Unit 4 Classes and Abstraction

Structure:

4.1 Introduction

In the previous unit, you have studied how to reuse code using function. You have also studied storage classes, strings (array of characters), structures and unions. Structures and unions are used to group heterogeneous data.

In this unit, we will discuss implementation of objects and classes. In real life we come across the situations where we have to model the functionality of data type along with data. Classes bind data and functions that act on data together. However, in C++ structures are used in the same way as classes by binding together functions and data. But you should use structures only in situations when it is required to group together the data and the functions. In C++ there are objects which are the instances of class. The objects have a similar relationship with classes as variables have with data types. Object is an instance of a class.

In this unit, we will also discuss important features of C++, such as, access specifiers, abstract classes, this pointer, friend functions, and static functions.

Objectives:

After studying this unit you should be able to:

- explain the role of class and objects in oops
- define different types of access specifiers
- describe this pointer
- discuss friend function, its advantages, and scope
- discuss static variables and static functions.

4.2 Creating classes

In unit 1, you have studied the basic features of Objects Oriented programming. In this section we will discuss the uses of classes and how to create a class. Classes provide users a way to create user defined data types.

Classes provide a convenient way to group related data and the methods that operate on data together. One advantage of creating a class is that when you create an object from the class, you automatically create all the related fields. Another advantage of using classes is that you can think about them and manipulate them in the same way as you do with real-life classes.

Characteristics of class:

A class is a template that units data and operations.

A class is an abstraction of the real world entities with similar properties.

A class identifies a set of similar objects.

A class is implementation of an abstract data type.

You can define a class by using the keyword class followed by the name of the class and an open and closed brace. Between curly braces, you define all the data and member functions of class. When you create or define a class you can describe what it has and can do.

General syntax of class:

```
 class classname
{
  variable declaration;
  function declaration;
}
```

Real world entities such as, employee, vehicle, animal, etc. can be modelled by class or it can model a user defined data type such as string, distance, etc.

Example:

```
 class Student
{
int rollnumber;
string name;
float grade;
};
```

Let see one more simple example of class

```
class product
{
   int product_id;   // data varibles declartion
   double cost;
    void getdat(int a, float b);  // member function decalrtion
    void putdat(void);
}
```

In the above example the class product contains two data variables and two member functions.

Self Assessment Questions

1. A ________ is an abstraction of the real world entities with similar properties.
2. A class is an implementation of _____________ datatype.

4.3 Creating objects

Objects hold the similar relationship with class as a variable holds with a data type. An instance of a class is known as an object of that class and is used in the program to store data. The objects are declared in the program like the variable declaration.

Objects are mainly used for the following purposes:

- Understanding real world and a practical base for designers.

- Decomposition of a problem into objects depends on judgment and nature of problem.

A blueprint for objects is given by class. Basically an object is created from a class. The object is created by using the class name followed by the object name. The statements shown below declare two objects of class product.

```
product  p1;     // declare p1 object of the type product
product p2;      // declare p2 object of the type product
```

Both of the objects p1 and p2 will have their own copy of data members. The object creation syntax that has been used so far called direct initialization. The object can also be value initialization.

```
product p=product();  // value initialization
```

The parentheses can supply constructor arguments, or remain empty to construct the object using parameter less constructor.

To understand the various syntactical requirements for implementing a class, let us take an example.

```
// distance.cpp
#include<iostream.h>
class distance
{ private:
 int feet;
 int inches;
public:
void setdistance(int c, int d)
{ feet=c;
   inches=c;
}
void printdistance()
{ cout<<feet<<"ft"<<inches<<"inches;
};
void main()
```

```
{ distance d1;
d1. setdistance(10, 2);
d1.printdistance();
}
```

In the above program, there is a class distance with two data members- feet and inches and two member functions or methods; setdistance() and printdistance(). You can define a class with the keyword class followed by the name of the class. As you can see in the main program, d1 is an object of class distance. Every object of a class has its own copy of data, but all the objects of a class share the functions. There is no separate copy of the member function for every object of class. The data can be accessed by the objects through member functions or methods. To invoke the member functions of class we have to use object of that class. The member functions cannot be invoked without the objects of the class. The methods defined in the class can only be invoked by the objects of the class. Without an object, methods of the function cannot be invoked.

Self Assessment Questions

3. An instance of a class is known as an ______ of that class and is used in the program to store data.
4. The object is created by using the class name followed by object name. (True/False)

4.4 Access Specifiers

As you have seen in the above program, we have defined the data elements -feet and inches- as private and member functions or methods as public. These are known as access specifiers. Data and functions in a class can have any one of the following access specifiers: private, public and protected. Only the member functions of the class can access the private data and functions but the external functions cannot access it. Any external function of the class can access public data and member functions and for that you need to have an object of that class. This feature of object oriented programming is sometimes referred to as data hiding. The data is hidden from accidently getting modified by an unknown function. Only a few sets of functions can access the data. By default, the data and functions are private in a class so there is no need to explicitly mention the private keyword these are depicted in figure 4.1.

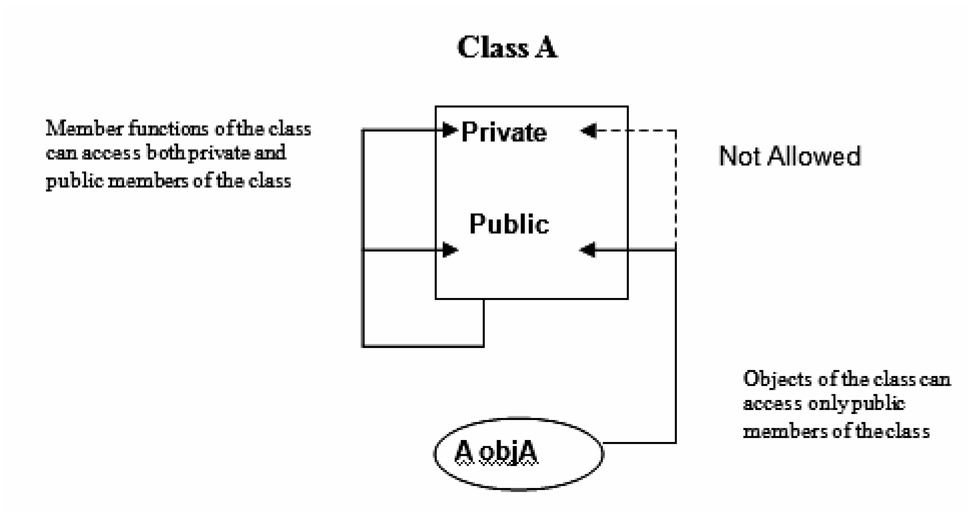
Class A
Member functions of the class can access both private and public members of the class
Private
Public
Not Allowed
Objects of the class can access only public members of the class
A objA

Figure 4.1: Access specifiers in a class

The methods or member functions are referred to as messages in some programming languages. Thus, d1.display() is a message to d1 to display itself. Generally, data are declared as private and functions as public. However, in certain cases, you may even declare functions as private and data as public. The class definition should be terminated with a semicolon. You can also create an array of objects that is same as the array of structures. For example distance d[10]; creates an array of objects of type distance. If the first distance object accesses the member function display you can use d[0].display().

Self Assessment Questions

5. Data and functions in a class can have any one of the following access specifiers ___________.
6. In general practice, data are declared as ________ and functions as _________.

4.5 Objects and Arrays

In the above section, we mentioned about creating an array of objects. In this section we shall discuss this concept in more detail. The data type of array can be of any type including struct. So there can be arrays of variables of type class and those variable are called arrays of objects. Let us consider the following definition of class:

```
class distance
{
   Private:
   int feet;
float inches;
public:
 void getdist();
void printdist();
};
```

The identifier distance is a name of the class which is a user defined datatype and can be used to create objects of type distance.

Example:

```
distance D[2]  // array of two distances
distance dist[5] //array of five distances
```

The array D contains two objects namely D[0] and D[1]. The array dist contains five objects namely dist[0], dist[1], dist[2], dist[3].

To access the member functions you can use the usual array accessing method followed by a dot operator.

For example the statement D[i].printdist(); will print the distance value of ith element of the array D.

We will demonstrate it by a program array.cpp containing array of distances.

```
//array.cpp
#include<iostream.h>
class distance
{
  private:
  int feet;
  float inches;
  public:
  void getdist() //input length from user
```

```
 {
   cout<< “ enter feet:\n”;
  cin>>feet;
 cout<<”enter inches:”;
 cin>> inches;
}
 void printdist() // print the distance
 {
   cout<< feet << “\ ‘ -“ << inches << ‘\” ’;
 }
};
int main()
{
 distance D[50];  //array of distances
int count=0;   // count the entries
char res;  //response of user (y or n)
cout<< endl;
do
{
 cout<< “ enter the value of distance:” << count+1;
D[n++].getdist(); //store distance in array
cout<< “ do you want to enter another distance (y/n)?:”;
cin>>res;
} while(res!= (’n’ || ‘N’))
for(j=0; j<count; j++)
{
  cout<< “\ndistance number” << j+1 << “ is “;
D[j].showdist();
}
cout<<endl;
return  0;
}
```

In this program the user can enter the required number of distances. After entering every distance. The program asks if user wants to enter another distance value. If not, then the program terminates and prints all the distances entered. Here is the output of the program when user enters three distances.

```
enter distance number:1
enter feet:4
enter inches:5
do you want to enter another distance(y/n)? y
enter the value of distance:2
enter feet: 7
enter inches: 8
do you want to enter another distance(y/n)? n
distance number 1 is 4' -5"
distance number 2 is 7' -8"
```

Self Assessment Questions

7. There can be arrays of variables of type class and those variable are called _________.

4.6 Objects and Functions

In this section you will learn to perform operations on data. We can start with a program which implements an add member function to the distance class. This can be done in two ways. Shown below are the prototypes of the function:

```
void add(distance, distance);
or
distance add(distance);
```

The first function takes two distance objects as arguments and stores the result in the object invoking the member function. It can be implemented as follows:

```
//adddistance.cpp
#include<iostream.h>
class distance
{ private:
int feet;
int inches;
public:
void setdistance(int c, int d)
{ feet=c;
inches=d;
}
void printdistance()
{ cout<<feet<<"ft"<<inches<<"inches;}
void adddist(distance d1, d2)
{ feet=d1.feet+d2.feet;
inches=d1.inches+d2.inches;
if (inches>12)
{ feet++;
inches=inches-12;
}
}
};
void main()
{ distance D1,D2,D3;
D1. setdistance(10, 2);
D2.setdistance(2,4);
D3.adddist(d1,d2);
D3.display();
}
```

In the above program, object D3 invokes the add function so the feet and inches refer to invoking object's data members. After adding the respective data of D1 and D2 it is stored in D3 object.

4.7 Objects and Pointers

Pointers can point to simple data types, arrays and to objects as well. Let us consider the following statement:

```
item A;
```

Here item is a class and A is the object of the class item. You can define a pointer iptr of type item as shown below:

item *iptr;

We use pointers in the situations where we do not know that how many objects we have to create in a program. In that situation we use new function to create objects at run time. The *new* function returns a pointer to an unnamed object. Let us understand this concept with the help of an example shown below.

```
//pointer.cpp
#include<iostream.h>
class distance
{
  private:
  int feet;
  float inches;
  public:
  void getdist() //input length from user
 {
   cout<< " enter feet:\n";
  cin>>feet;
 cout<<"enter inches:";
 cin>> inches;
}
 void printdist() // print the distance
 {
   cout<< feet << "\ ' -" << inches << '\" ';
 }
};
int main()
{
  distance D;  //define a named distance object
D.getdist();
D.printdist();
distance *dptr;  //pointer to distance class
dptr = new distance; //points to new distance object
dptr->getdsit();  // points to new distance object
dptr->printdist();   //access object members with -> operator
return 0;
}
```

The main() function defines dist, uses the Distance member function getdist() to get a distance from the user and then uses printdist() to display it.

Self Assessment Questions

8. We use ________ in the situations where we do not know how many objects we need to create in a program.
9. The *new* function returns a pointer to an unnamed object. (True/False)

4.8 Abstract class

An abstract class is the class which acts as a base class and can be inherited by other classes. It is not used to create objects. It provides a base upon which other classes can be built. In programming the concept of abstract class is of great importance and used deliberately in a program for creating derived class.

You can make a class abstract by declaring at least one of its virtual functions as pure virtual function. You can specify a pure by placing "= 0" in its declaration. An example is shown below:

```
class container
{
  public:
        virtual double getvolume() = 0; //pure virtual function
  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
```

The concept of pure virtual function is discussed in unit 8.

Abstract Class Example:

Let us see the following example where an interface is provided by parent class to the base class to implement a function called getArea().

```
include <iostream.h>

// Base class
class shape
{
public:
  // pure virtual function providing interface framework.
  virtual int getarea() = 0;
  void setlenght(int l)
  {
    length = l;
  }
  void setbreadtht(int b)
  {
    breadth = b;
  }
protected:
  int length;
  int breadth;
};
 // Derived classes
class rectangle: public shape
{
public:
  int getarea()
  {
    return (length * breadth);
  }
};
class square: public shape
{
public:
  int getarea()
  {
    return (length*length);
  }
};

int main()
{
  rectangle rect;
  square sq;;

  rect.setbreadth(5);
  rect.setlengtht(7);
  // Print the area of the object.
  cout <<  Area of rectangle is: " << rect.getarea() << endl;

  sq..setlenght(5);
    // Print the area of square.
  cout << "Area of square is: " << sq..getarea() << endl;

  return 0;
}
```

The result of the above program is as shown below:

```
Area of rectangle is: 35
Area of square is: 25
```

4.9 The *this* Pointer

Every time when the program creates a class instance, a special pointer is created in C++ called *this* and it contains the address of the current object instance. The address of the class instance is passed as an implicit parameter to the member functions. The example shown below describes how to use *this* pointer. C++ maintains a single copy of its member functions and memory is allocated to data members for all of their instances. And these many instances of data are maintained by *this* pointer.

The following are the features of *this* pointer:

- *this* pointer stores the address of the class instance, to enable pointer access of the members to the member functions of the class.
- *this* pointer is not counted for calculating the size of the object.
- *this* pointers are not accessible for static member functions.
- *This pointer can't be modified.* Let us see the following example to understand the concept of *this* pointer.

```
class this_pointer_example { // class for explaining C++ tutorial
    int data1;
    public:
        //Function using this pointer for C++ Tutorial
        int getdata() {
            return this->data1;
        }
        //Function without using this pointer
        void setdata(int newval) {
            data1 = newval;
        }
 };
```

Thus, a member function can gain the access of data member by either using this pointer or not.

Self Assessment Questions

10. __________________ contains address of the current object instance.
11. *this* pointers can be modified. (True/false)

4.10 Friend Functions

Friend functions allow us to access some non-member functions or other classes in C++. Access to non-member functions or to another class is given in C++ by using friend keyword. The private as well as protected class members can be accessed by friend classes. There are places where friends can lead to more intuitive code, and are often needed to correctly implement operator overloading.

From designer's point of view, friend functions can be treated similar to that of public member functions. The concept of a class interface can be extended from public members to include friend functions and friend classes. Put another way: Friend functions do not break encapsulation; instead they naturally extend the encapsulation barrier.

You can declare friend function anywhere in the class declaration. But in common practice we list friends at the beginning of the program as the class has no control over the scope of friends so the public and protected keywords do not apply to friend function

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

```
// friend functions
#include <iostream.h>
class Rect
{
   int w, h // w is the variable to store width and h is the variable to store
height
  public:
      void setvalue (int, int);
      int area () {return (w * h);}
      friend Rect duplicate (Rect);
};
void Rect::setalue (int c, int d) {
   w = c;
   h = d;
}
Rect duplicate (Rect rectparam)
{
   Rect r;
   r.w = rectparam.w*2;
   r.h = rectparam.h*2;
   return (r);
}
int main () {
Rect r1, r2;
   r1.setvalues (2,3);
   r2 = duplicate (r1);
   cout << r2.area();
   return 0;
}
```

The function duplicate is a friend of Rect function. With the friend function named duplicate, it is possible to access the members w and h of objects of type rectangle. You can observe that the function duplicate is not a member function of class Rect but it has access to the private members of the class Rect.

Friend Classes

You should use friend class in the situation when the two classes are strongly coupled. For example, suppose we have a class CCord that represents a coordinate, and a class CCollect that holds a list of points. The collections class may be used to change the point objects so we can declare CCollect as a friend class of CCord.

```
      // Forward declaration of friend class.
      class CCollect;
      // Point class.
      class CCord {
        friend CCollect;
        private:
          double m_x;
          double m_y;
        public:
          CCordt(const double x, const double y):
```

```
        m_x(x),
        m_y(y) { }
        ~C Cord(void) { }
        // ...
};
```

As you can see in the example above, the CCollect and CCord classes are friend classes. Hence, CCollect class can access the data of any object of CCord class. This is proved to be useful when you need to modify individual elements of class CCollect. For example a setvalue method ofclass CCollect can set all the values of the class CCord to a particular value.

```
class CCollect {
    private:
        vector<CCord> m_vecPoints;
    public:
        CCollect(const int nSize) :
        m_vecPoints(nSize) { }
        ~CCollect(void);
        void set(const double x, const double y);
        // ...
};
```

The set member can iterate over the collection and reset each point:

```
void CCollect::set(const double x, const double y) {
    // Get the number of elements in the collection.
    const int nElements = m_vecPoints.size();
    // Set each element.
    for(int i=0; i<nElements; i++) {
        m_vecPoints[i].m_x = x;
        m_vecPoints[i].m_y = y;
    }
}
```

The friendship is not mutual among friend classes. As in the above example, CCollect class can access the data of CCord but the vice versa is not true. The friendship is not passed down in the hierarchy of class. For example the derived classes of CCollect will not be able to access the data of CCord class. The principle is that it is not possible for a class to implicitly grant friendship; each class must choose its friends explicitly.

Friend Scope

The friend function's name or class which is introduced first in a friend declaration is not in the scope of class granting friendship (also known as the enclosing class). And also it is not a member of class granting friendship.

The name of a function first introduced in a friend declaration is in the scope of the first nonclass scope that contains the enclosing class. The body of a function provided in a friend declaration is handled in the same way as a member function defined within a class. Processing of the definition does not start until the end of the outermost enclosing class. In addition, unqualified names in the body of the function definition are searched for starting from the class containing the function definition.

If we introduce a friend's class name before the friend declaration, then the compiler will search for a class name that matches with the name of the friend class beginning at the scope of the friend declaration. If the name of a friend class is introduced before the friend declaration, the compiler searches for a class name that matches with the name of the friend class beginning at the scope of the friend declaration. If the declaration of a nested class is followed by the declaration of a friend class with the same name, the nested class is a friend of the enclosing class.

The scope of a friend class name is the first nonclass enclosing scope. For example:

```
class X {
   class Y { // arbitrary nested class definitions
      friend class Z;
   };
};
```

is equivalent to:

```
class Z;
class X {
  class Y { // arbitrary nested class definitions
    friend class Z;
  };
};
```

You have to use the scope resolution operator(::) if the friend function is a member of another class. Example to explain this concept is shown below.

```
class X {
  public:
    int func() { }
};
class Y {
  friend int A::func();
};
```

Friends of a base class are not inherited by any classes derived from that base class. The following example demonstrates this:

```
class X {
  friend class Y;
  int p;
};
class Y { };
class Z : public Y {
  void func(p* q) {
    //q->p = 2;
  }
};
```

The compiler would not allow the statement p->q= 2 because class Z is not a friend of class X, although Z inherits from a friend of X.

Friendship cannot be transitive. The example below explains this concept.

```
class X {
  friend class Y;
  int p;
};
class Y {
  friend class Z;
};
class Z {
  void func(pA* q) {
      //q->p = 2;
  }
};
```

The following statement q->a = 2 is not allowed by compiler as classes Z and x are not friends, even though class Z is a friend of friend of X.

If you declare a friend in a local class, and the friend's name is unqualified, the compiler will look for the name only within the innermost enclosing nonclass scope. Before declaring a friend of a local scope you must declare a function. However, the declaration of a friend class will hide a class in an enclosing scope with the same name. The example to explain the concept is shown below. The following example demonstrates this:

```
    class X { };
    void a();
    void f() {
        class Y { };
        void b();
        class A {
            friend class X;
            friend class Y;
            friend class Z;
            //friend void a();
            friend void b();
            //friend void c();
        };
        ::X moocow;
        //X moocow2;
    }
```

The following statements will be allowed by the compiler in the above example:

- friend class X: This statement does not declare ::X as a friend of A, but the local class X as a friend, even though this class is not otherwise declared.

- friend class Y: In the scope of f() the local class Y is declared. friend class Z: This statement declares the local class Z as a friend of A even though Z is not otherwise declared.
- friend void b():In the scope of f() the function b() has been declared.
- ::X moocow: The object of the nonlocal class ::X is created by this statement.

The following statements will not be allowed by the compiler: friend void a(): The function a() is not considered by this function in namespace scope. The compiler will not allow this statement as the function a() is not declared in scope of a(). friend void c(): The compiler will not allow this statement because the function c() is not declared in the scope of f().X moocow2: This statement attempt creating an object of the local class named X, and not the nonlocal class ::X. This statement is not allowed by the compiler because the local lass X is not defined.

Self Assessment Questions

12. Access to non-member functions or to another class is given in C++ by using ____________ keyword.
13. In general practice the friend functions are listed in __________ of the program.

4.11 Static variables and Static Functions

You have observed that a separate copy of data members is created for every object when it is created. But there is an exception in terms of static data members. Static data member of the class is one data member that is common for all the objects of the class and is accessible for the class. It is beneficial to store some common data about the class like count of number of objects created for a class. To declare static data member you should prefix it with the keyword ‘static’.

You have already learnt in the unit that member functions of a class are invoked via objects of the class. Let us suppose that we have a static data member that contains count of the total objects created and a function display_count prints the total. We cannot access the total without the support of Static functions or a special type of functions which can be invoked even without an object of the class. To define the static function you should prefix it with the keyword ‘static’. Static functions are also defined by prefixing the keyword static. You can call the static function by using the name of the class and then the scope resolution operator followed by the function name. The example below demonstrates the implementation of static data and static functions.

```
//static.cpp
# include  <iostream.h>
class X {
static int count;
int id;
public:
 X()
{ count++;
id=count;
}
~X()
{  count--;cout<<"\n destroying ID number="<<id;
}
static void display_count()
{cout<<"endl "<<countl;}
void showid()
{cout<<"endl "<<id;}
};
int X::count=0;
void main()
{
X::display_count();();X x1, x2;
X::display_count();
x1.showid();
x2.showid();
}
```

The output of the above program is:

```
0
2
1
2
```

In the above program count is a static variable which is shared by all objects of the class X. We have to explicitly initialize the static variables outside the class. Static variable is declared using the name of the class then the scope resolution operator followed by the name of the static variable. You should keep in mind that during initialization of static data member you need to specify the data type and no need to specify the static keyword. In the above program there is a static variable count which is incremented in the constructor when the new object is created. And whenever the object is destroyed, the count is decremented by the destructor. In the program the static function is display_count which can be invoked by using the name of the class without using the object of the class.

The comparison between a static member functions and non-static member functions are shown in table 4.1.

Table 4.1: Comparison between static and non-static member functions

Static member function	**Non-static member functions**
Only the static member data, static member functions , data and functions outside the class can be accessed by the static member functions	Whereas all of these including static data member can be accessed by non-static functions.
Even if the class is not initiated it is possible to call a static member function.	But these functions can be called only through the objects of the class.
It is not allowed in C++ to declare these functions as virtual	These functions can be declared as virtual.
Do not have access to *this* pointer of the class.	Have access to *this* pointer of the class.
These functions are not used very frequently in the class but there are very useful when the class is not initialized.	These function are used frequently in the programs.

You cannot declare the static and non-static member functions with the same name, having same number and same type of arguments. A static member function does not have *this* pointer. The following example demonstrates this:

```
#include <iostream.h>
struct X
 {
    private:
        int i;
        static int si;
    public:
        void set_i(int arg) {  = arg; }
        static void set_si(int arg) { si = arg; }
        void print_i() {
        cout << "Value of i = " << i << endl;
        cout << "Again, value of i = " << this->i << endl;
    }
    static void print_si() {
        cout << "Value of si = " << si << endl;
        //cout << "Again, value of si = " << this->si << endl;
    }
};
int X::si = 77; //Initialize static data member
int main() {
    X xobj;
    xobj.set_i(11);
    xobj.print_i();
    //static data members and functions belong to the class and
    //can be accessed without using an object of class X
    X::print_si();
    X::set_si(22);
    X::print_si();
}
```

Output:

```
Value of i = 11
Again, value of i = 11
Value of si = 77
Value of si = 22
```

As the member function A::print_si() is declared as static and hence, it does not have *this* pointer. So the member access operation this->si is not allowed by the compiler.

You can call a static member function using *this* pointer of a non-static member function. In the following example, the non-static member function *printall()* calls the static member function f() using *this* pointer:

```
#include <iostream.h>
class A {
   static void xyz() {
      cout << "The value of p is:: " << p<< endl;
   }
   static int p;
   int q;
   public:
      A(int q1): q(q1) { }
      void print();
};
void A::print() {
   cout << "the value of q is:: " << this->q << endl;
   this->xyz();
}
int A::p = 3;
int main() {
   A obj_A(0);
   obj_A.print();
}
```

Output:

```
the value of q is: 0
the value of p is: 3
```

It is not allowed in C++ to declare a static member function with the following keywords: virtual, const, volatile, or const volatile. Only the names of the following can be accessed by a static member function: static members, enumerators, and nested types of the class in which it is declared. A static member function cannot be declared with the keywords virtual, const, volatile, or const volatile. A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared. Suppose a static member function A() is a member of class C. The static member function A() cannot access the non-static members C or the non-static members of a base class of C. The figure 4.2 summarizes member functions and data of a class

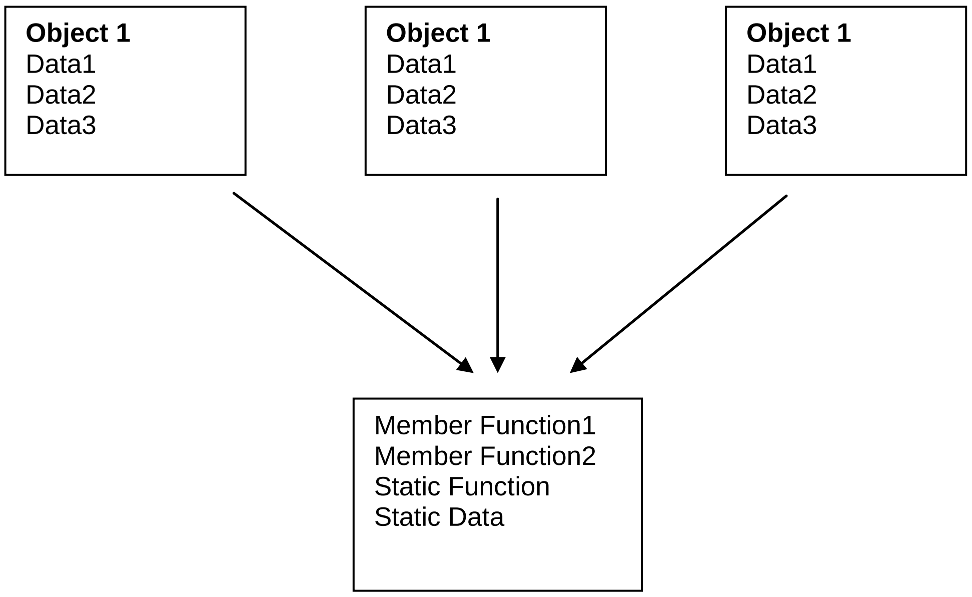

Figure 4.2: Data and Functions in a class

Self Assessment Questions

14. _______ is the data member that is common for all the objects of the class and is accessible for the class.

15. ___________ functions cannot be invoked without the objects of the class.
16. In C++ it is possible to declare static and non-static member functions with the same names. (True/False)

4.12 Summary

- Classes provide users a way to create user defined data type.
- Object is an instance of a class and is used in the program to store data.
- Pointers can point to simple data types, arrays and to objects as well. We use pointers in the situations where we do not know that how many objects we have to create in a program.
- An abstract class is the class which acts as a base class and can be inherited by other classes and is not used to create objects.
- Every time when the program creates a class instance, a special pointer is created in C++ called *this* and it contains the address of the current object instance.
- Friend functions allow us to access some non-member functions or other classes in C++.
- With the possibility of selecting a non-member friend function, there are two more options for its friends. These options are that either the class can declare as a friend a member function of another class or another class also can be declared as a friend.
- Static data member of the class is one data member that is common for all the objects of the class and are accessible for the class.

4.13 Terminal Questions

1. What is *this* pointer?
2. Brief about class and objects.
3. Explain the scope of friend functions.
4. Describe friend functions and friend classes.
5. Discuss static variable and function.

4.14 Answers

Self Assessment Questions

1. Class
2. Abstract
3. Object
4. True
5. Public, private and protected
6. private and public
7. Array of objects
8. Pointers
9. True
10. *this*
11. False
12. Friend
13. Beginning
14. Static Functions
15. Member functions
16. False

Terminal Questions

1. Every time when the program creates a class instance a special pointer is created in C++ called *this* that contains the address of the current object instance. For more details refer section 4.9.
2. Classes provide users a way to create user defined data types. Classes provide a convenient way to group related data and the methods that operate on data together. For more details refer section 4.2.
3. The name of a function first introduced in a friend declaration is in the scope of the first nonclass scope that contains the enclosing class. For more details refer section 4.10.3.
4. With the possibility of selecting a non-member friend function, there are two more options for its friends. These options are: either the class can declare a member function of another class as a friend or another class can also be declared as a friend. For more details refer section 4.10.2.

5. Static data member of the class is one data member that is common for all the objects of the class and is accessible for the class. Static functions are special type of functions which can be invoked even without an object of the class. For more details refer section 4.11.

References:

- Object Oriented Programming In C++, 4/E by Robert Lafore. Pearson Education India.
- Object Oriented Programming with C++ - Sixth Edition, by E Balagurusamy. Tata McGraw-Hill Education.
- Object-Oriented Programming Using C++, By Joyce Farrell. Cengage Learning.