is called the *copy constructor*.

Copyrighted material

6.4 Multiple Constructors in a Class

So far we have used two kinds of constructors. They are:

```
integer();              // No arguments
integer(int, int);      // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from **main()**. C++ permits us to use both these constructors in the same class. For example, we could define a class as follows:

```
class integer
{
        int m, n;
    public:
        integer(){m=0; n=0;}            // constructor 1
        integer(int a, int b)
        {m = a; n = b;}                 // constructor 2
        integer(integer & i)
        {m = i.m;  n = i.n;}            // constructor 3
};
```

This declares three constructors for an **integer** object. The first constructor receives no arguments, the second receives two **integer** arguments and the third receives one integer object as an argument. For example, the declaration

```
integer I1;
```

would automatically invoke the first constructor and set both **m** and **n** of **I1** to zero. The statement

```
integer I2(20,40);
```

would call the second constructor which will initialize the data members **m** and **n** of **I2** to 20 and 40 respectively. Finally, the statement

```
integer I3(I2);
```

would invoke the third constructor which copies the values of **I2** into **I3**. In other words, it sets the value of every data element of **I3** to the value of the corresponding data element of **I2**. As mentioned earlier, such a constructor is called the *copy constructor*. We learned in Chapter 4 that the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Copyrighted material

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)

Copyrighted material

```
        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.

Copyrighted material

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.

Copyrighted material

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)

Copyrighted material

```
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
```

Copyrighted material

```
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;
```

Copyrighted material

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

Copyrighted material

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)

Copyrighted material

```
        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
```

Copyrighted material

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)

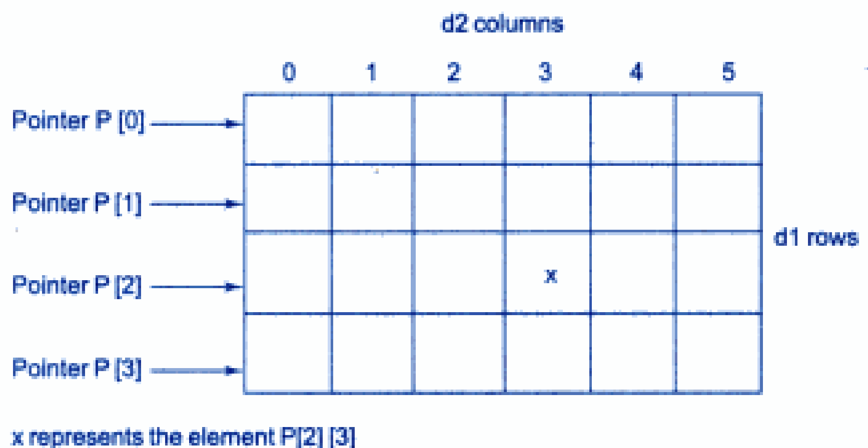
Copyrighted material

```
            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.

Copyrighted material

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<d1; 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.

Copyrighted material

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

Copyrighted material

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.

Copyrighted material

⇔ 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.

Copyrighted material

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:
```

Copyrighted material

```
        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;
```

Copyrighted material

```
            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)
        {
```

Copyrighted material

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

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

int Room::copyConsCount = 0;

void main()
{
        Room objRoom1;
        Room objRoom2 (objRoom1);
        Room objRoom3 = objRoom1;
        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 s1; // 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 = s1;
```

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.*

Copyrighted material

(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.*

Copyrighted material

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

Copyrighted material

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:

Copyrighted material

```
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

Copyrighted material

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.

OVERLOADING UNARY MINUS

```
#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)

Copyrighted material

```
        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.

Copyrighted material

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)

Copyrighted material

```
}
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
```

Copyrighted material

We know that a member function can be invoked only by an object of the same class. Here, the object **C1** takes the responsibility of invoking the function and **C2** plays the role of an argument that is passed to the function. The above **invocation** statement is equivalent to

```
C3 = C1.operator+(C2);          // usual function call syntax
```

Therefore, in the **operator+()** function, the data members of **C1** are accessed directly and the data members of **C2** (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x = x + c.x;
```

c.x refers to the object **C2** and x refers to the object **C1. temp.x** is the real part of **temp** that has been created specially to hold the results of addition of **C1** and **C2**. The function returns the complex temp to be assigned to **C3**. Figure 7.1 shows how this is implemented.

As a rule, in overloading of binary operators, the *left-hand* operand is used to invoke the operator function and the *right- hand* operand is passed as an argument.

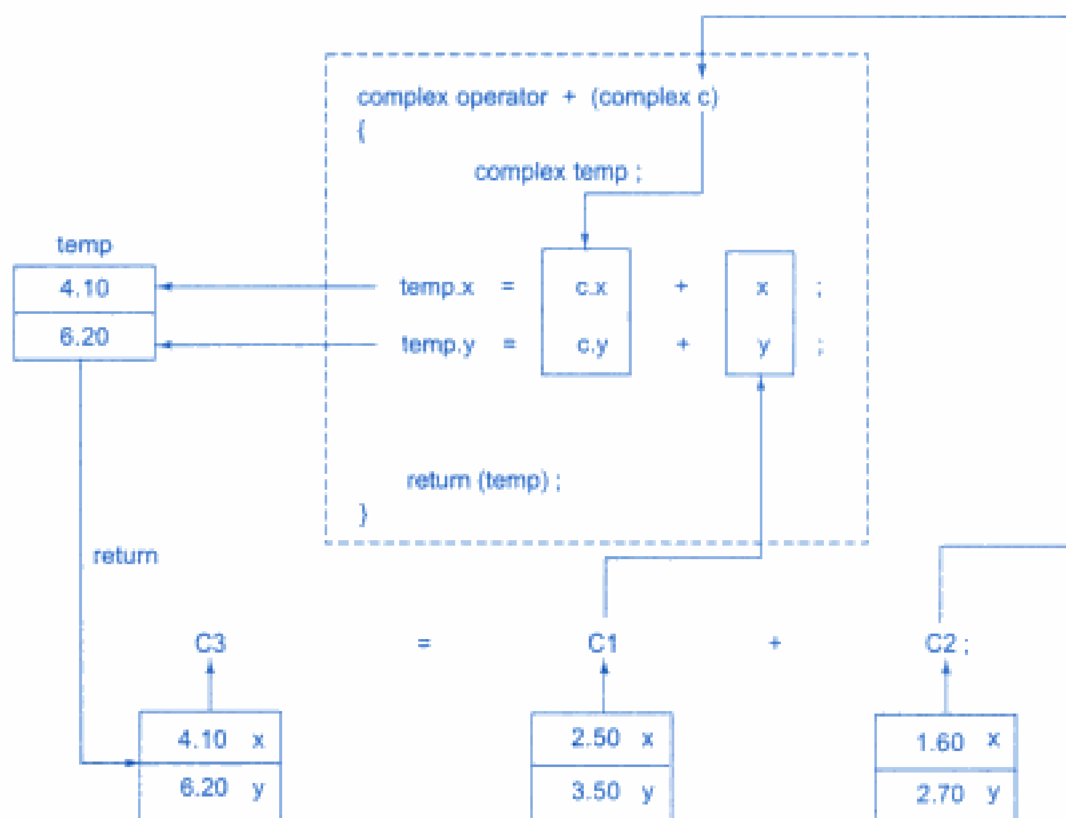


Fig. 7.1 ⇔ *Implementation of the overloded + operator*

Copyrighted material

We can avoid the creation of the **temp** object by replacing the entire function body by the following statement:

```
return complex((x+c.x),(y+c.y));      // invokes constructor 2
```

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using *temporary objects* can make the code shorter, more efficient and better to read.

7.5 Overloading Binary Operators Using Friends

As stated earlier, **friend** functions may be used in the place of member functions for overloading a binary operator, the only difference being that a **friend** function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a **friend** operator function as follows:

1. Replace the member function declaration by the **friend** function declaration.
   ```
   friend complex operator+(complex, complex);
   ```
2. Redefine the operator function as follows:
   ```
   complex operator+(complex a, complex b)
   {
         return complex((a.x+b.x),(a.y+b.y));
   }
   ```

In this case, the statement

```
C3 = C1 + C2;
```

is equivalent to

```
C3 = operator+(C1, C2);
```

In most cases, we will get the same results by the use of either a **friend** function or a member function. Why then an alternative is made available? There are certain situations where we would like to use a **friend** function rather than a member function. For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built-in type data as shown below,

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

Copyrighted material

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)

Copyrighted material

```
        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
```

Copyrighted material

```
        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)

Copyrighted material

```
        if(t1 <= t3)
        {
                show(t1);
                cout << " smaller than ";
                show(t3);
                cout << "\n";
        }
        else
        {
                show(t3);
                cout << " smaller than ";
                show(t1);
                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?

Copyrighted material

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

Copyrighted material

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.

Copyrighted material

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;
```

Copyrighted material

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()
```

Copyrighted material

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.

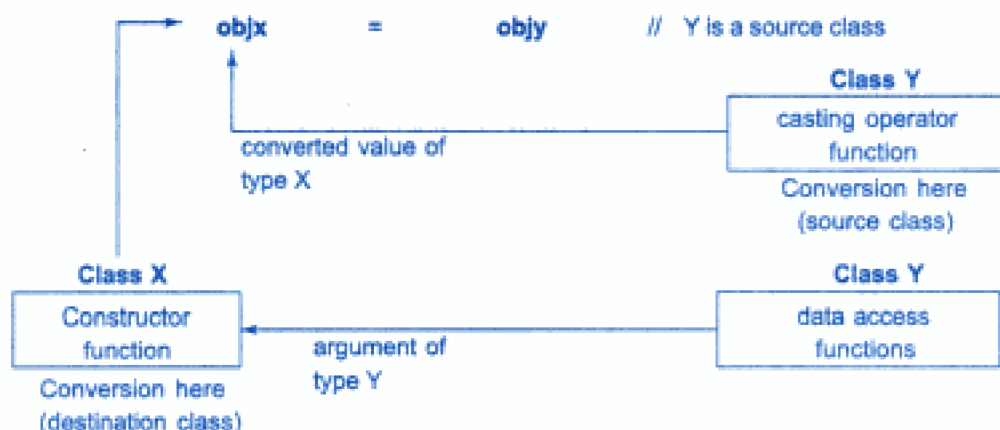


Fig. 7.2 ⇔ *Conversion between object*

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.

Copyrighted material

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)

Copyrighted material

```
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

Copyrighted material

Following is the output of Program 7.5:

```
Product details-invent1 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 **invent1** to convert the **invent1** type data to a **float**. The constructor

```
invent2 (invent1)
```

is used in the class **invent2** to convert the **invent1** type data to the **invent2** type data.

Remember that we can also use the casting operator function

```
operator invent2()
```

in the class invent1 to convert **invent1** 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 (::)

Copyrighted material

- 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?*

Copyrighted material

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++;
```

Copyrighted material

```
            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;
```

Copyrighted material

```
}
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()
```

Copyrighted material

```
{
    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);
```

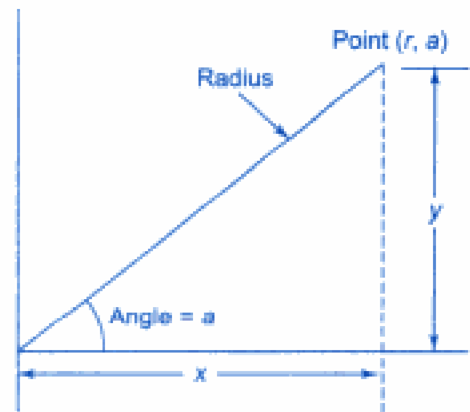


Fig. 7.3 ⇔ Polar coordinates of a point

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.*

Copyrighted material