

Constructors and Destructors



C++ Object Oriented Programming
Pei-yih Ting
NTOU CS

Contents

- ❖ House Keeping Problem
- ❖ Constructors
- ❖ Destructors
- ❖ Invoking Mechanisms
- ❖ Advantage of OOP
- ❖ Multiple Constructors
- ❖ Array of Objects
- ❖ Default Arguments
- ❖ Initialization Lists
- ❖ Constant Data Members Initialization

House Keeping Problems

- ❖ What is wrong with the following code?

```
class Array {
public:
    void initArray(int arraySize);
    void insertElement(int element, int slot);
    int getElement(int slot) const;
    void cleanUp();
```

```
private:
    int m_arraySize;
    int *m_arrayData;
```

```
};
void Array::initArray(int arraySize) {
    m_arrayData = new int[arraySize];
    m_arraySize = arraySize;
```

```
}
void main() {
    Array array;
    array.insertElement(10, 0);
}
```

Assume insertElement(), getElement(), and cleanUp() are defined elsewhere.

In the client code: main
1. Forget to initialize the object array (there is no call to initArray())
2. Forget to call cleanUp() code segment

Invalid Internal State

- ❖ Initialization
 - * Interface functions are used to maintain the internal state of an object valid and consistent
 - * Without suitable initialization, the object's initial state would be invalid.
 - * We need a method to guarantee that each new object is well initialized.
- ❖ Clean up
 - * Clean up is important if a program is to run for a long time. If resources (memory, file, ...) are occupied one by one and forget to released afterwards, sooner or later no program would be running.
 - * We need a method to guarantee that each object is well cleaned up.

Constructors

- ❖ **ctor**: A **constructor** is a function called automatically when an object comes into existence.
- ❖ Syntax
 - * The name of the constructor is **the same as the class name**
 - * Must not have a return type
 - * Parameters must be supplied when the object is defined.
 - * Cannot be called elsewhere (explicitly) inside the program

<pre>class Array { public: Array(int arraySize); void insertElement(int element, int slot); int getElement(int slot) const; private: int m_arraySize; int *m_array; };</pre>	<pre>void main() { Array array(20); array.insertElement(10, 0); } Array::Array(int arraySize) { m_array = new int[arraySize]; m_arraySize = arraySize; }</pre>
---	---

5

Destructors

- ❖ **dtor**: A **destructor** is a function called automatically when an object's life comes to an end. (goes out of scope, program ends, or deleted by the program)
- ❖ Syntax
 - * The name of the destructor must be the same as the name of the class preceded by ~ (tilde).
~Array();
 - * Destructors take no arguments and return no values
- ❖ Purpose: to free any resource allocated by the object.

<pre>class Array { public: ... ~Array(); ... };</pre>	<pre>Array::~~Array() { delete [] m_array; }</pre>
---	--

6

When are ctors and dtors called?

- ❖ Static variables (local, global)

```
void Foo() {
    Array array(20); // ctor invoked
    array.insertElement(10, 0);
    cout << array.getElement(0);
} // dtor invoked
```

What would happen if there were no destructor?

- * dtor of a global variable will be invoked when the program exits

- ❖ Dynamic variables

```
Array *Foo(int numElements) {
    Array *array;
    array = new Array(numElements); // ctor invoked
    return array;
}

void Bar() {
    Array *mainData = Foo(20);
    delete mainData; // dtor invoked
}
```

What would happen if we did not call delete?

7

Advantages Achieved by OOP

Automatic initialization

```
Array::Array(int arraySize) {
    m_array = new int[arraySize];
    m_arraySize = arraySize;
}
```

Reduced memory-leakage risks

```
Array::~~Array() {
    delete [] m_array;
}
```

Safe client/server programming

```
void Array::insertElement(int element, int slot) {
    if ((slot < m_arraySize) && (slot >= 0))
        m_array[slot] = element;
    else
        cout << "Warning, out of range!!";
}

int Array::getElement(int slot) const {
    if ((slot < m_arraySize) && (slot >= 0))
        return m_array[slot];
    else {
        cout << "Warning, out of range!!";
        return 0;
    }
}
```

Better abstraction

```
cout << array.getElement(0);
```

Conceptually, an array is no longer just a chunk of data storages.

8

Multiple Constructors

- ❖ A class can have more than one constructor (function overloading)

```
class Name
{
public:
    Name();
    Name(char *firstName, char *lastName);
    ~Name();
    void setName(char *firstName, char *lastName);
    void printName() const;
private:
    char *m_firstName;
    char *m_lastName;
};

Name::Name() ←----- This ctor is called "default constructor".
{
    m_firstName = 0;
    m_lastName = 0;
}

Name::Name(char *firstName, char *lastName)
{
    setName(firstName, lastName);
}
```

9

Multiple Constructors (cont'd)

```
void Name::setName(char *firstName, char *lastName)
{
    m_firstName = new char[strlen(firstName)+1];
    m_lastName = new char[strlen(lastName)+1];
    strcpy(m_firstName, firstName);
    strcpy(m_lastName, lastName);
}

Name::~~Name()
{
    delete[] m_firstName;
    delete[] m_lastName;
}

void Name::printName() const
{
    if (m_firstName) cout << m_firstName << ' ';
    if (m_lastName) cout << m_lastName << ' ';
}

➤ Usage:
void main()
{
    Name name1, name2("Mary", "Smith");
    name1.setName("Mark", "Anderson");
    name1.printName(); name2.printName();
}
```

10

Constructors and Arrays

- ❖ If you try to define an array of objects, you can NOT do this

```
class Name
{
public:
    Name(char *firstName, char *lastName);
    ~Name();
    void setName(char *firstName, char *lastName);
    void printName() const;
private:
    char *m_firstName;
    char *m_lastName;
};

void main()
{
    Name names[100];
    names[0].setName("Mark", "Anderson");
    names[0].printName();
}
```

- error C2512: 'Name' : no appropriate default constructor available

Name() is the so-called default constructor

11

Solutions to Array of Objects

- ❖ Solution 1: provide a ctor without arguments ... i.e. the default ctor

```
class Name {
public:
    Name();
    Name(char *firstName, char *lastName);
    ~Name();
    void setName(char *firstName, char *lastName);
    void printName() const;
private:
    char *m_firstName;
    char *m_lastName;
};
```

- ❖ Solution 2: have no ctor at all ... i.e. use the implicit default ctor

```
class Name {
public:
    ~Name();
    void setName(char *firstName, char *lastName);
    void printName() const;
private:
    char *m_firstName;
    char *m_lastName;
};
```

12

Constructors with Default Arguments

- Consider this class with two constructors

```
class Account {
public:
    Account();
    Account(double startingBalance);
    void changeBalance(double amount);
    void showBalance() const;
private:
    double m_balance;
};

Account::Account() {
    m_balance = 0.0;
}

Account::Account(double startingBalance) {
    m_balance = startingBalance;
}

void main() {
    Account client1, client2(100.0);
    client1.showBalance();
    client2.showBalance();
}
```

Output:
0.0
100.0

13

Ctor with Default Arguments (cont'd)

- The class is rewritten as follows

```
class Account {
public:
    Account(double startingBalance=0.0);
    void changeBalance(double amount);
    void showBalance() const;
private:
    double m_balance;
};

Account::Account(double startingBalance) {
    m_balance = startingBalance;
}
```

The single ctor is exactly the same as before

- We can now declare an array of Account.

```
void main() {
    Account clients[100];
    clients[0].changeBalance(100.0); clients[0].showBalance();
}
```

This works fine without default ctor.

14

Initialization Lists

- Consider the following class

```
enum Breed {undefined, collie, poodle, coca, bulldog};
class Dog {
public:
    Dog();
    Dog(char *name, Breed breed, int age);
    ~Dog();
    void list() const;
private:
    char *m_name;
    Breed m_breed;
    int m_age;
};
```

★ This ctor can be rewritten as:
Dog::Dog(char *name, Breed breed, int age)
: m_name(new char[strlen(name)+1]),
m_breed(breed), m_age(age) {
strcpy(m_name, name);
}

- The constructor might look like this

```
Dog::Dog(char *name, Breed breed, int age) {
    m_name = new char[strlen(name)+1];
    strcpy(m_name, name);
    m_breed = breed;
    m_age = age;
}
```

15

Constant Data Member Initialization

- The breed of the dog will not change, so let us make this a constant variable in the class declaration.

```
class Dog {
public:
    Dog();
    Dog(char *name, Breed breed, int age);
    ~Dog();
    void list() const;
private:
    char *m_name;
    const Breed m_breed;
    int m_age;
};
```

◇ Constant variables MUST be initialized in the initialization list
Dog::Dog():m_breed(undefined) {}

- Other good uses for const

```
Dog::Dog(const char *name, const Breed breed, const int age)
: m_name(new char[strlen(name)+1]),
m_breed(breed), m_age(age) {
    strcpy(m_name, name);
}
```

16

Initialization List (cont'd)

- ❖ There are several cases where initialization list **MUST** be used
 - * Constant data member
 - * Reference data member
 - * Non-default parent class constructor
 - * Non-default component object constructor
- ❖ Coding style: use initialization list as much as possible
 - * initialization list is inevitable in many cases
 - * initialization will be performed implicitly in the initialization list whether you use it or not. It saves some computation to do it in the initialization list.
- ❖ Caution:
 - * The order of expressions in the initialization list is NOT the order of execution, the defining order of member variables in the class definition defines the order of execution.

```
Dog::Dog(const char *name, const Breed breed, const int age)
: m_age(age), m_name(new char[strlen(name)+1]), m_breed(breed){
strcpy(m_name, name);
}
```

third first second