Subclassing

- In many ways, OOP is “all about” subclasses overriding methods
  - Often not what you want, but what makes OOP fundamentally different from, say, functional programming (Scheme, ML, Haskell, etc., cf. CSE413)
- C++ gives you lots more options than Java with different defaults, so it’s easy to scream “compiler bug” when you mean “I’m using the wrong feature”...
Subclassing in C++

• Basic subclassing:
  ```
  class D : public C { ... }
  ```

• This is *public inheritance*; C++ has other kinds too (won’t cover)
  – Differences affect visibility and issues when you have multiple superclasses (won’t cover)
  – So **do not forget** the *public* keyword
More on subclassing

• Not all classes have superclasses (unlike Java with Object)
  – (and classes can have multiple superclasses —
    more general and complexity-prone than Java)
• Terminology
  – Java (and others): “superclass” and “subclass”
  – C++ (and others): “base class” and “derived class”
• Our example code: House derives from Land which
derives from Property (read the code, no time for
detailed presentation)
• As in Java, can add fields/methods/constructors, and
override methods
Constructor and destructors

- Constructor of base class gets called before constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one after the : in the constructor
  - Initializer syntax:
    ```
    Foo::Foo(...): Bar(args); int(x) { ... }
    ```
    • Needed to execute superclass constructor with arguments; also works on instance variables and is preferred in production code (slogan: “initialization preferred over assignment”)
- Destructor of base class gets called after destructor of derived class
- So constructors/destructors really extend rather than override, since that is typically what you want
  - Java is the same
Method overriding, part 1

• If a derived class defines a method with the same method name and argument types as one defined in the base class (perhaps because of an ancestor), it overrides (i.e., replaces) rather than extends

• If you want to use the base-class code, you specify the base class when making a method call (\texttt{class::method(...)})
  – Like \texttt{super} in Java (no such keyword in C++ since there may be multiple inheritance)

• Warning: the title of this slide is \textit{part 1}
Casting and subtyping

• An object of a derived class cannot be cast to an object of a base class.
  – For the same reason a struct T1 {int x,y,z;} cannot be cast to type struct T2 {int x, y;}
    (different size)
• A pointer to an object of a derived class can be cast to a pointer to an object of a base class.
  – For the same reason a struct T1* can be cast to type struct T2* (pointers to a location in memory)
  – (Story not so simple with multiple inheritance)
• After such an upcast, field-access works fine (prefix), but what do method calls mean in the presence of overriding?
class A {
    public:
    
    void m1() { cout << "a1"; }
    
    virtual void m2() { cout << "a2"; }
};

class B : public A {
    
    void m1() { cout << "b1"; }
    
    void m2() { cout << "b2"; }
};

void f() {
    A* x = new B();
    
    x->m1();
    x->m2();
}
In words...

- A **non-virtual method-call** is *resolved* using the (compile-time) type of the *receiver* expression.
- A **virtual method-call** is *resolved* using the (run-time) class of the *receiver object* (what the expression evaluates to)
  - Like in Java
  - Called “dynamic dispatch”
- A method-call is virtual if the method called is marked `virtual` or overrides a virtual method.
  - So “one virtual” somewhere up the base-class chain is enough, but it’s probably better style to repeat it.
More on two method-call rules

• For software-engineering, virtual and non-virtual each have advantages:
  – Non-virtual – can look at the code to know what you’re calling (even if subclass defines the same function)
  – Virtual – easier to extend code already written
• The implementations are the same and different:
  – Same: Methods just become functions with one extra argument this (pointer to receiver)
  – Different:
    • Non-virtual: linker can plug in code pointer
    • Virtual: At run-time, look up code pointer via “secret field” in the object
Destructors revisited

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

... 
B * b = new B();
A * a = b;
delete a;
```

- Will `B::~B()` get called (before `A::~A()`)?
- Only if `A::~A()` was declared `virtual`
  - Rule of thumb: Declare destructors virtual; usually what you want
Downcasts

Old news:
• C pointer-casts: unchecked; better know what you are doing
• Java: checked; may raise ClassCastException (checks “secret field”)

New news:
• C++ has “all the above” (several different kinds of casts)
• If you use single-inheritance and know what you are doing, the C-style casts (same pointer, assume more about what is pointed to) should work fine for downcasts
• Worth learning about the differences on your own
A C++ “pure virtual” method is like a Java “abstract” method.

- Some subclass must override because there is no definition in base class.
- Makes sense with dynamic dispatch.
- Unlike Java, no need/way to mark the class specially.
- Funny syntax in base class; override as usual:
  ```cpp
class C {
  virtual t0 m(t1,t2,...,tn) = 0;
  ...
};
```
- Side-comment: with multiple inheritance and pure-virtual methods, no need for a separate notion of Java-style interfaces.
C++ summary

• Lots of new syntax and gotchas, but just a few new concepts:
  – Objects vs. pointers to objects
  – Destructors
  – virtual vs. non-virtual
  – pass-by-reference
  – Plus all the stuff we didn’t get to, especially templates, exceptions, and operator overloading.
  – Later (if time): why objects are better than code-pointers – coding up object-like idioms in C