C++ & Subtyping
Week 7

• Statically-typed OO languages: C++
• Closer look at subtyping
Why talk about C++?

• C++ is an OO extension of C
  ➤ Efficiency and flexibility from C
  ➤ OO program organization from Simula

• Interesting design decisions
  ➤ Features were and still are added incrementally
  ➤ Backwards compatibility is a huge priority
  ➤ “What you don’t use, you don’t pay for.”- Bjarne Stroustrup
Recall: C++ OO concepts in 1 slide

- Encapsulation
  - Public, private, protected + friend classes
- Dynamic lookup
  - Only for special functions: virtual functions
- Inheritance
  - Single and multiple inheritance!
  - Public and private base classes!
- Subtyping: tied to inheritance
Plan for C++

• Look at dynamic lookup as done in C++ (vtables)
  ➤ Why?

• Only interesting when inheritance comes into play
  ➤ Why?
Simple example

```cpp
class A {
    int a;
    void f(int);
}

A* pa;
pa->f(2);
```

compiles to

`__A_f(pa, 2);`

runtime representation of A object

info necessary to lookup function: type of pointer
Inheritance

class A {
    int a;
    void f(int);
}
class B : A {
    int b;
    void g(int)
}
class C : B {
    int c;
    void h(int)
}
Inheritance

class A {
  int a;
  void f(int);
}
class B : A {
  int b;
  void g(int)
}
class C : B {
  int c;
  void h(int)
}
Inheritance + virtual methods

class A {
    int a;
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
}

class B : A {
    int b;
    void g(int)
}

class C : B {
    int c;
    void h(int)
}

C* pc;
pc->g(2);

- runtime representation of C object
- vptr
  - int a
  - int b
  - int c
- vtable
  - A::f
  - B::g
  - C::h

- info necessary to lookup function: found at runtime
  - (*(pc->vptr[1]))(pc, 2)
Non-virtual vs. Virtual

- **Non-virtual functions**
  - Do they get called directly?  
    - A: yes, B: no

- **Virtual functions**
  - Do they get called directly?  
    - A: yes, B: no
  - They go through the vtable
Non-virtual vs. Virtual

• Non-virtual functions
  ➤ Can they be redefined? A: yes, B: no, C: ehhhh
  ➤ They can be overloaded

• Virtual functions
  ➤ Can they be redefined? A: yes, B: no, C: ehhhh
Virtual methods can be redefined

class A {
    int a;
    virtual void f() {
        printf("parent");
    }
}

class B : A {
    int b;
    virtual void f() {
        printf("child");
    }
}

A* pa = new B();
pa->f();

compiles to

(*((pa->vptr[0])))(pa)

info necessary to lookup
function: found at runtime

runtime representation of B object
Non-virtual functions are overloaded

class A {
    int a;
    void f() {
        printf("parent");
    }
}

class B {
    int b;
    void f() {
        printf("child");
    }
}

A* pa = new B();
pa->f();

A* pa = new B();
pa->f();

info necessary to lookup function: type of pointer

runtime representation of B object

__A_f(pa)
Dynamic vs. static OO systems

- Smalltalk and JavaScript: no static type system
  - In message obj.method(arg), the obj can refer to anything
  - Need to find method using pointer from obj
  - The location in dictionary/hashtable will vary
- In C++ compiler knows the superclass for obj
  - Offset of data and function pointers are the same in subclass and superclass
  - Invoke function pointer at fixed offset in vtable!
Virtual method call takeaway

Invoke function pointer at fixed offset in vtable!
Week 7

- Statically-typed OO languages: C++
- Closer look at subtyping
What is subtyping?

• Relationship between interfaces
  ➤ in contrast to inheritance: relationship between implementations

• If interface A contains all of interface B, then A <: B
  ➤ Interface = set of messages the object understands
  ➤ Eg., ColorPoint <: Point
Subtyping in JavaScript

- Objects implicitly have an interface
  - No recorded by some type system;

    ```
    Point {x, y, move}
    ColoredPoint {x, y, move, color}
    ```

- No relationship to inheritance
  - can delete methods, etc.

    ```
    Boo {x, y, move, boo}
    ```
Subtyping in C++

• Is implementing same functions enough?
  ➤ A: yes, B: no

```cpp
class ColoredPoint {
  public:
    virtual void move();
    virtual int color();
  private:
    ...
};
```

```cpp
class Point {
  public:
    virtual int move();
    private:
        ...
};
```
What is an interface in C++?

- Recall: everything gets compiled down to fn call
  - memory layout of objects
  - memory layout of vtables

- From inheritance, we get:
  - compatible memory layout
  - subtype relation
What does subtyping really mean?
Where does the name come from?

- ColoredPoint vs. Point
  - Interface is clearly bigger for Colored Point

  Point \{x, y, move\}
  ColoredPoint \{x, y, move, color\}

- Why subtype?
  - Think: Natural <: Integer
  - Think:
What does it mean in PL?

- S is a subtype of T if any term of type S can be used in a context where a term of type T is expected
  - This is a runtime phenomenon: when one term can be used where an object of another type is expected
  - Static type system can tell us if we got it right
What does it mean in PL?

\[
\begin{align*}
e &:: S \\
S &:: T \\
\hline \\
e &:: T
\end{align*}
\]
Who defines <: ?

• Language designers!

• How is <: defined in C++?
  ➤ Class definition: class B: public A { } tells us B <: A

• Why is the definition important?
  ➤ It may restrict how we can override functions in subclasses
Who defines <: ?

A: developers  B: language designers
Who defines <: ?

• How is <: defined in C++?
  ➤ Class definition: class B: public A { } tells us B <: A

• Why is the definition important?
  ➤ It may restrict how we can override functions in subclasses
Return covariance

• Is it OK to override clone as follows?

```
class A {
    public:
        virtual bool equals(A&);
        virtual A* clone();
};

class B: public A {
    public:
        bool equals(A&);
        B* clone();
};
```

➤ Yes! Why? any case we need clone of As, we can use B’s clone and upcast the B to an A.

➤ Suppose A* pa = new B(); then pa->clone(); returns a B* that can always be casted to an A*
Is it OK to override equals as follows?

```cpp
class A {
public:
    virtual bool equals(A&);  
    virtual A* clone();
}

class B: public A {
public:
    bool equals(B&);
    B* clone();
}
```

➤ No! Why? the implementation of equals must be prepared for any object of type A to be passed in; B is one kind of A

➤ Suppose A* pa1 = new B(); and A* pa2 = new C(); where C is a subclass of A then pa->equals(pa2); should fail since we should not be allowed to cast a C object to a B object.
Subtyping rule for functions

- Subtyping for function results
  - if $A <: B$ then $C \rightarrow A <: C \rightarrow B$ (covariance)
  - E.g., $C \rightarrow \text{ColorPoint} <: C \rightarrow \text{Point}$
  - Anywhere you expect a function that returns a $B$ you can use a function that returns an $A$ — $A$’s “are” $B$’s so you can upcast the return value
Subtyping rule for functions

- Subtyping for function arguments
  - if \( A <: B \) then \( B -> C <: A -> C \) (contravariance)
  - E.g., \( \text{Point} -> C <: \text{ColorPoint} -> C \)
  - Anywhere you expect a function that can operate on \( A \)s you can use a more general function that operates on \( B \)s — you can always cast the \( A \) argument you were going to call the function with to a \( B \)
Return covariance (w/ records)

class A {
   public:
      virtual bool equals(A&);
      virtual A* clone();
   }

class B: public A {
   public:
      bool equals(A&);
      B* clone();
   }

type A = {
   equals :: A -> bool;  <:
   clone :: () -> A;    <:
   }

type B = {
   equals :: A -> bool;  <:
   clone :: () -> B;    <:
   }
Argument covariance (w/ records)

class A {
    public:
        virtual bool equals(A&);
        virtual A* clone();
}

class B: public A {
    public:
        bool equals(B&);
        B* clone();
}

type A = {
    equals :: A -> bool;
    clone :: () -> A;
}

type B = {
    equals :: B -> bool;
    clone :: () -> B;
}
Example

Circle <: Shape

Circle -> Shape

Circle -> Circle

Shape -> Circle

Shape -> Shape
For other data types: can be tricky!

• E.g., Java screwed up <: definition for Arrays
  ➢ Generic arrays are covariant
  ➢ Breaks type and memory safety!
We are placing trust in <:
Today

- Statically-typed OO languages: C++
  - vtables
- Closer look at subtyping