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:

```c
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:

```c
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:

```c
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:
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 \textit{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.

\textbf{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. \textit{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 \texttt{struct} can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

\begin{verbatim}
    student A;     // C++ declaration
\end{verbatim}

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as \texttt{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.

\textbf{note}

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

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:

```cpp
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.
A Simple Class Example

A typical class declaration would look like:

```cpp
class item
{
    int number;
    float cost;
    // variables declaration
    // private by default

    public:
    void getData(int a, float b);
    // functions declaration
    // using prototype
    void putData(void);
    }:// 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,
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

```java
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

```java
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

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

has no meaning. Similarly, the statement

```java
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,

```java
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:

```java
class xyz
{
    int x;
    int y;
    public:
    int z;
};
```

```java
......
......
```

```java
xyz p;
p.x = 0;     // error, x is private
p.z = 10     // OK, z is public
......
......
```
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:

```c
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 \textit{friend} function discussed later.)
- A member function can call another member function directly, without using the dot operator.

\textbf{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:

\begin{verbatim}
class item
{
    int number;
    float cost;

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

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 \textit{inline} function are also applicable here. Normally, only small functions are defined inside the class definition.

\section{5.5 A C++ Program with Class}

All the details discussed so far are implemented in Program 5.1.
```cpp
#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;
}
```
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

```cpp
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.

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

Here is the output of Program 5.1:

```plaintext
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:

```cpp
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;
}

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 \texttt{read()} can be called by the function \texttt{update()} to update
the value of \texttt{m}.

\begin{verbatim}
void sample :: update(void)
{
    read();    // simple call; no object used
}
\end{verbatim}

\section*{5.9 \hspace{1em} Arrays within a Class}

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

\begin{verbatim}
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);
};
\end{verbatim}

The array variable \texttt{a[ ]} declared as a private member of the class \texttt{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 \texttt{setval()} sets the values of
elements of the array \texttt{a[ ]}, and \texttt{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.

\begin{verbatim}
#include <iostream>
using namespace std;
const m=50;
class ITEMS
{
\end{verbatim}

(Contd)
```c
{    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"
        << "\n1 : Add an item ";
        cout << "\n2 : Display total value"
        << "\n3 : Delete an item"
        << "\n4 : Display all items"
        << "\n5 : Quit"
        << "\n\nWhat is your option?";
        cin >> x;
        switch(x)
        {
            case 1 : order.addItem(); 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;
}
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

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
is incremented after the assignment operation is over. The function \texttt{displaySum( )} first evaluates the total value of the order and then prints the value. The fourth function \texttt{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 \texttt{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.
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.

```cpp
#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)
```
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  \(\Rightarrow\) 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.

```cpp
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:

  ```cpp
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.
#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;
}
Output of Program 5.5:

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

Note that the statement

```cpp
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:

```cpp
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:

```cpp
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:

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

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

will display the data of the i-th 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](image)

Program 5.6 illustrates the use of object arrays.

```cpp
#include <iostream>
using namespace std;

class employee
```
{  
    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;
}

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 manager 2
Enter name: yyy
Enter age: 37

Details of manager 3
Enter name: zzz
Enter age: 50

Program output
Manager 1
Name: xxx
Age: 45

Manager 2
Name: yyy
Age: 37

Manager 3
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.
#include <iostream>

using namespace std;

class time
{
    int hours;
    int minutes;

public:
    void getline(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;

    T1getline(2,45); // get T1
    T2getline(3,30); // get T2

    T3 sums(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;
}
The output of Program 5.7 would be:

\[ T1 = 2 \text{ hours and 45 minutes} \]
\[ T2 = 3 \text{ hours and 30 minutes} \]
\[ T3 = 6 \text{ hours and 15 minutes} \]

\textbf{note}

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

Figure 5.6 illustrates how the members are accessed inside the function \texttt{sum()}.

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

\section{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, \texttt{manager} and \texttt{scientist}, have been defined. We would like to use a function \texttt{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:

```cpp
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.

```cpp
FRIEND FUNCTION

#include <iostream>

using namespace std;

class sample

(Contd)
```
```cpp
{
    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;
}
```

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:

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

class 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:

```cpp
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**

```cpp
#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)
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

```cpp
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.
#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;

    C1ADATA(100);
    C2ADATA(200);

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

    (Cont'd)
The objects x and y are aliases of C1 and C2 respectively. The statements

```cpp
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.

```cpp
#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; }

    // Other methods...

    // Example:
    static complex add(complex a, complex b)
    { return complex(a.x + b.x, a.y + b.y); }

    // Usage:
    complex c = add(x, y);
}
```

(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;
}

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:

```cpp
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

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

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

```cpp
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:

```cpp
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:

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

Now, look at the following code:

```cpp
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:

```cpp
(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**

```cpp
#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)
```
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:

```cpp
void test(int a) // function
{
    
    
    class student // local class
    {
        
        // class definition
    }
```
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 binding
- 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