5.2.1 Giving Values to Members

As the members are not themselves variables they should be linked to the structure variables. The Link between a member and a variable is established using member operator '.' which is also known as dot operator.

This can be explained using following example:

```
e.g.: / * Programme to define a structure and assign value to members * /
        struct book
      {
            char * name;
            int pages;
            char *author;
      };
      main( )
      {
            struct book b1;
            printf ("\n Enter Values:");
            scanf ("%s %d %s", b1.name, &b1.page, b1.author);
            printf ("%s, %d, %s, b1.name, b1.page, b1.author);
      }
```

5.2.2 Creating Structure Variables

The structure declaration does not actually create variables. Instead, it defines data type only. For actual use a structure variable needs to be created. This can be done in two ways:

1. Declaration using tagname anywhere in the programme.

```
e.g.:          struct book
               {
                     char name [30];
                     char author [25];
                     float price;
               }
               struct book book1, book2;
```

2. It is also allowed to combine structure declaration and variable declaration in one statement.

   This declaration is given below:

```
struct person
{
char * name;
int age;
char *address;
```

```
}
p1, p2, p3;
```

While declaring structure variables along with their definition, the use of tag_name is optional.

```
struct
{
        char *name;
        int age;
        char *address;
}
p1, p2, p3;
```

5.2.3 Structure Initialization

A structure variable can be initialized as any other data type.

```
  main( )
{
        static struct
{
  int weight;
  float height;
}
  student = {60, 180.75};
```

This assigns the value 60 to student.weight and 180.75 student.height. There is a one-to-one correspondence between the members and their initializing values.

A structure must be declared as static if it is to be initialized inside a function (similar to arrays). The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
  main( )
{
  struct st_record
  {
        int weight;
        float height;
  }
  static struct st_record student1 = {60, 180.75};
  static struct st_record student2 = {53, 170.60};
  - - - - - -
  - - - - - - -
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record / * No static word */
{
        int weight;
    int height;
}
student1 = {60, 180.75};
```

5.3 NESTED STRUCTURES

Structures within a structure means nesting of structures. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
        char name [20];
        char department [10];
        int basic_pay;
        int dearness_allowance;
        int house_rent_allowance;
        int city_allowance;
}
employee;
```

This structure defines name, department, basic pay and three kinds of allowances. All the items related to allowance can be grouped together and declared under a sub-structure as shown below:

```
struct salary
{
    char name [2];
    char department [10];
    struct
    {
        int dearness;
        int house_rent;
        int city;
    }
    allowance;
}
employee;
```

The salary structure contains a member named allowance which itself is a structure with three members. The members contained in the inner structure, namely, dearness, house_rent and city can be referred to as:

```
employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city
```

Then inner most member in a nested structure can be accessed by chaining all the concerned structure variables (from outermost to inner most) with the member using dot operator.

The following statements are invalid:

```
employee.allowance          (actual member is missing)
employee.house_rent         (inner structure variable is missing)
```

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
{
      struct
      {
            int dearness;
            - - - - -
      }
      allowance, arrears;
}
employee [100];
```

The inner structure has two variables, allowance and arrears. This implies that both of them have the same structure template.

A base member can be accessed as follows:

```
employee[1].allowance.dearness
employee[1].arrears.dearness
```

Tag names can also be used to define inner structures.

```
e.g.:       struct pay
      {
            int dearness;
            int house_rent;
            int city;
      };
      struct salary
      {
            char name [20];
```

```
        char department [10];
        struct pay allowance;
        struct pay arrears;
    }
    struct salary employee [100];
```

The pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure.

It is also permissible to nest more than one type of structures:

```
struct personal_record
{
    struct name_part name;
    struct date date_of_birth;
    - - - - - -
    - - - - - -
};
struct personal_record person1;
```

The first member of the structure is name which is of the type struct name_part. Similarly, other members have their structure types.

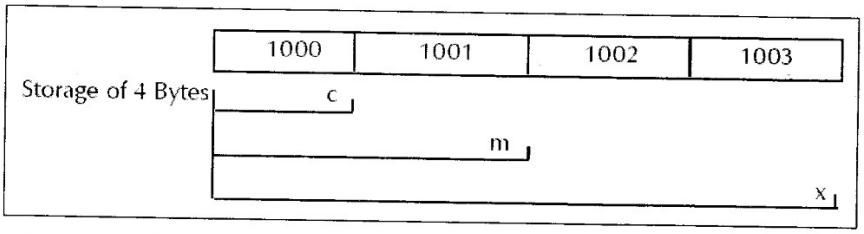
5.4 UNION - DEFINITION AND DECLARATION

Unions follow the same syntax as structures but differ in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time.

Like structures, a union can be declared using the keyword union as follows:

```
union item
{
   int m;
   float x;
   char c;
} code;
```

This declaration declares a variable code of type union item. The union contains three members, each with a different data type. However, only one can be used at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.



The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. As shown in the example declaration, the member x requires 4 bytes which is the largest among the members. It is assumed that a float variable requires 4 bytes of storage and the figure above shows how all the three variables share the same address.

5.4.1 Accessing a Union Member

To access a union member, you can use the same syntax that you use for structure members.

e.g.: code.m, code.x, code.c are all valid member variables.

During accessing, you should make sure that you are accessing the member whose value is currently stored.

For example, the statements such as

```
code.m      = 150;
code.x      = 785;
printf ("%d", code.m);
```

would produce erroneous output (which is machine dependent). The user must keep track of what type of information is stored at any given time.

Thus, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous member's value.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

5.4.2 Union of Structures

Just as one structure can be nested within another, a union too can be nested in another union. Not only that, there can be a union in a structure, or a structure in a union. Here is an example of structures nested in a union.

```
main( )
{
   struct a
   {
      int i;
      char c[2];
   };
   struct b
```

```
    {
      int j;
      char d[2];
    };
    union z
    {
      struct a key;
      struct b data;
    }strange;
    strange.key.i = 512;
    strange.data.d[0] = 0;
    strange.data.d[1] = 32;
    printf("%d\n", strange.key.i);
    printf("%d\n", strange.data.j);
    printf("%d\n", strange.key.c[0];
    printf("%d\n", strange.data.d[0];
    printf("%d\n", strange.key.c[1]);
    printf("%d\n", strange.data.d[1];
}
```

Output:

512
512
0
0
32
32

Structures and unions may be freely mixed with arrays.

e.g.

```
    union id
    {
      char color[12];
      int size;
    };
    struct clothes
    {
      char manufacturer[20];
```

```
    float cost;
    union id description;
} shirt, trouser;
```

Now shirt and trouser are structure variable of type clothes. Each variable will contain the following members: a string (manufacturer), a floating-point quantity (cost), and a union (description). The union may represent either a string (color), or an integer quantity (size). Another way to declare the structure variable shirt and trouser is to combine the above two declarations. This is shown as follows:

```
struct clothes
{
    char manufacturer[20];
    float cost;
    union    {
      char color[12];
      int size;
            } description;
} shirt, trouser;
```

This declaration is more concise, though perhaps less straightforward than the original declarations.

An individual union member can be accessed in the same manner as an individual structure member, using the operators " . " and " -> ". Thus, if variable is a union variable, then variable.member refers to a member of the union. Similarly, if ptvar is a pointer variable that points to a union, then ptvar->member refers to a member of that union.

e.g.:

```
#include <stdio.h>
main()
{
    union id
    {
          char color;
          int size;
    };
    struct
    {
          char manufacturer[20];
          float cost;
          union id description;
    }  shirt, trouser;
    printf("%d\n", sizeof(union id));
    shirt.description.color = ' w ' ; /* assigns a value to color */
```

```
    printf("%c %d\n", shirt.description.color, shirt. description. size);
    shirt.description.size = 12; /* assigns a value to size */
    printf("%c %d\n", shirt.description.color, shirt.description. size);
}
```

5.4.3 Initialization of a Union Variable

A union variable can be initialized, provided its storage class is either external or static. Only one member of a union can be assigned a value at any one time. The initialization value is assigned to the first member within the union.

e.g.:

```
/* Programme to demonstrate initialization of union variables. */
#include <stdio.h>
main( )
{
  union id
  {
        char color[12];
        int size;
  };
  struct clothes
  {
        char manufacturer[20];
        float cost;
        union id description;
  };
  static struct clothes shirt = {"American", "25.00", "White"};
  printf("%d\n", sizeof(union id));
  printf("%s %5.2f", shirt.manufacturer, shirt.cost);
  printf("%s %d\n", shirt.description.color, shirt.description.size);
  shirt.description.size = 12;
  printf("%s %5.2f", shirt.manufacturer, shirt.cost);
  printf("%s %d\n", shirt.description.color, shirt.description.size);
}
```

Output:

```
12
American  25.00  White  26743
American  25.00  ~      12
```

5.5 BITFIELDS

While we're on the subject of structures, we might as well look at bitfields. They can only be declared inside a structure or a union, and allow you to specify some very small objects of a given number of bits in length. Their usefulness is limited and they aren't seen in many programmes, but we'll deal with them anyway. This example should help to make things clear:

```
struct {
      /* field 4 bits wide */
      unsigned field1 :4;
      /*
       * unnamed 3 bit field
       * unnamed fields allow for padding
       */
      unsigned :3;
      /*
       * one-bit field
       * can only be 0 or -1 in two's complement!
       */
      signed field2 :1;
      /* align next field on a storage unit */
      unsigned       :0;
      unsigned field3 :6;
}full_of_fields;
```

Each field is accessed and manipulated as if it were an ordinary member of a structure. The keywords `signed` and `unsigned` mean what you would expect, except that it is interesting to note that a 1-bit signed field on a two's complement machine can only take the values `0` or `-1`. The declarations are permitted to include the `const` and `volatile` qualifiers.

The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files. C gives no guarantee of the ordering of fields within machine words, so if you do use them for the latter reason, you programme will not only be non-portable, it will be compiler-dependent too. The Standard says that fields are packed into 'storage units', which are typically machine words. The packing order, and whether or not a bitfield may cross a storage unit boundary, are implementation defined. To force alignment to a storage unit boundary, a zero width field is used before the one that you want to have aligned.

Be careful using them. It can require a surprising amount of run-time code to manipulate these things and you can end up using more space than they save.

Bit fields do not have addresses—you can't have pointers to them or arrays of them.

5.6 ENUMERATION

It is defined as enum identifier {value$_1$, value$_2$, ... value$_n$};

The identifier is a user defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as enumeration constants).

After this definition, we can declare variable to be of this 'new' type as below:

enum identifier v_1, v_2,...v_n;

The enumerated variables v_1, v_2, -- v_n can only have one of the values value$_1$, value$_2$, —value$_n$.

The assignments v_1 = value$_3$; v_5 = value$_1$; are valid.

```
e.g.: enum day{Monday, Tuesday------Sunday};
      enum day week_st, week_end;
      week_st = Monday;
      week_end = Friday;
      if (week_st == Tuesday) week_end = Saturday;
```

Note: The values that are in original declaration, can only be used.

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value$_1$, is assigned 0, value$_2$ is assigned 1, and so on.

The automatic assignments can be overridden by assigning values explicitly to the enumeration constants.

```
e.g.: enum day {Monday =1, Tuesday, ---Sunday};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement.

```
e.g.: enum day{Monday, --- Sunday}week_st, week_end;
```

Like structures this declaration has two parts:

(a) The first part declares the data type and specifies its possible values. These values are called 'enumerators'.

(b) The second part declares variables of this data type.

Check Your Progress

Fill in the blanks:

1. A structure is a collection of .....
2. A structure variable can be .....
3. Unions follow the same syntax as .....

5.7 LET US SUM UP

Structure is a derived data type used to store the instances of variables of different data types. Structure definition creates a format that may be used to declare structure variables in the programme later on. The structure operators like dot operator "." are used to assign values to structure members. Structure variable can be initialized as any other data type. An array of structure can be declared as any other array. In such an array, each element is a structure. Structures may contain arrays as well as structures. Union is a memory location that is shared by two or more variables. When union variable is declared, compiler automatically allocates enough storage to hold to largest member of union. Only the unions with storage class external or static can be initialized. Unions are useful for applications involving multiple members. They are also used in many DOS based application softwares.typedef and enum are two user defined data types.

5.8 KEYWORDS

Structure: A structure is a collection of variables referenced under one name providing a convenient means of keeping related information together.

Union: Unions follow the same syntax as structures but differ in terms of storage.

Bitfields: The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files.

5.9 QUESTIONS FOR DISCUSSION

What will be the output:

1.
```
main( )
    {
        struct
        {
            int i;
        }
        xyz;
        (&xyz) -> i = 10;
        printf ("%d", xyz.i);
    }
```

2.
```
main( )
    {
        struct
        {
            int i;
        }

        *xyz;
        (&*xyz)->i = 10;
        printf ("%d", xyz ->i);
    }
```

3.
```
main( )
   {
         struct xyz
         {
               int i;
         }
         struct xyz *p;
         struct xyz a;
         p = &a;
         p -> i = 10;
         printf ("%d", xyz.i);
   }
```

4.
```
main( )
   {
         struct xyz
         {
               int xyz;
         };
         struct xyz xyz;
         xyz.xyz = 10;
         printf ("%d", xyz.xyz);
   }
```

5. Is there any error Yes/No
```
   main( )
   {
         struct xyz
         {
               int i;
         }
         *pqr;
   }
```

6.
```
main( )
   {
         struct xxx
         {
               int i;
               char j;
         };
         struct xxx zzz = {I, 'a'];
         abc (zzz);
   }
         abc (struct xxx aaa)
         {
         printf ("%d. . . %d", aaa.i, aaa.j);
   }
```

5.
```
main( )
   {
         union a
         {
         int i;
               char ch[2];
```

```
        };
        union a z1 = {512};
        union a z2 = {0, 2};
    }
```

6.
```
main( )
    {
        union
        {
            int i;
            char j;
        }
        xyz;
        xyz. i = 300;
        printf("%d", xyz.j);
    }
```

7.
```
main( )
    {
        union
        {
            union
            {
                char a;
                char b;
                char c;
                char d;
            }
            car;
            union
            {
                char i;
                char j;
            }
            in;
            char z;
    }
        pqr;            printf ("%d", sizeof (pqr));}
```

Check Your Progress: Model Answer

1. Variables
2. initialized as any other data type
3. structures but differ in terms of storage

5.10 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo C++*, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with C++*, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference C++*, Tata Mc Graw Hill

LESSON

6

CLASSES AND OBJECTS

CONTENTS

6.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of declaration of class
- Discuss member functions
- Describe the object of a class
- Access a member of a class
- Identify and explain the arrays of class objects
- Discuss the pointers and classes
- Explain the concept of union and classes
- Discuss the constructors and destructors
- Explain the inline member functions
- Identify the static class members
- Define friend functions
- Discuss the dynamic memory allocations

6.1 INTRODUCTION

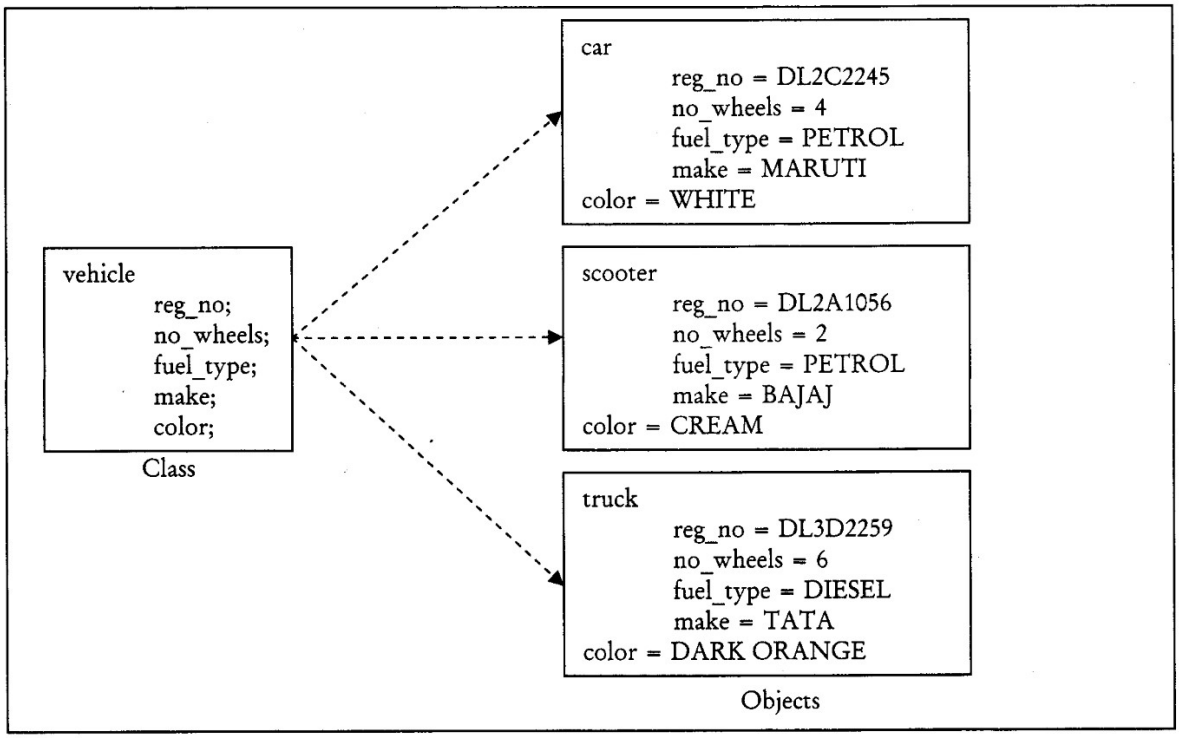
As has been explained earlier at various places, classes and objects are at the core of object-oriented programming in general and programming C++ in particular. Writing programmes in C++ essentially means writing classes and creating objects from them. In this lesson, we will learn to work with the same.

It is important to note the subtle differences between a class and an object, here. A class is a template that specifies different aspects of the object it models. It has no physical existence. It occupies no memory. It only defines various members (data and/or methods) that constitute the class.

An object, on the other hand, is an instance of a class. It has physical existence and hence occupies memory. You can create as many objects from a class once you have defined a class.

You can think of a class as a data type; and it behaves like one. Just as a data type like *int*, for example, does not have a physical existence and does not occupy any memory until a variable of that type is declared or created; a class also does not exist physically and occupies no memory until an object of that class is created.

To understand the difference clearly, consider a class of vehicle and a few objects of this type as depicted below:



In this example, vehicle is a class while car, scooter and truck are instances of the class vehicle and hence are objects of vehicle class. Each instance of the class vehicle - car, scooter and truck - are allocated individual memory spaces for the variables - reg_no, no_wheels, fuel_type, make and, color - so that they all have their own copies of these variables.

When an object is created all the members of the object are allocated memory spaces. Each object has its individual copy of member variables. However, the data members are not initialized automatically. If left uninitialized these members contain garbage values. Therefore, it is important that the data members are initialized to meaningful values at the time of object creation. Conventional methods of initializing data members have lot of limitations. In this lesson, we will learn alternative and more elegant ways initializing data members to initial values.

When a C++ programme runs it invariably creates certain objects in the memory and when the programme exits the objects must be destroyed so that the memory could be reclaimed for further use.

C++ provides mechanisms to cater to the above two necessary activities through constructors and destructors methods.

6.2 DECLARATION OF A CLASS

Like structures a *class* is just another derived data-type. While structure is a group of elements of different data-type, class is a group of elements of different data-types and functions that operate on them. C++ structure can also have functions defined in it.

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.

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

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented. The general form of a class declaration is:

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

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 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.

The 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.

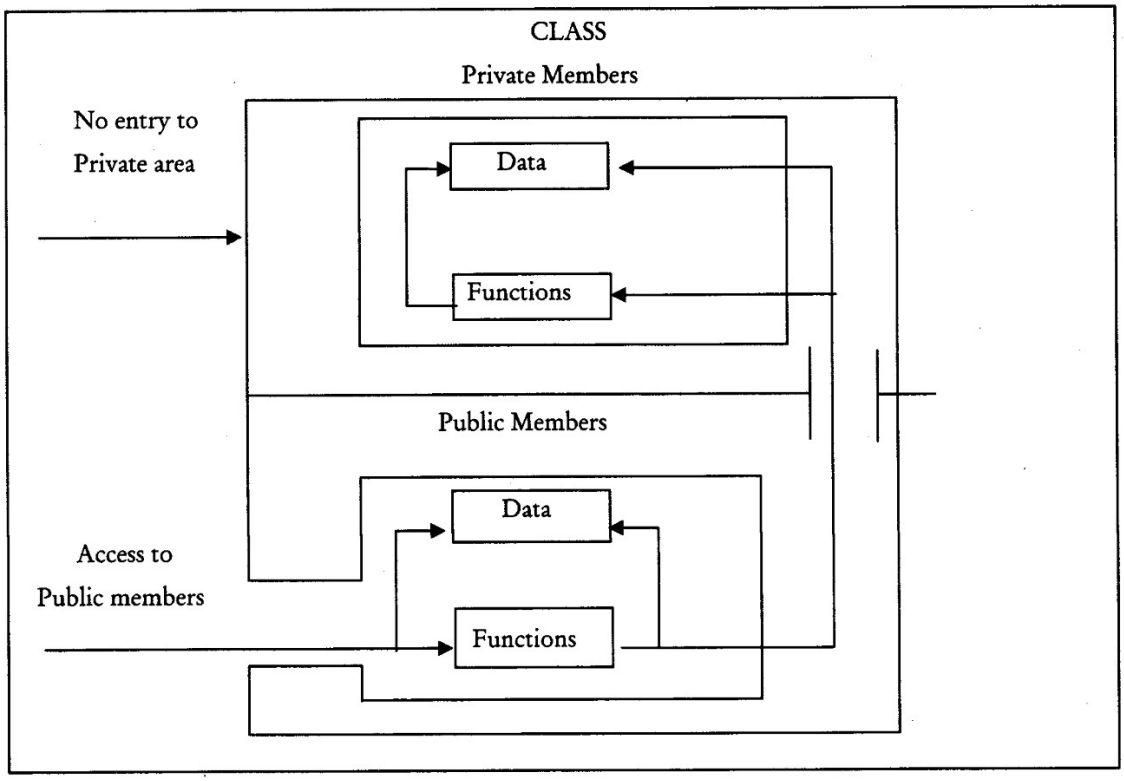


Figure 6.1: Data Access in Class

The variables declared inside a 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. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*. The access to private and public members of a class is well explained diagrammatically in the figure.

Let us consider the following declaration of a class for student:

```
class student
{
  private:
  int rollno;
  char name [20];
  public:
  void getdata(void);
  void disp(void);
};
```

The name of the class is *student*. With the help of this new type identifier, we can declare instances of class student. The data members of this class are *int rollno* and *char name [20]*. The two function members are *getdata()* and *disp()*. Only these functions can access the data members. Therefore, they

provide the only access to the data members from outside the class. The data members are usually declared as private and member functions as public. The member functions are only declared in the class. They shall be defined later.

A class declaration for a machine may be as follows:

```
class machine
    {
      int totparts, partno;
      char partname [20];
      public:
      void getparts (void);
      void disp(part_no);
    };
```

Having defined the class, we need to create object of this class. In general, a class is a user defined data type, while an object is an instance of a class. A class provides a template, which defines the member functions and variable that are required for objects of a class type. A class must be defined prior to the class declaration.

The general syntax for defining the object of a class is :

```
class<class_name>
{
    private:
      //data
      // functions
    public:
      //functions
};
<class_name> object1, object2,... objectN;
```

where object1, object2 and objectN are the instances of the class <class_name>.

A class definition is very similar to a structure definition except the class definition defines the variables and functions.

Consider the following programme segment which declares and creates a class of objects.

```
class student
{
private:
    int rollno;
    int age;
    float height;
    float weight;
public:
```

```
    void getinfo( );
    void disinfo( );
    void process( );
    void personal( );
};
student std;        //std is the object of the class student
```

In another example, the employee details such as name. code, designation, address, salary age can be grouped as follows.

```
class employee
{
private:
char name[20];
    int code;
    char designation[20];
    char address[30];
    float salary;
    int age;
public:
    void salary( );
    void get_info( );
    void display_info( );
};
```

employee x,y; //creates x and y - two objects of the class employee

By now it should be clear to you that you can use a class just as you use a data type. In fact you can also create an array of objects of a particular class.

6.3 MEMBER FUNCTIONS

We have learnt to declare member functions. Let us see how member functions of a class can be defined within a class or outside a class.

A member function is defined outside the class using the :: (double colon symbol) scope resolution operator. The general syntax of the member function of a class outside its scope is:

<return_type> < class_name>:: <member_function> (arg1, arg2....argN)

The type of member function arguments must exactly match with the types declared in the class definition of the <class_name>. The Scope resolution operator (::) is used along with the class name in the header of the function definition. It identifies the function as a member of a particular class. Without this scope operator the function definition would create an ordinary function, subject to the usual function rules of access and scope.

The following programme segment shows how a member function is declared outside the class declaration.

```
class sample
{
private:
    int x;
    int y;
public:
    int sum ();          // member function declaration
};
int sample:: sum( )      //member function definition
{
 return (x+y);
}
```

Please note the use of the scope operator double colon (::) is important for defining the member functions outside the class declaration. Let us consider the following programme snippet:

```
class first
{
private:
    int x;
inty;
public:
    int sum();
};
class second
{
private:
    intx;
    int y;
public:
    int sum();
};
first one;
second two;
int sum()
//error, scope of the member function is not defined
{
    return (x+y);
}
```

Both classes in the above case are defined with the same member function names. While accessing these member function, it gives an error. The scope of the member function *sum( )* is not defined. When accessing the member function *sum( )*, control will be transferred to both classes *one* and *two*. So the scope resolution operator (::) is absolutely necessary for defining the member functions outside the class declaration.

```
int one:: sum( )            // correct
{
return (x+y);
}
```

6.4 DEFINING THE OBJECT OF A CLASS

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data; they may also represent user-defined data such as vectors, time and lists.

They occupy space in memory that keeps its state and is operated on by the defined operations on the object. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other data or code.

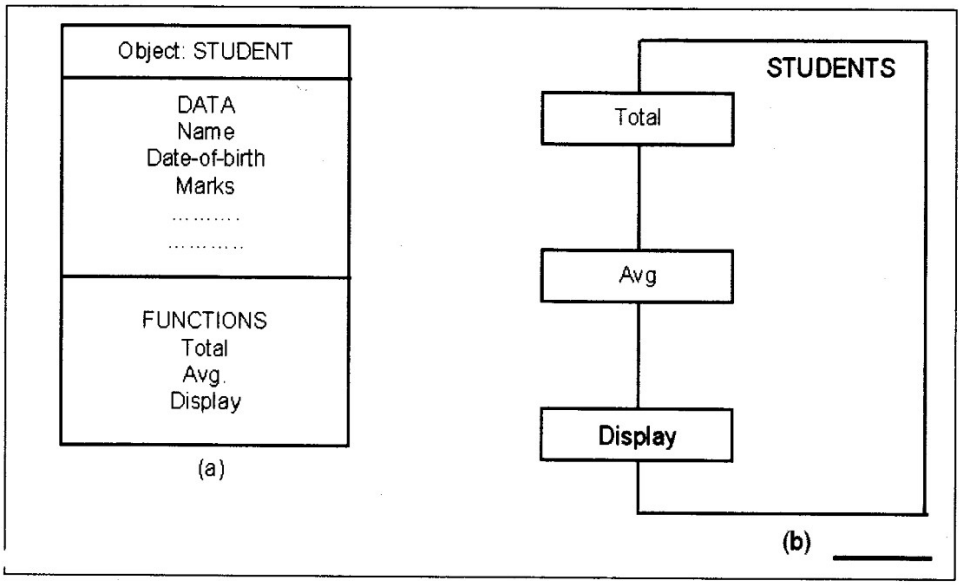


Figure 6.2: Example of an Object

6.4.1 Objects as Function Arguments

Like any other data type argument, objects can also be passed to a function. As you know arguments are passed to a function in two ways:

- by value
- by reference

Objects can also be passed as arguments to the function in these two ways. In the first method, a copy of the object is passed to the function. Any modification made to the object in the function does not affect the object used to call the function. The following programme illustrates the calling of functions by value. The programme declares a class *integer* representing a integer variable *x*. Only the values of the objects are passed to the function written to swap them.

```
#include<iostream.h>
#include<conio.h>
class integer
{
    int x;
public:
    void getdata()
{
      cout << "Enter a value for x";
      cin >> x;
}
    void disp()
{
      cout << x;
}
    void swap(integer a1 , integer a2)
{
      int temp;
      temp = a2.x;
      a2.x = a1.x;
      a1.x = temp;
}
};
main()
{
    integer int1, int2;
    int1.getdata();
    int2.getdata();
    cout <<"\nthe value of x belonging to object int1 is ";
    int1.disp();
    cout <<"\nthe value of x belonging to object int2 is ";
    int2.disp();
    integer int3; .
int3.swap(int1, int2); //int3 is just used to invoke the function
```

```
    cout <<" \nafter swapping ";
    cout <<"\nthe value ofxbelonging to object intI is ";
    intl.disp();
    cout <<"\nthe value of x belonging to object int2 is ";
    int2.disp() ;
    cout<< "\n";
    getche();
}
```

You should get the following output.

Enter a value for x 15

Enter a value for x 50

the value of x belonging to object int is 15

the value of x belonging to object int2 is 50

after swapping

the value of x belonging to object int is 15

the value of x belonging to object int2 is 50

In the second method, the address of the object is passed to the function instead of a copy of the object. The called function directly makes changes on the actual object used in the call. As against the first method any manipulations made on the object inside the function will occur in the actual object. The second method is more efficient as only the address of the object is passed and not the entire object.

The following programme illustrates calling of a function by reference. The programme declares a class *integer* to represent the variable *x* and defines functions to input and display the value. The function written to swap the integer values of the object takes the addresses of the objects.

```
#include<iostream.h>
#include<conio.h>
class integer
{
    int x;
public:
    void getdata()
    {
      cout << "Enter a value for x";
      cin >> x;
}
void disp()
{
```

```
        cout << x;
}
    void swap(integer *a1)
{
        int temp;
        temp = x;
        x = a1->x;
        a1->x = temp;
}
};
main()
{
    integer int1, int2;
    int1.getdata();
    int2.getdata();
    cout <<"\nthe value of x belonging to object int1 is ";
    int1.disp();
    cout <<"\nthe value of x belonging to object int2 is ";
    int2.disp();
    int1.swap(&int2);
    cout << "\nafter swapping ";
    cout <<"\nthe value of x belonging to object int1 is ";
    int1.disp();
    cout <<"\nthe value of x belonging to object int2 is ";
    int2.disp();
    cout << "\n";
    getche();
}
```

You should see the following output.

Enter a value for x 15

Enter a value for x 50

the value of x belonging to object int1 is 15

the value of x belonging to object int2 is 50

after swapping

the value of x belonging to object int1 is 50

the value of x belonging to object int2 is 15

6.4.2 Returning Objects

Just as a function takes an object as its argument, it can also return an object. The following program illustrates how objects are returned. The program declares a class *integer* representing an integer variable *x* and defines a function to calculate the sum of two integer values. This function finally returns an object which stores the sum in its data member x.

```
#include<iostream.h>
#include<conio.h>
class integer
{
int x;
public:
void getdata(int x1)
{
    x = x1;
}
    void disp()
    {
       cout << x;
}
integer sum(integer int2)
{
       integer int3;
       int3.x = x + int2.x;
       return(int3);
}
};
main()
{
    integer int1, int2, int3;
    int1.getdata(15);
    int2.getdata(25);
    cout<<"\nthe value of x for object int1 ";
    int1.disp();
    cout<<"\nthe value of x for object int2 ";
    int2.disp(); .
cout<<"\n the sum of private data values of x belonging to objects int1 and
int2 is ";
    int3 = int1.sum(int2);
```

```
    int3.disp();
    getche();
}
```

You should see the following output from the program.

The value of x for object int1 15

The value of x for object int2 25

The sum of private data values of x belonging to objects int1 and int2 is 40

6.5 ACCESSING A MEMBER OF CLASS

Member data items of a class can be static. Static data members are data objects that are common to all objects of a class. They exist only once in all objects of this class. The static members are used when the information is to be shared. They can be public or private data. The main advantage of using a static member is to declare the global data which should be updated while the program lives in memory.

When a static member is declared private, the non-member functions cannot access these members. But a public static member can be accessed by any member of the class. The static data member should be created and initialized before the main function control block begins.

For example, consider the class account as follows:

```
class account
{
private:
    int acc _no;
    static int balance;              //static data declaration
public:
    void disp(int acc _no);
    void getinfo( );
};
```

The static variable *balance* is initialized outside main() as follows:

```
int account::balance = 0;                    //static data definition
```

Consider the following programme which demonstrates the use of static data member *count*. The variable *count* is declared static in the class but initialized to 0 outside the class.

```
#include <iostream.h>
#include <conio.h>
class counter
{
private:
static int count;
public:
```

```
void disp();
};
int counter::count = 0;
void counter::disp()
{
    count++
    cout << "The present value of count is " << count << "\n";
}
main()
{
    counter cntl ;
    for(int i=0; i<5; i++)
      cntl.disp();
    getche();
}
```

You should get the following output from this programme.

The present value of count is 1

The present value of count is 2

The present value of count is 3

The present value of count is 4

The present value of count is 5

6.6 ARRAYS OF CLASS OBJECTS

An array is a user defined data type whose members are of the same type and stored in continuous memory locations. Just as one can create an array of any basic data type, one can also create arrays of objects.

The general syntax of the array of objects of a class is:

```
class class_name
{
private:
      // data
      // functions
public:
      //data
      // functions
};
class_name object [MAX]
```

where, MAX is a user defined size of the array of class objects e.g. 20 as shown in example given below.

```
class employee
{
private:
char name[20];
    int code;
    char designation[20];
    char address[30];
    float salary;
    int age;
public:
    void salary( );
    void gecinfo( );
    void display_info( );
};
employee obj[200]
```

The following programme illustrates the use of array of objects to store information of students. The programme declares a class *student* having data members - *rollno*, *age*, *height* and *weight*, and two functions to get and display this information. Since information of more than one student is stored, an array of objects that is *std[max]* is created.

```
//array of class objects
#include<iostream.h>
#include<conio.h>
#define max 30
class student
{
private:
int rollno;
    int age;
    float height, weight;
public:
void getinfo( )
{
    cout << "roll no: ";
    cin >>  rollno;
    cout << "age: ";
    cin >> age;
```

```
    cout << "Height: ";
    cin >> height;
    cout << "Weight: ";
    cin >> weight;
}
    void disinfo( )
{
    cout<<endl;
    cout<<"Roll no ="<< rollno<< endl;
    cout<<"Age ="<< age << endl;
    cout<<"Height =" << height << endl;
    cout<<"Weight = " << weight << endl;.
}
};
void main( )
{
    student std[max];    // arrary of objects having max = 30
    int i, n;
    cout << "How many students? \n" << endl;
    cin >> n;
    cout << "enter the student details \n" << endl;
    for(i=0; i < n; ++i)
{   .
      cout << endl;
      std[i].getinfo();
}
    cout <<" The list of student's is as follows \n";
    for (i = 0; i < n; ++i)
      std[i].disinfo();
getche();
}
```

You should see the following when you run the programme.

How many students?

3

roll no: 1

age: 18

Height: 134

Weight: 45

Roll no : 4

age: 20

Height: 143

Weight: 46

Roll no: 27

age: 20

Height: 147

Weight: 50

The list of student's is as follows:

Roll no= 1

Age = 18

Height = 134

Weight = 45

Roll no = 4

Age = 20

Height = 143

Weight = 46

Roll no = 27

Age = 20

Height = 147

Weight = 50

6.7 POINTER AND CLASSES

There are two cases in which a pointer to a class can be converted to a pointer to a base class.

The first case is when the specified base class is accessible and the conversion is unambiguous. Whether a base class is accessible depends on the kind of inheritance used in derivation. Consider the inheritance illustrated in the following figure.

Inheritance Graph for Illustration of Base-Class Accessibility

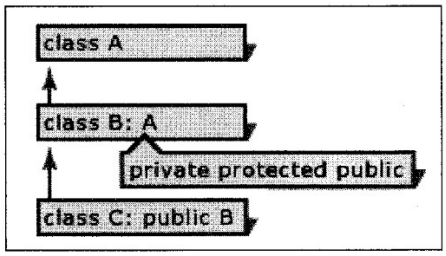


Figure 6.3

The following table shows the base-class accessibility for the situation illustrated in the figure.

Base-Class Accessibility

Type of Function	Derivation	Conversion from B* to A* Legal?
External (not class-scoped) function	Private	No
	Protected	No
	Public	Yes
B member function (in B scope)	Private	Yes
	Protected	Yes
	Public	Yes
C member function (in C scope)	Private	No
	Protected	Yes
	Public	Yes

The second case in which a pointer to a class can be converted to a pointer to a base class is when you use an explicit type conversion. (See Expressions with Explicit Type Conversions for more information about explicit type conversions.)

The result of such a conversion is a pointer to the "subobject," the portion of the object that is completely described by the base class.

The following code defines two classes, `A` and `B`, where `B` is derived from `A`. (For more information on inheritance, see Derived Classes.) It then defines `bObject`, an object of type `B`, and two pointers (`pA` and `pB`) that point to the object.

```
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

class B : public A
{
```

```
public:
    int BComponent;
    int BMemberFunc();
};
int main()
{
   B bObject;
   A *pA = &bObject;
   B *pB = &bObject;
   pA->AMemberFunc();    // OK in class A
   pB->AMemberFunc();    // OK: inherited from class A
   pA->BMemberFunc();    // Error: not in class A
}
```

The pointer `pA` is of type `A *`, which can be interpreted as meaning "pointer to an object of type `A`." Members of `bObject` (such as `BComponent` and `BMemberFunc`) are unique to type `B` and are therefore inaccessible through `pA`. The `pA` pointer allows access only to those characteristics (member functions and data) of the object that are defined in class `A`.

6.8 UNIONS AND CLASSES

The separating factor between a struct and a union is that a struct can also have member functions just like a class. The difference between a struct and a class is that all member functions and variables in a struct are by default public, but in a class, they default to private as previously discussed.

It is often a good idea to use constructors to initialize the member variables of a struct. Other than that, though, it is against current standards, and usually looked down upon, to use functions in a struct, and is usually considered just being lazy.

6.9 CONSTRUCTORS

Constructor is public method that is called automatically when the object of a particular class is created. C++ provides a default constructor method to all the classes. This constructor method takes no parameters. Actually the default constructor method has been defined in system. object class. Since every class that you create is an extension of system. object class, this method is inherited by all the classes.

The default constructor method is called automatically at the time of creation of an object and does nothing more than initializing the data variables of the object to valid initial values.

A Programmer can also define constructor methods for a class if she so desires. While writing a constructor function the following points must be kept in mind:

1. The name of constructor method must be the same as the class name in which it is defined.
2. A constructor method must be a public method.
3. Constructor method does not return any value.
4. A constructor method may or may not have parameters.

Let us examine a few classes for illustration purpose. The class abc as defined below does not have user defined constructor method.

```
class abc
{
    int x,y;
}

main()
{
    abc myabc;
    …;
}
```

The main function above an object named myabc has been created which belongs to abc class defined above. Since class abc does not have any constructor method, the default constructor method of C++ will be called which will initialize the member variables as:

```
myabc.x=0
```

and

```
myabc.y=0.
```

Let us now redefine myabc class and incorporate an explicit constructor method as shown below:

```
class abc
{
    int x,y;
    public:
    abc(int, int);
}
abc::abc(int a, int b)
{
    x=a;
    y=b;
}
```

Observed that myabc class has now a constructor defined to except two parameters of integer type. We can now create an object of myabc class passing two integer values for its construction, as listed below:

```
main()
{
    abc myabc(100,200);
    …;
}
```

In the main function myabc object is created value 100 is stored in data variable x and 200 is stored in data variable y. There is another way of creating an object as shown below.

```
main()
{
    myabc=abc(100,200);
    …;
}
```

Both the syntaxes for creating the class are identical in effect. The choice is left to the programmer.

There are other possibilities as well. Consider the following class differentials:

```
class abc
{
    int x,y;
    public:
      abc();
}
abc::abc()
{
    x=100;
    y=200;
}
```

In this class constructor has been defined to have no parameter. When an object of this class is created the programmer does not have to pass any parameter and yet the data variables x,y are initialized to 100 and 200 respectively.

Finally, look at the class differentials as given below:

```
class abc
{
    int x,y;
    public:
      abc();
abc(int);
abc(int, int);
}
abc::abc()
{
    x=100;
    y=200;
}
abc::abc(int a)
```

```
{
    x=a;
    y=200;
}
abc::abc(int a)
{
    x=100;
    y=a;
}
```

Class myabc has three constructors having no parameter, one parameter and two parameters respectively. When an object to this class is created depending on number of parameters one of these constructors is selected and is automatically executed.

Note that C++ selects one constructor by matching the signature of the method being called. Also, once you define a constructor method, the default constructor is overridden and is not available to the class. Therefore you must also define a constructor method resembling the default constructor method having no parameters.

6.9.1 Parameterized Constructors

If it is necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called 'Parameterized constructors.' The definition and declaration are as follows:

```
class dist
{
    int m, cm;
public:
    dist(int x, int y);
};
dist::dist(int x, int y)
{
    m = x;  n = y ;
}
main()
{
dist d(4,2);
d. show ();
}
```

6.9.2 Constructors with Default Arguments

This method is used to initialize object with user defined parameters at the time of creation.

Consider the following Programme that calculates simple interest. It declares a class interest representing principal, rate and year. The constructor function intializes the objects with principal and number of years. If rate of interest is not passed as an argument to it the Simple Interest is calculated taking the default value of rate of interest.

```
#include<iostream.h>
#include<conio.h>
class interest
{ int principal, rate, year;
    float amount;
    public
    interest (int p, int n, int r = 10);
    void cal (void);
};
    interest::interest (int p, int n, int r = 10)
{ principal = p; year = n; rate = r;
};
void interest::cal (void)
{
    cout<< "Principal" <<principal;
    cout << "\ Rate" <<rate;
    cout<< "\ Year" <<year;
    amount = (float) (p*n*r)/100;
    cout<< "\Amount" <<amount;
};
main ( )
{
    interest i1(1000,2);
    interest i2(1000, 2,15);
    clrscr( );
    il.cal();
    i2.cal();
}
```

Note the two objects created and initialized in the main( ) function.

```
interest i1(1000,2);
interest i2(1000,2, 15);
```

The data members principal and year of object i1 are initialized to 1000 and 2 respectively at the time when object i1 is created. The data member rate takes the default value 10 whereas when the object 12 is created, principal, year and rate are initialized to 1000, 2 and 15 respectively.

It is necessary to distinguish between the default

constructor::construct();

and default argument constructor

construct : : construct (int = 0)

The default argument constructor can be called with one or no arguments. When it is invoked with no arguments it becomes a default constructor. But when both these forms are used in a class, it causes ambiguity for a declaration like construct C1;

The ambiguity is whether to invoke construct: : construct ( ) or construct: : construct (int = O)

6.9.3 Copy Constructors

A copy constructor method allows an object to be initialized with another object of the same class. It implies that the values stored in data members of an existing object can be copied into the data variables of the object being constructed, provided the objects belong to the same class. A copy constructor has a single parameter of reference type that refers to the class itself as shown below:

```
abc::abc(abc & a)
{
    x=a.x;
    y=a.y;
}
```

Suppose we create an object myabc1 with two integer parameters as shown below:

```
abc myabc1(1,2);
```

Having created myabc1, we can create another object of abc type, say myabc2 from myabc1, as shown below:

```
myabc2=abc(& myabc1);
```

The data values of myabc1 will be copied into the corresponding data variables of object myabc2. Another way of activating copy constructor is through assignment operator. Copy constructors come into play when an object is assigned another object of the same type, as shown below:

```
abc myabc1(1,2);
abc myabc2;
myabc2=myabc1;
```

Actually assignment operator(=) has been overloaded in C++ so that copy constructor is invoked whenever an object is assigned another object of the same type.

6.9.4 Dynamic Constructors

Allocation of Memory during the creation of objects can be done by the constructors too.

The memory is saved as it allocates the right amount of memory for each object (Objects are not of the same size). Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. new operator is used to allocate memory.

The following programme concatenates two strings. The constructor function initializes the strings using constructor function which allocates memory during its creation.

```
#include <iostream.h>
#include <string.h>
class string
{
    char * name;
    intlength;
    public:
    string 0
{
    length = 0;
    name = new char [length = 1];
};
string (char*s)
{
length = strlen (s);
name = new char [length + 1];
strcpy(name,s );
};
void display (void)
{
    cout<<"\n Name :- "<<name;
};
void join (string & a, string & b)
{
    length = a.length + b.length;
    delete name;
    name = new char [length + 1];
    strcpy (name,a.name);
    strcat (name," ");
    strcat (name,b.name);
};
```

```
};
main()
{
    char * FirstName= "Mohan";
    string Fname(First name);
    string Mname("Kumar");
    string Sname("Singh");
    string Halfname, Fullname;
//Joining FirstName with Surname
Halfname.join (Fname, Sname);
//Joining Firstname with Middlename & Surname
Fullname.join (Halfname, Mname);
Fname.display ();
Mname.display ();
Sname.display ();
Halfname.display();
Fullname.displayO();
}
```

You should see the following output.

```
/*
    Name :- Ram
    Name :- Kumar
    Name :- Singh
    Name:- Mohan .Singh Name :- Mohan.Singh .Kumar
*/
```

The above programme uses *new* operator to allocate memory. The first constructor is an empty constructor that allows us to declare an array of string. The second constructor initializes length *of* the string, allocates necessary space for the string to be stored and creates the string itself. The member function join ( ) concatenates 2 strings

It actually adds the length of 2 strings and then allocates the memory for the combined string. After that the join function uses inbuilt string functions strcpy & strcat to fulfill the action.

The output of the programme will be in Full Name and Half name

That is Mohan Singh Kumar and Ram Singh respectively.

Another example of the dynamic constructor is the matrix programme. In two Dimensional matrix we need to allocate the memory for the values to be stored in. Using constructor we can allocate the memory for the matrix.

This programme declares a class matrix to represent the number *of* rows and columns *of* matrix as well as two dimensional array to store the contents of matrix. the constructor function used to initialize the objects allocates the required amount of memory for the matrix.

```
#include<iostream.h>
#include<conio.h>
//Class Definition
class matrix
{
    int **p;        //declaring two dimensional array
    int d1,d2;
public:
    matrix(intx,inty);
    void get_value( void);
    void dis_value( void);
    void square(void);
    void cube(void);
};
matrix: : matrix( int x,int y)
{
    d1=x;
    d2=y;
    p= new int *[d1];
    for (int i =0;i<d1 ;i++)
    p[i]=new int[d2];
} ;
void matrix:: get_value(void)
{
    for(int i = 0;i<d1;i++)
    for(int j=0;j<d2;j++)
{
    cout<<"Please Enter A No At"<<i<<j<< "Position :-";
    cin>>p[i][j];
}
} ;
voidmatrix:: dis_value(void)
{
    cout< <"The Matrix Entered";
    for (int i = 0;i<d1 ;i++)
{
```

```
    cout< <"\n";
    for(int j=0;j<d2;j++)
    cout<<"\t"<<p[i] [j];
}
};
void matrix:: square(void)
{
    cout < <"\n The Squared Matrix";
    for (int i = 0;i<d1 ;i++ )
{
    cout<<"\n";. for(intj=0;j<d2;j++)
cout<<"\t" <<p[i][j]*p[i] [j];
}
};
void matrix:: cube(void) {
    cout < <"\nThe Cubed Matrix";
    for (int i = 0;i<d1;i++)
{
    cout<<"\n";
    for(int j=0;j<d2;j++ )
    cout<<"\t"<<p[i][j] * p[im] * p[i][j];
}
};
//Start Of Main Programme
main()
{
    intm,n;
    elrser();
    cout<<"Enter Size Of Matrix :- ";
    cin> >rn>>n;
    matrix mat(m,n);
    mat.get_value();
    mat.dis_value();
    getch();
    mat.square();
    mat.cube();
    getch();
}
```

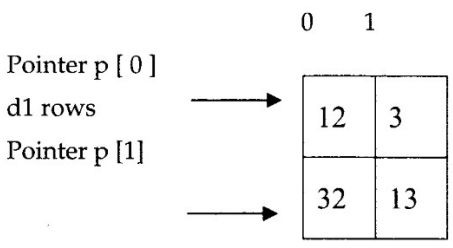
Let us assume we run this programme with the following input.

```
/*
Enter Size Of matrix :- 2  2
Please Enter A No At Position 00 :- 1
Please Enter A No At Position 01 : - 1
Please Enter A No At Position 10 : - 2
Please Enter A No At Position 11 :- 2
*1
```

You should see the following output.

```
/*
The Matrix Entered
1    1
2    2
The Squared Matrix
1    1
4    4
The Cubed Matrix
1    1
8    8
*/
```

The Constructor first creates a vector pointer to an int of size dl. Then it allocates, iteratively, an int type vector of size d2 pointed at each element p[i]. Thus space for the element of a d1x d2 matrix is allocated from free store as shown below.



6.10 DESTRUCTORS

Constructors create an object, allocate memory space to the data members and initialize the data members to appropriate values; at the time of object creation. Another member method called

destructor does just the opposite when the programme creating an object exits, thereby freeing the memory.

A destructive method has the following characteristics:

1. Name of the destructor method is the same as the name of the class preceded by a tilde(~).
2. The destructor method does not take any argument.
3. It does not return any value.

The following codes snippet shows the class abc with the destructor method;

```
class abc
{
    int x,y;
    public:
      abc();
abc(int);
abc(int, int);
~abc()
{
cout << "Object being destroyed!!";
      }
}
```

Whenever an object goes out of the scope of the method that created it, its destructor method is invoked automatically. However if the object was created using *new* operator, the destructor must be called explicitly using *delete* operator. The syntax of delete operator is as follows:

```
delete(object);
```

Note that whenever you create an object using new keyword, you must explicitly destroy it using delete keyword, failing which the object would remain homed in the memory and in the course of programme execution there may come a time when sufficient memory is not available for creation of the more objects. This phenomenon is referred to as *memory leak*. Programmers must consider memory leak seriously while writing programmes for the obvious reasons.

6.11 INLINE FUNCTIONS

An inline function is just a function whose declaration is accompanied with its definition. C++ supports the Inline functions to let you have a reduced execution time in your code.

With the normal functions, the compiler puts a jump instruction to your code when you call the function. This means that arguments and automatic variables have to be set up on the stack and the programme execution has to be transferred to the function itself, and so on.

On the other hand, when we use inline functions, the compiler actually puts the entire code for the function directly into the code that is inline. That means, no jump is needed and you need save a little

time. However, the programme size increases as the same code is copied many times wherever the inline function is called. We usually use small codes for inline functions.

To create an inline function, *inline* keyword is required at the beginning of function prototype as shown below:

```
inline int sum(int, int);
```

6.12 STATIC CLASS MEMBERS

Member data items of a class can be static. Static data members are data objects that are common to all objects of a class. They exist only once in all objects of this class. The static members are used when the information is to be shared. They can be public or private data. The main advantage of using a static member is to declare the global data which should be updated while the programme lives in memory.

When a static member is declared private, the non-member functions cannot access these members. But a public static member can be accessed by any member of the class. The static data member should be created and initialized before the main function control block begins.

For example, consider the class account as follows:

```
class account
{
private:
    int acc _no;
    static int balance;              //static data declaration
public:
    void disp(int acc _no);
    void getinfo( );
};
```

The static variable *balance* is initialized outside main() as follows:

```
int account::balance = 0;                    //static data definition
```

Consider the following Programme, which demonstrates the use of static data member *count*. The variable *count* is declared static in the class but initialized to 0 outside the class.

```
#include<iostream.h>
#include<conio.h>
class counter
{
private:
static int count;
public:
void disp();
};
```

```
int counter::count = 0;
void counter::disp()
{
    count++
    cout << "The present value of count is " << count << "\n";
}
main()
{
    counter cnt1;
    for(int i=0; i<5; i++)
      cnt1.disp();
    getche();
}
```

You should get the following output from this programme.

The present value of count is 1

The present value of count is 2

The present value of count is 3

The present value of count is 4

The present value of count is 5

6.13 FRIEND FUNCTIONS

Object oriented programming paradigm secures data because of the data hiding and data encapsulation features. Data members declared as private in a class are restricted from access by non-member functions. The private data values can be neither read nor written by non-member functions. Any attempt made directly to access these members, will result in an error message as "inaccessible data-type" during compilation.

The best way to access a private data member by a non-member function is to change a private data member to a public group. But this goes against the concept of data hiding and data encapsulation. A special mechanism available known as *friend function* allows non-member functions to access private data. A friend function may be either declared or defined within the scope of a class definition. The keyword *friend* informs the compiler that it is not a member function nor the property of the class.

The general syntax of the friend function is:

```
friend <return_type> <function_name>(argument list);
```

friend is a keyword. A friend declaration is valid only within or outside the class definition. The following code snippet shows how a friend function is defined.

```
class sample
{
private:
```

```
    int x;
    public:
      void getdata( );
      friend void disp(sample abc); //friend function
};
void disp(sample abc)          // non-member function without scope:: operator
{
cout<<"value of x ="<< abc.x;
cout<<endl;
}
```

Note that the function is declared as friend in the class and is defined outside the class. The keyword friend should not be in both the function declaration and definition. The friend declaration is unaffected by its location in the class. It can be declared either in a public or a private section, which does not affect its access right.

For example, the following declarations of a friend function are valid:

1. The friend function disp( ) is declared in the public group

```
class sample
{
  private :
        int x;
  public:
        void getdata( );
        friend void disp( );
};
```

2. The friend function disp( ) is declared in the private group

```
class sample
{
    private:
        int x;
        friend void disp();
    public:
        void getdata( );
};
```

Since private data members are available only to the particular class and not to any other part of the programme, a non-member function cannot access these private data. Therefore, the friend function is a special type of function which is used to access the private data of any class. In other words, they are defined as non-member functions with the ability to modify data directly or to call function members that are not part of the public interface. The friend class has the right to access as many members of its

class. As a result the level of privacy of the data encapsulation gets reduced. Only if it is necessary to access the private data by non-member functions, then a class may have a friend function, otherwise it is not necessary.

Let us look at a sample programme given below to access the private data of a class by non-member functions through friend function. The programme declares a private class example representing variable x and function to input the value for the same. The friend function to display the value of x takes an object as argument with the help of which private variable x can be accessed.

```
#include<iostream.h>
#include<conio.h>
class example
{
private:
int x;
public:
void getdata()
{
cout << "Enter the value ofx"<< "\n";
cin >> x;
}
friend void disp(example);
};
void disp(example e.g.)
{
      cout << "Display the entered number"<< e.g.<< "\n";
}
main()
{
      example eg1;
      eg1.getdata();
      disp(eg1);
      getche();
}
```

You should see the following output.

Enter the value of x

4

Display the entered number 4

There are also other areas of application for friend function. The friend function can also be defined within the scope of a class definition itself. Friend function may also have inline member functions. If

the friend function is defined in the class, then the inline code substitution is done automatically. But if defined outside the class, then it is necessary to precede the return type with the keyword inline to make the inline code substitution.

The following programme accesses the private data of a class through a friend function where the friend function is defined with inline code substitution.

```
#include<conio.h>
class example
{
private:
int x;
public:
inline void getdata();
friend void disp(example);
};
inline void example::getdata()
{
cout<<"Enter the value of x " << "\n";
cin>>x;
}
inline void disp(example e.g.) //Note the use of the keyword inline
{
    cout << "Display the entered number" << e.g. << "\n";
}
main()
{
    example eg1;
    eg1.getdata();
    disp(eg1);
    getche();
}
```

You should see the output as shown below:

Enter the value of x

20

Display the entered number 20

One class can be friendly with another class. Consider two classes, *first* and *second*. If the class *first* is friendly with the other class *second*, then the private data members of the class *first* are permitted to be accessed by the public members of the class *second*. But on the other hand, the public member functions of the class *first* cannot access the private members of the class *second*.