Object persistence and Visibility

- There are a number of factors which affect the persistence and visibility of an object during the program execution.
- Some objects may exist throughout the running of a program and may be visible in all modules.
- Other objects may exist momentarily within the limited scope of particular function body or some block.
- Therefore , there is always range of lifetimes and visibility among instantiated objects in OOPs.

Types of object

There are four types of objects:

- ***External or Global objects*** : Persistence or existence throughout the life time of a program and having "file scope" visibility.
- ***Automatic or Local objects***: persistence and visible only throughout the local scope in which they are created.
- ***Static objects:*** persistence throughout a program but only visible within their local scope.
- ***Dynamic objects:*** life time may be controlled within a particular scope.

External(Global) Objects:

- An external object is one which is persistence and visible throughout a program module "its scope is and entire module or source file". It may also be wide visible in other modules.
- An external object is declared outside the scope of any braces.

```
e.g. BankAccount acc1; // creation of an external object
void main()
{
   -----
}
```

Automatic(Local) objects:

- If an object is instantiated local to the scope or within the braces such an object is called automatic object.
- The persistence of this type object is destroyed automatically when fall out of the scope in which they were instantiated.
- This type of objects can not be visible in the module other than its definition.

```
void main()
{
        BankAccount acc1; // within the main scope
        ----------
        ---------
        {
                BankAccount acc2; // with in the local scope or nested scope
                ----------
        }
        {
                BankAccount acc3; // within another nested scope
```

```
}
```

Static Objects:

- Objects instantiated inside local scope having local visibility, but persisting from their declaration to the end of program are called static objects.

  e.g.

```
void function1()
{
     BankAccount acc1; // automatic object
     Static BankAccount acc2; // static object
}
```

- In above creation of objects, the `acc1` object will be created and destroyed every time the function is called, because this is local object within that function1(). The object `acc2` will be created only once(first time the function executes) and will persist until the end of entire program.
- Hence the advantage of static object is that it can retain its state data even when it is not in scope. But it is visible within that scope only. A static object cannot be declared "extern" because it can only be visible within the scope in which it is defined.

Instantiating objects of different types:

```
#include<iostream.h>
class object
{
     private:
     int  value;
     public:
     object();      //constructor
     int get_value();
     void add_value(int value_in);
};

object:: object()
{
     value=0;
}

int object:: get_value()
{ return value; }

void object:: add_value(int val)
{
     value+=val;
}

object ext_obj; //external object created.
```

```
void main()
{
      ext_obj.add_value(10);
      for(int i=0;i<3;i++)
      {
            cout<<"Pass:"<<(i+1)<<endl;
            object auto_obj;
            auto_obj.add_value(10);
            static object static_obj;
            static_obj.add_value(10);
            cout<<"The automatic object has the value”;
            cout<<auto_obj.get_value()<<endl;
            cout<<"The static object has value:";
            cout<<static_obj.get_value() <<endl;
      }
      cout<<"The external object has value :”;
      cout<<ext_obj.get_value()<<endl;
      //cout<<static_obj.get_value(); //not defined error !
      //cout<<static_obj.get_value(); //not defined error !
}
```

output

```
Pass:1
The automatic object has the value10
The static object has value:10
Pass:2
The automatic object has the value10
The static object has value:20
Pass:3
The automatic object has the value10
The static object has value:30
The external object has value : 10
```

What is the difference in life time and visibility between external , automatic and static objects ?

- External objects persists or exist for the life time of the program and their visibility is global.
- Automatic objects exists as long as they remain in the scope and are visible only within that scope.
- Static objects are created and visible within a particular scope, but persist from their point of creation until the end of program.

Desrtuctors:

- A destructor is a function appeared in public section of a class preceded by tilde(~) sign.
- The name of destructor is same as class name like constructor.
- The destructor never takes any argument not does it return any values.
- The constructor is always called to reserve the memory for the instantiated object.
- The role of destructor is to remove the object from the memory created by constructor.
- Like constructor, there are two types of destructors.
    - Default destructor
    - User defined destructor

Calling Destructor

- Whenever a program is stop external object and static object will be destroyed either calling default destructor or user-defined destructor(if exist in public area in the class definition)
- Whenever an automatic object is out of scope, then the default destructor or user-defined destructor will be called automatically.

Example:

```
#include<iostream.h>
int count=0;
class example
{
   public:
   example()
   {
        count++;
        cout<<"Number of object created  "<<count<<endl;
   }
   ~example()
   {
        cout<<"The number of object destroyed  "
             <<count<<endl;
        count--;
   }
};

void main()
{
   cout<<"Object created in main"<<endl;
   example e1,e2,e3,e4;
   {
        cout<<"Object created in block1"<<endl;
        example e5;
        }
   {
```

```
        cout<<"Object created in block2"<<endl;
        example e6;
    }
    cout<<"Return in main() "<<endl;

     }
```

```
output:
Object created in main
Number of object created  1
Number of object created  2
Number of object created  3
Number of object created  4
Object created in block1
Number of object created  5
The number of object destroyed  5
Object created in block2
Number of object created  5
The number of object destroyed  5
Return in main()
The number of object destroyed  4
The number of object destroyed  3
The number of object destroyed  2
The number of object destroyed  1
```

Dynamic Objects:

- The objects which are not predictable enough to be instantiated as external, static or automatic objects.
- This type of object will be created when the programmer are unable to predict the following at compile time.
  - Object identities
  - Object quantities
  - Object life times
- Only the dynamic object is possible to create at run time.

Creating and destroying dynamic objects:

- The new and delete operators are used to create and destroy the dynamic objects

Calling the methods of dynamic objects:

- The arrow operator(->) is used instead of dot operator.

Creating dynamic Objects:

- C++ includes a special memory allocation operator "**new**" for use in dynamic object constructors.
- The **new** operator allocate the memory via a pointer of the required type.
- The syntax is based on the creation of a pointer , and then direction of that pointer to an area of memory which will contain an object.

```
BankAccount *acc_ptr; // pointer type declaration
```

- Above syntax creates s pointer able to reference objects of class "`BankAccount`".
- This pointer "`acc_ptr`" is now able to point to an object of the `BankAccount` class
- The pointer `acc_ptr` can directed to any `BankAccount` object using the ***new*** operator as:

```
// object created directed by acc_ptr
Acc_ptr = new BankAccount;
Or
BankAccount *acc_ptr = new BankAccount;
//acc_ptr is not the name of object, it is the name of a pointer to an object of
//class BankAccount.
```

```
void main()
{
    BankAccount *acc_ptr1; // create pointer of type BankAccount
    BankAccount *acc_ptr2;// create pointer of type BankAccount
    Acc_ptr1=new BankAccount; //new object created
    Acc_ptr2= acc_ptr1; // redirect acc_ptr2 to point to the same object
    ……
    ……..
}
```

Calling the methods of a dynamic objects:

In order to call a method, defined in public area of the class, the de-referencing "arrow"operator is used for dynamic object.

```
Void main()
{
     BankAccount acc1; // object created
     BankAccount *acc2=new BankAccount; // dynamic object
created.
     acc1.deposit(1000.0);  // call of method by object
acc1
     acc2->deposit(2000.0); // call of method by dynamic
object
     ……..
     ……..
}
```