Creating and Destroying Objects

Теми

- Класове и обекти: разделяне на декларация и дефиниция. Създаване и унищожаване на обекти. Структура и обект.

- Конструктори и деструктори. Видове конструктори (виртуални и статични). Методи на клас.

Constructor

- Constructor function
  - Special member function
    - Initializes data members
    - Same name as class
  - Called when object instantiated
  - Several constructors
    - Function overloading
  - No return type

Rules for making a constructor (C++)

- A constructor must have the same name as the class.
- No return type; not even void.
- No return statement.
- Never call a constructor manually. The execution process takes care of that.
- Never declare constructor as virtual or static, const, volatile, or const volatile.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.

Let moving to C++

```
class XY {
      public:
            double x;
            double y;
};
```

User-defined type (named in OOP – class) is a declaration of data, used when type is instantiated in an object, and set of operation needed for object manipulation

- **Declaration of an object:**

```
XY alfa;// an uninitialized object
alfa.x = 2.0;
alfa.y = 3.0;
```

Default constructors

- A default constructor is a constructor that either has no parameters, or if it has parameters, all the parameters have default values.

- No explicit constructor declaration => the compiler assumes the class to have a *default constructor* with no arguments.

Constructors

- in a function we construct the declared object:

```
void f(){
    XY bottomRight;
    // construction: in stack,  x and y - uninitialized
}
```

- Don’t call default constructors as function – it seams like a call to forward declared function, returning XY:

```
XY bottomRight();
```

Initializing Class Objects: Constructors

- Initializers
  - Passed as arguments to constructor
  - In parentheses to right of class name before semicolon

    Class-type ObjectName(value1,value2,…);

Constructors

- Constructors can have multiple parameters: example with taking 2 parameters:

```
class XY {
      public:
         double x,y;
         XY(double a, double b)
            { x=a; y =b;}
};
```

- Now – declaring object:

  XY bottomRight(5.0, 7.0);

Constructors

- It's possible to have more than 1 constructors in a class:

```
class XY {
  public:
      double x,y;
      XY(){
            x = 0.0;
            y = 1.0;
      }
      XY(double a, double b;)
            {x =a; y =b;}
};
```

- Reference to a constructor:

```
XY mytop;
XY secondtop(2.0, 2.0);
```

- The constructor can not be declared as virtual or friend

- **Object of a class type must be initialized.**
- If a default constructor is present, it is called
- If not – a suitable constructor of the object is called
- If not a default constructor is produced by the compiler

- **An object can be member of a complex object if:**
  - Object's class possesses constructor without parameters;
  - Object's class does not possess constructor;
  - If the complex object has a constructor including values for initializing his member-object. So, a constructor of the member is called in the time the parent is constructed.

Example

```
class CBox
{     // ….
      //Constructor definition
  CBox(double lv = 1.0, double bv = 1.0,
                                  double hv = 1.0)
  {
      cout << endl << “Constructor called.”;
      m_Length = lv; // Set values of
      m_Width = bv; // data members
      m_Height = hv;
  }
};
```

Example

```
// Constructor definition using an
//initialization list
CBox(double lv = 1.0,
     double bv = 1.0, double hv = 1.0):
          m_Length(lv), m_Width(bv),
                        m_Height(hv)
{
     cout << endl << "Constructor
called.";
}
```

Example 1/2

```
// Ex7_07.cpp
// A class with private members
#include <iostream>
using std::cout;
using std::endl;

class CBox                                  // Class definition at global scope
{
  public:

    // Constructor definition using an initialisation list
    CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0):
                              m_Length(lv), m_Width(bv), m_Height(hv)
   {
     cout << endl << "Constructor called.";
   }

   // Function to calculate the volume of a box
   double Volume()
```

```
    {
      return m_Length*m_Width*m_Height;
    }

  private:
    double m_Length;                    // Length of a box in inches
    double m_Width;                     // Width of a box in inches
    double m_Height;                    // Height of a box in inches
};

int main()
{
  CBox match(2.2, 1.1, 0.5);            // Declare match box
  CBox box2;                            // Declare box2 - no initial values

  cout << endl
       << "Volume of match = "
       << match.Volume();

// Uncomment the following line to get an error
// box2.m_Length = 4.0;

  cout << endl
       << "Volume of box2 = "
       << box2.Volume();

  cout << endl;
  return 0;
}
```

Stack & Heap Objects

- Objects can be created with commands of the following form:
  - **On the stack:**

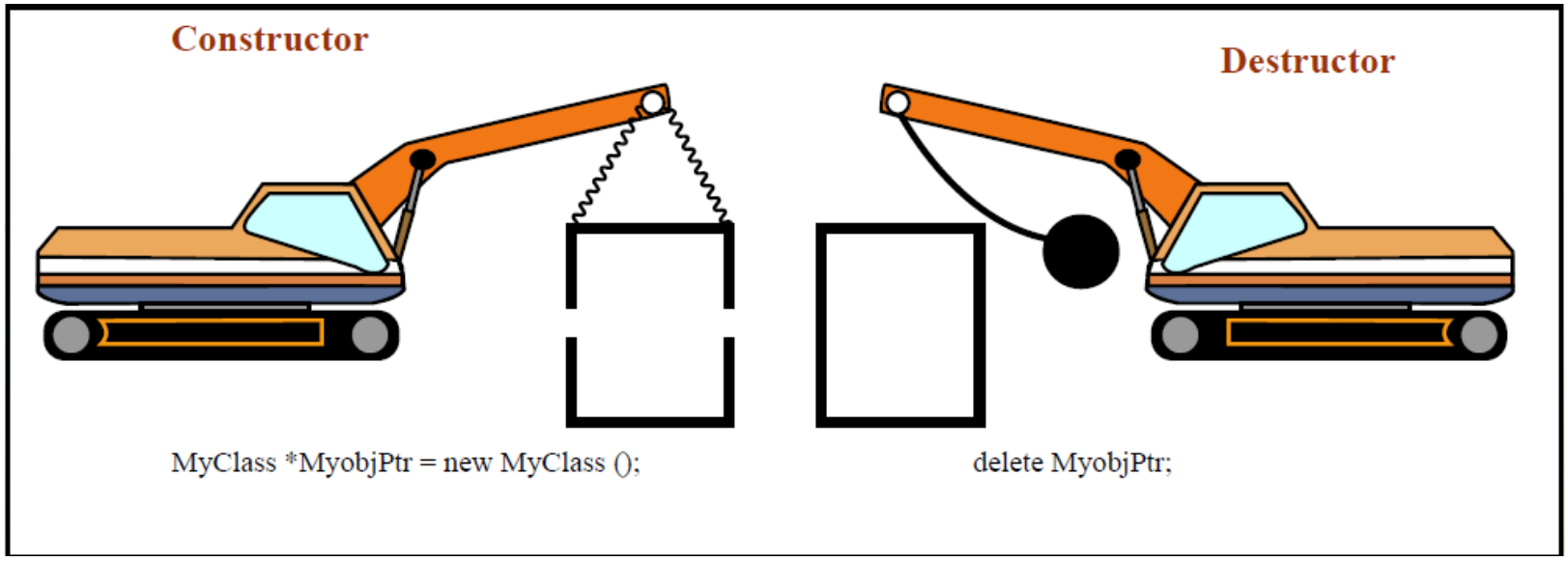
```
My_class my_object(arglist);
```

  - **On the heap:**

```
My_class* my_objptr = new My_class(arglist);
```

- To send a message to an object via its pointer use:

```
my_objptr->Message(arg_list);
```



- The code for the actual construction or destruction of an object is added on by the compiler and you do not see it.

Destructors

- Destructors
  - Same name as class
    - Preceded with tilde (~)
  - No arguments
  - Cannot be overloaded
  - Performs “termination housekeeping”

Destructors

- In a class, we can have no more than 1 destructor.
- He seams like a function with ~
- He take no arguments and return noting
- He is automatically called for any stack or global object, when that object goes out of scope.

```
class XY {
  public:
     double x,y;
     XY();      // default constructor
     XY( double a, double b);
     ~XY();                  // destructor
};
```

Destructors

- If you don't write a destructor, the compiler generates a default for you.
- For data members, that are C++ objects, the default destructor calls those object's destructors.
- When destructing, the compiler releases the storage, occupied by that object.

Destructors - examples

```
XY::XY()
{
   printf("default constructor called\n");
   x=y=0.0;
}

XY::XY(double a, double b)
{
   printf(" second explicit constructor called\n");
   x = a; y = b;
}

XY::~XY()
{
   printf("destructor called\n");
}
```

Summary

- When objects die, their destructor is called. Its name is of the form:

```
~My_class
```

- Stack objects are short-lived, in a C++ compiled program they die automatically when control leaves the innermost compound statement that contains.

```
gROOT->Reset();
```

- Heap objects are long lived, they are deleted using a command of the form:

```
delete my_objptr;
```

- Pointers to objects have to be carefully managed:
  - Forgetting to delete a heap object or losing its pointer results in a memory leak
  - Using a pointer after the object it points to has been deleted, is illegal and may well crash the program!

C++/CLI

- value class

```
value class Height
{

};
```

- ref class - This creates a reference type managed by the CLR.

```
ref class Height
{

};
```

Example - value class

```
// Class representing a height
value class Height
{
        private:
            // Records the height in feet and inches
            int feet;
            int inches;
        public:
            // Create a height from inches value
            Height(int ins)
            {
                        feet = ins/12;
                        inches = ins%12;
            }
        // Create a height from feet and inches
        Height(int ft, int ins) : feet(ft), inches(ins){}
};
```

Example - value class

```
int main(array<System::String ^> ^args)
{
    Height myHeight = Height(6,3);
    Height^ yourHeight = Height(70);
    Height hisHeight = *yourHeight;
    Console::WriteLine(L”My height is {0}”,
                       myHeight);
    Console::WriteLine(L”Your height is {0}”,
                       yourHeight);
    Console::WriteLine(L”His height is {0}”,
                       hisHeight);
    return 0;
}
```

```
ref class Box
{
    public:
        Box(): Length(1.0), Width(1.0), Height(1.0)
        {
            Console::WriteLine(L”No-arg constructor called.”);
        }
        Box(double lv, double bv, double hv):
                                    Length(lv), Width(bv), Height(hv)
        {
                        Console::WriteLine(L”Constructor called.”);
        }
        double Volume()
        {
            return Length*Width*Height;
        }
        private:
            double Length; // Length of a box in inches
            double Width; // Width of a box in inches
            double Height; // Height of a box in inches
};
```

```
int main(array<System::String ^> ^args)
{
    Box^ aBox; // Handle of type Box^
    Box^ newBox = gcnew Box(10, 15, 20);
    aBox = gcnew Box; // Initialize with default Box
    Console::WriteLine(L"Default box volume is {0}", aBox-
                                        >Volume());
    Console::WriteLine(L"New box volume is {0}", newBox-
                                        >Volume());
    return 0;
}
```

First real example 1/2

```
using namespace System;

__gc class animal

{

public:

        int    legs;

        void SetName(String *Name) { strName =
strName->Copy(Name); };

        String* GetName() { return strName; };

private:

        String  *strName;

};
```

First real example – 2/2

```
//This is the entry point for this application
int _tmain(void)
{ Cat = new animal;
   Dog = new animal;
   Cat->SetName("Cat");
   Cat->legs = 4;
   Dog->SetName("Dog");
   Dog->legs = 4;
   Console::Write("Name ");
   Console::WriteLine(Cat->GetName());
   Console::Write("Legs ");
   Console::WriteLine(Cat->legs);
   Console::WriteLine();
return 0;
}
```

Using Private Constructors

- A private constructor prevents unwanted objects from being created
  - Instance methods cannot be called
  - Static methods can be called
  - A useful way of implementing procedural functions

```
public class Math
{
    public static double Cos(double x) { ... }
    public static double Sin(double x) { ... }
    private Math( ) { }
}
```

Using Static Constructors

- Purpose
  - Called by the class loader at run time
  - Can be used to initialize static fields
  - Guaranteed to be called before instance constructor
- Restrictions
  - Cannot be called
  - Cannot have an access modifier
  - Must be parameterless

Example

```
public class Complex
 {
     public double real;
     public double imaginary;

     public static Complex FromCartesianFactory(double real, double imaginary )
     {
         return new Complex(real, imaginary);
     }

     public static Complex FromPolarFactory(double modulus , double angle )
     {
         return new Complex(modulus * Math.Cos(angle), modulus * Math.Sin(angle));
     }


     private Complex (double real, double imaginary)
     {
         this.real = real;
         this.imaginary = imaginary;
     }
 }

Complex product = Complex.FromPolarFactory(1,pi);
```