Objects in C++

Subtyping

C++ Object System

- Object-oriented features
  1. Classes and Data Abstraction
  2. Encapsulation
  3. Inheritance
     - Single and multiple inheritance
     - Public and private base classes
  4. Objects, with dynamic lookup of virtual functions
  5. Subtyping
     - Tied to inheritance mechanism

Subtyping (1)

- **Subtyping** is a relation on types that allows values of one type to be used in place of values of another.
  - If some object **a** has all of the functionality of another object **b**, then we may use **a** in any context expecting **b**.
- **Inheritance Is Not Subtyping**
  - *"Subtyping is a relation on interfaces, inheritance is a relation on implementations."*
- **A typical example is C++,** in which
  - A class A will be recognized by the compiler as a **subtype of** B only if B is a public base class of A

Subtyping (2)

- (A<:B = A subtype of B)

- Subtyping in principle
  - A <: B if every A object can be used without type error whenever a B object is required

Pt:	int getX();	Public members
	void move(int);	
ColorPt:	int getX();	Public members
	int getColor();	
	void move(int);	
	void darken(int tint);	

- C++: A <: B if class A has public base class B

Sample derived class

```
class ColorPt: public Pt {
public:
   ColorPt(int xv,int cv);
   ColorPt(Pt* pv,int cv);
   ColorPt(ColorPt* cp);
   int getColor();
   virtual void move(int dx);
   virtual void darken(int tint);
protected:
   void setColor(int cv);
private:
   int color;
 };
```

In C++: public base class gives supertype!

Overloaded constructor

Non-virtual function

Virtual functions

Protected write access

Private member data

Independent classes not subtypes

```
class Point {
  public:
    int getX();
    void move(int);
  ...
};
```

```
class ColorPoint {
  public:
    int getX();
    void move(int);
    int getColor();
    void darken(int);
  ...
};
```

- C++ does not treat ColorPoint <: Point as written
- Need public inheritance ColorPoint : public Pt
- Subtyping based on inheritance:
  - An efficiency issue
  - An encapsulation issue: preservation under modifications to base class …

Why C++ design?

- Client code depends only on public interface
  - In principle, if ColorPt interface contains Pt interface, then any client could use ColorPt in place of point
  - However -- offset in virtual function table may differ
  - Lose implementation efficiency
- Without link to inheritance
  - subtyping leads to loss of implementation efficiency
- Also encapsulation issue:
  - Subtyping based on inheritance is preserved under modifications to base class

Function subtyping- theory

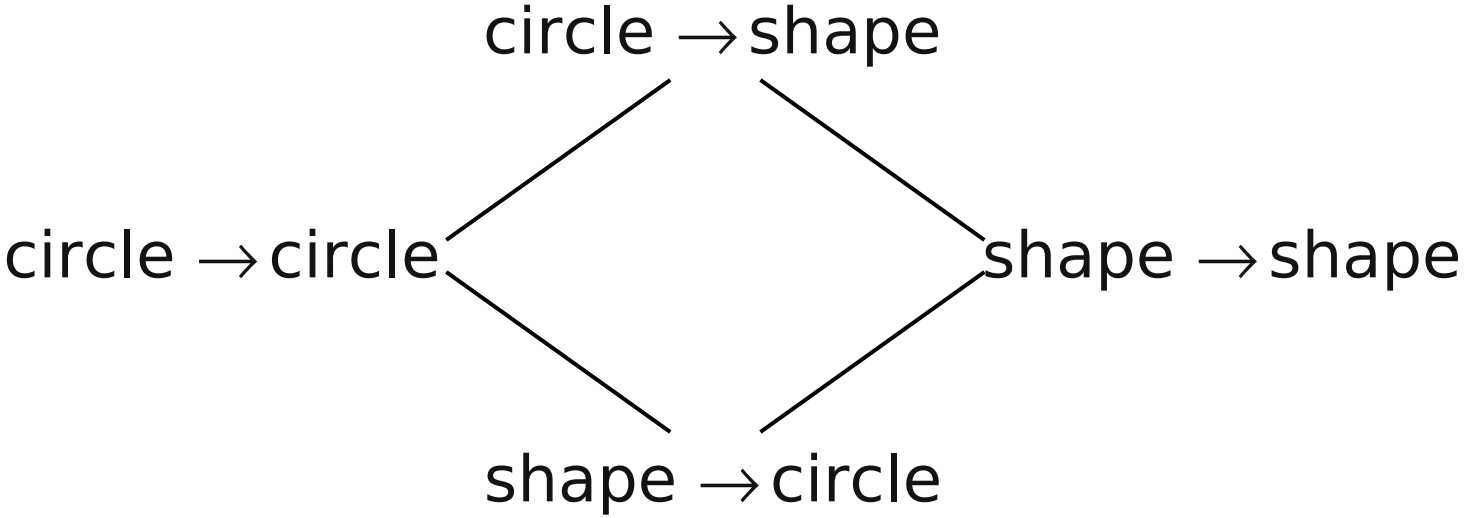
- Subtyping principle
  - A <: B if an A expression can be safely used in any context where a B expression is required
  - In questo modo vale il principio di sostituibilità
- Per le funzioni?
  - Posso estendere e dire che una funzione è sottotipo di un'altra se può essere usata al suo posto.
  - In teoria potrei ammettere l'overriding di funzioni che siano sottotipo
  - Quando una funzione f: $W \rightarrow Z$ è sottotipo di un altra?

Sottotipo per funzioni

- Rispetto il tipo ritornato, basta che ci sia covarianza:
  - If A <: B, then f1:C $\rightarrow$ A <: f2: C $\rightarrow$ B
    - Cioè f1 puà essere usata al postodi f2 se ritorna un sottotipo (A invece che B)
- Rispetto ai suoi argomenti (controvarianza)
  - If A <: B, then f1:B $\rightarrow$ C <: f2: A $\rightarrow$ C
    - Cioè f1 puà essere usata al posto di f2 se prende come argomento un supertipo (B invece che A)
- Terminology
  - Covariance: A <: B implies F(A) <: F(B)
  - Contravariance: A <: B implies F(B) <: F(A)

Examples

- If circle <: shape, then



C++ compilers recognize limited forms of function subtyping

In C++ - from 1998

- C++ supports the covariance of return types
  - Only virtual
  - Only pointers

Subtyping with functions

```
class Point {
  public:
    int getX();
    virtual Point *move(int);
  protected:   ...
  private:     ...
};
```

```
class ColorPoint: public Point {
  public:
    int getX();
    int getColor();
    ColorPoint * move(int);
    void darken(int);
  protected:   ...
  private:     ...
};
```

Inherited, but repeated here for clarity

- In principle: can have ColorPoint <: Point
- In practice: some compilers allow, others have not

This is covariant case; contravariance is another

Details, details

- This is legal

```
class Point { …
    virtual Point * move(int);
…   }
class ColorPoint: public Point {   …
    virtual ColorPoint * move(int);
 …  }
```

- But not legal if *'s are removed

```
class Point { …   virtual Point move(int); … }
class ColorPoint: public Point { …virtual ColorPoint move(int);… }
```

Related to subtyping distinctions for object L-values and object R-values

(Non-pointer return type is treated like an L-value for some reason)

Subtyping and Object L,R-Values

- If class B : public A { ... }

  Then
  - B r-value <: A r-value
    - If x = a is OK, then x = b is OK

      provided A's operator = is public
    - If f(a) is OK, then f(b) is OK

      provided A's copy constructor is public
  - B l-value ~~<:~~ A l-value
  - B* <: A*
  - B** ~~<:~~ A**

Generally, X <: Y → X* <: Y* is unsound.

Review

- Why C++ requires inheritance for subtyping
  - Need virtual function table to look the same
  - This includes private and protected members
  - Subtyping w/o inheritance weakens data abstraction
- Possible confusion regarding inlining
  - Cannot generally inline virtual functions
  - Inlining is possible for non virtual function

Inlining is very significant for efficiency; enables further optimization.

Abstract Classes

- Abstract class:
  - A class that has at least one *pure virtual member function*, i.e a function with an empty implementation
  - Declare by: **`virtual function_decl = 0;`**
  - A class without complete implementation
  - Useful because it can have derived classes

    Since subtyping follows inheritance in C++, use abstract classes to build subtype hierarchies.
  - Establishes layout of virtual function table (vtable)
- Example
  - Geometry classes

C++ Summary

- Objects
  - Created by classes
  - Contain member data and pointer to class
- Encapsulation
  - member can be declared public, private, protected
  - object initialization partly enforced
- Classes: virtual function table
- Inheritance
  - Public and private base classes, multiple inheritance
- Subtyping: Occurs with public base classes only

Some problem areas

- Casts
  - Sometimes no-op, sometimes not (esp multiple inher)
- Lack of garbage collection
  - Memory management is error prone
    - Constructors, destructors are helpful though
- Objects allocated on stack
  - Better efficiency, interaction with exceptions
  - BUT assignment works badly, possible dangling ptrs
- Overloading
  - Too many code selection mechanisms
- Multiple inheritance
  - Efforts at efficiency lead to complicated behavior

Additional topics if more time

- Style guides for C++:
  - Should a programming language enforce good style?
    - Make it easier to use good style than bad?
    - Simply make it possible to do whatever you want?

- Design patterns and use of OO
- Other topics of interest??