# CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 18
November 7, 2016

Demo: Craps Game

Polymorphic Derivation (continued)

Name Visibility

## Demo: Craps Game

## Game Rules

The player (known as the *shooter*) rolls a pair of fair dice.

1. If the sum is 7 or 11 on the first throw, the shooter wins; this event is called a natural.
2. If the sum is 2, 3, or 12 on the first throw, the shooter loses; this event is called craps.
3. If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, this number becomes the shooter's point. The shooter continues rolling the dice until either she rolls the point again (in which case she wins) or rolls a 7 (in which case she loses).

(From http://www.math.uah.edu/stat/games/Craps.html)

## A Craps simulator

Demo 18-Craps illustrates the use of derived classes in order to allow the simulator to work with both random dice and "prerecorded" dice throws stored in a file.

## Polymorphic Derivation (continued)

## Uses of polymorphism: 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 we never create instances of the base class `B`.
Rather, we want every derived class to provide a definition for `f()`.
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`.

## 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

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

Result:

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