Object Oriented Programming

Unit 2: Object Initialization and Cleanup

Object Initialization and Cleanup

2.1. Friend Function and Class

2.2. Static Data Members and Member Functions

2.3. Constructor:

- **2.3.1 Type of Constructor**
- **2.3.2 Constructor Overloading**
- **2.3.3 Constructor with Default Argument**

2.4. Destructor

2.5. Nameless Object

2.6. Dynamic Memory Allocation

CE: 2.1
Friend Function and Class

CE: 2.1.1. Friend Function

- A friend function of a class is defined outside the class scope.
- It has the right to access all ***private*** and ***protected members*** of the class.
- Even though the prototypes for friend functions appear in the class definition, friends are **not** member functions.

CE: 2.1.1. Friend Function (Conti…)

- **Characteristics of a friend function:**

1. It is not in the scope of the class to which it has been declared as friend.

2. Since it is not in the scope of the class, it cannot be called using the object of that class.

3. It can be invoked like a normal function without the help of any object.

4. Unlike member function, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.

5. It can be declared either in the public or in the private of a class without affecting its meaning.

6. Usually, **it has the object as arguments.**

CE: 2.1.1. Friend Function (Conti…)

When to use friend function?

- It can be generally used in two situations:
  1. In operator overloading
  2. If a particular function requires to be shared between two or more classes. A function that needs to be shared by two or more classes can be declared as a friend function in all the classes.

Class Work

- Write a program to make a friend function named calsalary(), which will calculate salary of two classes named “supervisor” and “manager”. Both the classes have three data members code, name and salary.

  [Consider DA as 4%, HRA as 8% and PF as 7% of basic salary for calculating salary.]

CE: 2.1.2. Friend Class

- Like a friend function, a class can also be made a friend of another class using keyword **friend**.
- A friend class ***can access private*** and ***protected members*** of other class in which it is declared as friend.
- It is sometimes useful to allow a particular class to access private members of other class.

CE: 2.1.2. Friend Class (Conti…)

- **For Example:**

```
... .. ...
class B;
class A
  {
        // class B is a friend class of class A
       friend class B;
       ... .. ...
   };
class B
  {
       ... .. ...
  };
```

- When a class is made a friend class, all the member functions of that class becomes friend functions.

Some important points about Friend Functions and Classes:

1. Friends should be used only for limited purpose.
   Too many functions or external classes are declared as friends of a class with protected or private data, it reduces the value of encapsulation of separate classes in object-oriented programming.

2. Friendship is not mutual. If a class A is friend of B, then B doesn't become friend of A automatically.

3. Friendship is not inherited.

4. The concept of friends is not there in Java.

Class Work

- By using friend class, write a simple program by just creating two classes named “classA” and “classB”. Take input of a single variable in a method of classA and display the value of that variable in a method of class B.

CE: 2.2
Static Data Members and Member Functions

CE: 2.2.1. Member Function

- A function which is declared or defined within the class definition like any other variable is called **member function**.
- And a function which is not declared or defined within the class definition is called **non-member function**.
- A *member function* operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

CE: 2.2.1. Member Function (Conti…)

- **Characteristics of member function:**

1. Several classes can use the same function name.

2. Member functions can access private data members of the class. That means a non-member function cannot do so.

3. A member function can call another member function directly, without using membership operator (dot(.) operator).

Class Work

- Write a program by creating a class named EMPLOYEE which will have data members as name, age and salary; and member function getdata() definition for taking the input from the user. Declare one more method named displaydata() within the class and define it outside the class to display the details of employee.

CE: 2.2.2. Static Data Members

- We can define class members static using **static** keyword.
- It is initialized to **0,** when the first object of its class is created. **No other initialization is permitted**.
- Only once copy of that member is created for entire class and is shared by all the object of that class (using :: operator), no matter how objects are created.
- When a variable is declared as static, space for it gets allocated for the lifetime of the program.
- It is visible only within the class, but it remains for lifetime of the entire program.

CE: 2.2.2. Static Data Members (Conti…)

- **For Example:**

```
class staticdemo
  {
        public:
              static int i;
  };

int main()
   {
        staticdemo s1, s2;
        s1.i = 2;
        s2.i = 3;
        cout << s1.i<<"\n"<<s2.i;
   }
```

CE: 2.2.2. Static Data Members (Conti…)

- A static variable inside a class should be initialized explicitly by the user using the class name and scope resolution operator outside the class.

CE: 2.2.2. Static Data Members (Conti…)

- **For Example:** Static variables inside a class.

```
class staticdemo
  {
        public:
              static int i;
  };
int staticdemo::i=0;
int main()
   {
        staticdemo s1, s2;
        s1.i = 10;
        s2.i = 20;
        cout << s1.i<<"\n"<<s2.i;
   }
```

CE: 2.2.3. Static Member Functions

- Just like the static variables or static data members inside the class, static member functions also does not depend on object of class.

- We are allowed to invoke a static member function using the object and the '.' operator, **but it is recommended to invoke the static members using the class name and the scope resolution operator.**

- **Static member functions are allowed to access only the static data members or other static member functions**, they can not access the non-static data members or member functions of the class.

CE: 2.2.3. Static Member Functions (Conti…)

- **For Example:** Static member function in a class.

```
class staticdemo
 {
      public:
          static int i;
          static void getdata()
            {
                  cin>>i;
            }
          static void display()
            {
                  cout<<"i = "<<i;
            }
 };
```

```
int staticdemo::i=0;
int main()
  {
        staticdemo::getdata();
        staticdemo::display();
        return 0;
  }
```

Class Work

- Create a class called “counter” by using appropriate methods count number of objects using static data members and member function.

CE: 2.3
Constructor

CE: 2.3. Constructor

- A **constructor** is a member function of a class which initializes objects of a class.

- It is special member function of the class, which has two reasons:
  1. Its name is same as class name.
  2. It is called automatically, when the object (instance of class) is created.

- It is not mandatory to use constructor in a class, but preferable.

- If we do not specify a constructor in user define class, then also the C++ compiler automatically generates a default constructor with empty body in compiled code.

CE: 2.3. Constructor (Conti…)

Why constructors is needed?

- Constructors are special member functions, with same name as the class name that are executed automatically when an object is created.

- A constructor has no return type but can take arguments.

- **It is often used to give initial values to object data members.**

CE: 2.3.1. Types of Constructor

- **Constructors are of different types:**
  1. Default Constructor
  2. Parameterized Constructor
  3. Copy Constructor

CE: 2.3.1. Types of Constructor (Conti…)

constructor.cpp

CE: 2.3.2. Constructor Overloading

constructor.cpp

CE: 2.3.3. Constructor with Default Argument

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

class Demo
   {
            int x,y;
        public:
            Demo()
              {  cout<<"\nDefault constructor is Called!!"; }
            Demo(int X, int Y=20)
              {
                    x = X;
                    y = Y;
                    cout<<"\nConstructor is Called!!";
              }
            void displaydata()
              {
                    cout <<"\nValue of X : " <<x;
                    cout <<"\nValue of Y : " <<y;
              }
          ~Demo()
              {
                    cout<<"\nDestructor is Called!!";
              }
     };
```

```
int main()
   {
        Demo d1;
        Demo d2= Demo(10);
        cout<<"\nFirst Call : ";
        d2.displaydata();

        Demo d3= Demo(30,40);
        cout <<"\nSecond Call : ";
        d3.displaydata();
        return 0;
   }
```

CE: 2.4
Destructor

CE: 2.4. Destructor

- A **destructor** is also a member function whose name is the same as the class name, but is preceded by tilde **('~')**.
- It is used to destroy the objects that have been created by a constructor.
- It takes no arguments; and no return types can be specified for it (not even "void").
- It is also called automatically by the complier when an object is destroyed.
- A destructor cleans up the storage that is no longer accessible.
- Therefore, **A *constructor* initializes an object** and **a *destructor* deinitializes an object when it is no longer needed.**

CE: 2.4. Destructor (Conti…)

Why destructor is needed?

- During the construction of an object by the constructor, resources are allocated for use.
- For Example:
  - ✓ Allocation of memory, opening of file, etc.
    - These allocated resources must be deallocated, before the object is destroyed.

- A destructor is responsible for this task and performs all clean-up jobs like…
  - ✓ Closing a file, deallocating and releasing memory area automatically.

- It is a good practice to declare destructors in a program, since it releases memory space for future use.

Difference between Constructor and Destructor:

Constructor	Destructor
1. Its name is same as class name.	1. Its name is same as class name preceded by tilde(~).
2. It is called automatically, when an object is created.	2. It is called automatically, when an object is destroyed.
3. It can have any number of arguments.	3. It never takes any argument.
4. Normally **new** is used in constructor to allocate memory to objects.	4. Normally **delete** is used in destructor to free memory from objects.

Class Work

- Create a class called “shape” by using types of constructors; and destructor for finding area of square and rectangle.

CE: 2.5
Nameless Object

CE: 2.5. Nameless Object

- C++ also supports unnamed objects.
- In the object creation statement, the name of an object is not needed to be mentioned.
- For Example:

<classname> (<arguments>);

- If no-arguments are mentioned, default constructor of the class is invoked.
- After the constructor is executed, the nameless objects are immediately destroyed.
- Therefore, the scope of a nameless object is limited only to the statement in which it is created.

CE: 2.5. Nameless Object (Conti…)

- **For Example:**

```
class BMIIT
 {
     public:
        BMIIT()
         {
             cout<<"Constructor is called!";
         }
        ~BMIIT()
         {
              cout<<"\ndestructor is called!";
         }
 };
```

```
int main()
 {
        BMIIT(); //
        BMIIT b1;
        return 0;
 }
```

Nameless object is created and destroyed

CE: 2.6
Dynamic Memory Allocation

CE: 2.6. Dynamic Memory Allocation

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by the programmer.

- *Dynamically allocated* memory is allocated on Heap; and
- *Non-static and local variables* get memory allocated on Stack.

- C uses **malloc()** and **calloc()** function to allocate memory dynamically at run time and uses **free()** function to free dynamically allocated memory.

- C++ supports these functions and also has two operators:
  - ✓ **new** that perform the task of allocating and
  - ✓ **delete** to free the memory,

  in a better and easier way.

CE: 2.6. Dynamic Memory Allocation (Conti…)

new operator:

- It denotes a request for memory allocation on the Heap.

- If sufficient memory is available, **new** operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- Syntax:
  pointer-variable = **new** data-type;

- For Example:
  int *p = NULL; //Pointer initialized with NULL
  p = new int; // request memory for the variable

  int *p = new int; declaration and their assignment of pointer

CE: 2.6. Dynamic Memory Allocation (Conti…)

new operator: (Conti…)

- We can also initialize the memory using **new** operator:

- Syntax:

  pointer-variable = **new** data-type(value);

- For Example:

```
int *p = new int(5);
float *q = new float(5.5);
```

CE: 2.6. Dynamic Memory Allocation (Conti…)

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

int main()
  {
        double* p= NULL;
        p = new double;


         *p = 1025.35;
         cout<<" *p = "<<*p;

         delete p;
         return 0;
   }
```

CE: 2.6. Dynamic Memory Allocation (Conti…)

new operator: (Conti…)

- It is also used to allocate a block of memory (using an array) of type data-type.

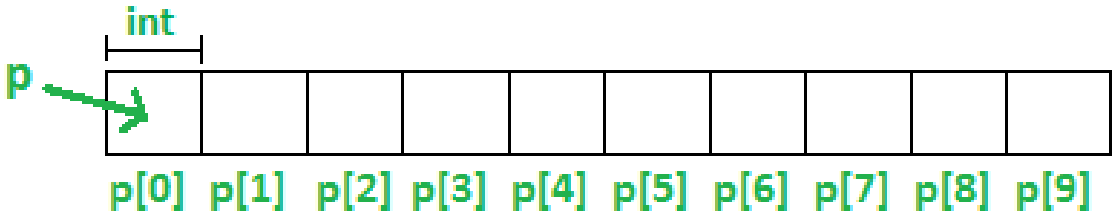
- Syntax:

  pointer-variable = **new** data-type[size];

  ✓where size(a variable) specifies the number of elements in an array.

- For Example:

  int *p = new int[10];



CE: 2.6. Dynamic Memory Allocation (Conti…)

delete operator:

- It is programmer's responsibility to deallocate dynamically allocated memory, programmers provides **delete** operator by C++ language.

- Syntax:
  - ✓ To release the memory pointed by pointer-variable…

    delete pointer-variable;

  - ✓ To free the dynamically allocated array pointed by pointer-variable…

    delete[] pointer-variable;

- For Example:

```
delete p;
delete[] p; // It will free the entire array pointed by p.
```

CE: 2.6. Dynamic Memory Allocation (Conti…)

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

class pointerdemo
  {
      public:
          pointerdemo()
            {
                cout << "Constructor called!";
            }
        ~pointerdemo()
            {
                cout << "\nDestructor called!";
            }
   };

int main()
   {
          pointerdemo* p = new pointerdemo[4];
          delete[] p;
          return 0;
   }
```


Thank You