CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 10
October 5, 2010

Name Visibility

Polymorphic Derivation

PS2 Craps Game Revisited

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

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

Visibility in derived class B.

Polymorphic Derivation

Polymorphism and Type Hierarchies

Consider following simple type hierarchy:

```
class B     { public: int f(); ... };
class U : B { int f(); ... };
class V : B { int f(); ... };
```

We have a base class `B` and derived classes `U` and `V`.

Declare `B* bp; U* up = new U; V* vp = new V`.
Can write `bp = up;` or `bp = vp;`.

Why does this make sense?
`*up` has an embedded instance of `B`.
`*vp` has an embedded instance of `B`.

Relationships: A `U` is a `B` (and more). A `V` is a `B` (and more).

Polymorphic pointers

Recall:

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

`bp` can point to objects of type `B`, type `U`, or type `V`.
Say `bp` is a polymorphic pointer.

Want `bp->f()` to refer to `U::f()` if `bp` contains a `U` pointer.
Want `bp->f()` to refer to `V::f()` if `bp` contains a `V` pointer.
In this example, `bp->f()` always refers to `B::f()`.

Virtual functions

Solution: Polymorphic derivation

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

A virtual function is dispatched at run time to the class of the actual object.

`bp->f()` refers to `U::f()` if `bp` points to a `U`.
`bp->f()` refers to `V::f()` if `bp` points to a `V`.
`bp->f()` refers to `B::f()` if `bp` points to a `B`.

Here, the type refers to the allocation type.

Unions and type tags

We can regard `bp` as a pointer to the union of types `U` and `V`.

To know which of `U::f()` or `V::f()` to use for the call `bp->f()` requires runtime type tags.

If a class has `virtual` functions, the compiler adds a type tag field to each object.
This takes space at run time.

The compiler also generates a vtable to use in dispatching calls on virtual functions.

Virtual destructors

Consider `delete bp;`, where `bp` points to a `U` but has type `B*`.

The `U` destructor will *not* be called unless destructor `B::~B()` is declared to be `virtual`.

Note: The base class destructor is always called, *whether or not it is* `virtual`.

In this way, destructors are different from other member methods.

Conclusion: If a derived class has a non-empty destructor, the *base class* destructor should be declared `virtual`.

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 abstract (pure virtual) 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.

PS2 Craps Game Revisited

Extending existing code

To test and debug randomized code, one needs to control the "random" data on which it depends in order to:

- have repeatable runs in which to track down manifest bugs.
- be able to force unlikely cases to occur.

Demo `10-Craps-extended` is a significant refactoring of `PS2-craps`, the posted solution to problem set 2.

Summary of extensions

The following significant changes were made to the PS2 code:

1. `Dice::roll()` can now use either `rand()` or a named file in order to determine the outcome of a dice roll.
2. The command line interface was changed to allow specification of the kind of dice to use.
3. The command line parser was moved into a new `Params` class.
4. The '_' suffix of data member names was dropped. The same name is now used for corresponding parameters to constructors. Ambiguity is not a problem in ctors. It is resolved in assignment using the `this->` prefix.
5. A print function useful for debugging was added to each class, and the output operator `<<` extended to use it.

1. Polymorphic dice

Dice are now represented by three classes:

- A base class `Dice`
- A derived class `RandDice: public Dice`
- A derived class `FileDice: public Dice`

A single `Dice*` pointer in `Simulator` holds the current dice. It can point to either a `RandDice` object or a `FileDice` object.

Both kinds of dice support the virtual functions `roll()` and `printSummary()`.

2. Command line interface

The new command line interface is
`craps [-s seed | -f filename] num_rounds`

With no options, random dice are seeded by time of day.

With `-s` option, random dice with specified seed are used.

With `-f` option, dice rolls come from specified file.

The two options are mutually exclusive; specifying both is an error.

3. Command line parser

Command line is parsed using the `getopt()` library function.

The parser code was moved to a new `Params` class.

Reasons for doing so:

- This unclutters `main()`.
- The simulator control parameters are grouped together as data members rather than being local variables in `main()`.
- Passing a `Params` object to the simulator is simpler and cleaner than passing the parameters individually.

4. Naming convention with underscores

In PS2 solution, I used the convention of appending an underscore character to the end of every data member name.

Pros:
- Allows name without underscore to be used for corresponding constructor parameter and/or get-function.
- Data members easily distinguished from other program elements.

Cons:
- Underscores are difficult to see on many screens.
- Adds unnecessary visual complexity to method definitions.

4. Naming convention without underscores

In `10-Craps-extended`, I changed to a convention without underscores.

- Use *same name* for both data member and initializing constructor parameter.
- No ambiguity in ctor, so no problem. Example: In `count(count)`, the first occurrence of `count` always refers to the data member.
- Ambiguity in body of constructor is avoided by writing `this->count` (or `className::count`) when referring to the data member `count` in class `className`.
- Capitalize name and prefix with `get` for get-function name, e.g., `getCount()`. This is a widely used convention for a readonly method to access a data member.

5. Print functions

It is very useful for debugging to add a `print()` function to each class and to extend operator `<<` to use it.

The print function should be declared as:

```
ostream& print(ostream& out) const;
```

and should return `out` as its result.

To extend the output operator to items of class `T`, put

```
inline ostream& operator<<(ostream& out, const T& t) {
    return t.print(out);
}
```

in the `.hpp` file following the class definition.

Other changes

There are several other minor changes to the code. Three that come to mind are:

- `Random` is now a composed object in dice rather than an aggregated object.
- I merged `main.hpp` and `main.cpp` since `main` is not a class, and nobody else should be including `main.hpp`.
- I separated setting up the simulator from running it.