C++ Inheritance

Or: Yet Another Thing That Is Simple In Java and Very Complicated In C++

Colin Gordon
csgordon@cs.washington.edu

University of Washington

CSE333 Section 6, 5/5/11

Today's Topics

Bad News

Did you think memory errors were the only thing Java simplified from C++? No, we've only scratched the surface

1. Basic Inheritance
   - Static Method Dispatch
   - Constructors
   - Destructors
2. Virtual Methods (Dynamic Dispatch)
   - Virtual Destructors
3. Pure Virtual Methods

Basic Public Inheritance

How do we do inheritance in C++?

```
class SubClass : public SuperClass {
    ...
};
```

- Much like in Java, this gives the subclass access to the base class's *public* and *protected* members, but not private members.
- There is also *private* or *protected inheritance*, where the class inherits the base class's properties, but clients don't see the subtyping relation
  - In private inheritance, subclasses of the subclass also don't see the privately-inherited base class's members

A Method Dispatch Surprise

Java and C++ dispatch methods differently... From BadOverride.cc:

```
class Employee {
    public:
        void print() {
            cout << "I am a mere employee" << endl;
        }
};
class Manager : public Employee {
    public:
        void print() {
            cout << "I am not only an employee, but a manager!" << endl;
        }
};
int main() {
    Manager *m = new Manager();
    Employee *e = m;
    e->print();
    delete m;
    return 0;
}
```

What does this print?

Static Method Dispatch

What Went Wrong?

By default, when calling method `m` on an object `o` statically typed as class `C`, C++ *directly calls* the method `C.m` — regardless of `o`'s actual runtime class!

Why Would They Do Such a Thing?

C++ was invented in the early 1980s. Back then a clock cycle was a valuable thing, and in cases where static dispatch was the right thing, it was considerably faster than dynamic dispatch.

C++ does have dynamic dispatch (what you're used to from Java) but you have to explicitly request it.

Constructors with Inheritance

Since base class may have inaccessible members, we must be able to trigger its constructor

- Default constructors can be called automatically
- If the superclass has no default constructor, we must be able to invoke non-default
  - Invoked similar to an initializer for a member of the base class
- Can have multiple / default constructors in the subclass invoke arbitrary constructor for the superclass

From ConstructorTest.cc:

```
class BaseClass {
  public:
    BaseClass(int x);
    int getX();
  private:
    int x_;
};
class SubClass : public BaseClass {
  public:
    SubClass(int x, int y);
    ...
  private:
    int y_;
};
...
SubClass::SubClass(int x, int y)
  : BaseClass(x),
    y_(y)
{ ... }
```

Destructors with Inheritance

- Superclass destructors are always called automatically when a subclass's destructor is invoked!
- But invoking the subclass destructor is subtle...
  - Because it might be statically dispatched.

From BadDestructors.cc:

```
class Employee {
  public:
    ~Employee() {
      cout << "employee destructor" << endl;
    }
};
class Manager : public Employee {
  public:
    ~Manager() {
      cout << "manager destructor" << endl;
    }
};
int main() {
  Manager *m = new Manager();
  Employee *e = m;
  delete e;
  return 0;
}
```

What does this print?

Virtual Methods (Dynamic Dispatch)

So how do we get *dynamic* dispatch?

```
class Employee {
    public:
        virtual void print() {
            cout << "I am a mere employee" << endl;
        }
};
class Manager : public Employee {
    public:
        void print() {
            Employee::print();  // Can still invoke base class's method
            cout << "I am not only an employee, but a manager!" << endl;
        }
};
int main() {
    Manager *m = new Manager();
    Employee *e = m;
    e->print();
    delete m;
    return 0;
}
```

Virtual Destructors

The problem with statically dispatched destructors can be handled the same way:

```
class Employee {
  public:
    virtual ~Employee() {
      cout << "employee destructor" << endl;
    }
};
class Manager : public Employee {
  public:
    ~Manager() {
      cout << "manager destructor" << endl;
    }
};
int main() {
  Manager *m = new Manager();
  Employee *e = m;
  delete e;
  return 0;
}
```

Pure Virtual Functions

Basically the C++ equivalent to Java's `abstract` keyword:

```
class BaseClass {
    public:
        virtual int m(int x) = 0;
};
```

- A class with at least one pure-virtual method is called an *abstract class*
- An abstract class with no methods implemented is roughly what Java calls an interface
- Abstract classes cannot be instantiated directly
  - e.g. for the code above, `new BaseClass()` is a compiler error!