Objects (cont.)

Deian Stefan

(Adopted from my & Edward Yang’s CS242 slides)
• Statically-typed OO languages: C++
  ➤ vtables
• 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

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

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

compiles to
__A_f(pa, 2);

info necessary to lookup function: type of pointer

runtime representation of A object

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

compiles to
(*((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

\(*(\text{pa->vptr}[0]))(\text{pa})\)
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();

compiles to

__A_f(pa)

info necessary to lookup
function: type of pointer
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!
Today

• Statically-typed OO languages: C++
  ➢ vtables

• 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, color, move\}

• No relationship to inheritance
  ➤ can delete methods, etc.

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

- Subtyping is explicit
  - $A <: B$ if $A$ has public base class $B$

- Why is this not enough?

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

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

  \[
  \begin{align*}
  \text{Point} & \quad \{x, y, \text{move}\} \\
  \text{ColoredPoint} & \quad \{x, y, \text{color}, \text{move}\}
  \end{align*}
  \]

- Why subtype?
  - Think: Natural <: Integer
  - Think:

- Points
  - ColoredPoints
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?

\[
\begin{align*}
  e :: S & \quad S <: T \\
  \hline \\
  e :: T
\end{align*}
\]
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
Return covariance

• Is it OK to override clone as follows?

```cpp
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.
Argument covariance

• Is it OK to override clone 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
Subtyping rule for functions

- Subtyping for function results
  - if \( A <: B \) then \( C -> A <: C -> B \) (covariance)

- Subtyping for function arguments
  - if \( A <: B \) then \( B -> C <: A -> C \) (contravariance)
Example

Circle <: Shape

Circle -> Shape
Circle -> Circle
Shape -> Circle
Shape -> Shape

Circle -> Shape
Shape -> Circle
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 <:
duck typing
Can we do better?

Behavioral subtyping (Liskov substitution principle)
Today

• Statically-typed OO languages: C++
  ➢ vtables

• Closer look at subtyping