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
  1. Objects, with dynamic lookup of virtual functions
  1. 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();
void move(int);

Public members

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

Public members

- 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?

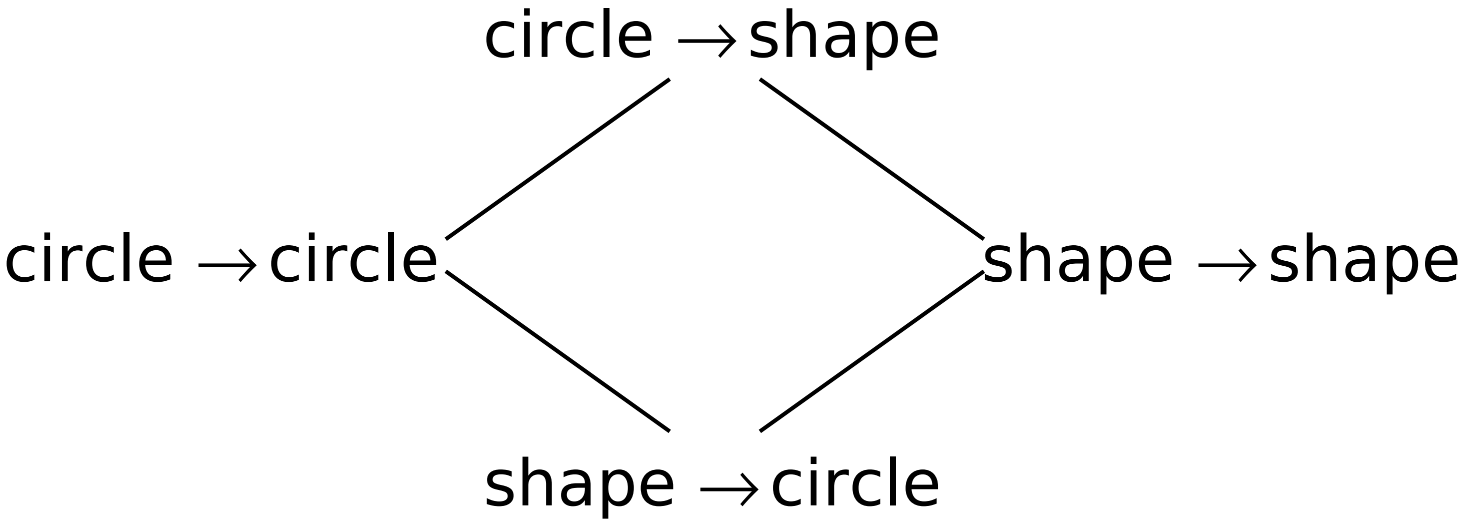
- Client code depends only on public interface
  - In principle, if ColorPt interface contains Pt interface, then any client could use ColorPt in place of point
  - However -- offset in virtual function table may differ
  - Lose implementation efficiency
- Without link to inheritance
  - subtyping leads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

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
  - If A <: B, then $C \to A \;<:\; C \to B$
- Subtyping for function arguments
  - If A <: B, then $B \to C \;<:\; A \to C$
- Terminology
  - Covariance: A <: B implies F(A) <: F(B)
  - Contravariance: A <: B implies F(B) <: F(A)

Examples

- If circle <: shape, then



circle → shape

circle → circle

shape → shape

shape → circle

C++ compilers recognize limited forms of function subtyping

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

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**

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

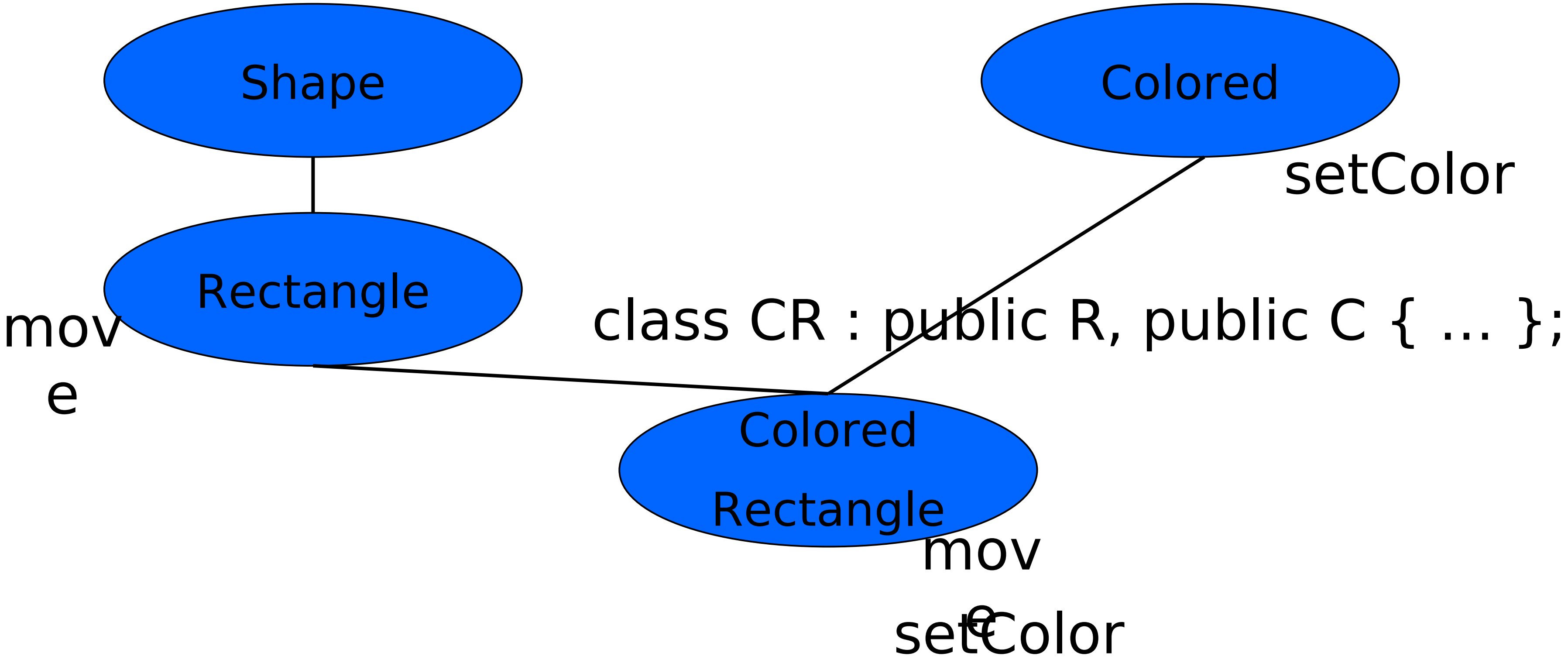
Inlining is very significant for efficiency; enables further optimization.

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

Multiple Inheritance



class CR : public R, public C { … };

Inherit independent functionality from independent classes

Problem: Name Clashes

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

same name
in 2 base
classes

Possible solutions to name clash

- Three general approaches
  - Implicit resolution
    - Language resolves name conflicts with arbitrary rule
  - Explicit resolution
    - Programmer must explicitly resolve name conflicts
  - Disallow name clashes
    - Programs are not allowed to contain name clashes
- No solution is always best
- C++ uses explicit resolution by using fully qualified names

Repair to previous example

- Rewrite class C to call A::f explicitly

```
 class C : public A, public B {
       public:
             void virtual f( ) {
                         A::f( );    // Call A::f(), not
   B::f();
             }
```

- Reasonable solution
  - This eliminates ambiguity
  - Preserves dependence on A

vtable for Multiple Inheritance

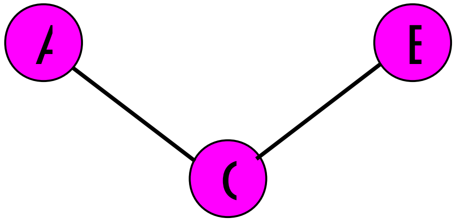
```
class A {
    public:
        int x;
        virtual void f();
};
class B {
    public:
        int y;
        virtual void g();
```

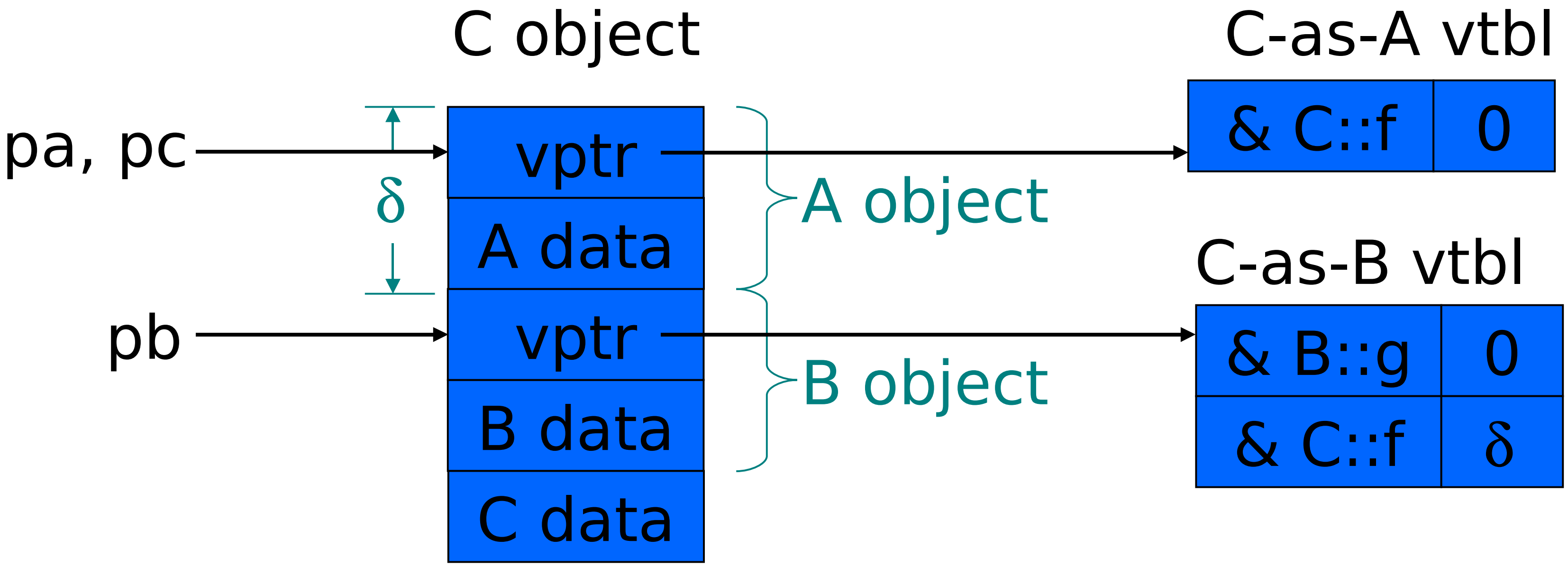
```
class C: public A, public B {
    public:
        int z;
        virtual void f();
};
```

```
C *pc = new C;
B *pb = pc;
A *pa = pc;
```

Three pointers to same object, but different static

Object and classes

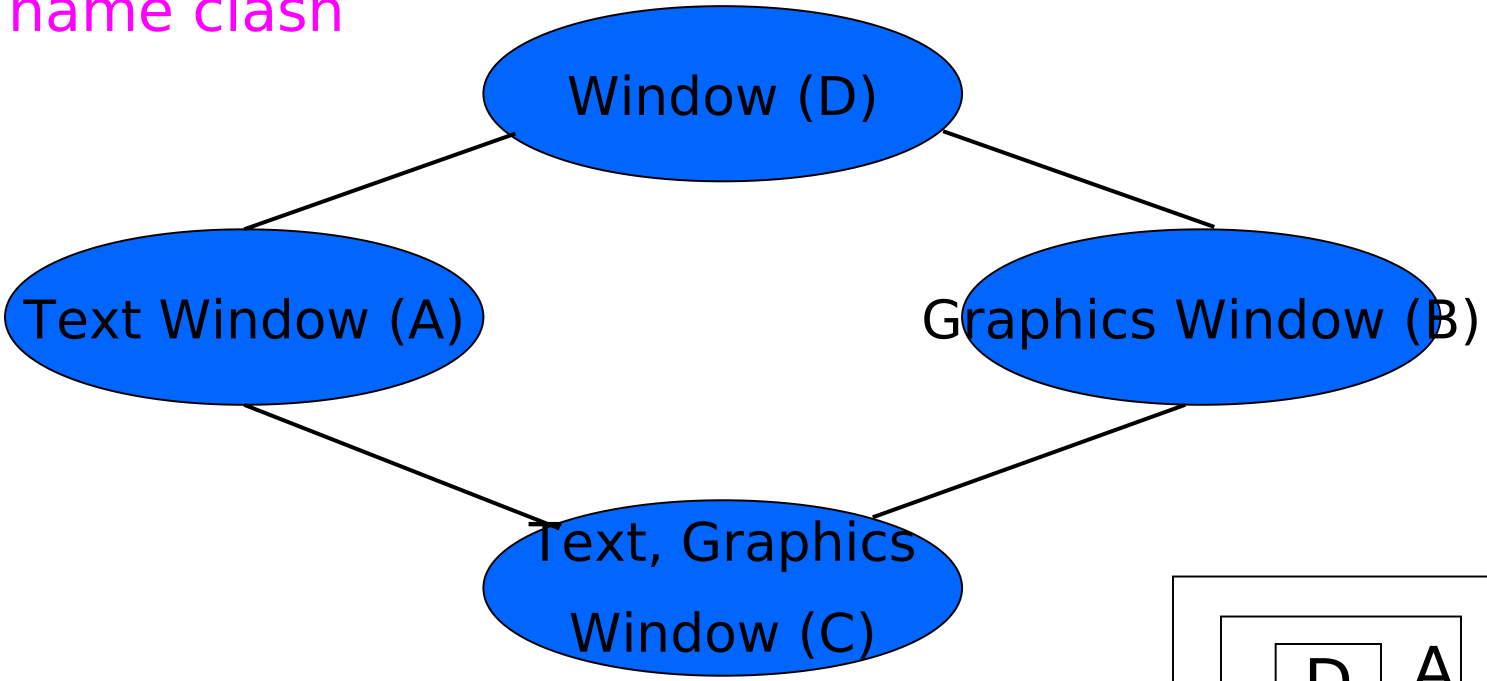


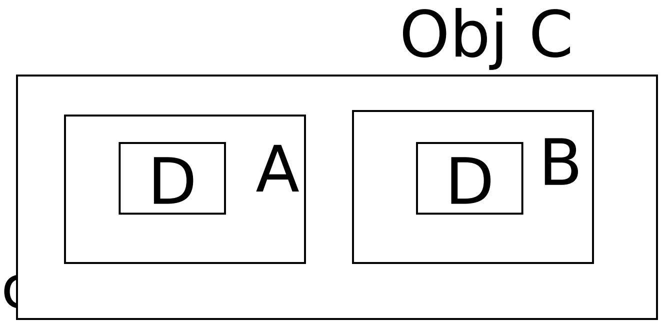


- Offset δ in vtbl is used in call to pb->f, since C::f may refer to A data that is above the pointer pb
- Call to pc->g can proceed through C-as-B vtbl

Multiple Inheritance “Diamond”

The *diamond inheritance* Problem: an interesting kind of name clash





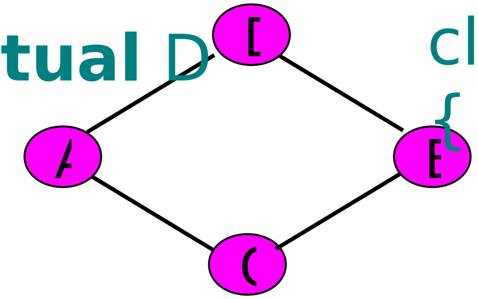
- The implementation is inherited twic
- C objects consist of two windows, one capable of displaying text and the other capable of displaying graphics!

A solution: virtual base classes

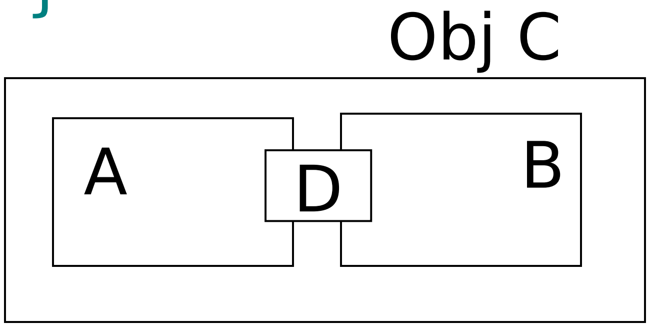
- C++ has a mechanism for eliminating multiple copies of duplicated base-class members,

- called ***virtual base classes*** and consists in declaring D as virtual base class of A and B

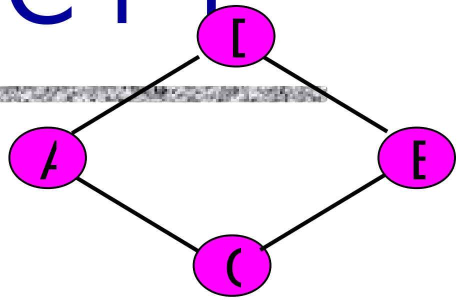
```
class A : public virtual D
{ … }
```



```
class B : public virtual D
{ … }
```



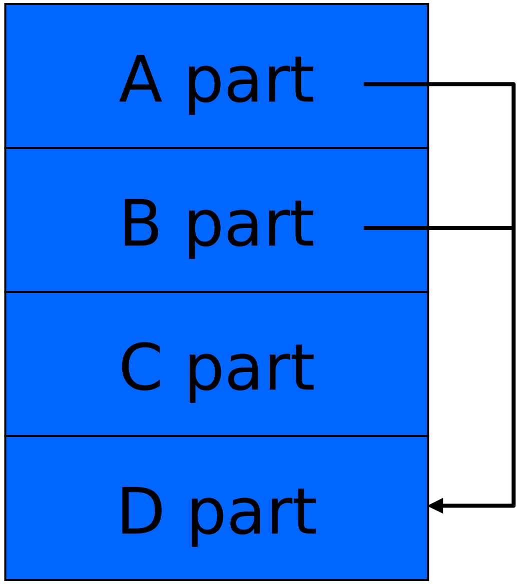
Diamond inheritance in C++



- 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


A part

B part

C part

D part

- C++ multiple inheritance is complicated in part because of desire to maintain efficient lookup
- Virtual base classes give rise to other type conversion problems

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??