BIT

# 6. Initialization & CleanUp

Hu Sikang
*skhu@163.com*

School of Computer
Beijing Institute of Technology

B17

# Initialization & CleanUp

- Encapsulation and access control make a significant step in improving the ease of library use.
- In safety C++ compiler can do more for us than C provides.
- Two of these safety issues are initialization and cleanup.

B17

# Contents

- Initialization with the constructor
- Cleanup with the destructor
- Aggregate initialization

# 6.1.1 Problem

```
#include <iostream>
using namespace std;
class Point {
public:
  void Init(double a, double b)
        {  coordX = a;    coordY = b;  }
   double GetX( )  {  return coordX; }
   double GetY( )  {  return coordY; }
private:
    double coordX, coordY;
};
```

```
int main( )
{
        Point p;
        p.Init(1, 2);
        p.GetX();
        return 0;
}
```

It's for programmer to wish that object could be *initialized automatically* when it is created.

B17

# 6.1.2 initialization with the constructor

- In C++, initialization is too important to leave to the client programmer.
- The compiler automatically calls the constructor at the point an object is created.

# 6.1.3 Constructors

```
#include <iostream>
using namespace std;
class Point {
public:
        Point( )   { coordX = coordY = 0; }
        void SetPoint(double x, double y);
        double GetX( )  { return coordX; }
private:
        double coordX, coordY;
};
```

Constructor

```
int main( )
{
    Point p;
    p.SetPoint(1, 2);
    p.GetX();
     return 0;
}
```

***Constructor*** **is recognized by having** ***the same name*** **as the class itself.**

# 6.1.3 Constructors

- *Constructor* is recognized by having the same name as the class itself.
- *Constructor* is called by C++ automatically.
- *Constructor* is called to create an object.
- If you don't define constructor, C++ provides a default constructors: no parameters, no realization.

```
class Point {
public:
  Point() { };  // Create constructor if you don't define
        …
private:
    double coordX, coordY;     //the coordinates
};
```

**Default Constructor**

# 6.1.3 Constructors

- *Overloaded Constructors*: with different parameters or types

```
class Point
{
public:
    Point( )  { coordX = coordY = 0;}
    Point(double, double);
private:
    double coordX, coordY;
};
```

```
Point::Point(double vx, double vy)
{
      coordX = vx;
      coordY = vy;
}
int main( )
{
    Point p1;
    Point p2(2, 6);
    return 0;
}
```

BIT

# 6.1.3 Constructors

1. **Constructor doesn't have returning type.**
2. **Constructor is called automatically when an object is created.**
3. **Constructor CANNOT be called by object.**

```
class A  {   public:   A( ) { }  };
main( ) {
      A  a;       // constructor is called automatically
      a.A( );     // error!
}
```

4. **There may be many constructors in a class.**

BIT

# 6.1.4 Objects of a class

*Exercises*

**Objects were defined as following, how to define the construtor?**

- **Point p1;** // default constructor
- **Point p2(20,30);** // overloaded constructor
- **Point pArray[3];** // default constructor

B17

# 6.2 Cleanup with the destructor

- In C++, cleanup is as important as initialization.
- A *destructor* clean up and release resources.
- A destructor is recognized by having *the same name* as the class itself with the complement symbol(~) .
- A destructor has not any arguments.

# 6.2.1 Destructors

- Called when an object is deleted.
- Defined with the name: *~classname();*

```
#include <iostream>
using namespace std;
class Point {
public:
   Point(double x, double y)
  {   coordX = x;      coordY = y;   }
    ~Point()    //destructor
   { cout << "This is destructor of Point class." << endl; }
private:
   double coordX, coordY;
};
```

```
int main()
{
      Point p(1, 1);
      return 0;
}  // Here destructor is
   //  called automatically
```

# 6.2.1 Destructors

**Notes:**

1. **Destructor doesn't have returning type.**
2. **Destructor is called when an object is destroyed.**
3. **Destructor can be called by object.**

   **class Point;  Point  p;  p.~A();**
4. **Destructor doesn't have argument.**
5. **There is only one destructors in a class.**
6. **Destructors are called in the reverse order of constructors.**

   **class Point;  Point  p1(1, 1),  p2(3, 5);**

# 6.2.1 Destructors

If client programmer need call destructor explicitly, he must define the pointer of class and use it with operator, *new* and *delete*.

```
#include <iostream>
using namespace std;
class Point
{
public:
   Point(double x, double y)  {
            coordX = x;
            coordY = y;
   }
    ~Point()     //destructor
   {   cout << "This is destructor of  Point class." << endl;   }
private:
   double  coordX, coordY;
};
```

If the statement is written as follows:
p = new Point[5];
How should we modified the codes?

```
int main( )
{
   Point *p;

   // Constructor is called.
   p = new Point(1,1);

  // Destructor is called.
   delete p;

   return 0;
}
```


# 6.2.2 Constructors and Destructors

① A constructor initializes objects and constructs values of a given type.
② A constructor is recognized by having *the same name* as the class itself.
③ A constructor can be *overloaded*.
④ A constructor has no *return*.
⑤ It can be invoked when an object is *created*.

① A destructor clean up and release resources.
② A destructor is recognized by having *the same name* as the class itself with the complement symbol(~) .
③ A destructor has not formal arguments and *cannot be overloaded*.
④ A destructor has no *return*.
⑤ It can be invoked when an object is *destroyed*.

# Example: Constructors and Destructors with dynamic memory

```
#include <iostream>
using namespace std;
class CMyString {
public:
  CMyString( )   { str = new char[50]; }
  void Copy(char*);
  ~CMyString( )
  {  if  (str != nullptr)   delete[ ] str; }
private:
   char* str;
};
```

```
void CMyString::Copy(char* ch)
{
        int i = 0;
        while (ch[i] != '\0')
            { str[i] = ch[i]; i++; }

        str[i] = ch[i];  // '\0'
        cout << str << endl;
}

int main( )
{
      CMyString  my;
      my.Copy("hello!");
      return 0;
}
```

B17

# 6.3 Aggregate initialization

- int a[5] = { 1, 2, 3, 4, 5 };
- int b[6] = { 2 };
- int c[ ] = { 1, 2, 3, 4 };

B17

# 6.3 initialization of Object Arrays

// Initialization

- *Point P*[3] = {Point(1,2), Point(3,4), Point(4,5)};

- *Point P*[3];

// Assignment. Compiler will create temporary objects.

- P[0] = Point(1,2); // create, assign, delete
- P[1] = Point(3,4);
- P[2] = Point(4,5);

# B17

```
#include <iostream>
using namespace std;
class DATE
 {
  private:
     int year, month, day;
  public:
    DATE( )   {
          year = month = day = 0;
          cout << "Default constructor called.“ << endl;
    }
    DATE(int y, int m, int d)  {
          year = y;    month = m;    day = d;
          cout << "Constructor called.“ << day << endl;
    }
    ~DATE( ) { cout<<"Destructor called."<<day<<endl; }
    void Print( ) {cout<<year<<":"<< month<<":"<<day<<endl; }
};
```

```
int main( )
{
      DATE dates[3] = { DATE(2024,3,4),  DATE(2024,3,7)};
      return 0;
 }
```