Administrivia

• HW6a due last night
  • We’ll grade ASAP, get you feedback before 6b is due
• HW6b due next Thurs
  • In order to use late days, both partners must have them
Subclassing

• Remember Java, where you could extend one class with another?
  • It turned out to not always be the best design…
  • You can do the same thing in C++

• C++ gives you lots more options than Java, and different defaults
  • If something seems not right, it’s probably “using the wrong feature” not “compiler bug”
Subclassing

• In C++, you can subclass in the following way:

```cpp
class D : public C { ... }
```

• This is public inheritance, C++ also has other kinds (which we won’t cover)

• **DO NOT FORGET** the public keyword above
Subclassing

- Not all classes have a super class (unlike Java with Object)
  - Classes don’t have to have just one parent (don’t do this, not only in 374, but also in life)
- Terminology:
  - Java: “Superclass” and “Subclass”
  - C++: “Base Class” and “Derived Class”
- As in Java, you can add fields/methods/constructors, and override methods
Constructors and Destructor

- Constructor of base class gets called before constructor of derived class
  - If not specified, default (0 arg) constructor is called
  - Can specify with initializer syntax (considered good style)
    ```cpp
    Foo::Foo() : Bar(args); other_data(x)
    ```
- Destructor of base class called after destructor of derived class
- Constructors really extend rather than override
Method Overriding (part 1)

• If a derived class defines the same method (name and param types) as the base class, that method gets overridden

• If you want to call base class code, use `class::method(…)`

  • Like super, but C++ has no super keyword

• Warning: This is just part 1, we’re not done yet
Casting and Subtyping

• An **object** of a derived class *cannot* be cast to an object of a base class
  • Same reason a **struct** T1 {int x, y, z;} can’t be cast to a **struct** T2 {int x, y;} (different sizes)
• A **pointer** to an object of a derived class *can* be cast to a pointer to an object of a base class
  • Same reason you can cast a **struct** T1* to a **struct** T2*
• This is called an upcast, field access works fine, method calls are **not** what you would expect
Important Example

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();
}
Explained

• A non-virtual method-call is resolved at compile time using the static type of the expression

• A virtual method-call is resolved at runtime using the dynamic type of the expression

  • Like Java

  • Called “dynamic dispatch”

• A method call is virtual if the method is marked virtual, or overrides a virtual method
When to use

• For good engineering, use non-virtual by default, only use virtual methods when actually needed
• This makes code easier to think about: at each method call you know what code is being called
• Implementations:
  • Non-virtual: same as normal method call, one hidden parameter to the object
  • Virtual: run-time lookup of what code to call via “secret field” in the object (more next lecture)
Destructors

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
  • Unlike methods, **ALWAYS** declare the destructor virtual
Downcasts

• Casting to a derived class from a base class is called a downcast

• If you do it right, it will work right

• If you do it wrong, no guarantee it is checked (hopefully you crash but who knows?)

• Not like Java, don’t assume it’s checked
Pure virtual methods

- A C++ “pure virtual” method is like a Java “abstract” method
  - Some subclass must override because there is no definition in the base class
  - Makes sense with dynamic dispatch
  - Unlike Java, no need or way to mark the class specially

```cpp
class C {
    virtual t0 m(t1, t2, t3...) = 0;
};
```

- No Java-style interfaces, instead this is it
C++ summary

• Lots of new syntax and ways to get it wrong, but just a few new concepts
  • Objects vs. Pointers to Objects
  • Destructors
  • virtual vs. non-virtual
  • pass-by-reference
  • Plus more (that we won’t cover): templates, exceptions, and operator overloading