Constructors and Destructors

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Contents

House Keeping Problems

```
class Array {
```

House Keeping Problems

```
class Array {

private:
    int m_arraySize;
```

House Keeping Problems

```
class Array {




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

House Keeping Problems

```
class Array {
public:
   void initArray(int arraySize);



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

House Keeping Problems

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


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

House Keeping Problems

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

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

House Keeping Problems

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

private:
   int m_arraySize;
   int *m_arrayData;
};

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

House Keeping Problems

```
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;
}
```

House Keeping Problems

```
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;
}
```

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

House Keeping Problems

```
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.

House Keeping Problems

- What is wrong with these 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.

House Keeping Problems

- What is wrong with these 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);
}
```

In the client code: main()

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

House Keeping Problems

- What is wrong with these 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);
}
```

In the client code: main()

1. **Forget** to initialize the object array (there is no call to initArray())

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

House Keeping Problems

- What is wrong with these 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);
}
```

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

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

Invalid Internal State

- Initialization

Invalid Internal State

- Initialization
  - Interface functions are required to maintain the internal state of an object such that they are valid and consistent all the time.

Invalid Internal State

- Initialization
  - Interface functions are required to maintain the internal state of an object such that they are valid and consistent all the time.
  - Without suitable initialization, the object’s initial state would be invalid.

Invalid Internal State

- Initialization
  - Interface functions are required to maintain the internal state of an object such that they are valid and consistent all the time.
  - Without suitable initialization, the object's initial state would be invalid.
  - We need a method to guarantee that each new object is well initialized. No additional care should be taken by the client codes.

Invalid Internal State

- Initialization
  - Interface functions are required to maintain the internal state of an object such that they are valid and consistent all the time.
  - Without suitable initialization, the object's initial state would be invalid.
  - We need a method to guarantee that each new object is well initialized. No additional care should be taken by the client codes.
- Clean up

Invalid Internal State

- Initialization
  - Interface functions are required to maintain the internal state of an object such that they are valid and consistent all the time.
  - Without suitable initialization, the object's initial state would be invalid.
  - We need a method to guarantee that each new object is well initialized. No additional care should be taken by the client codes.
- Clean up
  - Clean up is important if a program is supposed 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 have enough resources to finish their job correctly.

Invalid Internal State

- Initialization
  - Interface functions are required to maintain the internal state of an object such that they are valid and consistent all the time.
  - Without suitable initialization, the object's initial state would be invalid.
  - We need a method to guarantee that each new object is well initialized. No additional care should be taken by the client codes.
- Clean up
  - Clean up is important if a program is supposed 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 have enough resources to finish their job correctly.
  - We need a method to guarantee that each object is well cleaned up. No additional care should be taken by the client codes.

Constructors

- **ctor**: A **constructor** is a function executed automatically when an object comes into existence.

Constructors

- **ctor**: A **constructor** is a function executed automatically when an object comes into existence.
- **Syntax**
  - The name of the constructor is **the same as the class name**

```
class Array {
public:
   Array(int arraySize);


private:
   int m_arraySize;
   int *m_array;
};
```

Constructors

- **ctor**: A **constructor** is a function executed automatically when an object comes into existence.
- **Syntax**
  - The name of the constructor is **the same as the class name**

```
class Array {
public:
    Array(int arraySize);


private:
    int m_arraySize;
    int *m_array;
};
```

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

Constructors

- **ctor**: A **constructor** is a function executed automatically when an object comes into existence.
- **Syntax**
  - The name of the constructor is **the same as the class name**

class Array {
public:
   Array(int arraySize);
   void insertElement(int element, int slot);
   int getElement(int slot) const;
private:
   int m_arraySize;
   int *m_array;
};
	Array::Array(int arraySize) {
	   m_array = new int[arraySize];
	   m_arraySize = arraySize;
	}

Constructors

- **ctor**: A **constructor** is a function executed 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

```
class Array {
public:
   Array(int arraySize);
   void insertElement(int element, int slot);
   int getElement(int slot) const;
private:
   int m_arraySize;
   int *m_array;
};
```

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

Constructors

- **ctor**: A **constructor** is a function executed 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.

```
class Array {
public:
    Array(int arraySize);
    void insertElement(int element, int slot);
    int getElement(int slot) const;
private:
    int m_arraySize;
    int *m_array;
};
```

```
void main() {
    Array array(20);
    array.insertElement(10, 0);
}
Array::Array(int arraySize) {
    m_array = new int[arraySize];
    m_arraySize = arraySize;
}
```

Constructors

- **ctor**: A **constructor** is a function executed 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.
  - Do **not** call it (explicitly) except the following :

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

Constructors

- **ctor**: A **constructor** is a function executed <u>automatically</u> 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.
  - Do **not** call it (explicitly) except the following :

  1. new statements, 2. initialization lists, 3. temporary objects

```
class Array {
public:
    Array(int arraySize);
    void insertElement(int element, int slot);
    int getElement(int slot) const;
private:
    int m_arraySize;
    int *m_array;
};
```

```
void main() {
    Array array(20);
    array.insertElement(10, 0);
}
Array::Array(int arraySize) {
    m_array = new int[arraySize];
    m_arraySize = arraySize;
}
```

Destructors

- **dtor**: A **destructor** is a function executed automatically when an object's life comes to an end. (goes out of scope, program ends, or is deleted dynamically)

Destructors

- **dtor**: A **destructor** is a function executed automatically when an object's life comes to an end. (goes out of scope, program ends, or is deleted dynamically)
- **Syntax**
  - The name of the destructor must be **the same as the name of the class** preceded by **~** (tilde).

    ~Array();

Destructors

- **dtor**: A **destructor** is a function executed automatically when an object's life comes to an end. (goes out of scope, program ends, or is deleted dynamically)
- **Syntax**
  - The name of the destructor must be **the same as the name of the class** preceded by **~** (tilde).

    ~Array();

```
class Array
{
public:
    …
    ~Array();
    …
};
```

Destructors

- **dtor**: A **destructor** is a function executed automatically when an object's life comes to an end. (goes out of scope, program ends, or is deleted dynamically)
- **Syntax**
  - The name of the destructor must be **the same as the name of the class** preceded by **~** (tilde).

    ~Array();

Purpose: to free any resource (memory, file, connection) allocated by the object.

class Array
{
public:
  …
  ~Array();
  …
};
	Array::**~Array()** {
	  delete[] m_array;
	}

Destructors

- **dtor**: A **destructor** is a function executed automatically when an object's life comes to an end. (goes out of scope, program ends, or is deleted dynamically)
- **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 (memory, file, connection) allocated by the object.

class Array
{
public:
   …
   ~Array();
   …
};
	Array::**~Array()** {
	   delete[] m_array;
	}

Destructors

- **dtor**: A **destructor** is a function executed automatically when an object's life comes to an end. (goes out of scope, program ends, or is deleted dynamically)
- **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
  - Do **not** call it (explicitly)

Purpose: to free any resource (memory, file, connection) allocated by the object.

class Array
{
public:
…
~Array();
…
};
	Array::**~Array()** {
	delete[] m_array;
	}
	void main() {
	Array array**(20)**;
	array.insertElement(10, 0);
	}

When are ctors and dtors invoked?

- **Static** variables (local, global)

When are ctors and dtors invoked?

- **Static** variables (local, global)

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

When are ctors and dtors invoked?

- **Static** variables (local, global)

codes inserted by the C++ compiler

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

When are ctors and dtors invoked?

- **Static** variables (local, global)   codes inserted by the C++ compiler

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

  - ctor of a global variable is invoked before main() gets started

When are ctors and dtors invoked?

- **Static** variables (local, global)

codes inserted by the C++ compiler

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

  - ctor of a global variable is invoked before main() gets started
    dtor of a global variable is invoked when the program exits

When are ctors and dtors invoked?

- **Static** variables (local, global) codes inserted by the C++ compiler

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

What happens if the dtor was not defined?

- ctor of a global variable is invoked before main() gets started
  dtor of a global variable is invoked when the program exits

When are ctors and dtors invoked?

- **Static** variables (local, global)

codes inserted by the C++ compiler

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

What happens if the dtor was not defined?

  - ctor of a global variable is invoked before main() gets started
    dtor of a global variable is invoked when the program exits

- **Dynamic** variables

When are ctors and dtors invoked?

- **Static** variables (local, global)

codes inserted by the C++ compiler

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

What happens if the dtor was not defined?

  - ctor of a global variable is invoked before main() gets started
    dtor of a global variable is 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
}
```

When are ctors and dtors invoked?

- **Static** variables (local, global) — codes inserted by the C++ compiler

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

What happens if the dtor was not defined?

  - ctor of a global variable is invoked before main() gets started
    dtor of a global variable is 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 happens if you use malloc() to get the required memory for an object?

When are ctors and dtors invoked?

✧ **Static** variables (local, global) codes inserted by the C++ compiler

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

What happens if the dtor was not defined?

- ctor of a global variable is invoked before main() gets started
  dtor of a global variable is 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 happens if you use malloc() to get the required memory for an object?

What happens if we did not call delete?

Advantages Achieved by OOP

Automatic initialization

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

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;
}
```

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;
    }
}
```

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 encapsulation

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

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 encapsulation

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

Now, an array is no longer a fixed chunk of data storages. It serves data for the client codes reliably. It might even adjust its size dynamically.

Multiple Constructors

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

Multiple Constructors

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

```
class Name {
public:
   Name();
   Name(char *firstName, char *lastName);
```

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;
};
```

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() {
    m_firstName = 0;
    m_lastName = 0;
}
```

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() {
  m_firstName = 0;
  m_lastName = 0;
}
```

This ctor has special name: "**default constructor**".

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() {
    m_firstName = 0;
    m_lastName = 0;
}
```

VC, 『預設建構函式』

This ctor has special name: “**default constructor**”.

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;
};
```

VC, 『預設建構函式』

This ctor has special name: "**default constructor**".

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

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

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);
}
```

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;
}
```

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 << ' ';
}
```

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();
}
```

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();
}
```

Constructors and Arrays

- If you want to define an array of objects, your class **must have a default ctor**.

Constructors and Arrays

- If you want to define an array of objects, your class **must have a default ctor**.

```
class Name {
public:

   ~Name();
   void setName(char *firstName, char *lastName);
private:
   char *m_firstName;
   char *m_lastName;
};
void main() {
   Name names[100];
   names[12].setName("Mark", "Anderson");
}
```

Compiler accepts.

Constructors and Arrays

- If you want to define an array of objects, your class **must have a default ctor**.

```
class Name {
public:
    Name(char *firstName, char *lastName);
    ~Name();
    void setName(char *firstName, char *lastName);
private:
    char *m_firstName;
    char *m_lastName;
};
void main() {
    Name names[100];
    names[12].setName("Mark", "Anderson");
}
```

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

Constructors and Arrays

- If you want to define an array of objects, your class **must have a default ctor**.

Name() is the **default constructor**

```
class Name {
public:
   Name(char *firstName, char *lastName);
   ~Name();
   void setName(char *firstName, char *lastName);
private:
   char *m_firstName;
   char *m_lastName;
};
void main() {
   Name names[100];
   names[12].setName("Mark", "Anderson");
}
```

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

Constructors and Arrays

- If you want to define an array of objects, your class **must have a default ctor**.

Name() is the **default constructor**

```
class Name {
public:
   Name(char *firstName, char *lastName);
   ~Name();
   void setName(char *firstName, char *lastName);
private:
   char *m_firstName;
   char *m_lastName;
};
void main() {
   Name names[100];
   names[12].setName("Mark", "Anderson");
}
```

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

---

```
Name names[2] = {Name("Mark", "Anderson"), Name("Ron","Dale")}; // OK
```

Constructors and Arrays

- If you want to define an array of objects, your class **must have a default ctor**.

  C++ compiler does not give you a 'default' if you specify any ctor.

Name() is the **default constructor**

```
class Name {
public:
   Name(char *firstName, char *lastName);
   ~Name();
   void setName(char *firstName, char *lastName);
private:
   char *m_firstName;
   char *m_lastName;
};
void main() {
   Name names[100];
   names[12].setName("Mark", "Anderson");
}
```

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

---

```
Name names[2] = {Name("Mark", "Anderson"), Name("Ron","Dale")}; // OK
```

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);
private:
    char *m_firstName;
    char *m_lastName;
};
```

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);
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);
private:
    char *m_firstName;
    char *m_lastName;
};
```

Constructors with Default Arguments

- **Consider this class with two constructors**

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;
};
```

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;
}
```

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;
}
```

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;
};
```

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

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

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

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;
};
```

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

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

```
Output:
0.0
100.0
```

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

Ctor with Default Arguments (cont'd)

- The class is rewritten in the following way:

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

Ctor with Default Arguments (cont'd)

- The class is rewritten in the following way:

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

This ctor is exactly the same as before

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

Ctor with Default Arguments (cont'd)

- The class is rewritten in the following way:

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

This ctor is exactly the same as before

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

Ctor with Default Arguments (cont'd)

- The class is rewritten in the following way:

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

This ctor is exactly the same as before

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

- We can now declare an array of Account.

Ctor with Default Arguments (cont'd)

- The class is rewritten in the following way:

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

This ctor is exactly the same as before

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

- We can now declare an array of Account.

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

Ctor with Default Arguments (cont'd)

- The class is rewritten in the following way:

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

This ctor is exactly the same as before

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

- We can now declare an array of Account.

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

This works fine with a fake default ctor.

Initialization Lists

- Consider the following class

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();
private:
   char *m_name;
   Breed m_breed;
   int m_age;
};
```

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();
private:
   char *m_name;
   Breed m_breed;
   int m_age;
};
```

- The ctor might look like:

```
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;
}
```

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();
private:
    char *m_name;
    Breed m_breed;
    int m_age;
};
```

- The ctor might look like:

```
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 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);
}
```

Constant Data Member **Initialization**

- Consider the class:

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

Constant Data Member **Initialization**

- Consider the class:
- The breed of the dog will not change

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

Constant Data Member **Initialization**

- Consider the class:

- The breed of the dog will not change

  Let us make it a **constant variable** in the class declaration.

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

Constant Data Member **Initialization**

- Consider the class:

- The breed of the dog will not change

  Let us make it a **constant variable** in the class declaration.

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

Constant Data Member **Initialization**

- Consider the class:

- The breed of the dog will not change

  Let us make it a **constant variable** in the class declaration.

- Constant variables **MUST** be initialized in the initialization list

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

Constant Data Member **Initialization**

- Consider the class:

- The breed of the dog will not change

  Let us make it a **constant variable** in the class declaration.

- Constant variables **MUST** be initialized in the initialization list

```
Dog::Dog(): m_breed(undefined) {…}
```

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

Constant Data Member **Initialization**

- Consider the class:

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

- The breed of the dog will not change

  Let us make it a **constant variable** in the class declaration.

- Constant variables **MUST** be initialized in the initialization list

```
Dog::Dog(): m_breed(undefined) {…}
```

- Other preferred usages of **const**

Constant Data Member **Initialization**

- Consider the class:

- The breed of the dog will not change

  Let us make it a **constant variable** in the class declaration.

- Constant variables **MUST** be initialized in the initialization list

```
Dog::Dog(): m_breed(undefined) {…}
```

- Other preferred usages of **const**

```
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 Data Member **Initialization**

✧ Consider the class:

```
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;
};
```

✧ The breed of the dog will not change

Let us make it a **constant variable** in the class declaration.

✧ Constant variables **MUST** be initialized in the initialization list

```
Dog::Dog(): m_breed(undefined) {…}
```

✧ Other preferred usages of **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);
}
```

Initialization List (cont'd)

- ✧ There are several cases where initialization list **MUST** be used

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **Non-default component object constructor**

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **Non-default component object constructor**
- Coding style: use initialization list as much as possible

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **Non-default component object constructor**
- Coding style: use initialization list as much as possible
  - initialization list is inevitable in many cases

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **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.

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **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:

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **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.

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **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.

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **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);
}
```

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **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);
}
```

first (arrow pointing to m_name(new char[strlen(name)+1]))

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **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);
}
```

first (→ m_name) **second** (→ m_breed)

Initialization List (cont'd)

- There are several cases where initialization list **MUST** be used
  1. **Constant data member**
  2. **Reference data member**
  3. **Non-default parent class constructor**
  4. **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 (m_age(age)) **first** (m_name(...)) **second** (m_breed(breed))

Ctor of Intrinsic Data Type

- int, long, float, double, char

Ctor of Intrinsic Data Type

- int, long, float, double, char

    Initialize it with constructor syntax:

    ```
    int x(10);
    ```

Ctor of Intrinsic Data Type

- int, long, float, double, char

  Initialize it with constructor syntax:

  ```
  int x(10);
  ```

- Construction of a temporary variable

Ctor of Intrinsic Data Type

- int, long, float, double, char

  Initialize it with constructor syntax:

```
int x(10);
```

- Construction of a temporary variable

```
int x;
x = int(10.3); // assignment from a temporary integer
               // which is initialized with 10.3
```

Ctor of Intrinsic Data Type

- int, long, float, double, char

  Initialize it with constructor syntax:

  ```
  int x(10);
  ```

- Construction of a temporary variable

  ```
  int x;
  x = int(10.3); // assignment from a temporary integer
                 // which is initialized with 10.3
  ```

- Some people think that this is a kind of coercion like

  ```
  x =  (int) 10.3;
  ```

  Actually the above two are not the same mechanism in C++. Each invoke different procedures, although achieving similar functions.