Module- 3

Constructors & Destructors

Constructors

- A constructor is a member function of a class which initializes objects of a class.
- Constructor is automatically called when object(instance of class) create.
- It is special member function of the class. Becauseconstructor has same name as the class itself.
- Constructors don’t have return type.
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Syntax:

```
Class Class_Name {
//Private Mambers
Public:
Class_name(void);      //Constructor Declaraion
      };

Class_name::Class_Name(void)    //Constructor Definition
 {
   //Initialize Data Members
 }
```

Characterstics:

1. They should be declared in the public section.
2. They are invoked directly when an object is created.
3. They don’t have return type, not even void and hence can’t return any values.
4. They can’t be inherited; through a derived class, can call the base class constructor.
5. Like other C++ functions, they can have default arguments.
6. Constructors can’t be virtual.
7. Constructors can be inside the class definition or outside the class definition.
8. Constructor can’t be friend function.
9. They can’t be used in union.
10. They make implicit calls to the operators new and delete when memory allocation is required.
11. When a constructor is declared for a class, initialization of the class objects becomes necessary after declaring the constructor.

Default Constructors:

Default constructor is the constructor which doesn’t take any argument. It has no parameters.

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

```
{
public:
   int a, b;

      // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};
 int main()
{
    // Default constructor called automatically
    // when the object'c' is created.
   construct c;
// Access values assigned by constructor
   cout<< "a: "<<c.a<<endl<< "b: "<<c.b;
   return 0;
}
```

```
Output: a:10
        b:20
```

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly. The default value of variables is 0 in case of automatic initialization.

Parameterized Constructors:

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.

```
#include<iostream>
using namespace std;
class Point{
   private:
      int x, y;
   public:
      // Parameterized Constructor
      Point(int x1, int y1)
      {
         x = x1;
         y = y1;
      }
      intgetX()
      {
         return x;
      }
      intgetY()
      {
         return y;
      }
};

int main(){
   // Constructor called
   Point p1(10, 15);
   // Access values assigned by constructor
   cout<< "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
   return 0;
}
```

Output:
p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Point p1 = Point(0, 50); // Explicit call
Point p1(0, 50);         // Implicit call
```

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

Copy Constructor:

- A copy constructor is a member function which initializes an object using another object of the same class.
- A copy constructor has the following general function prototype:
  ClassName (constClassName&old_obj);
- Acopy constructor may be called in following cases:
  1. When an object of the class is returned by value.
  2. When an object of the class is passed (to a function) by value as an argument.
  3. When an object is constructed based on another object of the same class.
  4. When compiler generates a temporary object.

Example

```
#include<iostream>
usingnamespacestd;

classPoint
{
private:
   intx, y;
public:
   Point(intx1, inty1)
{
x = x1;
y = y1;
}
```

```
   // Copy constructor
   Point(Point &p2)
{
x = p2.x;
y = p2.y;
}
   intgetX(){  returnx; }
   intgetY(){  returny; }
};

intmain()
{
   Point p1(10, 15); // Normal constructor is called here
   Point p2 = p1; or Point p2(p1); // Copy constructor is called here
   // Let us access values assigned by constructors
   cout<< "p1.x = "<< p1.getX() << ", p1.y = "<< p1.getY();
   cout<< "\np2.x = "<< p2.getX() << ", p2.y = "<< p2.getY();
   return0;
}
```

Output:

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

Multiple constructors/Constructor Overloading

- Constructor can be overloaded in a similar way as function overloading.
- Overloaded constructors have the same name (name of the class) but different number of arguments.
- Depending upon the number and type of arguments passed, specific constructor is called.
- Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Example:

```
#include <iostream>
using namespace std;
class Area{
   private:
              int length;
              int breadth;
   public:
     // Constructor with no arguments
     Area(){
              length =5;
              breadth =2;
     }

     // Constructor with two arguments
     Area(int l, int b){
              length =l;
              breadth =b;
    }
```

```
     void GetLength()  {
                cout<< "Enter length and breadth respectively: ";
                cin>> length >> breadth;
     }

     intAreaCalculation() {
                return length * breadth;
     }

     void DisplayArea(int temp)  {
                cout<< "Area: " << temp <<endl;
     }
};

int main(){
     Area A1, A2(2, 1);
     int temp;
     cout<< "Default Area when no argument is passed." <<endl;
      temp = A1.AreaCalculation();
      A1.DisplayArea(temp);
     cout<< "Area when (2,1) is passed as argument." <<endl;
     temp = A2.AreaCalculation();
     A2.DisplayArea(temp);
    return 0;
}
```

- For object A1, no argument is passed while creating the object.Thus, the constructor with no argument is invoked which initializes length to 5 and breadthto 2. Hence, area of the object A1 will be 10.
- For object A2, 2 and 1 are passed as arguments while creating the object.Thus, the constructor with two arguments is invoked which initializes length to l (2 in this case) and breadth to b (1 in this case). Hence, area of the object A2 will be 2.
- Output of above Program:

```
Default Area when no argument is passed.

Area: 10

Area when (2,1) is passed as argument.

Area: 2
```

Destructors:

1. Destructors are special member functions of the class required to free the memory of the object whenever it goes out of scope.
2. Destructors are parameter less functions.
3. Name of the Destructor should be exactly same as that of name of the class. But preceded by '~' (tilde).
4. Destructors does not have any return type. Not even void.

5. The Destructor of class is automatically called when object goes out of scope.

   Example:

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

class Marks{
     public:
                  int maths;
                  int science;

                   //constructor
                  Marks() {
                  cout<< "Inside Constructor"<<endl;
                  cout<< "C++ Object created"<<endl;
                   }

              //Destructor
       ~Marks() {
                  cout<< "Inside Destructor"<<endl;
                  cout<< "C++ Object destructed"<<endl;
     }
};

int main( ){
     Marks m1;
     Marks m2;
     return 0;
}
```

Output

```
Inside Constructor
C++ Object created
Inside Constructor
C++ Object created

Inside Destructor
C++ Object destructed
Inside Destructor
C++ Object destructed
```

Operator overloading:

Why operator overloading?

*Let's say we have defined a class Integer for handling operations on integers. We can have functions add(), subtract(), multiply() and divide() for handling the respective operations. However, to make the code more intuitive and enhance readability, it is preferred to use operators that correspond to the given operations(+, -, *, / respectively) i.e. we can replace*

- The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used. However, for user-defined types (like: objects), you can redefine the way operator works. For example: If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings.
- This feature allows programmer to redefine the meaning of an operator (when they operate on class objects) is known as operator overloading.
- Why is operator overloading used?You can write any C++ program without the knowledge of operator overloading. However, operator operating are profoundly used by programmers to make program intuitive. For example, You can replace the code like:
  calculation = add(multiply(a, b),divide(a, b));
  to
  calculation = (a*b)+(a/b);
- How to overload operators? To overload an operator, a special operator function is defined inside the class as:

```
class className
{
    ... .. ...
    Public:
    returnType operator symbol (arguments){
    ... .. ...
    }
    ... .. ...
};
```

- Here, returnType is the return type of the function.The returnType of the function is followed by operator keyword.Symbol is the operator symbol you want to overload. Like: +, <, -, ++. You can pass arguments to the operator function in similar way as functions.

Things to remember

- Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).
- Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator. You do not need to create an operator function.
- Operator overloading cannot change the precedence and associatively of operators. However, if you want to change the order of evaluation, parenthesis should be used.
- There are 4 operators that cannot be overloaded in C++. They are :: (scope resolution), . (member selection), .* (member selection through pointer to function) and ?:(ternary operator).

Example:

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

class Test{
   private:
         int count;
```

```
  public:
        void Test(){
         Count=5;
        }


     void operator ++(){
                         count = count+1;
                    }
     void Display() {
                      cout <<"Count: "<<count;
        }
};

int main(){
    Test t;
    // this calls "function void operator ++()" function
    ++t;
   t.Display();
    return 0;
}
```

Output
Count: 6
This function is called when ++ operator operates on the object of Test class (object t in this case).In the program,void operator ++ () operator function is defined (inside Test class).This function increments the value of count by 1 for t object.

Unary Operator Overloading

C++ Program to overload pre-increment and post-increment operator

This C++ program overloads the pre-increment and post-increment operators for user-defined objects. The pre-increment operator is an operation where the value attribute of the object is incremented and the reference to resulting object returned whereas in the post-increment operator, a local copy of the object is saved, the value attribute of the object is incremented and the reference to the local copy of the object is returned.

```
/*
*  C++  Program  to  overload  pre-increment  and  post-increment
operator
*/
#include <iostream>
using namespace std;

class Integer {
    private:
        int value;
    public:
         Integer(int v){
           Value = v;
         }
        Integer operator++();
        Integer operator++(int);
        int getValue() {
            return value;
        }
};

// Pre-increment Operator
Integer Integer::operator++()
{
    value++;
    return *this;
}

// Post-increment Operator
Integer Integer::operator++(int)
{
    const Integer old(*this);
    ++(*this);
    return old;
}

int main()
{
    Integer i(10);

    cout << "Post Increment Operator" << endl;
    cout << "Integer++ : " << (i++).getValue() << endl;
    cout << "Pre Increment Operator" << endl;
    cout << "++Integer : " << (++i).getValue() << endl;
}
```

Operator Overloading of Decrement – Operator: Decrement operator can be overloaded in similar way as increment operator.

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

class Check{
  private:
    int i;
  public:
    Check(): i(3) {   }
    Check operator -- (){
        Check temp;
        temp.i = --i;
        return temp;
    }
    // Notice int inside barcket which indicates postfix decrement.
    Check operator -- (int){
        Check temp;
        temp.i = i--;
        return temp;
    }
    void Display(){
     cout << "i = "<< i <<endl;
    }
};

int main(){
    Check obj, obj1;
    obj.Display();
    obj1.Display();
    // Operator is called, only then value of obj is assigned to obj1
    obj1 = --obj;
    obj.Display();
    obj1.Display();
    // Assigns value of obj to obj1, only then operator function is called.
    obj1 = obj--;
    obj.Display();
    obj1.Display();
    return 0;
}
```

Binary Operators Overloading:

- The binary operators take two arguments
- When + operator is operated on obj1 and obj2, operator function complex operator+( ) is invoked which will add complex numbers.
- When - operator is operated on obj1 and obj2, operator function complex operator-( ) is invoked which will substract complex numbers.
- Right side of the binary operator is the parameter for operator function definition.

Example: Binary Operator Overloading to Subtract Complex Number

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

class Complex{
    private:
      float real;
      float imag;
    public:
       Complex(){ real= 0; imag= 0;}
       void input(){
           cout << "Enter re & imag parts respectively: ";
           cin >> real;
           cin >> imag;
       }
       // Operator overloading
       Complex operator - (Complex c2){
           Complex temp;
           temp.real = real - c2.real;
           temp.imag = imag - c2.imag;
           return temp;
       }
```

```
        void output(){
            if(imag < 0)
                cout << "O/p Comp: "<< real << imag << "i";
            else
                cout << "O/p Comp:"<< real << "+"<< imag << "i";
        }
};

int main(){
    Complex c1, c2, result;
    cout<<"Enter first complex number:\n";
    c1.input();
    cout<<"Enter second complex number:\n";
    c2.input();
    // In case of operator overloading of binary operators in C++ programming,
    // the object on right hand side of operator is always assumed as argument
    // by compiler.
    result = c1 - c2;
    result.output();
    return 0;
}
```

In this program, three objects of type *Complex* are created and user is asked to enter the real and imaginary parts for two complex numbers which are stored in objects `c1` and `c2`. Then statement `result = c1 -c 2` is executed. This statement invokes the operator function `Complex operator - (Complex c2)`. When `result = c1 - c2` is executed, `c2` is passed as argument to the operator function.

String manpulation using operator overloading

Given two strings. The task is to concatenate the two strings using Operator Overloading in C++.

Approach : Using binary operator overloading.

- Declare a class with a string variable and an operator function '+' that accepts an instance of the class and concatenates it's variable with the string variable of the current instance.
- Create two instances of the class and initialize their class variables with the two input strings respectively.
- Now, use the overloaded operator(+) function to concatenate the class variable of the two instances.

C++ program to compare two Strings using Operator Overloading

Given two strings, how to check if the two strings are equal or not, using Operator Overloading.

Approach: Using binary operator overloading.

```
# include <iostream>
# include<string.h>
using namespace std;

class String{
    private:
        char str[50];
    public:
        String();
        String(char[]);
        void get_string();
        void put_string();
        String operator +(String&);
};
String::String(){
  strcpy(str," ");
}
String::String(char temp[]){
    strcpy(str,temp);
}
void String::get_string(){
 cout<< "Enter the string:";
 cin>> str;
}
void String::put_string(){
cout<<"entered string:"<< str<<endl;
}

String String::operator +(String& ref1){
 String temp;
 strcpy(temp.str,(strcat(str,ref1.str)));
 return temp;
}

int main(){
    String ob1,ob2(".Bhat"),ob3;
    ob1.get_string();
    ob1.put_string();
    ob3=ob1+ob2;
    ob3.put_string();
    return 0;

}
```

- Declare a class with a string variable and operator function '==', '<=' and '>=' that accepts an instance of the class and compares it's variable with the string variable of the current instance.
- Create two instances of the class and initialize their class variables with the two input strings respectively.
- Now, use the overloaded operator(==, <= and >=) function to compare the class variable of the two instances.

```
#include <iostream>
#include<string.h>
using namespace std;
class String{
    private:
        char str[25];
    public:
        String();
        String(char[]);
        void put_string();
        void get_string();
        bool operator ==(String&);
        bool operator <=(String&);
        bool operator >=(String&);
 };

String::String(){
 strcpy(str," " );
 }

 String::String(char temp[]){
    strcpy(str,temp);
 }
 void String::get_string(){
 cout<< "Enter the string: ";
 cin>>str;
 }
 void String::put_string(){
 cout<<str<<endl;
 }

 bool String:: operator ==(String &ref){
     return strcmp(str,ref.str)==0?true:false ;
 }

 bool String::operator <=(String &ref){
     if (strlen(str)<=strlen(ref.str))
         return true;
    else
        return false;
 }
  bool String::operator >=(String &ref){
     if (strlen(str)>=strlen(ref.str))
         return true;
    else
         return false;
 }

int main(){
    String ob1;
    String ob2("AAA");
    ob1.get_string();
    ob1.put_string();
    ob2.put_string();
    cout<< (ob1==ob2);
    return 0;
}
```

Operator overloading using Friend Function

The friend functions are more useful in operator overloading. They offer better flexibility, which is not provided by the member function of the class. The difference between member function and friend function is that the member function takes argument explicitly. On the contrary, the friend function needs the parameters to be explicitly passed. The syntax of operator overloading with friend function is as follows:

Important points to be noted

- Friend function using operator overloading offers better flexibility to the class.
- These functions are not a members of the class and they do not have 'this' pointer.
- When you overload a unary operator you have to pass one argument.
- When you overload a binary operator you have to pass two arguments.
- Friend function can access private members of a class directly.

Syntax:

```
friend return-type operator operator-symbol (Variable 1, Varibale2)
{
     //Statements;
}
```

The keyword friend precedes function prototype declaration. It must be written inside the class. The function can be defined inside or outside the class. The arguments used in friend function are generally objects of the friend classes. A friend function is similar to normal function; the only difference being that friend function can access private member of the class through the objects. friend function has no permission to access private members of a class directly. However, it can access the private members through objects of the same class.

Example : Write a program to overload unary operator using friend function.

Explanation: In the below program, operator – is overloaded using friend function. The operator function is defined as friend. The statement c2=−c1 invokes the operator function. This statement also returns the negated values of c1 without affecting actual value of c1 and assigns it to object c2. The negation operation can also be used with an object to alter its own data member variables. In such a case, the object itself acts as a source and destination object. This can be accomplished by sending reference of object..

```
#include<iostream.h>
#include<constream.h>

class complex
{
   float real,imag;
   public:
   complex() // zero argument constructor
   {
   real=imag=0;
   }
   complex (float r, float i) //
   two argument constructor
   {
   real=r;
   imag=i;
   }
   friend complex operator - ( complex c)
   {
   c.real=-c.real;
   c.imag=-c.imag;
   return c;
   }
   void display()
   {
   cout<<"\n Real:"<<real;
   cout<<"\n Imag:"<<imag;
   }
};
void main()
{
   clrscr();
   complex c1(1.5,2.5),c2;
   c1.display();
   c2=-c1;
   cout<<"\n\n After Negation \n";
   c2.display();
}
```

OUTPUT

```
Real : 1.5
Imag : 2.5
After Negation
Real : -1.5
Imag : -2.5
```

```
#include <iostream>
using namespace std;
class Point {
  int x, y;
public:
  Point() {} // needed to construct temporaries
  Point(int px, int py) {
    x = px;
    y = py;
  }

  void show() {
    cout << x << " ";
    cout << y << "\n";
  }

  friend Point operator+(Point op1, Point op2); // now a friend
  Point operator-(Point op2);
  Point operator=(Point op2);
  Point operator++();
};

// Now, + is overloaded using friend function.
Point operator+(Point op1, Point op2){
  Point temp;
  temp.x = op1.x + op2.x;
  temp.y = op1.y + op2.y;
   return temp;
}

// Overload - for Point.
Point Point::operator-(Point op2){
  Point temp;
   // notice order of operands
  temp.x = x - op2.x;
  temp.y = y - op2.y;
   return temp;
}

// Overload assignment for Point.
Point Point::operator=(Point op2){
  x = op2.x;
  y = op2.y;
   return *this; // i.e., return object that generated call
}

// Overload ++ for Point.
Point Point::operator++(){
  x++;
  y++;
   return *this;
}

int main(){
  Point ob1(10, 20), ob2( 5, 30);
  ob1 = ob1 + ob2;
  ob1.show();  return 0;}
```

VTU Question Paper Questions :

1. How are constructors different from member functions?
2. Can a class have multiple constructors ? justify.
3. What is operator overloading? Give syntax and example. List the operators that cannot be overloaded in C++.
4. Explain the significance of friend functions to overload operators?
5. Discuss the importance of dynamic constructors and destructors in c+ class?
6. Wirte a c++ program to add two complex numbers by overloading the “+” operator . Alos overload the “<<” and “>>” operators for reading and displaying the complex numbers?
7. Create a class called clock with data members hour,minute and member functions readtime(), showtime(). Write a c++ code to input two clock objects and add using operator overloading + .
8. What are destructors? Mention the destructor rules?
9. Demonstrate the unary and binary operator overloading.
10. What is nesting of member functions ?
11. Explain how “++” and “—“ operators can be overloaded using a simple class?
12. Using general structure of a class, Explain how information hiding is implemented in c++?