UNIT-II
C++ CLASSES AND DATA ABSTRACTION

CLASS DEFINITION

In C++, the class forms the basis for object-oriented programming. The class is used to define the nature of an object, and it is C++'s basic unit of encapsulation. Classes are created using the keyword class. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an instance of a class. A class declaration is similar syntactically to a structure

General form: Class declaration that does not inherit any other class.

```
class class-name
{
        private: data and functions
        access-specifier:
                data and functions
        access-specifier:
                data and functions
        // ...
        access-specifier:
                data and functions
} object-list;
```

The object-list is optional. If present, it declares objects of the class. Here, access-specifier is one of these three C++ keywords: public, private, protected

Access Specifier

1. private

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class.

2. public

The public access specifier allows functions or data to be accessible to other parts of your program.

3. protected

The protected access specifier is needed only when inheritance is involved. Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

Member Functions

Functions that are declared within a class are called member functions. Member functions may access any element of the class of which they are a part. This includes all private elements.

Member Variables

Variables that are elements of a class are called member variables or data members. (The term instance variable is also used.) Collectively, any element of a class can be referred to as a member of that class.

Restrictions that apply to class members

There are a few restrictions that apply to class members.
1. A non-static member variable cannot have an initializer.
2. No member can be an object of the class that is being declared. (Although a member can be a pointer to the class that is being declared.)
3. No member can be declared as auto, extern, or register.

In general, you should make all data members of a class private to that class. This is part of the way that encapsulation is achieved.

PROGRAM 1: CLASS DEFINITION: LAB PROGRAM WEEK 4, 5

Class Function Definition

It describes how the class functions are implemented. Member functions can be defined in two places

a) Inside the class definition

Member functions can be defined within the class declaration. It simply adds the function directly to the class. This suits only for short function.

Example:

```
class stack
{
        ..
        ..
        ..
        int push(int i)
        {
                if(tos==SIZE)
                {
                        cout << "Stack is full.\n";
                        return;
                }
                stck[tos] = i;
                tos++;
        }
        ..
        ..
        ..
};
```

b) Outside the class definition

When it comes to code a function that is member of a class outside it, we must tell the compiler which class the function belongs to by qualifying its name with the name of the class of which it is a member along with Scope Resolution Operator(::)

Example:

```
void stack::push(int i)
{
        if(tos==SIZE)
        {
                cout << "Stack is full.\n";
                return;
        }
        stck[tos] = i;
        tos++;
}
```

Structures and Classes Are Related

In C++, the role of the structure was expanded, making it an alternative way to specify a class. In fact, the only difference between a class and a struct is that by default all members are public in a struct and private in a class. In all other respects, structures and classes are equivalent. That is, in C++, a structure defines a class type.

For example, consider this short program, which uses a structure to declare a class that controls access to a string:

```
using namespace std;
struct mystr {
void buildstr(char *s); // public
void showstr();
private: // now go private
char str[255];
} ;
void mystr::buildstr(char *s)
{
if(!*s) *str = '\0'; // initialize string
else strcat(str, s);
}
void mystr::showstr()
{
cout << str << "\n";
}
int main()
{
mystr s;
s.buildstr(""); // init
s.buildstr("Hello ");
s.buildstr("there!");
s.showstr();
return 0;
}
```

This program displays the string Hello there!.

The class mystr could be rewritten by using class as shown here:

```
class mystr {
char str[255];
public:
void buildstr(char *s); // public
void showstr();
} ;
```

PROGRAM : STRUCTURES AND CLASSES: LAB PROGRAM WEEK 2

Referencing to a member of class

When we want to refer to a member of a class we use objects name followed by the dot operator, followed by the name of the member. This rule applies whether you are accessing a data member or a function member.

FRIEND FUNCTIONS

It is possible to grant a non-member function access to the private members of a class by using a **friend**. A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**.

PROGRAM 3: FRIEND FUNCTION

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

class myclass
{
  int a, b;
public:
    friend int sum (myclass x);
  void set_ab (int i, int j);
};
void
myclass::set_ab (int i, int j)
{
  a = i;
  b = j;
}

// Note: sum() is not a member function of any class.
int
sum (myclass x)
{
/* Because sum() is a friend of myclass, it can
directly access a and b. */
```

```
  return x.a + x.b;
}

int main ()
{
  myclass n;
  n.set_ab (3, 4);
  cout << sum (n);
  return 0;
}
```

In this example, the sum( ) function is not a member of myclass. However, it still has full access to its private members. Also, notice that sum( ) is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.

OUTPUT:

```
7
```

Circumstances in which friend functions are valuable

1. Friends can be useful when you are overloading certain types of operators
2. Friend functions make the creation of some types of I/O functions easier.
3. Friend functions may be desirable when two or more classes may contain members that are interrelated relative to other parts of your program.

Friend function for interrelated parts of the program

Imagine two different classes, each of which displays a pop-up message on the screen when error conditions occur. Other parts of your program may wish to know if a pop-up message is currently being displayed before writing to the screen so that no message is accidentally overwritten. Although you can create member functions in each class that return a value indicating whether a message is active, this means additional overhead when the condition is checked (that is, two function calls, not just one). If the condition needs to be checked frequently, this additional overhead may not be acceptable. However, using a function that is a **friend** of each class, it is possible to check the status of each object by calling only this one function. Thus, in situations like this, a **friend** function allows you to generate more efficient code.

PROGRAM 4: FRIEND FUNCTION FOR INTERRELATED PARTS OF THE PROGRAM

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

const int IDLE = 0;
const int INUSE = 1;
```

```
class C2;                       // forward declaration
class C1
{
int status;                     // IDLE if off, INUSE if on screen
// ...
public:
void set_status (int state);
friend int idle (C1 a, C2 b);
};

class C2
{
int status;                     // IDLE if off, INUSE if on screen
// ...
public:
void set_status (int state);
friend int idle (C1 a, C2 b);
};

void C1::set_status (int state)
{
  status = state;
}
void C2::set_status (int state)
{
  status = state;
}
int idle (C1 a, C2 b)
{
    if (a.status || b.status)
    return 0;
    else
    return 1;
}
int main ()
{
  C1 x;
  C2 y;
  x.set_status (IDLE);
  y.set_status (IDLE);
  if (idle (x, y))
    cout << "Screen can be used.\n";
  else
    cout << "In use.\n";
  x.set_status (INUSE);
  if (idle (x, y))
```

```
  cout << "Screen can be used.\n";
 else
  cout << "In use.\n";
 return 0;
}
```

this program uses a *forward declaration* (also called a *forward reference*) for the class C2. This is necessary because the declaration of idle( ) inside C1 refers to C2 before it is declared. To create a forward declaration to a class, simply use the form shown in this program.

OUTPUT:

```
Screen can be used.
In use.
```

A friend of one class may be a member of another.

PROGRAM 5: FRIEND OF ONE CLASS MAY BE MEMBER OF ANOTHER

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

const int IDLE = 0;
const int INUSE = 1;

class C2;                       // forward declaration
class C1
{
int status;                     // IDLE if off, INUSE if on screen
// ...
public:
void set_status (int state);
int idle (C2 b);          // now a member of C1
};

class C2
{
int status;                     // IDLE if off, INUSE if on screen
// ...
public:
void set_status (int state);
friend int C1::idle (C2 b);
};

void C1::set_status (int state)
{
 status = state;
}
void C2::set_status (int state)
```

```
{
  status = state;
}

// idle() is member of C1, but friend of C2
 int C1::idle (C2 b)
{
  if (status || b.status)
    return 0;
  else
    return 1;
}

int main ()
{
 C1 x;
 C2 y;
 x.set_status (IDLE);
 y.set_status (IDLE);
  if (x.idle (y))
    cout << "Screen can be used.\n";
  else
    cout << "In use.\n";
  x.set_status (INUSE);
  if (x.idle (y))
    cout << "Screen can be used.\n";
  else
    cout << "In use.\n";
  return 0;
}
```

Because idle( ) is a member of C1, it can access the status variable of objects of type C1 directly. Thus, only objects of type C2 need be passed to idle( ).

OUTPUT:

```
Screen can be used.
In use.
```

Restrictions that apply to friend functions

There are two important restrictions that apply to **friend** functions.

1. A derived class does not inherit **friend** functions.
2. **Friend** functions may not have a storage-class specifier. That is, they may not be declared as **static** or **extern**.

Friend Classes

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class.

When one class is a **friend** of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the **friend** class. Friend classes are seldom used. They are supported to allow certain special case situations to be handled.

PROGRAM 6: FRIEND CLASSES

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

class TwoValues
{
int a;
int b;
public:
TwoValues (int i, int j)
  {
    a = i;
    b = j;
  }
friend class Min;
};

class Min
{
public:
int min (TwoValues x);
};
int Min::min (TwoValues x)
{
 return x.a < x.b ? x.a : x.b;
}

int main ()
{
TwoValues ob (10, 20);
Min m;
cout << m.min (ob);
return 0;
}
```

class Min has access to the private variables a and b declared within the TwoValues class.

OUTPUT:

```
10
```

STATIC CLASS MEMBERS

Both function and data members of a class can be made **static**. This section explains the consequences of each.

Static Data Members

When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created.

When you declare a **static** data member within a class, you are *not* defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class.

This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

PROGRAM 7: STATIC DATA MEMBERS

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

class shared
{
static int a;
int b;
public:
void set (int i, int j)
  {
    a = i;
    b = j;
  }
void show ();
};

int  shared::a;                    // define a
void
shared::show ()
{
  cout << "This is static a: " << a;
  cout << "\nThis is non-static b: " << b;
```

```
  cout << "\n";
}
int main ()
{
 shared x, y;
 x.set (1, 1);              // set a to 1
 x.show ();
 y.set (2, 2);              // change a to 2
 y.show ();
 x.show ();                         /* Here, a has been changed for both x and y
                                       because a is shared by both objects. */

 return 0;
}
```

the integer a is declared both inside shared and outside of it. this is necessary because the declaration of a inside shared does not allocate storage.

OUTPUT:

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

A **static** member variable exists *before* any object of its class is created.

Use of a static member variable

1. Used to provide access control to some shared resource used by all objects of a class.
   Example:
   We might create several objects, each of which needs to write to a specific disk file. only one object can be allowed to write to the file at a time. In this case, you will want to declare a **static** variable that indicates when the file is in use and when it is free. Each object then interrogates this variable before writing to the file.

PROGRAM 8: STATIC MEMBER VARIABLE TO PROVIDE ACCESS CONTROL TO SOME SHARED RESOURCE (USE-1)

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

 class cl
{
 static int resource;
public:
int get_resource ();
```

```
void free_resource ()
  {
    resource = 0;
   }
};
int  cl::resource;                          // define resource
int  cl::get_resource ()
{
  if (resource)
    return 0;                       // resource already in use
  else
    {
          resource = 1;
          return 1;                 // resource allocated to this object
    }
}
int main ()
{
cl ob1, ob2;
if (ob1.get_resource ())
    cout << "ob1 has resource\n";

if (!ob2.get_resource ())
    cout << "ob2 denied resource\n";

ob1.free_resource ();  // let someone else use it
  if (ob2.get_resource ())

cout << "ob2 can now use resource\n";
return 0;
}
```

OUTPUT:

```
ob1 has resource
ob2 denied resource
ob2 can now use resource
```

2. Used to keep track of the number of objects of a particular class type that are in existence.

PROGRAM 9: STATIC MEMBER VARIABLE IS TO KEEP TRACK OF THE NUMBER OF OBJECTS OF A PARTICULAR CLASS TYPE

```
#include <iostream>
```

```
using namespace std;

 class Counter
{
public:
static int count;
Counter ()
  {
    count++;
  }
~Counter ()
  {
    count--;
}
};
int   Counter::count;
void  f ();
int main (void)
{
Counter o1;
cout << "Objects in existence: ";
cout << Counter::count << "\n";
Counter o2;
cout << "Objects in existence: ";
cout << Counter::count << "\n";
f ();

cout << "Objects in existence: ";
cout << Counter::count << "\n";
return 0;
}
void f ()
{
  Counter temp;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  // temp is destroyed when f() returns
}
```

the static member variable count is incremented whenever an object is created and decremented when an object is destroyed. This way, it keeps track of how many objects of type Counter are currently in existence.

OUTPUT:

```
Objects in existence: 1
Objects in existence: 2
```

```
Objects in existence: 3
Objects in existence: 2
```

3. Using **static** member variables, you should be able to virtually eliminate any need for global variables. The trouble with global variables relative to OOP is that they almost always violate the principle of encapsulation.

Static Member Functions

Member functions may also be declared as **static**.

Restrictions on **static** member functions

1. They may only directly refer to other **static** members of the class. (Of course, global functions and data may be accessed by **static** member functions.)
2. A **static** member function does not have a **this** pointer.
3. There cannot be a **static** and a non-**static** version of the same function.
4. A **static** member function may not be virtual.
5. They cannot be declared as **const** or **volatile**.

PROGRAM 10: STATIC MEMBER FUNCTIONS

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

 class cl
{
 static int resource;
 public:
 static int get_resource ();
void free_resource ()
  {
    resource = 0;
  }
};

int  cl::resource;                          // define resource
int  cl::get_resource ()
{
if (resource)
    return 0;                          // resource already in use
else
    {

resource = 1;
return 1;                    // resource allocated to this object
```

```
    }
}

int main ()
{
  cl ob1, ob2;
  /* get_resource() is static so may be called independent of any object. */
    if (cl::get_resource ())
    cout << "ob1 has resource\n";

     if (!cl::get_resource ())
    cout << "ob2 denied resource\n";
     ob1.free_resource ();

     if (ob2.get_resource ())      // can still call using object syntax
    cout << "ob2 can now use resource\n";

 return 0;
}
```

OUTPUT:

```
ob1 has resource
ob2 denied resource
ob2 can now use resource
```

Applications

Static member functions have limited applications, but one good use for them is to "preinitialize" private **static** data before any object is actually created.

PROGRAM 11: STATIC MEMBER FUNCTIONS APPLICATION TO "PREINITIALIZE" PRIVATE STATIC DATA

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

class static_type
{
  static int i;
  public:
  static void init (int x)
  {
    i = x;
  }
void show ()
```

```
  {
    cout << i;
}
};
int  static_type::i;                    // define i
int main ()
{
// init static data before object creation
    static_type::init (100);
    static_type x;

    x.show ();                          // displays 100
    return 0;
}
```

CONSTRUCTORS AND DESTRUCTORS

It is very common for some part of an object to require initialization before it can be used. Because the requirement for initialization is so common, C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor function.

Definition: A *constructor* is a special function that is a member of a class and has the same name as that class.

Example:

stack class to initialize a constructor

```
        // This creates the class stack.
        class stack
        {
                int stck[SIZE];
                int tos;
                public:
                        stack(); // constructor
                        void push(int i);
                        int pop();
        };
```

stack( ) constructor is coded as,

```
        // stack's constructor
        stack::stack()
        {
                tos = 0;
                cout << "Stack Initialized\n";
        }
```

Characteristics

1. Constructors have the same name as that of class.
2. Constructors cannot return values i.e they have no return type.
3. An object's constructor is automatically called when the object is created. This means that it is called when the objects declaration is executed.
4. An objects constructor is called once for global or static local objects
5. For local objects, the constructor is called each time the object declaration is encountered
6. They should be declared as public.
7. They cannot be inherited or virtual.
8. They can have default arguments.

PROGRAM 13: CONSTRUCTORS AND DESTRUCTORS

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

#define SIZE 100
// This creates the class stack.
class stack
{
  int stck[SIZE];
  int tos;
  public:
    stack ();                 // constructor
  ~stack ();                          // destructor
  void push (int i);
  int pop ();
};
// stack's constructor
stack::stack ()
{
    tos = 0;
    cout << "Stack Initialized\n";
}
// stack's destructor
stack::~stack ()
{
  cout << "Stack Destroyed\n";
}
void stack::push (int i)
{
  if (tos == SIZE)
    {
        cout << "Stack is full.\n";
```

```
    return;
   }
  stck[tos] = i;
  tos++;
}

int  stack::pop ()
{
  if (tos == 0)
   {
     cout << "Stack underflow.\n";
    return 0;
   }
  tos--;
  return stck[tos];
}

int main ()
{
   stack a, b;                    // create two stack objects
   a.push (1);
   b.push (2);
   a.push (3);
   b.push (4);
   cout << a.pop () << " ";
   cout << a.pop () << " ";
   cout << b.pop () << " ";
   cout << b.pop () << "\n";
   return 0;
}
```

OUTPUT:

```
Stack Initialized
Stack Initialized
3 1 4 2
Stack Destroyed
Stack Destroyed
```

Types of Constructors

The following are the various types of constructors,

1. **Default Constructor**

   A constructor that accepts no parameters is called constructor.

   Example:

```
test :: test()
{
        ......
        ......
}
```

PROGRAM 14: DEFAULT CONSTRUCTORS

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

class Cube
{
   public:
   int side;
   Cube()
   {
      side = 10;
   }
};

int main()
{
   Cube c;
   cout << c.side;
}
```

OUTPUT:

```
10
```

2. Parameterized Constructor

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created.

To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

PROGRAM 15: PARAMETERIZED CONSTRUCTOR

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

 class myclass
```

```
{

int a, b;
public:
myclass (int i, int j)
  {
    a = i;
    b = j;
  }
void show ()
  {
    cout << a << " " << b;
}
};
int main ()
{
   myclass ob (3, 5);
   ob.show ();
    return 0;
}
```

OUTPUT:

```
3 5
```

The most common way to specify arguments when you declare an object that uses a parameterized constructor is , myclass ob(3, 4); causes an object called **ob** to be created and passes the arguments **3** and **4** to the **i** and **j** parameters of **myclass( )**. You may also pass arguments using this type of declaration statement:

myclass ob = myclass(3, 4);

PROGRAM 16: PARAMETERIZED CONSTRUCTOR

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

class Employee
{
   public:
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)
      float salary;
      Employee(int i, string n, float s)
       {
          id = i;
```

```
        name = n;
        salary = s;
    }
    void display()
    {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
    }
};
int main(void)
{
   Employee e1 =Employee(101, "Son", 890000); //creating an object of Employee
   Employee e2=Employee(102, "moon", 59000);
   e1.display();
   e2.display();
   return 0;
}
```

OUTPUT:

```
101  Son  890000
102  moon  59000
```

When useful

Parameterized constructors are very useful because they allow you to avoid having to make an additional function call simply to initialize one or more variables in an object. Each function call you can avoid makes your program more efficient.

Constructors with One Parameter: A Special Case

If a constructor only has one parameter, there is a third way to pass an initial value to that constructor.

For example, consider the following short program.

```
#include <iostream>
using namespace std;
class X
{
        int a;
        public:
        X(int j)
        {
                a = j;
         }
        int geta()
        {
                return a;
        }
};
```

```
int main()
{
        X ob = 99; // passes 99 to j
        cout << ob.geta(); // outputs 99
        return 0;
}
```

99 is automatically passed to the **j** parameter in the **X( )** constructor. That is, the declaration statement is handled by the compiler as if it were written like this:

X ob = X(99);

In general, any time you have a constructor that requires only one argument, you can use either *ob*(*i*) or *ob = i* to initialize an object. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.

PROGRAM 17: PARAMETERIZED CONSTRUCTOR WITH ONE PARAMETER

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

 class X
{
   int a;
   public:
  X (int j)
  {
    a = j;
  }
  int geta ()
  {
    return a;
  }
};

int main ()
{
  X ob = 99;                    // passes 99 to j
  cout << ob.geta ();           // outputs 99
  return 0;
}
```

OUTPUT:

```
99
```

PROGRAM 18: PARAMETERIZED CONSTRUCTOR WITH ONE PARAMETER

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

class Cube
{
   public:
   int side;
   Cube(int x)
   {
      side=x;
   }
};

int main()
{
   Cube c1(10);
   Cube c2(20);
   Cube c3(30);
   cout << c1.side<<endl;
   cout << c2.side<<endl;
   cout << c3.side<<endl;
}
```

OUTPUT:

```
10
20
30
```