CLASSES AND DATA ABSTRACTION:

CLASS AND OBJECT:

- Class is similar to the concept of user-defined data type.
- Class can also be called as collection of objects.
- Object is a variable of type class.

SYNTAX TO DECLARE CLASS IS:

```
class class_name
{
private:        var_declaration;
                function declaration;
 public:        var_declaration;
                function declaration;
 protected:     var_declaration;
                function declaration;
};
```

- **private, public and protected** are **access specifiers or visibility labels.**
- **Private members** can be accessed only with in class and functions of that class
- **Public members** can be accessed anywhere in the program
- **Protected members** can be accessed by a class and its derived class, and also by the derived class of the derived class.

SYNTAX TO DECALRE OBJECT IS:

```
Class_name    object_name;
```

TO ACCESS MEMBERS OF CLASS:

```
Object_name.function();
```

Program to find average of 3 numbers using the concept of class, object and member function

```
#include<iostream>
using namespace std;
class demo
{
private:
    int x,y,z;
    float k;
public:
    void input(int,int,int);
    void output();
    void average();
};

void demo::input(int p,int q,int r)
{
 x=p; y=q; z=r;
}

void demo::output()
{
 cout<<k;
}

void demo::average()
{
 k=(x+y+z)/3;
}

main()
{
```

```
 demo d;
 d.input(10,20,30);
 d.average();
 d.output();
}
```

- **A function can be defined either inside or outside the class.**
- **If it is defined outside then:**
  It should be written along with its class name and ::
  Scope Rules:

  Class Scope: Data members and member functions of a class belongs to the class scope.
  File Scope: Non-member functions will have file scope.
  Function Scope: Variable defined in a member function has function scope. i.e.., they are known to that function.

  Note:
  If a member function defines a variable with the same name as a variable with the class scope, the class scope variable is hidden by function scope variable in the function scope such variable can be accessed by **class name::variable name**
  Example: (refer class notes)

  <u>Initializing Class Objects:</u>

  <u>CONSTRUCTOR</u>

- In c-language values will be passed through function call as:
  function_name(arguments);
- In cpp it is done as follows:
  Values are passed by calling the function along with object name.
- In c-language variables are initialized at the time of declaration itself.
- That is, int x =10;
- In the same way, in order to enable an object to pass values at the time of its creation only, we use a special member function called as CONSTRUCTOR.

- **Constructor is a special member function that is called automatically when an object is created to a class.**

  PROPERTIES OF CONSTRUCTOR:

  - Name of the constructor function is same as class name
  - It has no return type not even void
  - There is no private constructor i.e., Constructor must be public.
  - Constructor may or may not have parameters.
  - Constructor cannot return a value.
  - Constructor cannot be friend function to any other class.

- **Syntax of constructor is:**

  class_name()
  {
  ….
  ….}

- **Types of constructors are:**

  1. **Default constructor**
  - When a class is created then compiler automatically allocates a constructor called as default constructor. **(**A class can contain only one default constructor i.e default constructor cannot be overloaded**)**
  2. **Non-parameterized constructor**
  - Constructor with no parameters
  3. **Parameterized constructor**
  - Constructor with parameters. **(**We can pass arguments to constructor while creating the object within the parenthesis beside object**)**
  4. **Copy constructor**
  - Passing object as an argument.
  - That is initializing object of one class with other object of same class is called **copy initialization.**
  - Such a constructor is called as copy constructor

Rules for copy constructor:

- It can have only one parameter
- It must be reference parameter
- Parameter type must be same as class name.

OVERLOADING OF CONSTRUCTOR MEANS GIVING A DIFFERENT MEANING TO THE CONSTRUCTOR.

Multiple Constructor: (Overloading Constructors)
That is more than one constructor can be defined for a class.

Program for Constructor Overloading

```
#include<iostream>
using namespace std;
class abc
{
private: int a,b;

public:
        abc( )
        {
        cout<<"default constructor \n";
        a=10;
        b=20;
        }

        abc( int x, int y)
        {
        cout<<"parameterized constructor \n";
        a=x;
```

```
        b=y;
        }

        abc(abc &i)
        {
        cout<<"copy constructor \n";
        a=i.a;
        b=i.b;
        }

         void diaplay()
        {
        cout<<"a"<<a<<"b"<<b<<endl;
        }
}
main()
{
abc     ob1;
abc     ob2(30,40);
abc     ob3(ob2);
ob1.display();
ob2.display();
ob3.display();
}
```

Output:

```
a  10   b  10
a  30   b  40
a  30   b  40
```

DESTRUCTOR

Destructor is a member function which is automatically called when an object of the class is destroyed or terminated.

Syntax : ~class_name()

{

}

Note:

- The destructor function also has the same name as a class but preceded by tilde(~) character.
- The name of the destructor for a class is a tilde followed by the class name.
- Class destructor is called when an object is destroyed.
- A destructor freezes the memory occupied by the object so that it can be reused to hold new object.
- Destructor is declared in the public section of a class
- Destructor receives no parameters and no return value.
- **Destructor cannot be overloaded.**
-

MAIN DIFFERENCE BETWEEN CONSTRUCTOR AND DESTRUCTOR:

While a class can have **multiple constructors** but can have only **one destructor.**
Default constructor cannot be overloaded.

Program to demonstrate destructor

Ex:1

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

class abc
{
public:
    abc()
    {
            cout<<"constructor is called\n";
    }
```

```
    ~abc()
    {
            cout<<"destructor is called\n";
    }
};

main( )
{
abc ob1;
}
```

Output:

Constructor is called
Destructor is called

Program to demonstrate destructor
Ex:2

```
#include<iostream>
using namespace std;
int c=0;
class abc
{
```

```
public:
    abc()
    {
            c++;
            cout<<" object"<<c<<”is created\n”;
    }
    ~abc()
    {
            cout<<"object"<<c<<”is destroyed\n”;
            c--;
    }
};
main()
{
    cout<<" In main";
    abc   a1,a2;
    {
            cout<<"block-1";
            abc   a3;
    }
}
```

Output:

```
In main
Object 1 is created
Object 2 is created
Object 3 is created
Object 3 is destroyed
Object 2 is destroyed
Object 1 is destroyed
```

FRIEND FUNCTION

[Generally in C++, only public functions have right to access private variables of the class. In addition C++ has one more function to access private variables of the same class even outside the class.]

- **"friend"** is a keyword.
- **Friend function** is used to access the private data of the class.
- Friend function takes object of a class as an argument
- Friend function cannot refer members of class directly. It uses the object to refer them
- Friend function cannot be called using object of class, it should be called directly.
- Normal values cannot be passed as arguments in friend function.

A friend function of a class is defined outside the class scope and yet has right to access private members of the class.
A function or entire class may be declared to be a friend of another class.
To declare a function as a friend of a class receive the function prototype in class declaration with friend keyword.
Syntax:

```
friend     datatype   functionname( );
```

Friend function is not a member function of the class. i.e., it is a non-member function.

Characteristics:

It is not in the scope of the class to which it has been declared as friend.
It cannot be called using the object of the class.
It can be called like a normal function without the help of any object.
It can be declared either in **public** or **private** part of the class.
It has an object as an argument.

Program to demonstrate friend function

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

class sample
{
private:
        int m1,m2,m3;
public:
        void getmarks();
        friend void total(sample );
};

void sample:: getmarks()
{
    cout<<"enter m1 m2 m3 marks\n";
    cin>>m1>>m2>>m3;
}

Int total(sample s)
{
    return (s.nm1+s.m2+s.m3);
 }

main()
{
 sample    e;
 e.getmarks();
 cout<<"total marks "<<total(e);
}
```

Output:

```
        enter m1 m2 m3 marks
```

90 80 70
total marks 240

“this” POINTER

“this” is a pointer that points to the object for which the member function is called.

or

- “this” pointer is used to represent an object that invokes a member function.

Program to demonstrate “this” pointer

```
#include<iostream>
using namespace std;
class abc
{
    int x;
public:
    abc()
    {
      x=10;
    }
    void function(int  x)
    {
     cout<< x;//20
     cout<< this->x;//10
    }
};

main()
{
```

```
abc   a1;
a1.function(20);
}
```

Output: 20
10

Application of 'this' pointer:

- To return the object to which it points to.
- To compare two or more objects inside a member function and return the invoking object as a result.

Program to find maximum of 2 numbers using "this" pointer

```
#include<iostream>
using namespace std;
class max
{
        int a;
public:
        void get()
        {
        cout<<"enter a";
        cin>>a;
        }

        void compare(max m)
        {
                if(this->a  > m.a)
                cout<<"max no. is"<<this->a;
```

```
                else
                cout<<"max no.  is"<<x.a;
            }
};
main()
{
max p,q;
p.get();
q.get();
p.compare(q);
}
```

Out put:

Enter a 10
Enter a 20
Max no. is 20

Dynamic Memory Allocation:

C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time.

C++ also supports dynamic memory allocation functions, called **malloc()** and **free().**
.
The **new** operator allocates memory and returns a pointer to the start of it.
The **delete** operator frees memory previously allocated using **new**.

The general forms of **new** and **delete** are shown here:

*Datatype *variable* = new data*type*;

delete v*ariable*;

Here,variable is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

```
main()
{
int *p = new int; // allocates memory( space )  for an int
```

```
*p = 100;
cout << "At " << p << " ";
cout << "is the value " << *p << "\n";
delete p;          // deallocates memory for variable p
}
```

Allocating for Arrays:

You can allocate arrays using **new** by using this general form:

Datatype * variable* =new *datatype [size];

Here, *size* specifies the number of elements in the array.
To free an array, use this form of **delete:**

delete [ ] *variable;*

Here, the **[ ]** informs **delete** that an array is being released.

In C language we are provided with dynamic memory allocation functions like malloc(),calloc(),free().

For Allocating memory at run time:
malloc(): allocate memory dynamically at run time

Syntax:
Data type *variable = (datatype *) malloc(sizeof(datatype));

Ex: int *p = (int *) malloc(sizeof(int));

For Arrays:
Calloc(): allocates memory dynamically at run time

Syntax:
Data type *variable = (datatype *) calloc(n, sizeof(datatype));
Where: n is size of array.

Ex: int *p = (int *)calloc(10,sizeof(int)); // which allocates 20 bytes for the variable p at run time.

For De-Allocation:
free(): Deallocate memory at run time.
Syntax:
free(variable name);
Ex: free(p);

DYNAMIC MEMORY ALLOCATION

int x[10];

- For the above statement 20 bytes of memory is allocated (10*2 bytes)
- This memory is allocated when that statement is executed by compiler
- This type of allocation is known as STATIC MEMORY ALLOCATION.
- But in this kind of declaration the memory might be wasted.
- That is, in the above declaration 20 bytes of memory is allocated for 10 elements, but if we assign only 2 elements that is only 4 bytes of memory is being used, thereby wasting 16 bytes of total memory.

- In order to avoid this we go for the concept of DYNAMIC MEMORY ALLOCATION.

Allocation memory at run-time is called as dynamic memory allocation

- OPERATORS USED FOR DYNAMIC MEMORY ALLOCATION ARE:
  - **new**
  - **delete**

- "new" operator ised to allocate memory at run-time
- Syntax for new operator is:
  datatype *pointer_variable = new datatype[size];

- "delete" operator is used to de-allocate memory at run-time.

  Syntax for delete operator is:
  delete[size] pointer_variable;

PROGRAM FOR DYNAMIC MEMORY ALLOCATION

```
#include<iostream>
using namespace std;
main()
{
int size,i;
```

```
cout<<"enter array size";
cin>>size;

int *p = new int[5];

cout"<<"enter elements";
for(i=0;i<size;i++)
cin>>p[i];
cout<<"elements are";
for(i=0;i<size;i++)
cout<<p[i];

delete [ ]p;
}
```

- ‘p’ acts as a constant pointer
- It holds the address of memory allocated.
- Delete operator is used because to free memory that is allocated at run-time.
- If we don’t use delete then that particular memory space cannot be used by other variables.
- Both in compile-time and run-time memory allocation after closing program memory is cleared.
  In compile time allocation memory is automatically cleared, where as in run-time allocation memory should be cleared by using “delete” operator

CONSTANT OBJECTS AND CONSTANT MEMBER FUNCTIONS

- If we don’t want to modify value of some object then we can use the keyword “const”.

- If a member function does not modify any data member in the class then we can declare it as “const” member function.
- A “const” object can call only constant member function.
- A non-const object can call both const member function and non-const member function.

Program to demonstrate const object and member function

```
#include<iostream>
using namespace std;
class demo
{
    int x,y;
public:
        demo()
        {
            x=0;
            y=0;
        }

        demo(int p,int q)
        {
            x=p;
            y=q;
        }

        vois show() const
        {
        cout<<x;
        cout<<y;
        }

        void display()
        {
         cout<<"hello";
        }
```

```
};
main()
{
demo d1;
const demo d2(20,40);
d1.show();
d2.show();
d1.display();
d2.display();// error
}
```

STATIC VARIABLES AND STATIC MEMBER FUNCTIONS

- 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 re declaring 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. (Remember, a class declaration is simply a logical construct that does not have physical reality.)

To understand the usage and effect of a static data member, consider this program:

```
#include<iostream>
using namespace std;
class A
{
public:
```

```
static int count;
A()
{
count++;
}
};



int A::count;
main()
{
A a1,a2,a3;
cout<<"objects created "<<a1.count;
A a4,a5;
cout<<"objects created "<<A::count;
}
```

output:

objects created 3

objects created 5

Static Member Functions

- Member functions may also be declared as static.
- **There are several restrictions placed on static member functions**.
- 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.)
- A static member function does not have a ‘this’ pointer.
- A static member function may not be ‘virtual’.
- They cannot be declared as const.

Consider the following program

```
#include<iostream>
```

```
using namespace std;
class A
{
public:
static int x,y;
A()
{
x++;  y=5;
}
void static put();
};

void A::put()
{
cout<<x;
cout<<y;
}
int A::x;
main()
{
A p,q,r;
p.put();
A s,t;
A::put();
}
```

Output: 3 5