UNIT 8 : INTRODUCTION TO CLASSES AND OBJECT

UNIT STRUCTURE

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- identify the basic components of a class
- define a class and create objects
- define member function of a class
- describe array of objects
- use objects as function arguments

8.2 INTRODUCTION

So far, we have learnt that C ++ lets you create variables which can be of a range of basic data types : *int, long, double* and so on. However, the variables of the basic type do not allow you to model realworld objects (or even imaginary objects) adequately. We come to know that the basic theme of the object oriented approach is to model the real –world problems. So, object oriented programming language C++ introduces a new data type called ***class*** by which you can define your own data types as *class*. Defining the variables of a class data type is known as a class instantiation and such variables are called ***objects***. In this unit, we will introduce you how to declare a class and how to create objects of a class.We will also discuss how a member function declare inside a class or outside a class and how it can be accessed. Moreover, the techniques of passing objects as function arguments are also illustrated in this unit.

8.3 CLASSES IN C++

We are already familiar with the term encapsulation which is a fundamental feature of OOP. The encapsulation is nothing but a mechanism that binds together the data and functions into a single component, and keeps both safe from outside interference and misuse.

The data cannot be accessible by outside functions. With encapsulation data hiding can be accomplished.

In C++, the encapsulation is supported by a construct called- "*class*". First, let us think a bit about-what is an object . From the general concept, we can say that an object is something that has fixed shape or well defined boundary. In other words, an object can be a person, a place, or a thing with which the computer must deal. If you look at your surroundings some objects may correspond to real-world entities such as– student, bank account, book, cars, bags, computer, lock, watch etc. You will observe two characteristics about objects–

(i) each objects has certain attributes.

(ii) each objects has some behaviours or actions or operations associated with it.

For example, the objects 'computer' & 'car' have the following attributes and operations–

- Object : car

  Attributes: company, model, colour, & capacity

  Operation: speed ( ), average ( ), & break ( )

- Object: Computer

  Attributes: brand, price, monitor resolution, hard disk and RAM size

  Operation: Processing( ), display( ) & printing ( )

Each object will have its own identity though its attributes and operation are same, the objects will never become equal. In case of person object, for instance, two person have the same attributes like name, age and sex, but they are not equal. Objects are the basic run time entities in an object – oriented system. Thus, in C++, an object is a collection of **related variables** and **function** bound together to form a high level entity. Thus,

- The variable defines – the state of the object
- function defines – the action or operation that can be performed on the object.

Now, let us come back to the discussion of **class**.

A **class** is a grouping of objects having identical attributes and common behaviour (operations). It means all objects possessing similar attributes or properties are grouped into the same unit which is called a class. A class encloses both the data and function into a single unit as shown in the following Figure 8.1

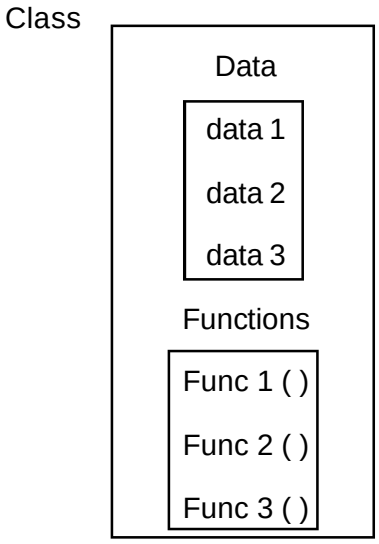
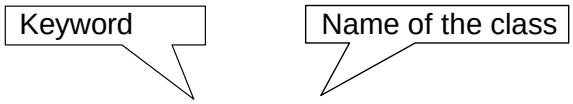


Figure 8.1 Grouping of data and function in a class

Thus, the entire group of data and code of an object can be built as a user-define data type using class. It is obvious that classes are the basic language construct in C++ for creating the user defined data types. Once a class has been declared, the programmer can create a number of objects associated with that class. Objects are nothing but the variable of the class data type. Defining variables of a class data type is known as a **class instantiation.** The syntax used to create an object of the class data type is similar to the syntax used to create an integer variable in C. In the next section, we will learn how to declare a class and an object.

8.4 CLASS DECLARATION

We have come to know that classes contain not only data but also functions. Data and functions within a class are called members of a class. The data inside a class is called a **data member** and the functions are called **member function**. The member functions define the set of operations that can be performed on the data member of a class. The syntax of a class declaration is shown below—




```
class <Class Name>
    {
    private :
        data member;
        member function;
    public:
        data member;
        member function;
    };
```

The keyword **'class'** indicates the name which follows class name, is an abstract data type. The declaration of a class is enclosed with curly braces and terminated by a semi-colon. The body of a class contains declaration of data members and member functions.

The members of a class are usually grouped in two sections i.e. *private* and *public*, which define the visibility of members.

The following declaration illustrates a class specification:

```
class employee
    {
    Private :
        char name[30];
        float age;
    Public :
        void insert_data (void);
        void show_data (void);
    };
```

A class name should be meaningful, reflecting the information it holds. Here in our example, the class 'employee' contains two data members and two member functions. The member function *insert_data( )* is used to assign value to the member variable or data member 'name' and 'age'. The member function *show_data( )* is used to display the values of the

data members. The data member of the class ‘employee’ cannot be accessed by any other functions that are defined outside the class. It means only the member functions of a class are permitted to access its data members.

In general, the data members are declared as private and member functions are declared as public. In our example, though, we have not specifed the data members as private; yet they are treated as private by default.

Figure 8.2 represent the class ‘employee’.

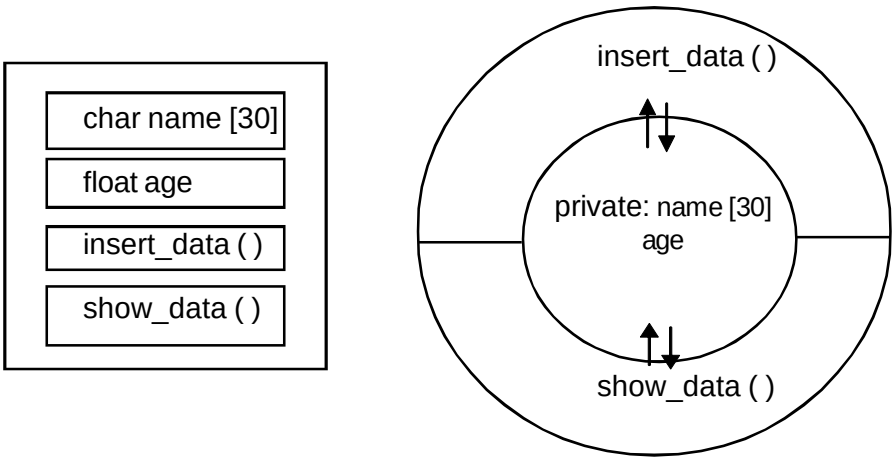


Figure 8.2 Representation of ‘employee’ class

8.4.1 Access Control in a Class

The members of a class are generally grouped into three sections by using the following keyword–

- private
- public
- protected

These keywords are called ***access control specifiers***. These control specifiers are written inside the class and terminated by this **‘:’**

symbol. All the members that follow a keyword (upto another keyword) belong to that type. If no keyword is specified, then the members are assumed as private member. We will discuss later about the protected keyword. Let us now briefly discuss about private and public keywords.

- **Private**: Private members are accessible to their own class members only. They cannot be accessed from outside the class by any member functions. The members at the beginning of class without any access specifier are private by default. Hence, writing the keyword 'private' at the beginning of a class is optional.

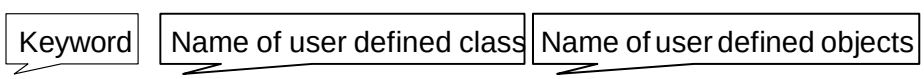
- **Public:** Public members are visible (accessible) outside the class, they should be declared in public section. All data members are not only accessible to their own members of a class but also can be accessible from anywhere in the program, either by functions that belong to the class or by those external to the class.

8.5 DECLARING OBJECTS

A class declaration only builds the structure of objects. In our example, the class *employee* does not define any objects of employee but only specifies what it will contain. Once a class has been declared, we can create variable of that type by using the class name (like any other built-in type variable). The process of creating objects (variables) of the class is called *class **instantiation***. For example–

```
int x, y, z;        // declaration of integer variables
float m, n; // declaration of float variables
employee a; // declaration of object or class variable
```

The syntax for creating objects are shown below :



class classname object name, ……

Remember, the use of the keyword *'class'* is optional at the time creating objects.

For example, ***class employee e1,***

or

employee e1;

In a single statement we can create more than one object as shown below.

employee e1, e2, e3, e4;

As in the case of structures, we can create objects by placing their names immediately just after the closing braces as follows –

```
class employee
    {
        // body of the class
    } e1, e2, e3;
```

This practice is rarely followed because we would like to define the objects as and when required, or at the point of their use.

Always remember that, at the time of declaration of object, necessary memory space is allocated for an object. Suppose, we have declared two objects as –

employee e1, e2;

Both ***e1*** and ***e2*** have the same data members and it is illustrated by the following figure.

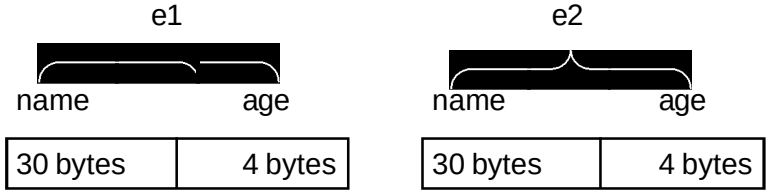


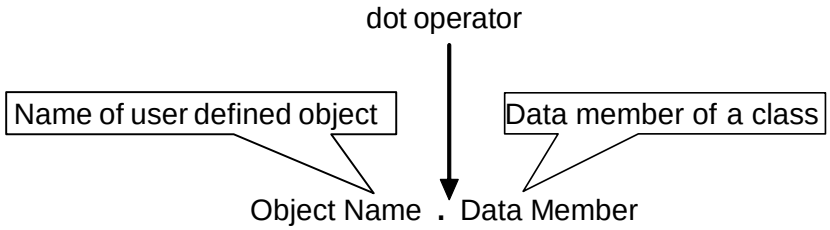
Fig. 8.3 : Allocation of memory for objects

Here, in the figure the objects **e1** and **e2** occupy the memory area. They are not initialized to anything; however the data members of each object will simply contain junk values. So, our main task is to access the data members of the object and setting them to some specific values.
An object is a conceptual entity having the following properties:

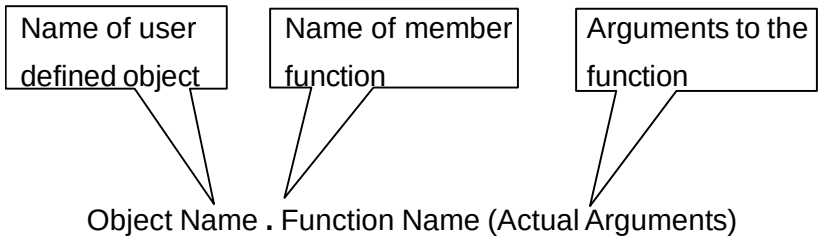
- it is individual
- it points to a thing, either physical or logical that is identifiable by the user.
- it holds data as well as operation method that handles data.
- its scope is limited to the block in which it is defined.

8.5.1 Accessing Class Members

After creating an object of a class, it is the time to access its members. This is the achieved by using the member access operator, dot (.). The syntax for accessing members (data and functions) of a class is shown below–



(a) Accessing data member of a class



(b) Accessing member function of a class

The following program demonstrates how the objects are used for accessing the class data members.

//Program 8.1

```
# include <iostream.h>
# include <string.h>
# include <conio.h>
class employee
{
    private:
    char name [30];          // name of an employee
    float age; // age of an employee
    public :   // initializing data members
    void insert_data (char * name1, float age)
    { strcpy (name, name1);
          age = age1;
    }
    void show_data ( )      //displaying the data members
    {
      cout << "Name :"<<name<<endl;
      cout << "Age :"<<age<<endl;
    }
};
void main ( )
{
    employee e1; //first object of class employee
    employee e2; //second object of class employee
    clrscr( );
    e1.insert_data("Hemanga",30);
    //e1 calls member insert_data ( )
    e2.insert_data ("Prakash",32);
    //e2 calls member insert_data ( )
    cout << "Employee Details:"<<endl;
    e1.show_data ( ); // e2 calls member show_data (
)
    e2.show_data ( ); // e2 calls member show_data (
)
      getch ( );
    }
```

OUTPUT : Employee Details:

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

In the above program, in main( ) we have declared two objects through the statements

```
employee e1;    and  employee e2;
```

The statements

```
e1.insert_data ("Hemanga", 30);
e2.insert_data ("Prakash", 32);
```

initialize the data members of object e1 and e2. The object e1's data member 'name' is assigned 'Hemanga' and age is assigned 30. Similarly, the object e2's data member 'name' is assigned 'Prakash' and age is assigned 32.

The statements

```
e1.show_data ( );
e2.show_data ( );
```

call their member show_data ( ) to display the contents of data members namely, 'name' and 'age' of employee objects e1 and e2. The two objects e1 and e2 will hold different data values.

CHECK YOUR PROGRESS

1. Answer the following by selecting the appropriate option:

   a) The members of a class are by default.

   (i) Private (ii) Public
   (iii) Protected (iv) None of these

   b) The private data of any class is accessed by -

   (i) Only public member function
   (ii) Only private member function

(iii) Boths (i) & (ii) (iv) None of these

c) Encapsulation means

(i) Protecting data (ii) Allowing global access

(iii) Data hiding (iv) Both (i) & (iii)

d) The size of object is equal to

(i) Total size of member data variables

(ii) Total size of member functions

(iii) Both (i) & (ii) (iv) None of these

e) In a class, only member function can access data which is not accessible to out side. This feature is called

(i) data security (ii) data hiding

(iii) data manipulation (iv) data definition

8.6 DEFINING MEMBER FUNCTIONS

We have already come to know that a class holds both the data and functions which are called *data members and member functions*. The data member of a class must be declared within the body of the class. The member functions of a class can be defined in two places

- inside the class definition
- outside the class definition

It is obvious that a function should perform the same task, no matter where it is defined. But the syntax of a member function definition changes depending on the place of its definition, i.e. inside a class or outside a class. We will now discuss both the approaches.

8.6.1 Member Function Inside a Class

In this method, the function is defined inside the class body. When a function is defined inside a class, it is treated as an **inline** function. We will discuss inline function in the next section.

In the program 8.1 we have defined the member functions–

```
void insert_data (char * name1, float age);
```

and

void show_data ( );

inside the class 'employee'. We have seen that these function definition are similar to the normal function definition except that they are enclosed within the body of a class. They will be treated as an inline function. Remember that if a function contains loop instruction i.e. *for, while do, while* ...etc. then that function will not be treated as inline function.

8.6.2 Member Function Outside a Class

In this method of defining a member function outside a class - first you will have to declare a function prototype within the body of the class and then define the function outside the body of the class.

The function defined outside the body of a class have the same syntax as normal functions i.e. they should have a function header and a function body. But, there must have a mechanism of binding the functions to the class to which they belong. This is done by using the **scope resolution operation (: :)**, in the header of the function. The scope resolution operator acts as an 'identity-label' and tells the compiler the class to which the function belongs. The general form of a member function definition is shown below–

ClassName

```
{  .....
   .....
   ReturnType MemberFunction (arguments); // Function Prototype
   .....
   .....
} ;
```

Return Type ClassName : : MemberFunction (arguments)

```
{
      // body of the function
  }
```

Here, the label **ClassName : :** tells the compiler that the function **MemberFunction** is the member of class **ClassName**. The scope of the function is restricted to only the objects and other members of the class. We can modify the Program 8.1 by defining the member functions outside the class body, as shown below:

//Program 8.2

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
    private:
      Char name [30];
      float age;
    public:
      void insert_data (char *name1, float age1);
      void show_data ( );// Member Function
};
void employee::insert_data (char *name1, float age1)
{
    strcpy (name, name1);
    age = age1;
}
void employee ::show_data ( )// Function definition
{
    cout <<“Name :”<<name<<endl;
    cout <<“Age:”<<age<<endl;
}
void main ( )
{
    employee e1, e2;
    clrscr( );
    e1 . insert_data (“Hemanga”, 30);
    e2 . insert_data (“Prakash”, 32);
    cout<<“EMPLOYEE DETAILS....”<<endl;
    e1 . show_data ( );
          e2 . show_data ( );
          getch ( );
      }
```

OUTPUT : EMPLOYEE DETAILS:

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

In the above definitions, the label '***employee : :***' informs the compiler that the functions insert_data ( ) and show_data ( ) are the members of the employee class.

The member functions have some special characteristics. There are–

- A program can have several different classes and they can use the same name for different member functions. The 'membership label' (ClassName : :) resolves the ambiguity of the compiler in deciding which function belong' to which class.
- Member functions can access the private data of the class, whereas non-member functions are not allowed to access. But 'friend function' can access the private data member of a class we will discuss later the friend functions.
- o Member functions of the same class can access all other members of their own class without the use of dot operator.

8.7 INLINE MEMBER FUNCTION

Let us first see what an inline function is. C++ provides a mechanism called *inline function*. When a function is declared as inline, function body is inserted in place of function call during compilation. In this mechanism, passing of control between *caller* and *callee* functions is avoided. The use of the inline function is most effective when calling function is small. If the calling function is very large, in such a case also, compiler copies the content of the inline function in the called function which reduces the program's execution speed. So, in such a case inline function should not be used.

Now, let us see how an inline function behaves with class specification. We have come to know that we can define a member function outside the class definition. The same member function can be defined as inline function by just using the qualifier inline in the header line of the function defining. In the followng, the syntax for defining inline function outside the class declaration is shown

Keyword indicates function defined outside a class body is inline

inline ReturnType ClassName : : FunctionName (arguments)

In fact, the inline mechanism reduces overhead relating to accessing the member function. It provides better efficiency and allows quick execution of functions. An inline member function is treated like a **macro** i.e. any call to this function in a program is replaced by the function itself. This is known as *inline expansion*. Inline functions are also called open subroutines because their code is replaced at the place of function call in the caller function. The normal functions are known as closed subroutines because when such functions are called, the control passes to the function. By default, all member functions defined inside the class are inline function.

We can modify the *Program 8.1* by defining the member functions as inline function as shown below:

```
    // Program 8.3
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
    private:
      char name [30];
      float age;
    public :
      void insert_data (char *name1, float age1);
      void show_data ( );
```

```
} ;
inline void employee::insert_data(char *name1, float age)
{
     strcpy (name, name1);
     age = age1;
}
inline void employee : : show_data ( )
{
       cout <<“Name :”<<name <<endl;
       cout <<“Age:” <<age <<endl;
}
void main()
{
       employee e1, e2;
       clrscr( );
       e1.insert_data (“Hemanga”, 30);
       e2.insert_data (“Prakash”, 32);
       cout<< “Employee Details ..” << endl;
       e1.show_data ( );
       e2.show _data ( );
       getch ( );
}
```

OUTPUT :

Employee Details–

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

8.8 ARRAY OF OBJECTS

We know that arrays hold data of similar type. Arrays can be of any data type including user defined data type, created by using *struct, class* etc. We can create an array of variables by using class data type. Such an

array of variables of class data type is also known as an array of objects which handle a group of objects.

Let us consider the following class definition :

```
class employee
{
      private :
      char name [30]
      float age ;
      Public :
      void insert_data (char* name1, float age1);
      void show_data ( );
};
```

Here, the identifier '*employee*' is a user defined data type and can be used to create objects that relate to different categories of employees. For example, the following definition will creates an array of objects of '*employee*' class–

```
employee consultant [30];  // array of cosultant
employee clerk [15];       // array of clerk
employee lecturer [20];    // array of lecturer
```

The array consultant contain 30 objects (consultant), namely, consultant [0], consultant [1], ...consultant [29], of type employee class. Similarly, clerk array contains 15 objects and lecturer array contains 20 objects.

We know that as array elements occupy continuous memory locations like the same way as array of objects occupy contiguous memory locations as shown in the fig. 8.4

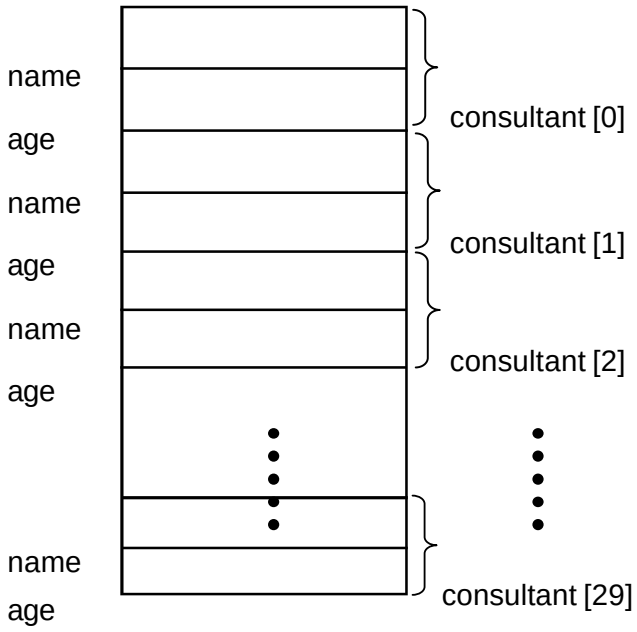


Fig.8.4 Stoage of data items in '*consultant'* array of objects.

By using index an individual element of an array of objects can be referred i.e. ***consultant [15], consultant [9] ..... etc***. By using the dot operator [**.**] we can access any member of an object. For example

consultant [30] . show_data ( )

will display the data of 30th consultant.

We can rewrite the Program 8.1 by using array of objects as shown below:

***//* Program 8.4**

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
    private:
    char name [30];
    float age;
    public:
    void insert_data (char *name1, float age)
```

```
    {
        strcpy (name, name1);
        age = age1;
      }
      void show_data ( )
      {
        cout<<"Name:"<<name<<endl;
        cout<<"Age:"<<age<<endl;
      }
};
void main ( )
{
    int i, age, count;
    char name [30], tag;
    employee consultant [30];
    clrscr ( );
    count = 0;
    for (i=0; i<30; i++)
    {
      cout<<"Enter Data For Employee (Y/N):";
      cin >> tag;
      if (tag = = 'y' || tag = = 'Y')
      {
        cout <<"\n Enter Name of Employee:";
        cin >> name;
        cout << "Age:";
        cin >> age;
        consultant[i].insert_data(name, age);
        count ++;
       }
      else
        break;
    }
    cout <<"\n\n Employee Details ......\n" ;
```

```
    for (i=0; i < count; i ++)
    consultant[i].show_data();
          getch();
    }
```

OUTPUT :

```
Enter Data for Employee (Y/N) : y
Enter Name of Employee        : Prakash
Age                           : 30

Enter Data for Employee (Y/N) : y
Enter Name of Employee        : Hemanga
Age : 28
Enter Data for Employee (Y/N) : n

Employee Details...
Name                          : Prakash
Age                           : 30
Name                          : Hemanga
Age                           : 28
```

8.9 OBJECTS AS FUNCTION ARGUMENTS

Objects can be passed as an argument to a function. There are three ways of passing objects as function arguments:

- a copy of the entire object is passed to the function, which is also called ***pass-by-value***
- only the address of the object is sent implicitly to the function, which is also called – ***pass by-reference***.
- the address of the object is sent explicitly to the function, which is also called – ***pass-by-pointer***.

8.9.1 Pass-by-value

In this technique, a copy of the object is passed to the called function (callee) from the calling function (caller). Since a copy of the object is passed so any changes made to the object inside the called function do

not affect the object used to call the function

The following program demonstrates the use of objects as function arguments in pass-by-value mechanism.

//Program 8.5

```
#include<iostream.h>
#include<conio.h>
class age
{
    private:
    int birthyr ;
    int presentyr ;
    int year ;
    public:
    void getdata ( );
    void period (age);
};
void age : : getdata ( )
{
    cout<<" \ n Year of Birth:";
    cin >> birthyr ;
    cout << "Current year:" ;
    cin >> presentyr ;
}
void age : : period (age x1)
{
    year = x1 . presentyr - x1 . birthyr ;
    cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( ){
    clrscr ( );
    age a1 ;
    a1 . getdata ( );
    a1 . period (a1) ;
    getch( );
```

```
}
```

OUTPUT:
```
Year of Birth      : 1990
Current Year       : 2002
Your Present Age   : 19 years
```

In the above program, the class age has three data member. The function *getdata ( )* reads integers through keyboard. The function *period ( )* calculates the difference between the two integers. In function main ( ), a1 is an object to the class age. The object a1 calls the function *getdata ( )*. The same object a1 is passed to the function *period ( )*, which calculates the difference between the two integers. Thus, an object can be passed to a function.

8.9.2 Pass-by-Reference

In this technique, only the address of the object is sent to the function. When an address of the object is passed, the address acts as reference pointer to the actual object in the calling funciton. Therefore, any change made to the objects inside the called function will reflect in the actual object in the calling funciton. We can modify the program 8.5 by using the pass by reference mechanism

```
// Program 8.6
#include<iostream.h>
#include<conio.h>
class age
  {
      private:
          int birthyr ;
          int presentyr ;
          int year ;
      public:
          void getdata ( );
          void period (age);
```

```
      };
void age : : getdata ( )
{
    cout<<" Year of Birth:";
    cin >> birthyr ;
    cout << "Current year:" ;
    cin >> presentyr ;
}
void age : : period (age & x1)
{
    x1. year = x1 . presentyr - x1 . birthyr ;
    cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( )
{
    clrscr ( );
    age a1 ;
    a1 . getdata ( );
    a1 . period (a1) ;
    getch ( );
}
```

RUN :

Year of Birth	:	1990
Current Year	:	2009
Your Present Age	:	19 years

8.9.3 Pass-by-Pointer

In this mechanism also, the address of the object is passed explicitly to the called function from the calling function. The program 8.6 is modified by using the mechanism pass-by-pointer as follows:

```
//Program 8.7
#include<iostream.h>
#include<conio.h>
class age
{
      private:
          int birthyr ;
          int presentyr ;
          int year ;
      public:
          void getdata ( );
          void period (age * );
};
void age : : getdata ( )
{
      cout<<" \n Year of Birth:";
      cin >> birthyr ;
      cout << "current year:" ;
      cin >> presentyr ;
}
void age : : period (age * x1)
{
    year = x1→presentyr - x1→birthyr ;
    cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( )
{
    clrscr ( );
    age a1 ;
    a1 . getdata ( );
    a1 . period (&a1) ;
    getch ( );
}
```

OUTPUT:
```
Year of Birth      : 1990
Current Year       : 2009
Your Present Age   : 19 years
```

In *Program 8.6* and *Program 8.7* we need to keep an eye on the symbols '.', '$\rightarrow$', '*', '&', and the statements(bold lines) where we have appropriately used them.

8.10 FRIEND FUNCITON AND FRIEND CLASS

We have already discussed the fact that the private members of a class cannot be accessible from the outside of the class. Only the member functions of that class have permission for accessing the private members. This policy enforces the encaptulation and data hiding techniques.

Let us think of a situation where a user need a funciton to operate on objects of two different classes. It means that the function will be allowed to access the private data of both the classes. In C++, this situation is over come by using the concept of friend function. It permits a friend function to access the private members of different classes.

The declaration of a friend function must be prefixed by the keyword "**friend**". In the following class is shown a declaration of a friend function.

```
class test
{
        private:
            ---------
            ---------
        public :
            ---------
            ---------
            friend void sum( ) ;
};
```

The function can be defined anywhere in the program similar to any 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 called friend functions. A friend function can be a friend to a multiple classes. The friend function have the following properties :

- There is no scope restriction for the friend function; hence they can be called directly without using objects.
- Unlike member functions of class, friend function cannot access the members directly. On the other hand, it uses object and dot operator to access the private and public member variables of the class.
- Use of friend function is rarely done, because it violates the rule of encapsulation and data hiding.
- The function can be declared in public or private sections without changing its meaning.

The following program demonstrates the use of friend function:

// Program 8.8

```
#include<iostream.h>
#include<conio.h>
class first ;     /*forward declaration like function Prototype*/
class second
{
    int x ;
    public :
    void get value ( )
    {
      cout << "\n Enter a number :" ;
      cin >> x ;
    }
    friend void sum(second, first);//declaration of friend
dunction
} ;
class first
{
```

```
            int y ;
            public :
            void getvalue ( )
            {
                  cout << "\n Enter a number:" ;
                  cin >> y ;
            }
            friend void sum (second, first) ;
} ;
void sum (second m, first n)
{
    cout << "\n Sum of two numbers :" << n.y + m.x
}
void main( )
{
    clrscr ( );
    first a ;
    second b ;
    a.get value ( ) ;
    b.get value ( ) ;
    sum(b,a);//funciton is called like a general function in C++
}
```

OUTPUT:

```
Enter a number       : 9
Enter a number       : 12
Sum of two numbers   : 21
```

In the above program each of the two classes 'first' and 'second' has a member function named getvalue ( ) and one private data member. Notice that, the function sum ( ) is declared as friend function in both the class. Hence, this function has the ability to access the members of both the classes. Using sum ( ) function, addition of integers is calculated and displayed.

It is possible to declare all the member functions of a class as the friend functions of another class. When all the functions need to access another class in such a situation we can declare an entire class as ***friend class***. Always remember that friendship is not exchangeable. Its meaning is that

- declaring *class A* to be a friend of *class B* does not mean that *class B* is also a friend of *class A*. The declaration of a friend class is as follows:

```
class second
  {
      -------
      -------
      friend class first;
  };   /* all member functions of class first are friends to
          class second */
```

The following program demonstrates the use of friend class :

//Program 8.9

```
#include<iostream.h>
#include<conio.h>
class smallvalue;
class value
{
    int a;
    int b;
    public:
    value (int i, int j)   // declaration of constructor with
arguments
    {
      a = i;
      b = j;
    }
 friend class smallvalue;
};
class smallvalue
{
    public:
    int minimum(value x)
    {
      return x.a < x.b ? x.a : x.b;
    }
```

```
  };
void main ( )
{
        value x (15, 25);
        clrscr ( ) ;
        smallvalue y;
        cout << y. minimum(x);
        getch ( );
}
```

In the above program we have used the constructor with arguments. The concept of constructor is illustrated in unit 9 'Constructors and Destructors'.

CHECK YOUR PROGRESS

2. Fill in the blanks of the following :
   (i) Member functions defined within the class definition are implicity ___.
   (ii) When only the address of the object is sent explicity, it is called ____
   (iii) ____ function can access the private data members of a class.
3. State whether the following statements are true or false:
   (a) To reference an object using a pointer to object, uses the<> operator.
   (b) In the prototype void sum (int &) arguments are passed by reference.
   (c) If class A is a friend class of class B then a member function of class B can access the data members of class A.

8.11 STATIC DATA MEMBER AND MEMBER FUNCTON

After studying public and private members, let us study about the static members of a class. Recall what we have learnt from C Programming:

(i) A variable can be declared as static inside a function or outside main().

(ii) Static variables value do not disappear when function is no longer active; their last updated value always persists. That is, when the control comes back to the same function again the static variables have the same value as they leave at the last time.

in C++ also. However, C++ has objects. Hence, the meaning of static with respect to member variables of an object is different.

We have already gained the idea that each object has its separate set of data member variable in memory. The member functions are created only once and all object share the function. No separate copy of the function of each object is created in the memory like data member variables. Figure 8.5 shows the accessing of member function by objects.

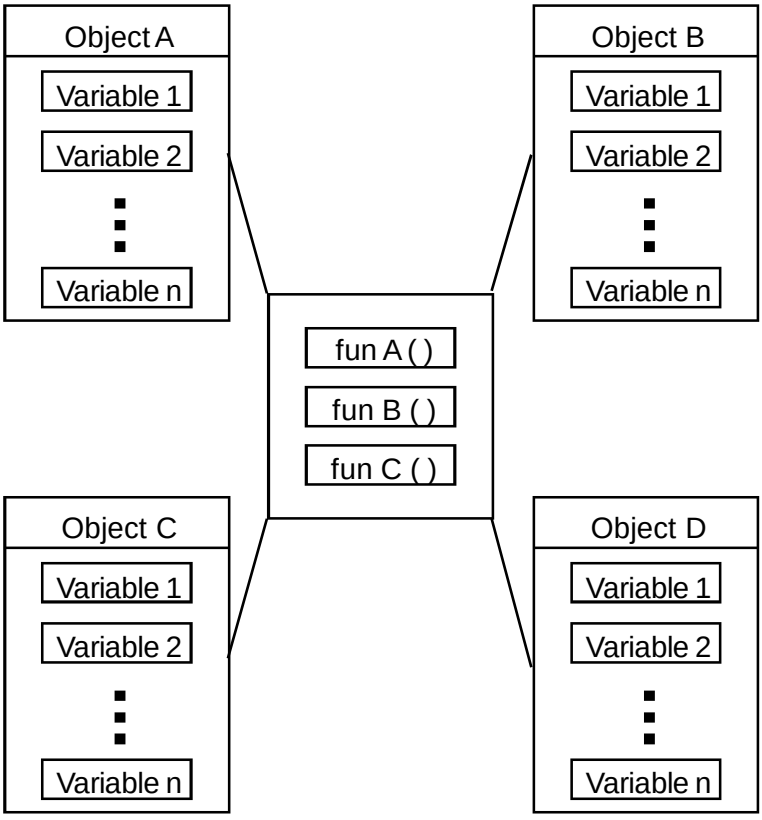


Fig. 8.5 Data members and member functions in memory

In C++, it is possible to create common member variables like function using the **static** keyword. Once a data member variable is declared as static, only one copy of the member is created for the whole class and all objects of the class will share that variable.

Always remember–

- A static variable preserves the value of a variable.
- When a variable is declared as static it is initialized to zero.
- A static data member or member function is only recognized inside the scope of the present class.
- A static variable can be a public or private.

The syntax for declaring static data member or member function within a class is shown below:

static <variable name> ;

static <function name> ;

When you declare a static data member within a class, you are not defining it i.e. you are not allocating storage for it. Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

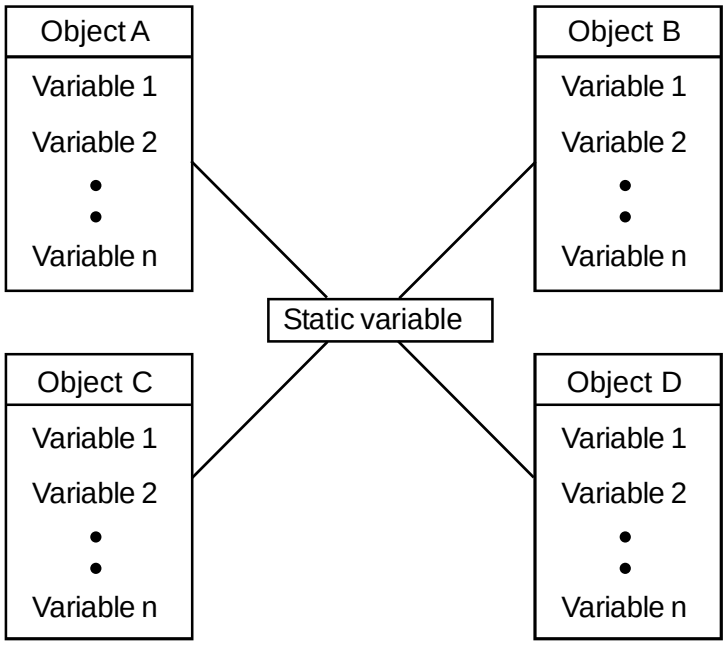


Fig. 8.6 Static member in memory

The declaration of static member is shown below :

```
class number
{
      static int C;
      public:
            ----------
            ----------
};
int number : : C = 0  // initializaiton of static member variable
```

The following program demonstrates the use of static data member in a class–

***//* Program 8.10**

```
#include<stdio.h>
#include<conio.h>
class number
{
      static int C;
      public:
      void count ( )
      {
            C ++;
            cout << "\n C =" << C;
      }
};
int number : : C = 0;
void main( )
 {
      number a, b, c;
      clrscr ( ) ;
      a.count ( );
      b.count ( );
      c.count ( );
      getch ( );
  }
```

OUTPUT: c = 1
c = 2
c = 3

In the above program, the class number has one static data variable C. The *count( )* is a member funciton, increment value of static member variable C by 1 when called. The statement *int number : : C = 0* initiatize the static member with 0. It is possible to initialize the static data members with other values. In the function *main( )*, a, b and c are three objects of *class number*. Each object calls the funciton *count( )*. At each call to the function *count( )* the variable C gets incremented and the count statement displays the value of variable C. The objects a, b and c share the same copy of static data member C.

STATIC MEMBER FUNCTION

In C++, like member variables, funcitons can also be declared as static. When a function is defined as static, it can access only static member variable and functions of the same class. The non-static members are not available to these functions. The static member function declared in public section can be invoked using its class name without using its objects. The static keyword makes the funciton free from the individual object of the class and its scope is global in the class without creating any side effect for other part of the program.

The following points should be remembered while declaring static function:

a) Just one copy of the static member is created in the memory for the entire class. All objects of the class share the same copy of static member.

b) Static member functions can access only static data member, or funcitons.

c) Static member functions can be invoked using class name.

d) It is also possible to invoke static member funcitons using objects.

e) When one of the objects changes the value of data member variables, the effect is visible to all the objects of the class.

The following program demonstrates the use of the static member function in a class.

***//*Program 8.11**

```
#include<iostream.h>
#include<conio.h>
class number
{
    private:
    static int X;
    public:
    static void count ( ) {X++; }
    static void display ( )
    {
          cout << "\n value of X =" << X;
    }
};
int number : : X = 0;
void main ( )
{
clrscr ( ) ;
number::display( );  //invokes display function
number::count( );    //invokes count function
number::count( );    //invokes display function
number::display( );  //invokes display function
getch ( );
}
```

OUTPUT: Value of X : 0
Value of X : 2

In the above program, the member variable X and functions *count ( ) & display ( )* of *class number* are static. The function *count ( )* when called, increases the value of static variable X. The function *display ( )* prints the current value of the variable X. The static functions can be called using class name and scope resolution operator as shown in the program–

number : : count ( );
number : : display ( );

8.12 LET US SUM UP

- Classes are the basic language construct in C++ for creating the user defined data types.
- A class contains member variable or data members and member functions.
- The members of a class are grouped into two sections, namely, private and public.
- Defining variables of a class data type is known as a class instantiation and such variables are called objects.
- Using the member accessed operator, dot(**.**), the class members can be access by the objects.
- The member function can be defined as a) private or public b) inside the class or outside the class.
- The scope resolution operator (**::**) is used, when a member function is defined outside the class body.
- Inline member function is treated like a macro, when a function is declared as inline, function body is inserted in place of function call during compilation.
- We can create an array of variables by using the class data type, then these variables are called array of objects, which occupies contiguous memory locations inmemory.
- There are three methods of passing objects to function, namely, pass-by-value, pass-by-reference, and pass-by-pointer.
- The function that are declared with the keyword ***friend*** are called friend function. A function can be a friend to multiple classes.
- static is the keyword used to preserve value of a variable. When a variable is declared as static, it is initialized to zero. A static function or data element is only recognized inside the scope of the present class.
- When a function is defined as static, it can access only static member variables and functions of the same class. The static member functions are called using its class name without using its objects.

8.13 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

8.14 ANSWERS TO CHECK YOUR PROGRESS

1. a) i. b) iii. c) iv. d) i. e) ii
2. i) inline, ii) pass-by-pointer, iii) friend
3. a) False, b) True, c) False

8.15 MODEL QUESTIONS

1. What is a class ? How does it accomplish data hiding ?
2. What is an object ? How is an object created ?
3. How is a member function of a class defined or declared ?
4. Explain the use of *private* and *public* keywords. How are they different from each other ?
5. What is the significance of scope resolution operator **::** ?
6. When will you make a function inline and why ?
7. Explain the different methods of passing objects to functions.
8. What is a friend function and a friend class ? Explain with example.

****