

Subtyping

(Dynamic Polymorphism)

Yu Zhang

Course web site: <http://staff.ustc.edu.cn/~yuzhang/fopl>

- Pure dynamically-typed OO languages
 - Object implementation and run-time lookup
 - Class-based languages (Smalltalk)
 - Prototype-based languages (Self, JavaScript)
- Statically-typed OO languages
 - C++
 - using static typing to eliminate search
 - problems with C++ multiple inheritance
 - Java
 - using Interfaces to avoid multiple inheritance

References

- [PFPL](#)
 - Chapter 24 Structural Subtyping
 - Chapter 27 Inheritance
- [TAPL \(pdf\)](#)
 - Chapter 15 Subtyping
- [\[Concepts in PLs\]](#)

Smalltalk

- Developed at Xerox PARC: Smalltalk-76, -80
- Object metaphor extended and refined
 - Used some ideas from Simula, but very different lang
 - Everything is an object, even a class
 - All operations are “messages to objects”
- Terminology

Object	Instance of some class	Class	Defines behavior of its object
Selector	Name of a message	Message	Selector together with parameter values
Method	Code used by a class to respond to message	Instance variable	Data stored in object
Subclass	Class defined by giving incremental modifications to some superclass		

Subtyping and Inheritance

- Interface
 - The external view of an object
- Subtyping
 - Relation between interfaces
- Implementation
 - The internal representation of an object
- Inheritance
 - Relation between implementations

Smalltalk: Example

• Point class

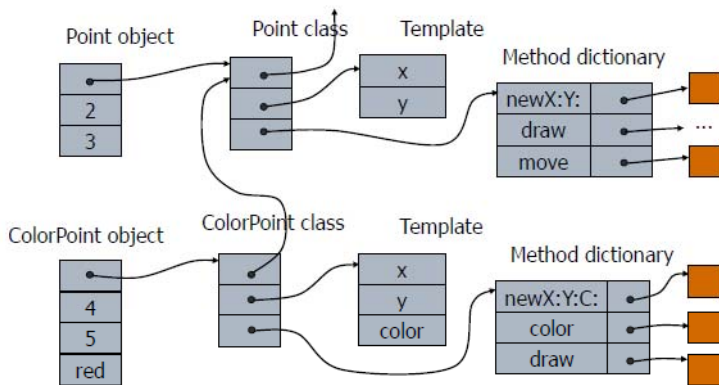
class name	Point
super class	Object
class var	pi
instance var	x y
class messages and methods	
<...names and code for 3 methods, i.e. newX:Y:, newOrigin, initialize>	
instance messages and methods	
<...names and code for 5 methods i.e. x:y:, moveDx:Dy:, x, y, draw>	

• ColorPoint class

class name	ColorPoint
super class	Point
class var	
instance var	color
class messages and methods	
newX:xv Y:yv C:cv <...code...>	
instance messages and methods	
color <...code...> draw <...code...>	

add instance variable
override
override
add method

Smalltalk: Run-time Representation



This is a schematic diagram meant to illustrate the main idea. Actual implementations may differ.

Smalltalk: Subtyping

- If interface **A** contains all of interface **B**, then **A** objects can also be used **B** objects.

Point	ColorPoint
x:y:	x:y:
moveDx:Dy:	moveDx:Dy:
x	x
y	y
draw	color
	draw

ColorPoint interface contains Point
ColorPoint is a subtype of Point

Smalltalk Summary

- **Class**
 - creates objects that share methods
 - pointers to template, dictionary, parent class
- **Objects:** created by a class, contains instance variables
- **Encapsulation**
 - methods public, instance variables hidden
- **Subtyping:** implicit, no static type system
- **Inheritance:** subclasses, self, super
Single inheritance in Smalltalk-76, Smalltalk-80

Subtyping and Inheritance

- Smalltalk/JavaScript subtyping is implicit
 - Not a part of the programming language
 - Important aspect of how systems are built
- Inheritance is explicit
 - Used to implement systems
 - No forced relationship to subtyping

Smalltalk: Object Interfaces

- **Interface**
 - The messages understood by an object
- **Example: point**
 - x:y: set x,y coordinates of point
 - moveDx:Dy: method for changing location
 - x returns x-coordinate of a point
 - y returns y-coordinate of a point
 - draw display point in x,y location on screen
- The interface of an object is its *type*

C++

- C++ is an object-oriented extension of C, Bell Labs
- Object-oriented features
 - **Classes**
 - **Objects**, with dynamic lookup of **virtual functions**
 - **Inheritance**
 - Single and multiple inheritance
 - Public and private base classes
 - **Subtyping**
 - Tied to **inheritance** mechanism
 - **Encapsulation**
 - Public, private, protected visibility

C++: Virtual functions

- Member functions are either
 - **Virtual**, if explicitly declared or inherited as virtual
 - **Non-virtual** otherwise
- Virtual functions
 - Accessed by indirection through ptr in object
 - May be redefined in derived (sub) classes
- Non-virtual functions
 - Are called in the usual way. *Just ordinary functions.*
 - Cannot redefine in derived classes (except overloading)
- Pay overhead only if you use virtual functions

Multiple Inheritance

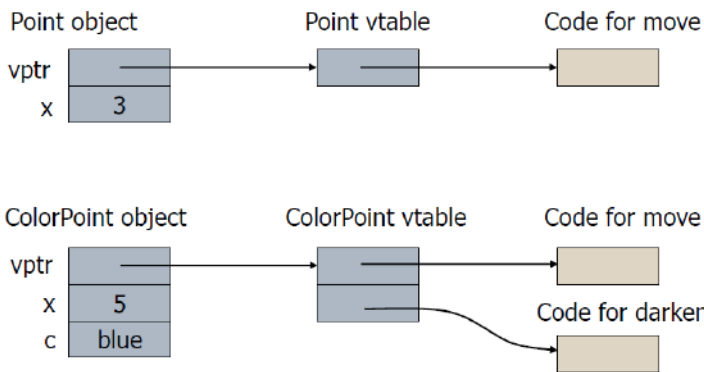
- Name clashes

```
class A {
public:
    virtual void f() { ... }
};
class B {
public:
    virtual void f() { ... }
};
class C : public A, public B{...};
...
C* p;
p->f(); // error
```

- Implicit resolution
 - Language resolves name conflicts with arbitrary rule
- Explicit resolution (C++)
 - Programmer must explicitly resolve name conflicts
- Disallow name clashes
 - Programs are not allowed to contain name clashes

No solution is always best

Run-time Representation



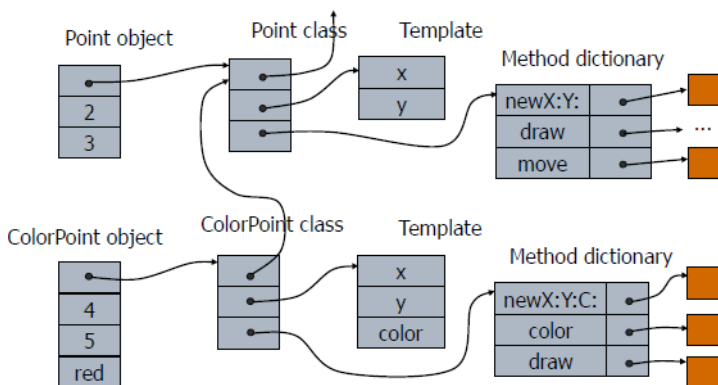
Multiple Inheritance

- Name clashes

```
class A {
public:
    virtual void f() { ... }
};
class B {
public:
    virtual void f() { ... }
};
class C : public A, public B{...};
...
C* p;
p->f(); // error
```

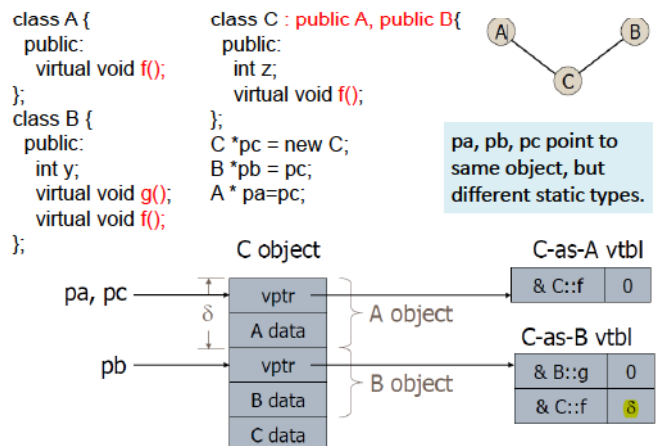
- Rewrite class C to call **A::f** explicitly
 - => eliminate ambiguity
- ```
class C : public A, public B{
public:
 void virtual f() {
 A::f();
 }
}
```

## Compare to Smalltalk/JavaScript

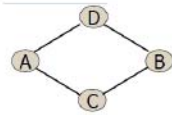


This is a schematic diagram meant to illustrate the main idea. Actual implementations may differ.

## vtable for Multiple Inheritance



## Diamond Inheritance in C++



- Standard base classes
  - D members appear twice in C
- Virtual base classes
 

```
class A : public virtual D { ... }
```

  - Avoid duplication of base class members
  - Require additional pointers so that D part of A, B parts of object can be shared
- C++ multiple inheritance is complicated in because of desire to maintain efficient lookup

## Subtyping Principles

- Products
  - Width subtyping (一个较宽的元组类型是一个较窄的元组类型的子类型)
 
$$\frac{i > j}{[m_1 : \tau_1, \dots, m_i : \tau_i] <: [m_1 : \tau_1, \dots, m_j : \tau_j]}$$
  - Depth subtyping (Covariance)
 
$$\frac{\sigma_i <: \tau_i}{[m_1 : \sigma_1, \dots, m_j : \sigma_j] <: [m_1 : \tau_1, \dots, m_j : \tau_j]}$$
  - Function subtyping
 
$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \rightarrow \tau <: \sigma' \rightarrow \tau'}$$

## C++ Subtyping

- Subtyping in principle
  - $A <: B$  if every A object can be used without type error whenever a B object is required
- C++:  $A <: B$  if class A has public base class B
  - Independent classes not subtypes

## Java

- 1990-95 James Gosling and others at Sun
- Syntax similar to C++
- Object
  - has fields and methods
  - is allocated on heap, not run-time stack
  - accessible through reference (only ptr assignment)
  - garbage collected
- Dynamic lookup
  - Similar in behavior to other languages
  - Static typing => more efficient than Smalltalk
  - Dynamic linking, interfaces => slower than C++

## Function Subtyping

- Subtyping principle
  - $A <: B$  if an A expression can be safely used in any context where a B expression is required
- Subtyping for function results (Covariance)
  - If  $A <: B$ , then  $C \rightarrow A <: C \rightarrow B$
- Subtyping for function arguments (Contravariance)
  - If  $A <: B$ , then  $B \rightarrow C <: A \rightarrow C$
- Terminology
  - Covariance(协变):  $A <: B$  implies  $F(A) <: F(B)$
  - Contravariance(逆变):  $A <: B$  implies  $F(B) <: F(A)$

## Inheritance

- Similar to Smalltalk, C++
- Subclass inherits from superclass
  - Single inheritance only (but Java has interfaces)
- Some additional features
  - Conventions regarding *super* in constructor and *finalize* methods
  - Final classes and methods cannot be redefined

# Interfaces vs Multiple Inheritance

- C++ multiple inheritance
  - A single class may inherit from two base classes
  - Constraints of C++ require derived class representation to resemble *all* base classes
- Java interfaces
  - A single class may implement two interfaces
  - No inheritance (of implementation) involved
  - Java implementation (discussed later) does not require similarity between class representations

## Subtyping

- Primitive types
  - Conversions: int -> long, double -> long, ...
- Class subtyping similar to C++
  - Subclass produces subtype
  - Single inheritance => subclasses form tree
- Interfaces
  - Completely abstract classes
    - no implementation
  - Multiple subtyping
    - Interface can have multiple subtypes (implements, extends)
- Arrays
  - Covariant subtyping – not consistent with semantic principles