First Year First Semester Course

M.Tech. (CS) [Batch 2021-23]

Lecture #14

Introduction to Programming

C++: Constructors and Destructors, Operator Overloading

Course Instructor:

Dr. Monidipa Das

DST-INSPIRE Faculty

Machine Intelligence Unit (MIU), Centre for Artificial Intelligence and Machine Learning (CAIML)

Indian Statistical Institute (ISI) Kolkata, India

Constructor

- A special member function whose task is to initialize the objects of its class .
- Function name is the sane as the class name
- It is invoked when the whenever an object of its associated class is created

```
class integer
{
    int m,n;
  public:
    integer(void);//constructor declared
    ------------
};
integer :: integer(void) //constructor definition
{
    m=0; n=0;
}
```

Constructor [contd.]

- Properties
  - They should be declared in the public section.
  - They are invoked automatically when the objects are created.
  - They don't have return types, not even void.
  - They cannot be inherited, though a derived class can call the base class constructor.
  - Like other C++ function, they can have default arguments.

- Default constructor: that accept no parameter

Parameterized Constructor

- The constructors that can take arguments
- The argument can be passed to the constructor by calling the constructor implicitly.

```
class integer
{
    int m,n;
  public:
    integer( int x, int y);
    --------
};
integer:: integer (int x, int y){
   m=x; n=y;
}
```

```
integer int1 = integer(0,100);
// above one is explicit call


integer int1(0,100);
// above one is implicite call
```

Parameterized Constructor[contd.]

Parameter cannot be of the type of the class in which it belongs

```
class A
{
        -----
        -----
  public:
        A(A);
        ------
        ------
};     //illegal
```

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

```
class A
{
        -----
        -----
  public:
        A(A&);
        ------
        ------
};
```

- This is called **copy constructor**
- Used to declare and initialize an object from another object
- This is called **copy initialization**

Example: Copy Constructor


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

class code
{
    int id;

    public:
      code(){}
      code(int a){id=a;}
      code(code& x) //copy constructor
      {
          id=x.id;  //copy of value
      }
      void display(void)
      {
          cout<<id;
      }
};
```

```
int main()
{
    code A(100);
    code B(A);  //copy constructor called
    code C = A; //copy constructor called again

    code D;
    D=A;

    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();

    return 0;
}
```

Multiple Constructors in a Class

- C++ permits us to use multiple constructors in the same class

```
#include<iostream>
using namespace std;
class complex{
                    float x,y;
          public:
                    complex(){} //required when other definitions of the constructor is here
                    complex(float a){x=y=a;}
                    complex(float a, float b){x=a; y=b;}

                    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);
}
```

Constructor Overloading

Multiple Constructors in a Class


```
void show(complex c){
          cout<<c.x<<" + i"<<c.y<<"\n";
}
int main(){
          complex A(1.5, 3.6);
          complex B(1.4);
          complex C;
          C=sum(A,B);
          cout<<"A= "; show(A);
          cout<<"B= "; show(B);
          cout<<"C= "; show(C);

          complex M,N,P;
          M=complex(1.5,3.6);
          N=complex(1.4);
          P=sum(M,N);
          cout<<"\n\nM= "; show(M);
          cout<<"N= "; show(N);
          cout<<"P= "; show(P);
          return 0;
}
```

Constructors as Inline Functions


```
class integer
{
    int m,n;
  public:
    integer( int x, int y){
    m=x; n=y;
   }
   ------
   ------
};
```

Constructors with Default Arguments

- Example

complex(float real, float imag=0)

complex C(0.5) → assign value 0.5 to real and 0.0 to imag (by default)

complex C(6.0, 3.0) → assign value 6.0 to real and 3.0 to imag

Dynamic Constructors

- Used to allocate memory at the time of creating objects when these are not of the same size

```
#include<iostream>
#include<string.h>
using namespace std;
class String{
            char * name;
            int len;

            public:
            String(){
                        len=0;
                        name=new char[len+1];
            }
            String(char *s){
                        len=strlen(s);
                        name=new char[len+1];
                        strcpy(name,s);
            }

            void display(void){   cout<<name<<"\n";
            }
            void join(String & a, String & b);
};
```

```
void String::join(String &a, String &b)
{
            len=a.len+b.len;
            delete name;
            name=new char[len+1];
            strcpy(name,a.name);
            strcat(name,b.name);

}
```

Dynamic Constructors [contd.]


```
int main()
{
        char *first="ISI";
        String name1(first), name2(" Kolkata"),s1;
        s1.join(name1,name2);

        name1.display();
        name2.display();
        s1.display();

        return 0;
}
```

Destructor

- used to destroy the objects that have been created by a constructor.
- a member function whose name is the same as the class name but is preceded by a tilde.
- never takes any argument nor does it return any value
- invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible
- if **new** is used for allocating memory in the constructor, we should use **delete** to free that memory

Example: Constructor and Destructor


```
#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<<"Enter main:\n";
   alpha A1,A2,A3,A4;
   {
       cout<<"\n \n Enter block 1 :\n";
       alpha A5;
   }
   {
        cout<<"\n \n Enter block2 \n";
        alpha A6;
   }
   cout<<"\n \n Re-enter main: \n";
   return(0);
}
```

Operator Overloading

Operator Overloading

- Creating new definitions for the operators in C++

- **Exceptions**
  - Class member access operators (., .*)
  - Scope resolution operators (::)
  - Size operator (sizeof)
  - Conditional Operator (?:)

- **Remember:**
  - During operator overloading, we can *extend the semantics but not the syntax*
  - *Original meaning* of the operator *does not lost*

Operator Overloading [contd.]

- We must specify what it means in relation to the class to which the operator is applied
- This is done with the help of *operator function*

- **General form of *operator function***

```
return type classname :: operator op(arglist)
{
        Function body //task defined
}
```

Operator Overloading [contd.]

- Example prototypes

```
//vector addition
        vector operator+(vector);
        friend vector operator+(vector, vector);

//unary minus
        vector operator-();
        friend vector operator-(vector);

//comparison
        int operator==(vector);
        friend int operator==(vector,vector);
```

Operator Overloading [contd.]

- Steps:
  - Create a class that defines the data type that is used in the overloading operation.
  - Declare the operator function **operator op()** in the public part of the class
  - It may be either a ***member function*** or ***friend function***.
  - Define the operator function to implement the required operations

Operator Overloading [contd.]

- How to invoke?

```
op x;
or
x op;          //for unary operators
x op y;        //for binary operators

operator op(x);     //for unary operator using friend function
operator op(x, y);  //for binary operator using friend function
```

Example: Unary Operator Overloading


```
#include<String.h>
#include<iostream>
using namespace std;
class abc
{
          int x,y,z;
      public:
          void getdata(int a, int b, int c){x=a; y=b; z=c;}
          void display(){cout<<x<<" ";cout<<y<<" ";cout<<z<<"\n";}
          void operator-(); //overload unary minus
};
void abc :: operator-()
{
          x=-x;
          y=-y;
          z=-z;
}
```

```
int main()
{
      abc A;
      A.getdata(60,-45,-72);

      cout<<"A: ";
      A.display();

      -A;
      cout<<"Now A: ";
      A.display();

      return 0;
 }
```

```
friend void operator-(abc &s);
```

```
void operator-(abc &s)
{
          s.x=-s.x;
          s.y=-s.y;
          s.z=-s.z;
}
```

Example: Binary Operator Overloading


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

class complex
{
            float x,y;
      public:
            complex(){}
            complex(float a, float b){x=a; y=b;}
            complex operator+(complex);
            void show();
};

complex complex :: operator+(complex c)
{
            complex tmp;
            tmp.x= x + c.x;
            tmp.y= y + c.y;
            return(tmp);
}
```

```
friend complex operator+(complex a, complex b);
```

```
void complex :: show()
{
            cout<<x<<" + i"<<y<<"\n";
}
int main()
{
            complex C1(2.5,3.6), C2, C3;
            C2=complex(1.5,1.4);
            C3=C1+C2;
            cout<<"C1= "; C1.show();
            cout<<"C2= "; C2.show();
            cout<<"C3= "; C3.show();

            return 0;
}
```

```
complex operator+(complex a, complex b)
{
    return complex((a.x+b.x),(a.y+b.y));
}
```

Mathematical Operations on Strings


```
#include<String.h>
#include<iostream>
using namespace std;
class String{
       char *p;
       int len;
    public:
       String(){len=0;p=0;}
       String(const char*s);
       String(const String &s);
       ~String(){delete p;}

        friend String operator+(const String &s, const String &t);
        friend int operator<=(const String &s, const String &t);
        friend void show(const String s);
};
String :: String(const char *s){
        len=strlen(s);
        p =new char[len+1];
        strcpy(p,s);
}
```

```
String :: String(const String & s)
{
            len=s.len;
            p =new char[len+1];
            strcpy(p,s.p);
}

String operator+(const String &s,
const String &t)
{
            String tmp;
            tmp.len=s.len+t.len;
            tmp.p=new char[tmp.len+1];
            strcpy(tmp.p,s.p);
            strcat(tmp.p,t.p);
            return(tmp);
}
void show(const String s)
{
            cout<<s.p;
}
```

Mathematical Operations on Strings


```
int operator<=(const String &s, const String &t)
{
            int l1=s.len;
            int l2=t.len;
            if(l1<=l2) return(1);
            else return(0);
}
```

```
int main(){
            String s1="ISI ";
            String s2="Kolkata";
            String t1,t2,t3;
            t1=s1;
            t2=s2;
            t3=s1+s2;
            cout<<"\nt1= ";show(t1);
            cout<<"\nt2= ";show(t2);
            cout<<"\nt3= ";show(t3);
            cout<<"\n\n";
            if(t2<=t3){
                        show(t2);
                        cout<<" smaller than ";
                        show(t3);cout<<"\n";
            }
            else{
                        show(t3);
                        cout<<" smaller than ";
                        show(t2);cout<<"\n";
            }
            return 0;
}
```

Some more Restrictions on Operator Overloading

- Only existing operators can be overloaded
- New operators cannot be created
- Basic meaning of the operator cannot be changed
- Syntax should remain the same
- Not all operator can be overloaded

Operators that cannot be overloaded by friend function: =, (), [], ->

Questions?