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();	
	int getColor();	Public members
	void move(int);	
	void darken(int tint);	

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

Sample public 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

Public inheritance and subtyping

```
class ColorPt: public Pt {
  ….
};
```

ColorPt is a subtype of Pt.

I can write

```
Pt * p = new ColorPt;

// not so good
ColorPt cpt;
Pt p = cpt;
```

private derived class are not subtypes

```
class ColorPt: private Pt {
  ....
};
```

ColorPt is not a subtype of Pt.

I cannot write

```
Pt * p = new ColorPt;

ColorPt cpt;
Pt p = cpt;
```

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 … *inheritance breaks encapsulation*
    - *We will see "duck subtyping"*

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 ...

In C++ - from 1998

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

```
class A{
public:
  virtual A * create() ...
};
class B : public A{
public:
  virtual B * create() … // overriding
};
```

Subtyping with functions

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

```
class ColorPoint: public Point {
   public:
      int getX();          // Inherited, but repeated here for clarity
      int getColor();
      ColorPoint * move(int);
      void darken(int);
   protected:    ...
   private:      ...
};
```

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

This is covariant case; contravariance is another story

In Java

- Covarianza del tipo restituito, già visto

Slicing - attenzione

```
class A {
   int foo;
};
class B : public A
{
   int bar;
};
```

- So an object of type B has two data members, foo and bar

- Polimorfism does not work without pointers, but copy constructor:
- B b;
- A a = b

- a will have only the foo attribute ! The member bar of b is lost

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)

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
- Shape is abstract supertype of circle, rectangle, ...

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