Constructors and Destructors

C++ Object Oriented Programming
Pei-yih Ting
93/02 NTOU CS

House Keeping Problems

What is wrong with the following code?

class CArray {
public:
    void initArray(int arraySize);
    void insertElement(int element, int slot);
    int getElement(int slot) const;
private:
    int m_arraySize;
    int *m_arrayData;
};

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

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

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

1. Forget to initialize the object array (there is no call to initArray())
2. There is no deallocation code segments

Invalid Internal State

Initialization

* Interface functions are used to keep 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 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
  - Must not have a return type
  - Parameters must be supplied when the object is defined.
  - Cannot be called elsewhere (explicitly) inside the program

```cpp
class Array
{
    public:
        Array(int arraySize);
    private:
        int m_arraySize;
        int *m_array;
};

void main()
{
    Array array(20);
    array.insertElement(10, 0);
}
```

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.

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

When are ctors and dtors called?

- **Static variables (local, global)**
  - void Foo() {
    // ctor invoked
    Array array(20); // ctor invoked
    cout << array.getElement(0);
    // dtor invoked
  }
  // dtor of a global variable will be invoked when the program exits
- **Dynamic variables**
  - Array *Foo(int numElements) {
    // ctor invoked
    Array *array;
    array = new Array(numElements);
    return array;
  }
  void Bar()
  {
    Array *mainData = Foo(20);
    delete mainData; // dtor invoked
  }

Advantages Achieved by OOP

- **Automatic initialization**
  - Array::Array(int arraySize) {
    m_array = new int[arraySize];
    m_arraySize = arraySize;
  }
- **Reduced memory-leak 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**
  - No longer the image of a whole block of memories
Multiple Constructors

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

  ```cpp
  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;
  }
  ```

  ```cpp
  Name::Name()
  {
    m_firstName = 0;
    m_lastName = 0;
  }
  ```

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

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

Usage:

```cpp
void main()
{
  Name name1, name2("Mary", "Smith");
  name1.setName("Mark", "Anderson");
  name1.printName();
  name2.printName();
}
```
Constructors with Default Arguments

- Consider this class with two constructors
  ```cpp
class Account {
public:
    Account();
    Account(double startingBalance);
    void changeBalance(double amount);
    void showBalance() const;
private:
    double m_balance;
};
```
  ```cpp
  Account::Account() {
    m_balance = 0.0;
  }
  ```
  ```cpp
  Account::Account(double startingBalance) {
    m_balance = startingBalance;
  }
  ```

- The class is rewritten as follows
  ```cpp
class Account {
public:
    Account(double startingBalance=0.0);
    void changeBalance(double amount);
    void showBalance() const;
private:
    double m_balance;
};
```
  ```cpp
  Account::Account(double startingBalance) {
    m_balance = startingBalance;
  }
  ```

- We can now declare an array of Account.
  ```cpp
  void main() {
    Account clients[100];
    clients[0].changeBalance(100.0); clients[0].showBalance();
  }
  ```

Initialization Lists

- Consider the following class
  ```cpp
  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;
  };
```

- The constructor might look like this
  ```cpp
  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;
  }
  ```

- This can be rewritten as:
  ```cpp
  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);
  }
  ```

- Constant Data Member Initialization

- The breed of the dog will not change, so let us make this a constant variable in the class declaration.
  ```cpp
  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
  ```cpp
  Dog::Dog():m_breed(undefined) {}  
  ```

- Other good uses for const
  ```cpp
  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);
  }  
  ```
Initialization List

- 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.

```cpp
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);
}  
```