TATA ELXSI

OBJECT ORIENTED PROGRAMMING USING C++
Module 3

Learning & Development Team

Constructors and Destructors

Constructors: Features

- A special member function which is invoked automatically when an object is created.
- Generally declared in the public section.
- Has the same name as the class.
- Does not have return types.
- Can be overloaded.

Default constructors

- A default constructor is a constructor that can be called with no arguments.
- Ex1: Constructor that takes no parameters
  - C() { x = 5;}
- Ex2: Constructor that takes parameters, but they take default values.
  - C(int i = 5, int j=40) { }
- can be called with no arguments.

Constructors and Destructors

- An object's constructor is called when an object comes into existence and destructor is called when the object is destroyed.
- A constructor function's name is the same as the class name and destructor function name is preceded with a '~'. The constructor function can take input parameters.
- Local object's constructor is executed when the object's declaration statement is encountered. Destructor functions are executed in the reverse order of constructor functions.

Constructors and Destructors

- Global object's constructor functions are executed before main() and are executed in the order of their declaration, and destructors are executed in reverse order after main() has terminated.
- It is not necessary to define a constructor. If no constructor is specified, a default, public, parameter-less constructor is provided.
- A constructors generally are not private.

Constructor - Example

```
class myclass
{
   int a, b;
  public:
   myclass(int i, int j)
   {
      a=i;
      b=j;
   }
   void show()
   {
      cout << a;
      cout << b;
   }
};
```

```
int main()
{
   myclass ob(3, 5);
   ob.show();
   return 0;
}
```

Constructor with one parameter and multi parameter

```
#include <iostream>
using namespace std;
class X
{
   int a;int b;
  public:
   X(int j) {
      a = j;b =0;
   }
    X(int i, int j){
      a = i; b =j;
   }
   int getA()   {      return a;   }
   int getB()   {      return b;   }
};
```

```
int main()
{
   X ob(99);
 // passes 99 to j

   cout << ob.geta();
 // outputs 99

   X ob2(10,20);
   cout << ob2.getA();  // 10
   cout << ob2.getB();  // 20

   return 0;
}
```

Destructors

- Function that is invoked implicitly when the object is destroyed.
- They clean up and release resources.
- Common use of destructor is to release memory acquired in a constructor.
- The destructor notation is **~cl();**
- **Never call a destructor**
- **Destructor does not take any parameter and does not return any value.**

Execution of constructors and destructors

```
#include <iostream>
using namespace std;
class myclass
{
  public:
   int who;
   myclass(int id);
   ~myclass();
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id)
{
   cout << "Initializing\n";
   who = id;
```

```
myclass::~myclass()
{
   cout << "Destructor\n";
}

int main()
{
   myclass local_ob1(3);
   cout << "In main()\n";
   myclass local_ob2(4);
   return 0;
}
```

Constructors - Initializing using initializer list

```
class X
{
   int a;
   float f;
  public:
   X(int j, float x) :a( j) , f(x) { }
   void show()
   {
      cout << a << “ ”  << f << endl;
   }
};

int main()
{
   X ob (99, 99.99);
   cout << ob.show;
   return 0;
```

Initializing using initializer list

- Constructors should initialize all member objects using initializer list as a rule.
- It directly creates the object with the value given. Otherwise, the right side value of the assignment statement is temporarily stored in one address, copied and then erased.
- Non-static const and non-static reference data members can't be assigned a value in the constructor, so for symmetry it makes sense to initialize everything in the initialization list.
- There are some instances where initialiser list cannot be used, such cases use assignment within constructor. For example, if try-throw-catch statements have to be included.

'this' in Constructors

```
class X
{
   int a;
   int b;
  public:
   X(int j) :a(j) , b(a) { } //this→a = j, b=this→ a
   void show( ) { cout << a << b;}
};

int main()
{
   X ob (5);
   ob.show;
   return 0;
}
```

Named constructors

- If you want to differentiate between various constructors of a class through different names rather than their parameter list then you can use the Named Constructor Idiom.
- The approach is the following: you declare all the constructors in the private section and you provide public static methods to return an object. These static methods are called the Named Constructors.

Named Constructors

```
class Point
{
  public:
   static Point rectangular(float x, float y);
   static Point polar(float radius, float angle);
  private:
   Point(float x, float y);
   float x_, y_;
};
inline Point::Point(float x, float y) : x_(x), y_(y)
{}
inline Point Point::rectangular(float x, float y)
{return Point(x, y);}
```

Named Constructors

```
inline Point Point::polar(float radius, float angle)
{
   return Point(radius*cos(angle),
   radius*sin(angle));
}
main()
{
   Point p1 = Point::rectangular(5.7, 1.2);
   Point p2 = Point::polar(5.7, 1.8);
}
```

Copy constructors

- Copy constructor is a constructor function with the same name as the class and used to make deep copy of objects.

- There are 3 important places where a copy constructor is called.
  - When an object is created from another object of the same type.
  - When an object is passed by value as a parameter to a function.
  - When an object is returned from a function.

- If a copy constructor is not defined, the compiler creates one.

Copy constructors

```
class mycl
{
   int x;
  public:
   mycl()   {
      x = 10;
   }
   void changex(int ii) {
      x = ii;
   }
   void printx()   {
      cout << x << endl;
   }
};
```

```
int main()
{
   mycl a;
   a.printx();
   mycl b(a);
   a.changex(55);
   b.printx();
   a.printx();
}
```

```
Output: 10
        10
        55
```

Copy constructors

- It only makes a shallow copy. For instance, if there are pointers in the class and an object is created from the existing class, we cant be sure that the memory is allocated.
- Also the delete in destructor may be called twice.
- Hence if a class has a pointer variable, a copy constructor has to be defined.

Copy constructors

```
class mycl
{
   char *p;
  public:
   mycl()   {
      p=new char[10];
   }
   void fillp()   {
      strcpy(p, “Hello”);
      cout << p << endl;
   }
    ~mycl()   {
      delete [] p;
   }
```

```
   ~myclass()    {
      delete p;
   }
};
int main()
{
   myclass a;
   a.fillp();
   {
        myclass b(a);
    }//Object b is destructed
// Created dangling pointer in object a
}
```

Copy constructors

```
class myclass
{
    char *name;
  public:
   myclass()
   {name = new char[10];}

   myclass(const myclass &x)
   {
       name = new char[10];
       strcpy ( name, x.name);
       cout << name << end;
   }
```

```
  void fillp()
  {
     strcpy(name, “Hello”);
     cout << name << endl;
  }
  ~myclass()
  { delete []name;}
};    // end of myclass
int main()
{
   myclass a;
   a.fillp();
   myclass b(a);
}
```

Explicit constructors

- In C++, a constructor with only one required parameter is considered an implicit conversion function.
- It converts the parameter type to the class type.
- Whether this is a good thing or not depends on the semantics of the constructor.

Explicit constructors

```
class test {
     int i , j;
public:
    test() { i=100; j=200;    }
   explicit test(int x) { i=x;  j=x+10; }
   test(int x , int y){  i = x;     j = y;     }
    void disp() { cout<<"I="<<i<<endl; cout<<"J="<<j<<endl<<endl; }
};

int main() {
    test m2(10); // invokes single parameter constructor → OK
    test m3;
    m3 = 500;  // also, invokes single parameter constructor →NOT-OK
}
```

Array of objects

```
class cl
{
   int i;
  public:
   void set_print(int j) {i=j;}
};
cl obj[3];
// Access array elements
obj[1].set_print(3);
```

Array of objects

```
class cl
{
   int i;
  public:
   cl(int j) { i = j;}
};
cl obj[3] = {1,2,3};
//same as
cl obj[3] = { cl(1),
             cl(2),
             cl(3)};
```

```
class cl
{
   int a,b;
  public:
   cl(int i, int j)
   { a = i; b = j;}
};

cl obj[3] = { cl(1,2), cl(3,4),
cl(5,6)};
```

Are you ready to solve…

1. Common use of _________ is to release memory acquired in a constructor.
   a. Move constructor b. Copy constructor
   c. destructor d. none of them

   Ans: **c. destructor**

2. initialization list executes more faster than normal constructor.

   a. True b. False

   Ans: **a. True**

End of Module 3

Disclaimer

- Some examples and concepts have been sourced from the below links and are open source material
  - ❖ **http://cppreference.com**
  - ❖ **www.cplusplus.com**

- References:
  - ❖ *C++: The Complete Reference* - 4th Edition by Herbert Schildt, Tata McGraw-Hill publications.
  - ❖ *The C++ Programming Language-* by Bjarne Stroustrup.
  - ❖ *Practical C++ Programming-* by Steve Oualline, O'Reilly publications.

Learning & Development Team

ITPB Road Whitefield
Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com

Confidentiality Notice

This document and all information contained herein is the sole property of Tata Elxsi Limited and shall not be reproduced or disclosed to a third party without the express written consent of Tata Elxsi Limited.

TATA ELXSI