Unit 3: Classes & Objects [8 Hrs.]

Class and Objects

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.

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank a bank account, a table of data or any item that the program has to handle.

Example

```
#include<iostream>
using namespace std;
class addition
{
        private:
        int a, b, sum;
        public:
                void getdata();
                void processing();
                void display();


};
void addition::getdata()
                {
                        cout<<"\n Enter two numbers";
                        cin>>a>>b;
                }

void addition::display()
                {
                        cout<<"\n Sum of two numbers is ="<<sum;
                }

void addition:: processing()
                {
                        sum=a+b;
                }

int main()
{
```

```
        addition a;
        a.getdata();
        a.processing();
        a.display();
        return 0;
}
```

Nesting of member functions:

We know that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as nesting of member function.

Example:

```
#include<iostream>
using namespace std;
class set
{
int m,n;
public:
void input(void);
void display(void);
int largest(void);
};

        int set :: largest(void)
        {
                if(m >= n)
                return(m);
                else
                return(n);
        }
        void set :: input(void)
        {
                cout << "Input value of m and n"<<"\n";
                cin >> m>>n;
        }
        void set :: display(void)
        {
                cout << "largest value=" << largest() <<"\n";
                }

        int main()
        {
```

```
        set A;
        A.input();
        A.display();
        return 0;
}
```

Initialization of class Objects (Constructor, Destructor)

Constructor

A constructor is a special member function whose task is to initialize the object of its class . It is special because it name is same as that of class name. Constructor is invoked whenever the object of its associated class is created. It is called constructor because it constructs the values of data members of the class . It has no return type, not even void but can take arguments.

Examples:

```
class  sample
{
        int a, b;
        public:
        Sample () // constructor
        {
        a = 0; b=0;
        }
} ;
int  main ()
{
        Sample s1; // object s1 created
        ……
        ……..
}
```

Above object declaration not only creates the object s1, but also initializes its data member's 'a' and 'b' to zero.

Some characteristics of constructor

1. They should always be declared in public section.
2. They are invoked automatically when the object are created.
3. They don't have return types, not even void so they can't return values.
4. They can't be inherited, though a derived class can call the base class constructor.
5. Like other C++ function, they can have default arguments.

Default Constructor

A constructor that takes no arguments is called the default constructor. It is invoked automatically when the object is created.

Example

```
#include <iostream>
using namespace std;
class  Demo
{
        int p,g;
        public:
        Demo () // constructor declared / defined
        {
        p = 0;
        g = 0;
        }
void display ()
        {
        cout <<p<<g;
        }
};
        int main ()
        {
        Demo d;
        d.display ();
        }
```

Parameterized Constructor

Sometime, it may be necessary to initialize the various data elements of different objects with different values when they are created. Constructors that take arguments as parameters are called parameterized constructor.

Example:

```
class  integer
{
        int m, n;
        public:
        integer (int x, int y); // parameterized constructor
};
integer:: integer (int x,int y)
```

```
{
        m = x;
        n = y;
}
```

When a constructor has been parameterized. The object declaration statement such as:

Integer int1; may not work.

Now, we must pass the initial values as arguments to the constructor function, when an object is declared. This can be done in 2 ways:-

1. By calling the constructor explicitly
2. By calling the constructor implicitly

Example:

integer int1 = integer (0, 100); // explicit call

This statement creates an integer object int1 and passes the values 0 and 100

Eg: integer int1 (0, 100); // implicit call

This method is sometimes called shorthand method and is used very often as it is shorter, looks better and is easy to implement.

Eg:

```
#include <iostream>
 using namespace std;
class  param
        {
                int a, b;
                public:
                param (int, int);
                void display ()
                {
                cout <<a<<"  "<<b<<endl;
                }
        };
        param:: param (int x, int y)
                {
                a = x;
                b = y;
                }
        int main()
        {
                param p1 = param (10, 20); // expilit call
                param p2(50,60);//implicit call
                p1.display();
                p2.display();
        }
```

Notes:

The parameter of a constructor can be of any type except that of the class to which it belongs

Eg.

```
class  B
{
…….
B(B); // it is illegal
……..
};
```

However, a constructor can accept reference to its own class as parameters.

Eg:

```
class  B
{
B(&B); // it is legal
…….
……
};
```

Copy Constructor

It is used to declare and initialized an object from another object i.e it is used to copy data members of an object of a class into another object of the same class .

For eg:

```
sample (sample & s); // copy constructor
```

Eg:

```
sample s2(s1);
```

Would define the object s2 and at the same time initialize it to the value of s1.

Another form:

```
Sample s2 = s1;
```

The process of initializing through a copy constructor is known as copy initialization.

Note:

S2 = S1;

Will not invoke the copy constructor but, if S1 and S2 are objects, then this statement is legal and assigns the value of S1 and S2. This is the task of overloaded assignment operator (=).

A copy constructors takes a reference to an object of the same class as itself as an argument.

Example of copy constructor

```
#include<iostream>
using namespace std;
class  Copy
        {
                int id;
```

```
                public:
                Copy() { }// constructor

                Copy(int x) //constructor
                {
                id = x;
                }
                Copy(Copy & y) //copy constructor
                {
                id = y.id;
                }
                void display()
                { cout<<id; }
        };
int main()
{
        Copy A (20);        // object A is created & initialized
        Copy B(A);          // copy constructor called
        Copy C = A;         // copy constructor called again
        Copy D;     // D is created, not initialized
        D = A;      // copy constructor not called

        cout<<"\n id of A: "; A.display();
        cout<<"\n id of B: "; B.display();
        cout<<"\n id of C: "; C.display();
        cout<<"\n id of D: "; D.display();
}
```

O/P

id of A:20
id of B:20
id of C:20
id of D:20

Notes:

A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.

When no copy constructor is defined, the compiler supplies its own copy constructor. If we don't define our own copy constructor, the C++ compiler creates a ***default copy constructor*** for each class which does a member-wise copy between objects. The compiler created copy constructor works fine in general. (*NOTE: Students are advised to check above program without using copy constructor definition*)

Multiple constructors in a class

integer (); // no arguments.
Integer (int, int); // two arguments
C++ allows both these constructors in the same class :

```
class integer
{
        int x,y;
        public:
        integer ()  // constructor 1
        { x=0; y=0; }

        integer(int a, int b) // constructor 2
        { x=a; y=b; }

        integer (integer & i) // constructor 3
        { m=i.m; n=i.n;}

};
```

This example given declares these constructors for an integer object.

* First one receives no arguments i.e.
  Integer I1;
  Would automatically invoice 1st constructor and sets both x and y of I1 to 0.
* Second one receives two integer arguments i.e.
  integer I2 (30,80);
  Would call the second constructor, which will initialized the data members x & y of I2 to 30 and 80 respectively.
* Third one receives one integer object as an argument.
  integer I3 (I2);
  Would invoke the third constructor which copies the values of I2 into I3. It sets the value of every data elements of I3 to the value of corresponding data elements of I2. Such a constructor is called copy constructor.

As the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, a constructor is overloaded when more than one constructor is defined in a class .

Overloaded Constructor:

Example 1

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

```
{
        float x,y;
        public:
        complex(){}
        complex(float a){x=y=a; }
        complex(float real, float imag)
        {x= real; y= imag;}
        void sum(complex,complex);
        void show();
};
void complex::sum (complex c1, complex c2)
{
        x = c1.x + c2.x;
        y= c1.y + c2.y;

}

void complex::show()
{
        cout<<x<<"+j"<<y<<"\n";
}
int main()
{
        complex A(2.7,3.5);
        complex B(1.6);
        complex C;
        C.sum(A,B);

        cout<<"A="; A.show();
        cout<<"B=";B.show();
        cout<<"C=";C.show();
        return 0;
}
```

Example 2: Using Friend function

```
# include <iostream>
using namespace std;
class  complex
{
    float x,y;
    public:
```

```
complex() { }
complex(float a)
{    x=y = a; }

complex(float real, float imag)
     {   x=real;   y=imag; }
     friend complex sum(complex, complex);
     friend void show (complex);
};
     complex sum (complex c1, complex c2)
     {
     complex c3;
     c3.x = c1 . x + c2 . x ;
     c3.y = c1 . y + c2 .y;
     return(c3);
     }
     void show (complex c)
     {
     cout << c . x << "+j" << c.y << "\n";
}
     int main()
{
     complex A (2.7, 3.5);
     complex B (1.6);
     complex C;
     C = sum(A,B);
     cout<<"A = ";  show (A);
     cout <<"B = ";  show (B);
     cout <<"C = ";  show (C);
}
```

O/P:

A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1

Destructor:

⇒ It is used to destroy the objects that have be created by a constructor.
⇒ The destructor is a member function whose name is same as the class name but is preceded by a tilde (~)
⇒ For eg: the destructor for the class integer can be defined as shown below:
~ integer () { }
⇒ A detector never takes any arguments nor does it return any values.
⇒ Object are destroyed in the reverse order of their creation.

Example:

```
#include <iostream>
 using namespace std;
int count = 0;
class  destruct
{
public:
destruct ()
     {
     count++;
     cout <<"\n no. of object created is" << count;
     }
  ~destruct ()
     {
     cout <<"\n no. of object destroyed" <<count;
     count --;
     }
};
int main()
     {
     destruct d1, d2, d3;
     return 0;
     }
```

O/P

```
No. of object created : 1
No. of object created : 2
No. of object created : 3

No. of object destroyed : 3
No. of object destroyed : 2
No. of object destroyed : 1
```

Example:

```
# include <iostream>
using namespace std;
int count=0;
class  alpha
{
public:
alpha()
     {
```

```
        count++;
        cout<<"\n no. of object created"<<count;
        }
     ~alpha()
        {
        cout<<"\n no. of object destroyed"<<count;
        count--;
        }
    };
    int main ()
    {   cout <<"\nenter main";
        {  alpha A1, A2, A3, A4;
                {  cout<<"\n enter block1";
                        alpha A5;
                }

                {
                cout<<"\n enter block2";
                alpha A6;
                }
        cout<<"\n RE-ENTER MAIN";
        }
return 0;
    }
```

O/P

Enter Main
No. of object created 1
No. of object created 2
No. of object created 3
No. of object created 4

Enter Block 1
No. of object created 5
No. of object destroyed 5

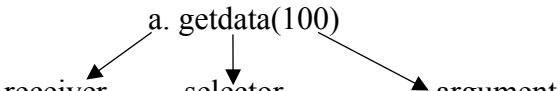
Enter Block 2
No. of object created 5
No. of object destroyed 5
RE-ENTER MAIN
No of object destroyed 4
No of object destroyed 3
No of object destroyed 2

No of object destroyed 1

Message Passing (Method-Lookup):

It is a dynamic process of asking an object to perform action:

1. A message is always given to some object called receiver.
2. The action performed in response to message is not fixed but may differ depending upon the class of the receiver. i.e. different object may accept the same message yet perform different actions.
3. There are three identifiable parts to any message:
   i) Passing expression: These are receiver (object to which message is being sent)
   ii) The message selector (the text that indicates the particular message being sent)
   iii) The argument (used in responding to the message).



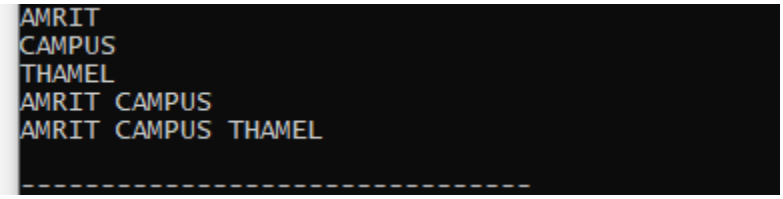
Where a is declared as an instance of class .

Dynamic Constructor:

```
# include<iostream>
# include<string.h>
using namespace std;
class  string1
    {
    char *name;
    int length;
    public:
    string1 ()
    {
    length = 0;
    name = new char [length +1];
    }
    string1 (char *s)
    {
    length = strlen(s);
    name = new char [length +1]; //+1 for additional null character
    strcpy (name, s);
    }
    void display ()
    {
    cout << name <<"\n";
    }
void join (string1 &a, string1 &b);
```

```
};
void string1 :: join (string1 &a, string1 &b)
    {
    length = a.length + b. length;
    delete name;
    name = new char [length +1]; // dynamic allocation
    strcpy (name, a.name);
    strcat (name, b.name);
    }
int main ()
{
char *first = "AMRIT ";
string1 name1 (first), name2("CAMPUS "), name3("THAMEL "), S1, S2;
S1.join (name1, name2);
S2.join (S1, name3);
name1.display ();
name2.display ();
name3.display ();
S1.display ();
S2.display ();
return 0;
}
```

Output



Note:

The constructor can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of object. The memory is allocated with the help of "new" operator.

Object as Function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways.

i) A copy of the entire object is passed to the function.
ii) Only the address of the object is transferred to the function

First method is pass-by-value. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called pass-by-references. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Example program of object as function argument:

Example 1

```
#include<iostream>
using namespace std;
class time
{
int hours;
int minutes;
public:
void gettime(int h, int m)
{
hours=h;
minutes=m;
}
void puttime(void)
{
cout<< hours<<"hours and ";
cout<<minutes<<"minutes"<<endl;
}

void sum( time ,time);
};
void time :: sum (time t1,time t2)
{
minutes=t1.minutes + t2.minutes;
hours=minutes%60;
minutes=minutes%60;
hours=hours+t1.hours+t2.hours;
}
int main()
{
time T1,T2,T3;
T1.gettime(2,45);
T2.gettime(3,30);
T3.sum(T1,T2);
```

```
cout<<"T1=";
T1.puttime( );
cout<<"T2=";
T2.puttime( );
cout<<"T3=";
T3.puttime( );
return(0);
}
```

Output:

```
T1=2hours and 45minutes
T2=3hours and 30minutes
T3=20hours and 15minutes
```

Example 2

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

class Demo
{
        private:
                int a;

        public:
                void set(int x)
                {
                        a = x;
                }

                void sum(Demo ob1, Demo ob2)
                {
                        a  = ob1.a + ob2.a;
                }

                void print()
                {
                        cout<<"Value of A :  "<<a<<endl;
                }
};

int main()
{
```

```
        //object declarations
        Demo d1;
        Demo d2;
        Demo d3;

        //assigning values to the data member of objects
        d1.set(10);
        d2.set(20);

        //passing object d1 and d2
        d3.sum(d1,d2);

        //printing the values
        d1.print();
        d2.print();
        d3.print();

        return 0;
}
```

Output:

```
Value of A :  10
Value of A :  20
Value of A :  30
```

Returning Objects from Function

```
#include <iostream>
using namespace std;
class Example {
public:
        int a;

        Example add(Example Ea, Example Eb)
        {
                Example Ec;
                Ec.a = Ea.a + Eb.a;
                // returning the object
                return Ec;
        }
};
```

```
int main()
{
        Example E1, E2, E3;
        E1.a = 50;
        E2.a = 100;
        E3.a = 0;

        cout << "Initial Values \n";
        cout << "Value of object 1: " << E1.a
                << "\nobject 2: " << E2.a
                << "\nobject 3: " << E3.a
                << "\n";

        // Passing object as an argument
        // to function add()
        E3 = E3.add(E1, E2);

        cout << "\n\nNew values \n";
        cout << "Value of object 1: " << E1.a
                << "\nobject 2: " << E2.a
                << "\nobject 3: " << E3.a
                << "\n";

        return 0;
}
```

Output

```
Initial Values
Value of object 1: 50
object 2: 100
object 3: 0


New values
Value of object 1: 50
object 2: 100
object 3: 150

--------------------------------
```

Structures and Classes

Structures in C

Structure provides method for packing together the data of different types.
Convenient tool for handling a group of logically related data items.
Consider:

```
struct complex
{
```

int real;
int img;
};
struct complex c1, c2, c3;
The complex number c1, c2, c3 are assigned values using a dot or period operator.

We cannot add or subtract two structure variables in C.
Also data hiding is not permitted in C

Structures in C++

C++, Structure supports all features of Structures in C.
C++, Structure can have both data and function. Data are called data member while functions are called the member function.
In C++, structure names are Stand-alone i.e. Keyword struct may be omitted in the declaration of structure variable.
E.g.

```
struct student
{
char name[20];
int roll;
};
```

Student s1, s2; //C++ declaration. It is invalid in C.
In C++, a structure member can be declared as *"Private"* so that they cannot be accessed directly by the external function.

Memory Allocation for Objects

"The memory space for objects is allocated when they are declared and not when the class is specified." This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for the member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects.

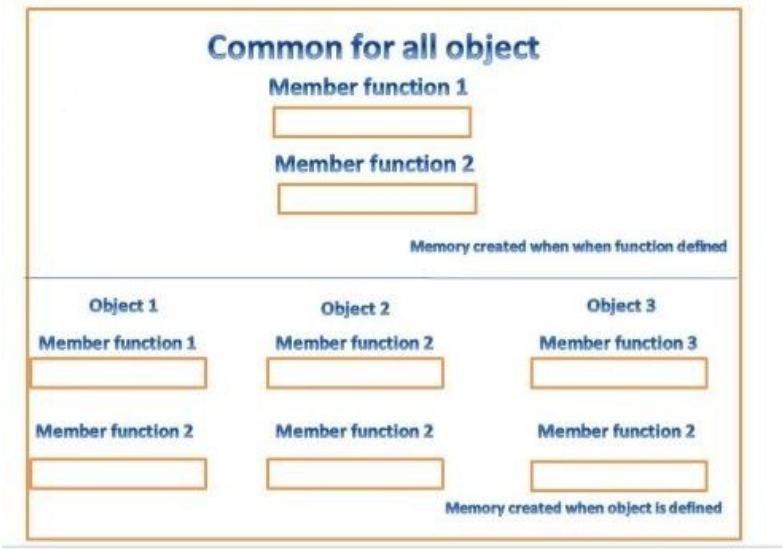


Fig: Object of Memory

Static Members

- The properties of static member variable are similar to that of a C static variable.
- A static member variable has certain special characteristics. These are:
  - It is initialized to zero when the first object of its class is created. No other initialization is permitted.
  - Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
  - It is visible only within the class, but its lifetime is the entire program.
  - Static variable are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects.

Example 1 :

```
#include<iostream>
using namespace std;
class item{

        static int X ;
                public:
                void get()
                {
```

```
                        cout<<"X = "<<X<<"\n";
                        X++;
        }


};

int item :: X;//definition of static data member. can also be initialized as
                        // int item :: count = 10;
int main()
{
        item a,b,c;

        a.get();
        b.get();
        c.get();

return 0;
}
```

Output:

```
X = 0
X = 1
X = 2
```

Example 2

```
#include<iostream>
using namespace std;
class item
{
static int count; //count is static
int number;
public:
void getdata(int a)
{
number=a;
count++;
}
void get_count(void)
{
cout<<"count:";
cout<<count<<endl;
}
};
```

```
int item :: count ; //count defined

int main( )
{
item a,b,c;
a.get_count( );
b.get_count( );
c.get_count( );
a.getdata(100);
b.getdata(200);
c.getdata(300);
cout<<"after reading data : "<<endl;
a.get_count( );
b.get_count( );
c.get_count( );
return(0);
}
```

Output:

```
Count: 0
Count: 0
Count: 0
after reading data :
Count: 3
Count: 3
Count: 3
```

Member Function defined Outside the Class

Member functions can be defined in two places

- Outside the class definition
- Inside the class definition

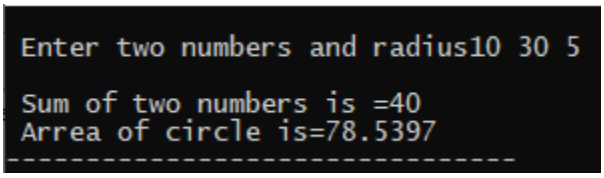
The general form of a member function definition outside the class is:

```
return-type class-name : : function-name ( argument declaration )
{
        Function body
}
```

Example

```
#include<iostream>
#define pi 3.14159
using namespace std;
class addition
{
	private:
	int a, b, sum;
	float r,area;
	public:
void getdata();
void display();
void processing();

};

void addition::getdata()
		{
			cout<<"\n Enter two numbers and radius";
			cin>>a>>b>>r;
		}


void addition::display()
		{
			cout<<"\n Sum of two numbers is ="<<sum;
			cout<<"\n Arrea of circle is="<<area;
		}

void addition::processing()
		{
			sum=a+b;
			area=pi*r*r;
		}


int main()
{
	addition a;
	a.getdata();
	a.processing();
	a.display();
	return 0;
}
```

Output


Enter two numbers and radius10 30 5
Sum of two numbers is =40
Arrea of circle is=78.5397