The Big Three

C++ Object Oriented Programming

Pei-yih Ting
NTOUCS

Contents

- Destructor
- Copy constructor
- Assignment operator
- Move constructor (C++11)
- Move assignment operator (C++11)
- The managed pointer

Introduction

- When the class has the functionality of **resource management**, it is very likely that the destructor (dtor), the copy constructor (copy ctor), and the assignment operator occur together.

called the BIG 3

Introduction

- When the class has the functionality of **resource management**, it is very likely that the destructor (dtor), the copy constructor (copy ctor), and the assignment operator occur together.
- Resource management: ex.

called the BIG 3

```
class Account {
public:



private:
   char *m_name;
   char *m_phone;
   char *m_address;
};
```

Introduction

- When the class has the functionality of **resource management**, it is very likely that the destructor (dtor), the copy constructor (copy ctor), and the assignment operator occur together.
- Resource management: ex.

called the BIG 3

```
class Account {
public:
    Account(const char *name, const char *phone, const char *address);


private:
    char *m_name;
    char *m_phone;
    char *m_address;
};
Account::Account(const char *name, const char *phone, const char *address) {
    m_name = new char[strlen(name)+1]; strcpy(m_name, name);
    m_phone = new char[strlen(phone)+1]; strcpy(m_phone, phone);
    m_address = new char[strlen(address)+1]; strcpy(m_address, address);
}
```

remote ownership

Introduction

- When the class has the functionality of **resource management**, it is very likely that the destructor (dtor), the copy constructor (copy ctor), and the assignment operator occur together.
- Resource management: ex.

called the BIG 3

```
class Account {
public:
    Account(const char *name, const char *phone, const char *address);
    ~Account();
    ....
private:
    char *m_name;
    char *m_phone;
    char *m_address;
};

Account::Account(const char *name, const char *phone, const char *address) {
    m_name = new char[strlen(name)+1]; strcpy(m_name, name);
    m_phone = new char[strlen(phone)+1]; strcpy(m_phone, phone);
    m_address = new char[strlen(address)+1]; strcpy(m_address, address);
}

Account::~Account() {
    delete[] m_name; delete[] m_phone; delete[] m_address;
}
```

remote ownership

dtor

Copy Constructor (copy ctor)

- What is a copy constructor? **X(X&)**

Copy Constructor (copy ctor)

- What is a copy constructor? **X(X&)**

  Account(Account &src); and Account(const Account &src);

Copy Constructor (copy ctor)

- What is a copy constructor? **X(X&)**

  Account(Account &src); and Account(const Account &src);

- When is the copy constructor invoked?

Copy Constructor (copy ctor)

- What is a copy constructor? **X(X&)**

  Account(Account &src); and Account(const Account &src);

- When is the copy constructor invoked? **object being copied**

Copy Constructor (copy ctor)

- What is a copy constructor? **X(X&)**

  Account(Account &src); and Account(const Account &src);

- When is the copy constructor invoked? **object being copied**

  Case 1: Account customer1("Sean Pan",

  "123-4567890", "1234 Sunset Blvd.");

  Account customer2(customer1);

  Account customer3 = customer1;

Copy Constructor (copy ctor)

- What is a copy constructor? **X(X&)**

  Account(Account &src); and Account(const Account &src);

- When is the copy constructor invoked? **object being copied**

  Case 1: Account customer1("Sean Pan",
  "123-4567890", "1234 Sunset Blvd.");

  Account customer2(customer1);

  Account customer3 = customer1;

  Case 2: void fun1(Account customer) {
  …
  }

Copy Constructor (copy ctor)

- What is a copy constructor? **X(X&)**

  Account(Account &src); and Account(const Account &src);

- When is the copy constructor invoked? **object being copied**

Case 1:
```
Account customer1("Sean Pan",
                  "123-4567890", "1234 Sunset Blvd.");
Account customer2(customer1);
Account customer3 = customer1;
```

Case 2:
```
void fun1(Account customer) {
    …
}
```

Case 3:
```
Account fun2() {
    Account x;
    …
    return x;
}
```

Copy Constructor

- If copy ctor is not defined, **compiler will synthesize one** for you.

Copy Constructor

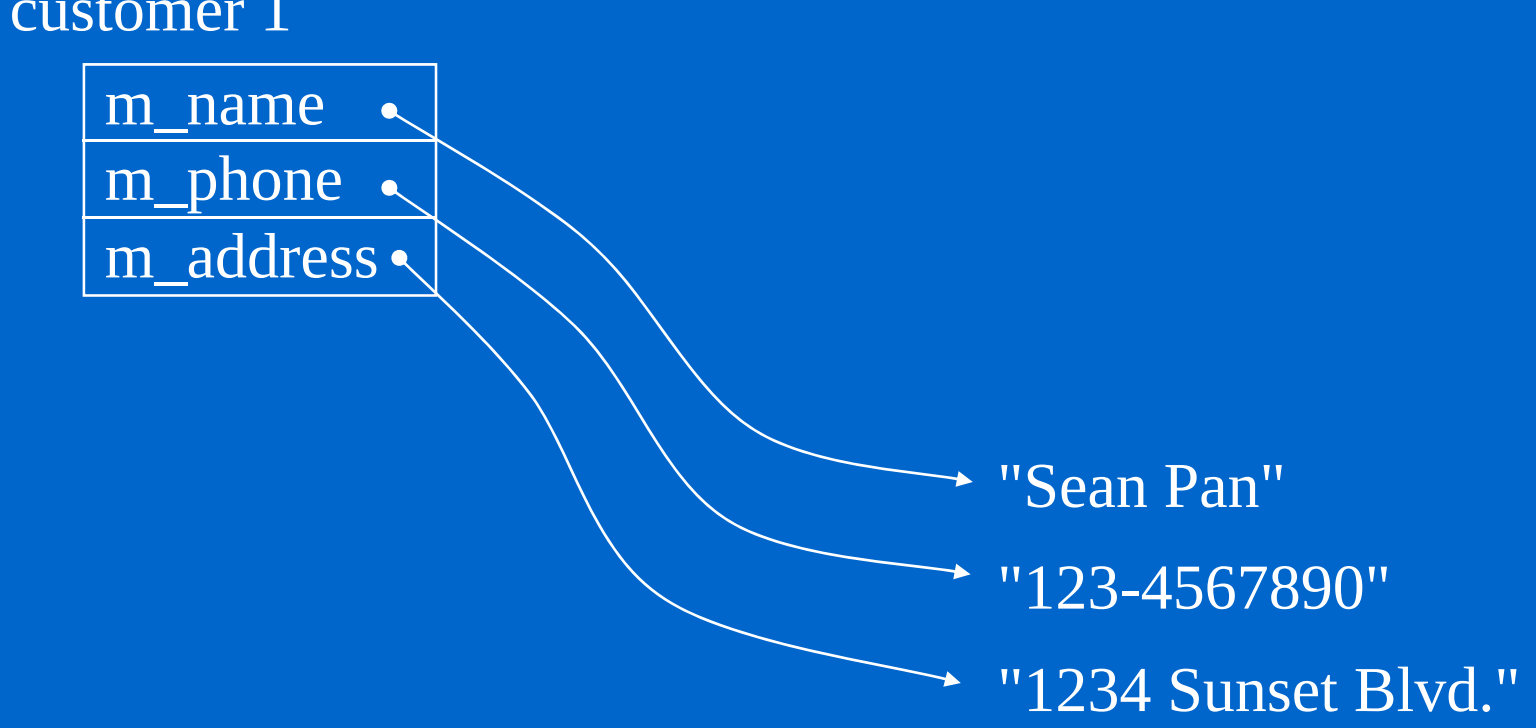
- If copy ctor is not defined, **compiler will synthesize one** for you.
- This synthesized copy ctor **copies all the bits** of the object.

Copy Constructor

- If copy ctor is not defined, **compiler will synthesize one** for you.
- This synthesized copy ctor **copies all the bits** of the object.
- For many cases this implementation is good but for a class which allocates memory or handles other resources itself, this usually leads to errors. A trap to ***dangling reference***.

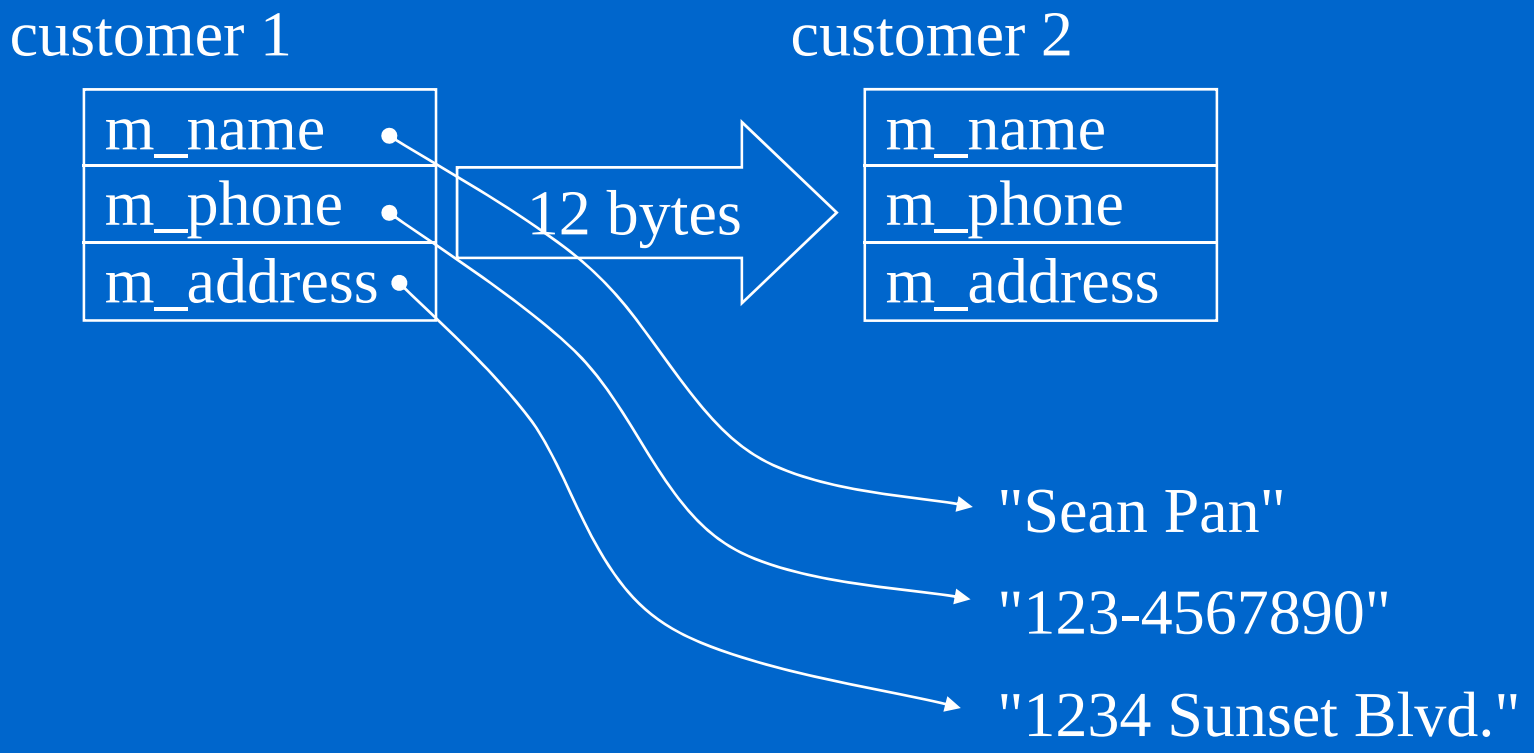
Copy Constructor

- If copy ctor is not defined, **compiler will synthesize one** for you.
- This synthesized copy ctor **copies all the bits** of the object.
- For many cases this implementation is good but for a class which allocates memory or handles other resources itself, this usually leads to errors. A trap to ***dangling reference***.



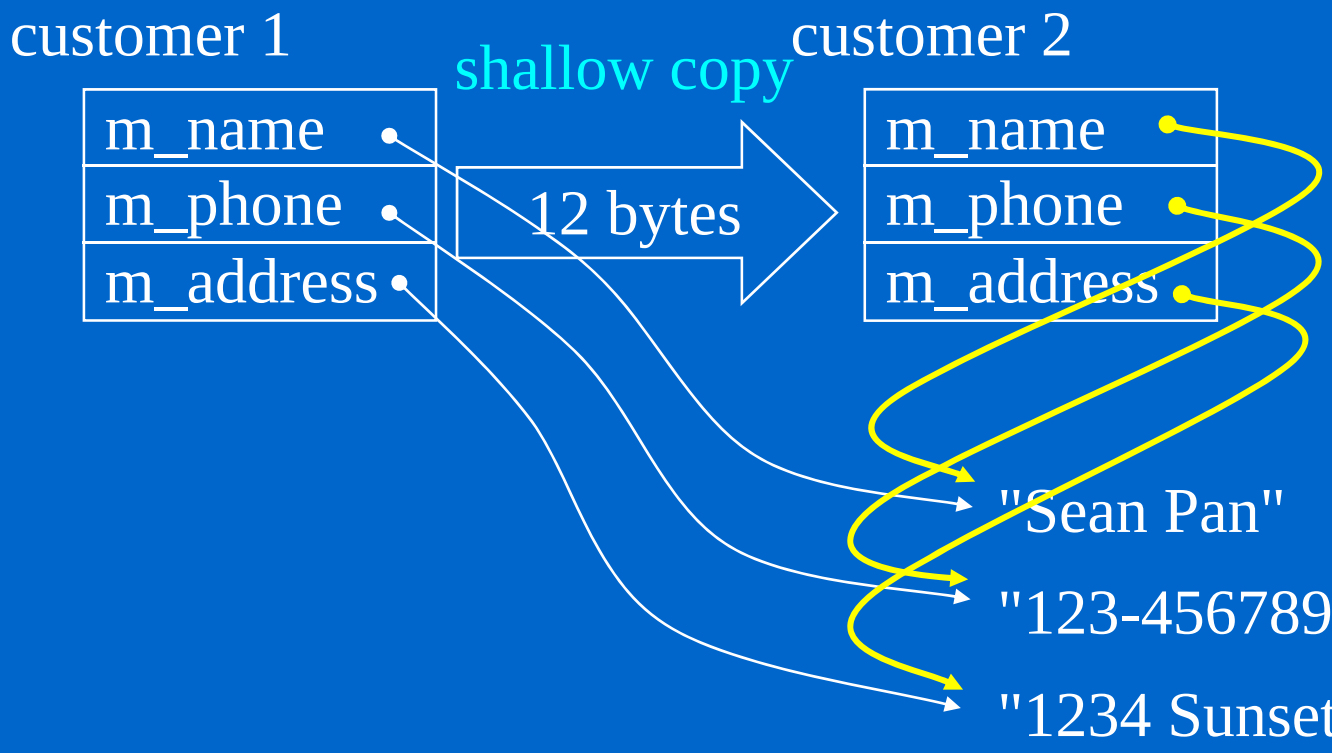
Copy Constructor

- If copy ctor is not defined, **compiler will synthesize one** for you.
- This synthesized copy ctor **copies all the bits** of the object.
- For many cases this implementation is good but for a class which allocates memory or handles other resources itself, this usually leads to errors. A trap to ***dangling reference***.



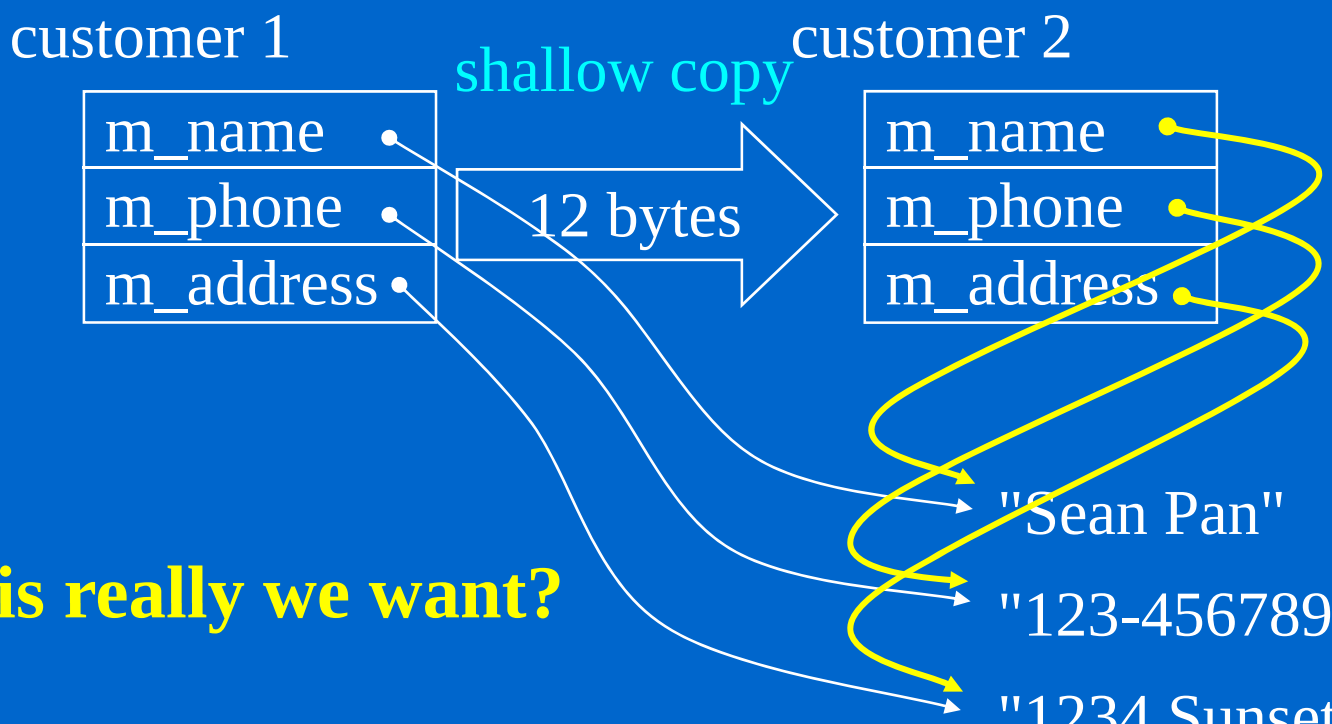
Copy Constructor

- If copy ctor is not defined, **compiler will synthesize one** for you.
- This synthesized copy ctor **copies all the bits** of the object.
- For many cases this implementation is good but for a class which allocates memory or handles other resources itself, this usually leads to errors. A trap to ***dangling reference***.



Copy Constructor

- If copy ctor is not defined, **compiler will synthesize one** for you.
- This synthesized copy ctor **copies all the bits** of the object.
- For many cases this implementation is good but for a class which allocates memory or handles other resources itself, this usually leads to errors. A trap to ***dangling reference***.



Is this really we want?

Copy Constructor

- If copy ctor is not defined, **compiler will synthesize one** for you.
- This synthesized copy ctor **copies all the bits** of the object.
- For many cases this implementation is good but for a class which allocates memory or handles other resources itself, this usually leads to errors. A trap to ***dangling reference***.

Is this really we want?

Dangling Reference

- Consider the following codes

Dangling Reference

- Consider the following codes

```
void main() {
   Account customer("Sean Pan", "123-4567890", "1234 sunset Blvd.");
      …
   fun(customer);
      …
   customer.display(); // show all the customer information
}
```

Dangling Reference

- Consider the following codes

```
void main() {
   Account customer("Sean Pan", "123-4567890", "1234 sunset Blvd.");
      …
   fun(customer);
      …
   customer.display(); // show all the customer information
}

void fun(Account customerLocal) {
         ….
} // the dtor would deallocate the memory belongs to customerLocal
  //  however, these memory blocks are the same as those of customer
```

Dangling Reference

- Consider the following codes

```
void main() {
   Account customer("Sean Pan", "123-4567890", "1234 sunset Blvd.");
      …
   fun(customer);
      …
   customer.display(); // show all the customer information
}

void fun(Account customerLocal) {
         ….
} // the dtor would deallocate the memory belongs to customerLocal
  //  however, these memory blocks are the same as those of customer
```

- The statement **fun(customer)** would cause ***dangling reference*** and the statement **customer.display()** would access memory blocks previously belonged to this customer object and display some strange contents.

Unexpected Release

- Sometimes, the resource might be unexpectedly released, ex.

Unexpected Release

- Sometimes, the resource might be unexpectedly released, ex.

```
void main() {
    ifstream infile("input.dat");
    …
    readFile(infile);
    …
}
```

Unexpected Release

- Sometimes, the resource might be unexpectedly released, ex.

```
void main() {
    ifstream infile("input.dat");
    …
    readFile(infile);
    …
}
```

```
void readFile(ifstream is) {
    …
}
```

Unexpected Release

- Sometimes, the resource might be unexpectedly released, ex.

```
void main() {
    ifstream infile("input.dat");
    …
    readFile(infile);
    …
}
```

```
void readFile(ifstream is) {
    …
}
```

VC 2010 does not allow this

Unexpected Release

- Sometimes, the resource might be unexpectedly released, ex.

```
void main() {
    ifstream infile("input.dat");
    …
    readFile(infile);
    …
}
```

```
void readFile(ifstream is) {
    …
}
```

VC 2010 does not allow this

- This is a complex problem. The program will have runtime error. Why does the error occurs? You won't be able to correct this by supplying a copy constructor for ifstream because it is a library class. The only thing you can easily do is **not invoking the copy ctor** by passing the parameter with **reference**.

Example Copy Constructor

```
Account::Account(const Account &src) {
    m_name = new char[strlen(src.m_name)+1];
    strcpy(m_name, src.m_name);
    m_phone = new char[strlen(src.m_phone)+1];
    strcpy(m_phone, src.m_phone);
    m_address = new char[strlen(src.m_address)+1];
    strcpy(m_address, src.m_address);
}
```

Example Copy Constructor

```
Account::Account(const Account &src) {
    m_name = new char[strlen(src.m_name)+1];
    strcpy(m_name, src.m_name);
    m_phone = new char[strlen(src.m_phone)+1];
    strcpy(m_phone, src.m_phone);
    m_address = new char[strlen(src.m_address)+1];
    strcpy(m_address, src.m_address);
}
```

- Copy ctor is a kind of ctor. You should use **initialization list** whenever possible. Especially, you should invoke the **base class copy ctor** if it is a derived class. You should invoke **the component class copy ctor** if it contains a member object.

Example Copy Constructor

```
Account::Account(const Account &src) {
    m_name = new char[strlen(src.m_name)+1];
    strcpy(m_name, src.m_name);
    m_phone = new char[strlen(src.m_phone)+1];
    strcpy(m_phone, src.m_phone);
    m_address = new char[strlen(src.m_address)+1];
    strcpy(m_address, src.m_address);
}
```

- Copy ctor is a kind of ctor. You should use **initialization list** whenever possible. Especially, you should invoke the **base class copy ctor** if it is a derived class. You should invoke **the component class copy ctor** if it contains a member object.
- In a copy ctor, you are initiating an object from another object. The memory space for the object is allocated by the system.

Example Copy Constructor

```
Account::Account(const Account &src) {
    m_name = new char[strlen(src.m_name)+1];
    strcpy(m_name, src.m_name);
    m_phone = new char[strlen(src.m_phone)+1];
    strcpy(m_phone, src.m_phone);
    m_address = new char[strlen(src.m_address)+1];
    strcpy(m_address, src.m_address);
}
```

- Copy ctor is a kind of ctor. You should use **initialization list** whenever possible. Especially, you should invoke the **base class copy ctor** if it is a derived class. You should invoke **the component class copy ctor** if it contains a member object.
- In a copy ctor, you are initiating an object from another object. The memory space for the object is allocated by the system.
- If you want to forbid public usage of **call-by-value** semantics of an object, you can declare a **private copy ctor** for that class.

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
    …

    …
private:
    Component m_obj;
};
```

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
    …
    Derived(const Derived &src);
    …
private:
    Component m_obj;
};
```

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
    …
    Derived(const Derived &src);
    …
private:
    Component m_obj;
};

Derived::Derived(const Derived &src)
        : Base(src), m_obj(src.m_obj) {
    …
}
```

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
    …
    Derived(const Derived &src);
    …
private:
    Component m_obj;
};

Derived::Derived(const Derived &src)
        : Base(src), m_obj(src.m_obj) {
    …
}
```

You have to chain manually.

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
   …
   Derived(const Derived &src);
   …
private:
   Component m_obj;
};

Derived::Derived(const Derived &src)
      : Base(src), m_obj(src.m_obj) {
   …
}
```

You have to chain manually.

Compiler supplied copy ctor also chains correctly.

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
   …
   Derived(const Derived &src);
   …
private:
   Component m_obj;
};

Derived::Derived(const Derived &src)
      : Base(src), m_obj(src.m_obj) {
   …
}
```

Note:

```
Derived::Derived(const Derived &src)
      : m_obj(src.m_obj)  {
   …
}
```

You have to chain manually.

Compiler supplied copy ctor also chains correctly.

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
   …
   Derived(const Derived &src);
   …
private:
   Component m_obj;
};

Derived::Derived(const Derived &src)
      : Base(src), m_obj(src.m_obj) {
   …
}
```

Compiler adds **Base()** invocation automatically

Note:

```
Derived::Derived(const Derived &src)
      : m_obj(src.m_obj)  {
   …
}
```

You have to chain manually.

Compiler supplied copy ctor also chains correctly.

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
   …
   Derived(const Derived &src);
   …
private:
   Component m_obj;
};

Derived::Derived(const Derived &src)
      : Base(src), m_obj(src.m_obj) {
   …
}
```

Compiler adds **Base()** invocation automatically

Note:

```
Derived::Derived(const Derived &src)
      : m_obj(src.m_obj)  {
   …
}

Derived::Derived(const Derived &src) {
   …
}
```

You have to chain manually.

Compiler supplied copy ctor also chains correctly.

Member Object and Base Class

- Copy constructor is a constructor, member objects and base class must be initialized through initialization list
- For example:

```
class Derived: public Base {
public:
   …
   Derived(const Derived &src);
   …
private:
   Component m_obj;
};
```

```
Derived::Derived(const Derived &src)
      : Base(src), m_obj(src.m_obj) {
   …
}
```

Compiler adds **Base()** invocation automatically

Note:

```
Derived::Derived(const Derived &src)
      : m_obj(src.m_obj)  {
   …
}
```

```
Derived::Derived(const Derived &src) {
   …
}
```

Compiler adds **Base(), m_obj()** invocations automatically

You have to chain manually.

Compiler supplied copy ctor also chains correctly.

Assignment Operator

- When/where is the assignment operator invoked?

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
```

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
customer2 = customer1;
```

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
customer2 = customer1;
customer2.operator=(customer1);
```

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
customer2 = customer1;
customer2.operator=(customer1);
customer3 = customer2 = customer1;
```

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
customer2 = customer1;
customer2.operator=(customer1);
customer3 = customer2 = customer1;
```

- Note: Account customer2 = customer1;

  does **not** invoke the assignment operator

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
customer2 = customer1;
customer2.operator=(customer1);
customer3 = customer2 = customer1;
```

- Note: Account customer2 = customer1;

  does **not** invoke the assignment operator

- What is its prototypes?

```
Account &operator=(const Account &rhs);
```

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
customer2 = customer1;
customer2.operator=(customer1);
customer3 = customer2 = customer1;
```

- Note: Account customer2 = customer1;

  does **not** invoke the assignment operator

- What is its prototypes?

```
Account &operator=(const Account &rhs);
```

No extra copy ctor invoked

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
customer2 = customer1;
customer2.operator=(customer1);
customer3 = customer2 = customer1;
```

- Note: Account customer2 = customer1;

  does **not** invoke the assignment operator

- What is its prototypes?

```
Account &operator=(const Account &rhs);
```

No extra copy ctor invoked

Designed for continuously assignment

```
customer3.operator=(customer2.operator=(customer1));
```

Assignment Operator

- When/where is the assignment operator invoked?

```
Account customer1("abc", "1234", "ABC street");
Account customer2, customer3;  // assume default ctor defined
customer2 = customer1;
customer2.operator=(customer1);
customer3 = customer2 = customer1;
```

- Note: Account customer2 = customer1;

  does **not** invoke the assignment operator

- What is its prototypes?

```
Account &operator=(const Account &rhs);
```

No extra copy ctor invoked

Designed for continuously assignment

```
customer3.operator=(customer2.operator=(customer1));
```

Note: this does not contradict the rule that reference does not bind to temporary object

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

```
Account &Account::operator=(const Account &rhs)
{




}
```

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

```
Account &Account::operator=(const Account &rhs)
{
① if (&rhs == this) return *this;
}
```

Detecting self assignments

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

```
Account &Account::operator=(const Account &rhs)
{
①   if (&rhs == this) return *this;
②   delete[] m_name;  delete[] m_phone;  delete[] m_address;


}
```

Detecting self assignments

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

```
Account &Account::operator=(const Account &rhs)
{
①   if (&rhs == this) return *this;
②   delete[] m_name;  delete[] m_phone;  delete[] m_address;
③   m_name = new char[strlen(rhs.m_name)+1];
    m_phone = new char[strlen(rhs.m_phone)+1];
    m_address = new char[strlen(rhs.m_address)+1];

}
```

Detecting self assignments

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

```
Account &Account::operator=(const Account &rhs)
{
①   if (&rhs == this) return *this;
②   delete[] m_name;  delete[] m_phone;  delete[] m_address;
③   m_name = new char[strlen(rhs.m_name)+1];
    m_phone = new char[strlen(rhs.m_phone)+1];
    m_address = new char[strlen(rhs.m_address)+1];
④   strcpy(m_name, rhs.m_name);
    strcpy(m_phone, rhs.m_phone);
    strcpy(m_address, rhs.m_address);

}
```

Detecting self assignments

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

```
Account &Account::operator=(const Account &rhs)
{
①   if (&rhs == this) return *this;
②   delete[] m_name;  delete[] m_phone;  delete[] m_address;
③   m_name = new char[strlen(rhs.m_name)+1];
    m_phone = new char[strlen(rhs.m_phone)+1];
    m_address = new char[strlen(rhs.m_address)+1];
④   strcpy(m_name, rhs.m_name);
    strcpy(m_phone, rhs.m_phone);
    strcpy(m_address, rhs.m_address);
⑤   // invoke the base class assignment operator


}
```

Detecting self assignments

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

```
Account &Account::operator=(const Account &rhs)
{
①   if (&rhs == this) return *this;
②   delete[] m_name;  delete[] m_phone;  delete[] m_address;
③   m_name = new char[strlen(rhs.m_name)+1];
    m_phone = new char[strlen(rhs.m_phone)+1];
    m_address = new char[strlen(rhs.m_address)+1];
④   strcpy(m_name, rhs.m_name);
    strcpy(m_phone, rhs.m_phone);
    strcpy(m_address, rhs.m_address);
⑤   // invoke the base class assignment operator
⑥   // invoke the component object assignment operator

}
```

Detecting self assignments

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operator** occur together.
- There are **seven** important things to do in an assignment operator

```
Account &Account::operator=(const Account &rhs)
{
①   if (&rhs == this) return *this;
②   delete[] m_name;  delete[] m_phone;  delete[] m_address;
③   m_name = new char[strlen(rhs.m_name)+1];
    m_phone = new char[strlen(rhs.m_phone)+1];
    m_address = new char[strlen(rhs.m_address)+1];
④   strcpy(m_name, rhs.m_name);
    strcpy(m_phone, rhs.m_phone);
    strcpy(m_address, rhs.m_address);
⑤   // invoke the base class assignment operator
⑥   // invoke the component object assignment operator
⑦   return *this;
}
```

Detecting self assignments

Assignment Operator

- You can declare the assignment operator in the private section to forbid public usage of the assignment semantics.

Assignment Operator

- You can declare the assignment operator in the private section to forbid public usage of the assignment semantics.
- If there is a reference variable or a const variable defined in the class, there is no way to define the assignment operator.

Assignment Operator

- You can declare the assignment operator in the private section to forbid public usage of the assignment semantics.
- If there is a reference variable or a const variable defined in the class, there is no way to define the assignment operator.

- Usually, the assignment operator repeats the codes both in the copy ctor and the dtor. It is common to prepare common functions to be called in assignment operator, copy ctor and the dtor.

Assignment Operator

- You can declare the assignment operator in the private section to forbid public usage of the assignment semantics.
- If there is a reference variable or a const variable defined in the class, there is no way to define the assignment operator.
- Usually, the assignment operator repeats the codes both in the copy ctor and the dtor. It is common to prepare common functions to be called in assignment operator, copy ctor and the dtor.
- The Big 3 are never inherited because based class functions are not sufficient to initialize, copy, or destroy a derived instance.

Assignment Operator

- You can declare the assignment operator in the private section to forbid public usage of the assignment semantics.
- If there is a reference variable or a const variable defined in the class, there is no way to define the assignment operator.
- Usually, the assignment operator repeats the codes both in the copy ctor and the dtor. It is common to prepare common functions to be called in assignment operator, copy ctor and the dtor.
- The Big 3 are never inherited because based class functions are not sufficient to initialize, copy, or destroy a derived instance.
- Three make a team. Do not forget any one of them.

Managed Pointer

- template class **auto_ptr<T>**: **#include <memory>**

Managed Pointer

- template class **auto_ptr<T>**: **#include <memory>**

  auto_ptr<Fred> acts like a **Fred*** except that **it owns the referent** (the Fred object)

Managed Pointer

- template class **auto_ptr<T>**: **#include <memory>**

  auto_ptr<Fred> acts like a **Fred*** except that **it owns the referent** (the Fred object)

  1. Declare a managed pointer with NULL value
     auto_ptr<Fred> ptr;

Managed Pointer

- template class **auto_ptr<T>**: **#include <memory>**

  auto_ptr<Fred> acts like a **Fred*** except that **it owns the referent** (the Fred object)

  1. Declare a managed pointer with NULL value
     auto_ptr<Fred> ptr;
  2. Invoke the assignment operator later
     ptr = auto_ptr<Fred>(new Fred());

ptr now owns this new Fred object

Managed Pointer

- template class **auto_ptr<T>**: **#include <memory>**

  auto_ptr<Fred> acts like a **Fred*** except that **it owns the referent** (the Fred object)

  1. Declare a managed pointer with NULL value
     auto_ptr<Fred> ptr;
  2. Invoke the assignment operator later
     ptr = auto_ptr<Fred>(new Fred());
  3. Construct a managed pointer with a pointer
     auto_ptr<Fred> ptr(new Fred()); or
     auto_ptr<Fred> ptr = new Fred();

ptr now owns this new Fred object

Managed Pointer

- template class **auto_ptr<T>**: **#include <memory>**

  auto_ptr<Fred> acts like a **Fred*** except that **it owns the referent** (the Fred object)

  1. Declare a managed pointer with NULL value
     auto_ptr<Fred> ptr;
  2. Invoke the assignment operator later
     ptr = auto_ptr<Fred>(new Fred());
  3. Construct a managed pointer with a pointer
     auto_ptr<Fred> ptr(new Fred()); or
     auto_ptr<Fred> ptr = new Fred();
  4. Can be used anywhere like a Fred* pointer
     ptr–>services(); or **(*ptr).services();**

ptr now owns this new Fred object

Managed Pointer

- template class **auto_ptr<T>**: **#include <memory>**

  auto_ptr<Fred> acts like a **Fred*** except that **it owns the referent** (the Fred object)

  1. Declare a managed pointer with NULL value
     auto_ptr<Fred> ptr;
  2. Invoke the assignment operator later
     ptr = auto_ptr<Fred>(new Fred());
  3. Construct a managed pointer with a pointer
     auto_ptr<Fred> ptr(new Fred()); or
     auto_ptr<Fred> ptr = new Fred();
  4. Can be used anywhere like a Fred* pointer
     ptr–>services(); or **(*ptr).services();**
  5. Retrieve the raw Fred pointer
     Fred *ptrRaw = ptr.get();

ptr now owns this new Fred object

Managed Pointer (cont'd)

6. Copy ctor is implemented with **ownership transfer** (surprise!!)

auto_ptr<Fred> newPtr = ptr; or

auto_ptr<Fred> newPtr(ptr);

newPtr now owns the Fred object originally owned by ptr, ptr will point to the same object afterwards but will not own it anymore.

Managed Pointer (cont'd)

6. Copy ctor is implemented with **ownership transfer** (surprise!!)

   auto_ptr<Fred> newPtr = ptr; or

   auto_ptr<Fred> newPtr(ptr);

   newPtr now owns the Fred object originally owned by ptr, ptr will point to the same object afterwards but will not own it anymore.

7. When this object goes out of scope, its dtor will delete the owned Fred object.

Managed Pointer (cont'd)

6. Copy ctor is implemented with **ownership transfer** (surprise!!)
   auto_ptr<Fred> newPtr = ptr; or
   auto_ptr<Fred> newPtr(ptr);

   newPtr now owns the Fred object originally owned by ptr, ptr will point to the same object afterwards but will not own it anymore.

7. When this object goes out of scope, its dtor will delete the owned Fred object.

8. What about an explicit delete?
   delete ptr; // syntax **error**, do not new an auto_ptr, do not keep the raw Fred pointer, pass by reference to a function

Managed Pointer (cont'd)

6. Copy ctor is implemented with **ownership transfer** (surprise!!)
   auto_ptr<Fred> newPtr = ptr; or
   auto_ptr<Fred> newPtr(ptr);

   newPtr now owns the Fred object originally owned by ptr, ptr will point to the same object afterwards but will not own it anymore.

7. When this object goes out of scope, its dtor will delete the owned Fred object.

8. What about an explicit delete?
   delete ptr; // syntax **error**, do not new an auto_ptr, do not keep the raw Fred pointer, pass by reference to a function

9. If you copy the managed pointer from another managed pointer without ownership to the real object, the new managed pointer does not have ownership to the real object. If you construct a new managed pointer with a raw pointer twice, both objects have ownership. Fortunately, delete in its dtor will only succeed once. But using a pointer without ownership to the real object is likely to be a dangling reference like a raw pointer.

Managed Pointer (cont'd)

- auto_ptr is part of C++98, C++03 and is more commonly called a **smart pointer**, do not get confused with **operator->** overloading

Managed Pointer (cont'd)

- auto_ptr is part of C++98, C++03 and is more commonly called a **smart pointer**, do not get confused with **operator->** overloading
- **auto_ptr** implements copy and assignment **with implicit ownership transfer** due to the lack of ***move semantics*** in C++98/03.
  The compiler allows you to pass an auto_ptr by value to a function, the original auto_ptr would lose the ownership and ① the managed resource is going to be deleted as the function exits unless another auto_ptr is returned back. ② auto_ptr cannot manage an array and ③ cannot be used in a container.

Managed Pointer (cont'd)

- auto_ptr is part of C++98, C++03 and is more commonly called a **smart pointer**, do not get confused with **operator->** overloading
- **auto_ptr** implements copy and assignment **with implicit ownership transfer** due to the lack of ***move semantics*** in C++98/03.
  The compiler allows you to pass an auto_ptr by value to a function, the original auto_ptr would lose the ownership and ① the managed resource is going to be deleted as the function exits unless another auto_ptr is returned back. ② auto_ptr cannot manage an array and ③ cannot be used in a container.
- Do **NOT** use auto_ptr!!

Managed Pointer (cont'd)

- auto_ptr is part of C++98, C++03 and is more commonly called a **smart pointer**, do not get confused with **operator->** overloading
- **auto_ptr** implements copy and assignment **with implicit ownership transfer** due to the lack of ***move semantics*** in C++98/03.
  The compiler allows you to pass an auto_ptr by value to a function, the original auto_ptr would lose the ownership and ① the managed resource is going to be deleted as the function exits unless another auto_ptr is returned back. ② auto_ptr cannot manage an array and ③ cannot be used in a container.
- Do **NOT** use auto_ptr!!
- The following smart pointers are designed to replace it
  - **boost::shared_ptr**, **boost::scope_ptr**, **boost::shared_array**, **boost::scope_array**, **boost::weak_ptr**

Managed Pointer (cont'd)

- auto_ptr is part of C++98, C++03 and is more commonly called a **smart pointer**, do not get confused with **operator->** overloading
- **auto_ptr** implements copy and assignment **with implicit ownership transfer** due to the lack of ***move semantics*** in C++98/03.
  The compiler allows you to pass an auto_ptr by value to a function, the original auto_ptr would lose the ownership and ① the managed resource is going to be deleted as the function exits unless another auto_ptr is returned back. ② auto_ptr cannot manage an array and ③ cannot be used in a container.
- Do **NOT** use auto_ptr!!
- The following smart pointers are designed to replace it
  - **boost::shared_ptr**, **boost::scope_ptr**, **boost::shared_array**, **boost::scope_array**, **boost::weak_ptr**
  - C++11: **std::shared_ptr**, **std::weak_ptr**, **std::unique_ptr**