CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 10
October 4, 2011

Polymorphic Derivation (cont.)

Name Visibility

Uses of polymorphism

Some uses of polymorphism:

- To define an extensible set of representations for a class.
- To allow containers to store mixtures of different but related types of objects.
- To support run-time variability of within a restricted set of related types.

Multiple representations

Might want different representations for an object.

Example: A point in the plane can be represented by either Cartesian or Polar coordinates.

A `Point` base class can provide abstract operations on points. E.g., `virtual int quadrant() const` returns the quadrant of `*this`.

For Cartesian coordinates, quadrant is determined by the signs of the x and y coordinates of the point.
For polar coordinates, quadrant is determined by the angle θ.

Both `Cartesian` and `Polar` derived classes should contain a method for `int quadrant() const`.

Heterogeneous containers

One might wish to have a stack of `Point` objects.

The element type of the stack would be `Point*`.

The actual values would have type either `Cartesian*` or `Polar*`.

The automatically generated type tags and dynamic dispatching obviates the need to cast the result of `pop()` to the correct type.

Example:

```
Stack st; Point* p;
p = st.pop();   // no need to cast result
p->quadrant(); // automatic dispatch
```

Run-time variability

Two types are closely related; differ only slightly.

Example: Company has several different kinds of employees.

- `Employee` base class has a large and complicated payroll function.
- Payroll is same for all kinds of employees except for a function `pay()` that computes the actual weekly pay.
- Each employee kind has its own `pay()` function.
- Big payroll function is in base class.
- It calls `pay()` to get the actual pay for this `Employee`.

Pure virtual functions

Suppose we don't want `B::f()` and never create instances of `B`.
We make `B::f()` into a pure virtual function by writing `=0`.

```
class B     { public: virtual int f()=0; ... };
class U : B { virtual int f(); ... };
class V : B { virtual int f(); ... };
B* bp;
```

A pure virtual function is sometimes called a promise.
It tells the compiler that a construct like `bp->f()` is legal.
The compiler requires every derived class to contain a method `f()`.

Abstract classes

An abstract class is a class with one or more pure virtual functions.

An abstract class cannot be instantiated.
It can only be used as the base for another class.

The destructor can never be a pure virtual function but will generally be `virtual`.

A pure abstract class is one where all member functions are pure virtual (except for the destructor) and there are no data members,

Pure abstract classes define an interface à la Java.

An interface allows user-supplied code to integrate into a large system.

Name visibility

Private derivation (default)

`class B : A { ... };` specifies private derivation of `B` from `A`.

A class member inherited from `A` become `private` in `B`.
Like other private members, it is inaccessible outside of `B`.

If `public` in `A`, it can be accessed from within `A` or `B` or via an instance of `A`, but not via an instance of `B`.

If `private` in `A`, it can only be accessed from within `A`.
It cannot even be accessed from within `B`.

Private derivation example

Example:

```
class A {
private:  int x;
public:   int y;
};
class B : A {
    ... f() {... x++; ...} // privacy violation
};
//-------- outside of class definitions --------
A a; B b;
a.x   // privacy violation
a.y   // ok
b.x   // privacy violation
b.y   // privacy violation
```

Public derivation

`class B : public A { ... };` specifies public derivation of `B` from `A`.

A class member inherited from `A` retains its privacy status from `A`.

If `public` in `A`, it can be accessed from within `B` and also via instances of `A` or `B`.

If `private` in `A`, it can only be accessed from within `A`.
It cannot even be accessed from within `B`.

Public derivation example

Example:

```
class A {
private:  int x;
public:   int y;
};
class B : public A {
    ... f() {... x++; ...} // privacy violation
};
//-------- outside of class definitions --------
A a; B b;
a.x   // privacy violation
a.y   // ok
b.x   // privacy violation
b.y   // ok
```

The `protected` keyword

`protected` is a privacy status between `public` and `private`.

Protected class members are inaccessible from outside the class (like `private`) but accessible within a derived class (like `public`).

Example:

```
class A {
protected: int z;
};
class B : A {
    ... f() {... z++; ...} // ok
};
```

Protected derivation

`class B : protected A { ... };` specifies protected derivation of `B` from `A`.

A `public` or `protected` class member inherited from `A` becomes `protected` in `B`.

If `public` in `A`, it can be accessed from within `B` and also via instances of `A` but not via instances of `B`.

If `protected` in `A`, it can be accessed from within `A` or `B` but not from outside.

If `private` in `A`, it can only be accessed from within `A`.
It cannot be accessed from within `B`.

Privacy summary

Class `A`	Kind of Derivation: public	Kind of Derivation: protected	Kind of Derivation: private
public	public	protected	private
protected	protected	protected	private
private	invisible	invisible	invisible

Visibility in derived class `B`.

Surprising example 1

```
1   class A {
2   protected:
3     int x;
4   };
5   class B : public A {
6   public:
7     int f() { return x; }          //  ok
8     int g(A* a) { return a->x; }  //  privacy violation
9   };
```

Result:

```
tryme1.cpp: In member function 'int B::g(A*)':
tryme1.cpp:3: error: 'int A::x' is protected
tryme1.cpp:9: error: within this context
```

Surprising example 2: contrast the following

```
1   class A { };
2   class B : public A {};    // <-- public derivation
3   int main() { A* ap; B* bp;
4     ap = bp; }
```

Result: OK.

```
1   class A { };
2   class B : private A {};    // <-- private derivation
3   int main() { A* ap; B* bp;
4     ap = bp; }
```

Result:

```
tryme2.cpp: In function 'int main()':
tryme2.cpp:4: error: 'A' is an inaccessible base of 'B'
```

Surprising example 3

```
1   class A { protected: int x; };
2   class B : protected A {};
3   int main() { A* ap; B* bp;
4     ap = bp; }
```

Result:

```
tryme3.cpp: In function 'int main()':
tryme3.cpp:4: error: 'A' is an inaccessible base of 'B'
```