

Introduction to C++

Part II

Søren Debois

Department of Theoretical Computer Science
IT University of Copenhagen

September 12th, 2005

Outline

- 1 References
- 2 Inheritance
- 3 Special members
- 4 Case study: auto_ptr

Outline

- 1 References
- 2 Inheritance
- 3 Special members
- 4 Case study: auto_ptr

Parameter passing

C++ uses **pass-by-value**. (Java uses pass-by-reference.)

Example

```
struct A {  
    char data[0x100000];  
};  
  
A f(A a) {  
    ...  
    a.data[3293] = ...  
    ...  
    return a;  
}
```

This approach has two shortcomings:

- 1 at least 0x100000 is copied onto the run-time stack twice,
- 2 the function `f` can't modify `a`.

Parameter passing (2)

We could use pointers instead.

Example

```
struct A {  
    char data[0x100000];  
};  
  
A* f(A* p) {  
    ...  
    p->data[3293] = ...  
    ...  
    return p;  
}
```

(The syntax `a->data[3293]` is equivalent to `(*a).data[3293]`.)
But then, the calling function will have to worry about pointers. Also, we might accidentally do `++p`, which might not make sense.

Reference types

What we want is **reference types**, just like in Java.

Example

```
struct A {  
    char data[0x100000];  
};  
  
void f(A& a) {  
    ...  
    a.data[3293] = ...  
    ...  
    // No need to return anything.  
}
```

The `a` is really a pointer, but it is automatically dereference, and we can't accidentally move it by, say, adding to it.

Constant references

Very commonly, we can guarantee that we will not modify an argument. In this case, we use **constant references**.

Example

```
struct A {
    char data[0x100000];
};

int f(const A& a) {
    ...
    a.data[3293] = ... // this is now an error
    ...
    return a[132];
}
```

(We'll see later how to specify that member functions does not “modify anything”.)

Outline

1 References

2 Inheritance

3 Special members

4 Case study: auto_ptr

Access specifiers

C++ has three **access specifiers**: **public**, **private** and **protected**.

Example

```
class A {  
    public:    void pub();  
    private: void priv();  
    protected: void prot();  
};
```

- **public** members (`pub`) can be accessed by anyone,
- **private** members can be accessed only by A's members, and
- **protected** members can be accessed only by A's members and by classes derived from A.

Inheritance

- Inheritance in C++ works somewhat like in Java. The derived class inherits methods and members from the base classes.
- However, unlike Java, C++ superclasses may have **multiple** base classes. Also, there are three forms of inheritance, **public**, **private** and **protected**. They differ in what access specifier members of the base class will have as members of the superclass.
- Inheritance in C++ looks like this:

Example

```
class A : public B, private C ...
```

A derives publicly from B and privately from C.

Inheritance (**public**)

In **public** inheritance, public resp. protected members of A become public resp. protected members of B.

Example

```
class B : public A {  
    public: void f();  
};
```

Here, anybody can invoke `B::f()` and `B::pub()`. Only the implementation of `f` can access `B::prot()`.

Example

```
B::f() {  
    ...  
    prot();  
    ...  
}
```

Inheritance (**private**)

In **private** inheritance, the public and protected members of `A` become private members of `B`.

Example

```
class B : private A {  
    public: void f();  
};
```

Here, anybody can invoke `B::f()`, but only the implementation of `f` can access `B::pub()` and `B::prot()`.

Inheritance (protected)

In **protected** inheritance, the public and protected members of A become protected members of B.

Example

```
class B : private A {  
    public: void f();  
};
```

Here, anybody can invoke `B::f()`, but only the implementation of `f` can access `B::pub()` and `B::prot()`.

Inheritance (**private** vs. **protected**)

The difference between **private** and **protected** inheritance is that

- with **private** inheritance, classes derived from B **cannot** access `B::prot()` and `B::pub()`
- with **protected** inheritance, classes derived from B **can** access `B::prot()` and `B::pub()`

Inheritance (2)

We can use an object of a derived class (B) wherever we need an object of one of its base classes (A)

Inheritance (2)

We can use an object of a derived class (B) wherever we need an object of one of its base classes (A) — subject to access specifiers.

Inheritance (2)

We can use an object of a derived class (B) wherever we need an object of one of its base classes (A) — subject to access specifiers.

Example

```
class A { ... };  
class B : public A { ... };
```

Here, anybody can use a B instead of an A:

Example

```
void f(A& a);  
...  
f(B());
```

Inheritance (3)

Example

```
class A { ... };  
class B : private A { ... };
```

Here, only members of B can use a B instead of an A.

Virtual methods

Pop quiz! What is the output of this program?

Example

```
class A {
    public: void f() { cout << "I'm A." << endl; }
};
class B : public A {
    public: void f() { cout << "I'm B." << endl; }
};
main () {
    B b;
    b.f();
    A& a(b);
    a.f();
}
```

Virtual methods

Pop quiz! What is the output of this program?

Example

```
class A {
    public: void f() { cout << "I'm A." << endl; }
};
class B : public A {
    public: void f() { cout << "I'm B." << endl; }
};
main () {
    B b;
    b.f();
    A& a(b);
    a.f();
}
```

Answer:

"I'm B."

"I'm A."

Virtual methods (2)

In C++ you don't get the most specific derived function, **unless** you declared the method **virtual**:

Example

```
class A {
    public: virtual void f() { cout << "I'm A." << endl; }
};
class B : public A {
    public: virtual void f() { cout << "I'm B." << endl; }
};
main () {
    B b;
    b.f();
    A& a(b);
    a.f();
}
```

Output:

"I'm B."

"I'm B."

Virtual methods (3)

Recall that destructors should do whatever cleaning up. They have a second role: Information about **how much** space an object of a class takes up is associated with its destructor. So what happens here?

Example

```
class A { ... /* No destructor */ };
class B : public A { ... };
main () {
    A* p = new B();
    ...
    delete p;
}
```

Virtual methods (3)

Recall that destructors should do whatever cleaning up. They have a second role: Information about **how much** space an object of a class takes up is associated with its destructor. So what happens here?

Example

```
class A { ... /* No destructor */ };
class B : public A { ... };
main () {
    A* p = new B();
    ...
    delete p;
}
```

Because A's destructor was not declared **virtual**, `delete` calls the destructor of A, even though we know `p` points to an object of type B. As a side effect, only the storage used by the A-part of `*p` will be deallocated.

Virtual methods (4)

Rule of thumb: Unless you **really** know what you are doing, always implement a virtual destructor in any base class you make:

Virtual methods (4)

Rule of thumb: Unless you **really** know what you are doing, always implement a virtual destructor in any base class you make:

a.h

```
class A {  
    ...  
    virtual ~A()  
};
```

a.cc

```
#include "a.h"  
...  
A::~~A() {}
```

(Some older compilers will claim that the `A`s destructor is either undefined or multiply defined if you inline the implementation of the destructor.)

Virtual Inheritance (5)

Abstract classes are classes that have one or more **abstract methods**, which are declared like this:

Example

```
class C {
    public:
        virtual void f() = 0;
        ...
}
```

Abstract classes cannot be instantiated. They serve a purpose similar to the “interface” of Java.

Multiple inheritance

Most of the time, multiple inheritance is convenient.

Multiple inheritance

Most of the time, multiple inheritance is convenient. However, **evil** lurks under the surface!

Multiple inheritance

Most of the time, multiple inheritance is convenient. However, **evil** lurks under the surface! It's **perverse**!

Multiple inheritance

Most of the time, multiple inheritance is convenient. However, **evil** lurks under the surface! It's **perverse**! It's **unclean**!

Multiple inheritance

Most of the time, multiple inheritance is convenient. However, **evil** lurks under the surface! It's **perverse**! It's **unclean**! It's **inbreeding**, now in a programming language!

Multiple inheritance

Most of the time, multiple inheritance is convenient. However, **evil** lurks under the surface! It's **perverse**! It's **unclean**! It's **inbreeding**, now in a programming language!

Example

```
class A {
    public: int i;
    ...
};
class B : public A { ... };
class C : public A { ... };
class D : public B, public C { ... };
```

How many members `i` will `D` have?

Multiple inheritance

Most of the time, multiple inheritance is convenient. However, **evil** lurks under the surface! It's **perverse**! It's **unclean**! It's **inbreeding**, now in a programming language!

Example

```
class A {
    public: int i;
    ...
};
class B : public A { ... };
class C : public A { ... };
class D : public B, public C { ... };
```

How many members `i` will `D` have? Two. Derive **virtually** if you want only one. And unless you can read and appreciate chapter 15 of Stroustrup's "The C++ Programming Language" (copies available upon request), avoid such shared base classes.

Outline

- 1 References
- 2 Inheritance
- 3 Special members**
- 4 Case study: auto_ptr

Overview

C++ classes have several special members.

- The default constructor `A()`. This constructor is called when the class is instantiated with no arguments, e.g., the statements `A a` or `A* p = new A`.

Overview

C++ classes have several special members.

- The default constructor `A()`. This constructor is called when the class is instantiated with no arguments, e.g., the statements `A a` or `A* p = new A`.
- Overloaded constructors, say `A(int)`. These constructors are called when the class is instantiated with the given arguments, e.g., the statement `A* p = new A(10)`.

Overview

C++ classes have several special members.

- The default constructor `A()`. This constructor is called when the class is instantiated with no arguments, e.g., the statements `A a` or `A* p = new A`.
- Overloaded constructors, say `A(int)`. These constructors are called when the class is instantiated with the given arguments, e.g., the statement `A* p = new A(10)`.
- The copy constructor `A(const A&)`. This constructor is called when the class must be copied, typically when passing an object of the class as an argument by value, or when returning an object of the class.

Overview

C++ classes have several special members.

- The default constructor `A()`. This constructor is called when the class is instantiated with no arguments, e.g., the statements `A a` or `A* p = new A`.
- Overloaded constructors, say `A(int)`. These constructors are called when the class is instantiated with the given arguments, e.g., the statement `A* p = new A(10)`.
- The copy constructor `A(const A&)`. This constructor is called when the class must be copied, typically when passing an object of the class as an argument by value, or when returning an object of the class.
- The copy assignment `A&operator=(const A&)`. This operator is called whenever an object of the class is assigned to.

Initializer lists

All constructors can have **initializer lists**. These are generally handy, and they are the only way to initialize reference members.

a.h

```
class A {  
    int x;  
    B& b;  
public:  
    A(B& b);  
    ...  
}
```

a.cc

```
A::A(B& b_) : x(5), b(b_) {}
```

Automatically generated constructors

- If you do not specify a default constructor, the compiler will **make one up!** (If it can.)
- If you do not specify a copy constructor, the compiler will **make one up!**
- If you do not specify a copy assignment, the compiler will **make one up!**

Automatically generated constructors (2)

The automatically generated copy constructor and copy-assignment operators are implemented something like this.

```
A::(const A& a) {  
    memcpy(this, <address of a>, sizeof(a));  
}
```

In particular, if A has pointer members, only the pointers themselves are copied.

Automatically generated constructors (3)

Suppose that the `Stack` of last week dynamically allocated its storage:

Example

```
class Stack {  
    char* stck;  
    ...  
    Stack() : stck(new char[100]) {}  
    ~Stack() { delete stck; }  
};
```

Automatically generated constructors (3)

Suppose that the `Stack` of last week dynamically allocated its storage:

Example

```
class Stack {
    char* stck;
    ...
    Stack() : stck(new char[100]) {}
    ~Stack() { delete stck; }
};
```

What happens in this sequence of statements?

Example

```
Stack a;
Stack b;
b = a;
```

Automatically generated constructors (3)

Suppose that the `Stack` of last week dynamically allocated its storage:

Example

```
class Stack {
    char* stck;
    ...
    Stack() : stck(new char[100]) {}
    ~Stack() { delete stck; }
};
```

What happens in this sequence of statements?

Example

```
Stack a;
Stack b;
b = a;
```

We will get `a.stck` and `b.stck` to be the same pointer.

Automatically generated constructors (3)

Suppose that the `Stack` of last week dynamically allocated its storage:

Example

```
class Stack {
    char* stck;
    ...
    Stack() : stck(new char[100]) {}
    ~Stack() { delete stck; }
};
```

What happens in this sequence of statements?

Example

```
Stack a;
Stack b;
b = a;
```

We will get `a.stck` and `b.stck` to be the same pointer.

CATASTROPHE WAITS WHEN THE DESTRUCTOR IS CALLED!

Outline

- 1 References
- 2 Inheritance
- 3 Special members
- 4 Case study: auto_ptr**

The idiom

Suppose we need to do this:

Example

```
void f(...) {  
    A* p = new A();  
    ...  
    g(p);  
    ...  
    delete p;  
}
```

If an exception is thrown by `g`, we leak storage.

The idiom (2)

Idea: Create a wrapper class for pointers, where the destructor automatically deallocates the pointer. That way, exceptions won't hurt us.

Example

```
void f(...) {  
    aptr p(new A());  
    ...  
    g(p);  
}
```

We don't even have to deallocate `p` manually! (This relies on C++ allowing us to specify a method that is called when `p` must be converted to a regular pointer, as happens in the call to `g`.)

Implementation

Here is the class declaration for `aptr`.

`aptr.h`

```
class aptr {
public:
    aptr(A* p_);
    operator A* ();
    ~aptr();
private:
    A* p;
};
```

Implementation (2)

And here is the corresponding definition.

`aptr.cc`

```
#include "aptr.h"

aptr::aptr(A* p_) : p(p_) {}

aptr::operator A* () {
    return p;
}

aptr::~aptr() {
    delete p;
}
```

(A note on templates)

In modern C++ variants, there is a mechanism called **templates**, that allow us to parameterize the `aptr` class over the type of the pointer. Actually, the `aptr` is in the standard library. It is called `auto_ptr<T>`.

(SIGSEGV)

Thank you!