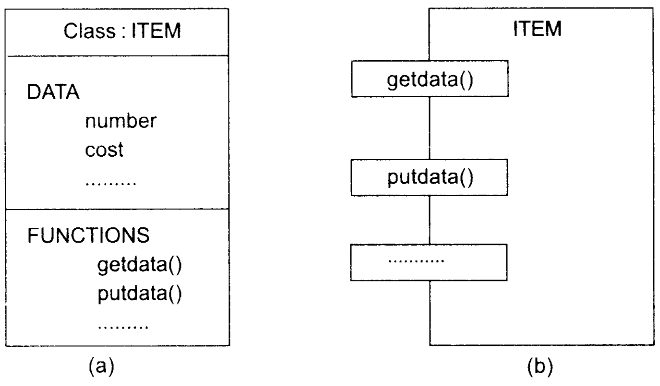

Fig. 5.2 ⇔ *Representation of a class*

```
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:

```
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
     .....
     .....
```

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;
}
```

```
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.

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

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
};
```

```
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)

```
        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
```

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)

```
{
        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)

```
        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

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)

```
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 **itemPrice[count]**. Note that inside this function **count**

```
        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.

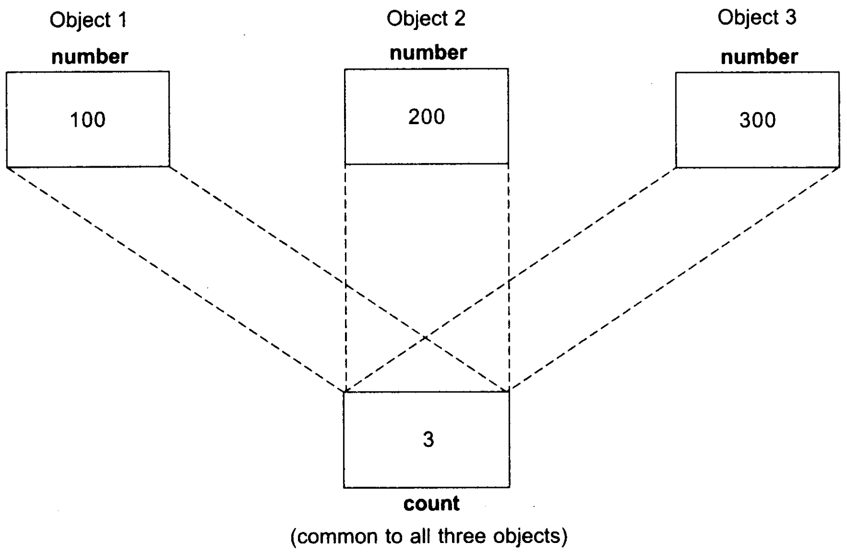

Fig. 5.4 ⇔ *Sharing of a static data member*

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.

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.

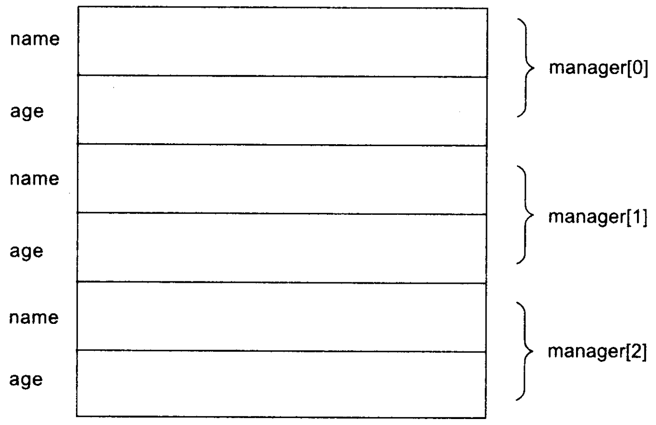

Fig. 5.5 ⇔ *Storage of data items of an object array*

Program 5.6 illustrates the use of object arrays.

ARRAYS OF OBJECTS

```
#include <iostream>

using namespace std;

class employee
```

(Contd)

```
{
        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
```

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

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

Program output

```
        Managerl
        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.

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

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()**.

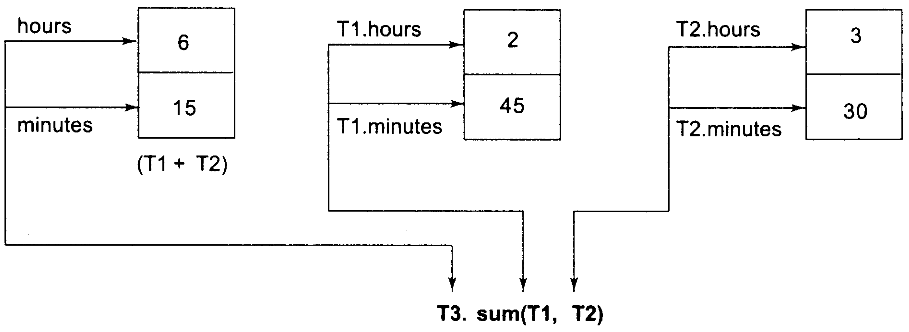

Fig. 5.6 ⇔ *Accessing members of objects within a called function*

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.

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)

```
{
        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 funl();          // member function of X
        .....
};

class Y
{
        .....
        .....
        friend int X :: funl();          // funl() of X
                                         // is friend of Y
        .....
    };
```

The function **funl()** 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)

```
//-------------------------------------------------//
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)

```
        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)

```
          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.

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.

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)

```
};
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
```

```
            .....
        };
        .....
        .....
        student sl(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.

⇔ 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;
        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()
        {
                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;
        }
};
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;
        }
        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;
```

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.

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;
}
```

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
```

```
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*.

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.