Introduction to C++

Vitaly Shmatikov

Reading Assignment

- Mitchell, Chapter 12

History

- C++ is an object-oriented extension of C
- Designed by Bjarne Stroustrup at Bell Labs
  - His original interest at Bell Labs was research on simulation
  - Early extensions to C are based primarily on Simula
  - Called “C with classes” in early 1980s
  - Features were added incrementally
    - Classes, templates, exceptions, multiple inheritance, type tests...

Design Goals

- Provide object-oriented features in C-based language, without compromising efficiency
  - Backwards compatibility with C
  - Better static type checking
  - Data abstraction
  - Objects and classes
  - Prefer efficiency of compiled code where possible
- Important principle
  - If you do not use a feature, your compiled code should be as efficient as if the language did not include the feature (compare to Smalltalk)

How Successful?

- Many users, tremendous popular success
- Given the design goals and constraints, very well-designed language
- Very complicated design, however
  - Many features with complex interactions
  - Difficult to predict from basic principles
  - Most serious users chose subset of language
    - Full language is complex and unpredictable
  - Many implementation-dependent properties

Significant Constraints

- C has specific machine model
  - Access to underlying architecture (BCPL legacy)
- No garbage collection
  - Consistent with the goal of efficiency
  - Need to manage object memory explicitly
- Local variables stored in activation records
  - Objects treated as generalization of structs
    - Objects may be allocated on stack and treated as l-values
    - Stack/heap difference is visible to programmer

Non-Object-Oriented Additions

- Function templates (generic programming)
- Pass-by-reference
- User-defined overloading
- Boolean type

C++ Object System

- Classes
- Objects
  - With dynamic lookup of virtual functions
- Inheritance
  - Single and multiple inheritance
  - Public and private base classes
- Subtyping
  - Tied to inheritance mechanism
- Encapsulation

Good Decisions

- Public, private, protected levels of visibility
  - Public: visible everywhere
  - Protected: within class and subclass declarations
  - Private: visible only in class where declared
- Friend functions and classes
  - Careful attention to visibility and data abstraction
- Allow inheritance without subtyping
  - Private and protected base classes
  - Useful to separate subtyping and inheritance (why?)

Problem Areas

- Casts
  - Sometimes no-op, sometimes not (multiple inheritance)
- Lack of garbage collection
- 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

Sample Class: Points

```
class Pt {
   public:
      Pt(int xv);              Overloaded constructor
      Pt(Pt* pv);
      int getX();              Public read access to private data
      virtual void move(int dx);  Virtual function
    protected:
      void setX(int xv);       Protected write access
    private:
      int x;                   Private member data
 };
```

Virtual Functions

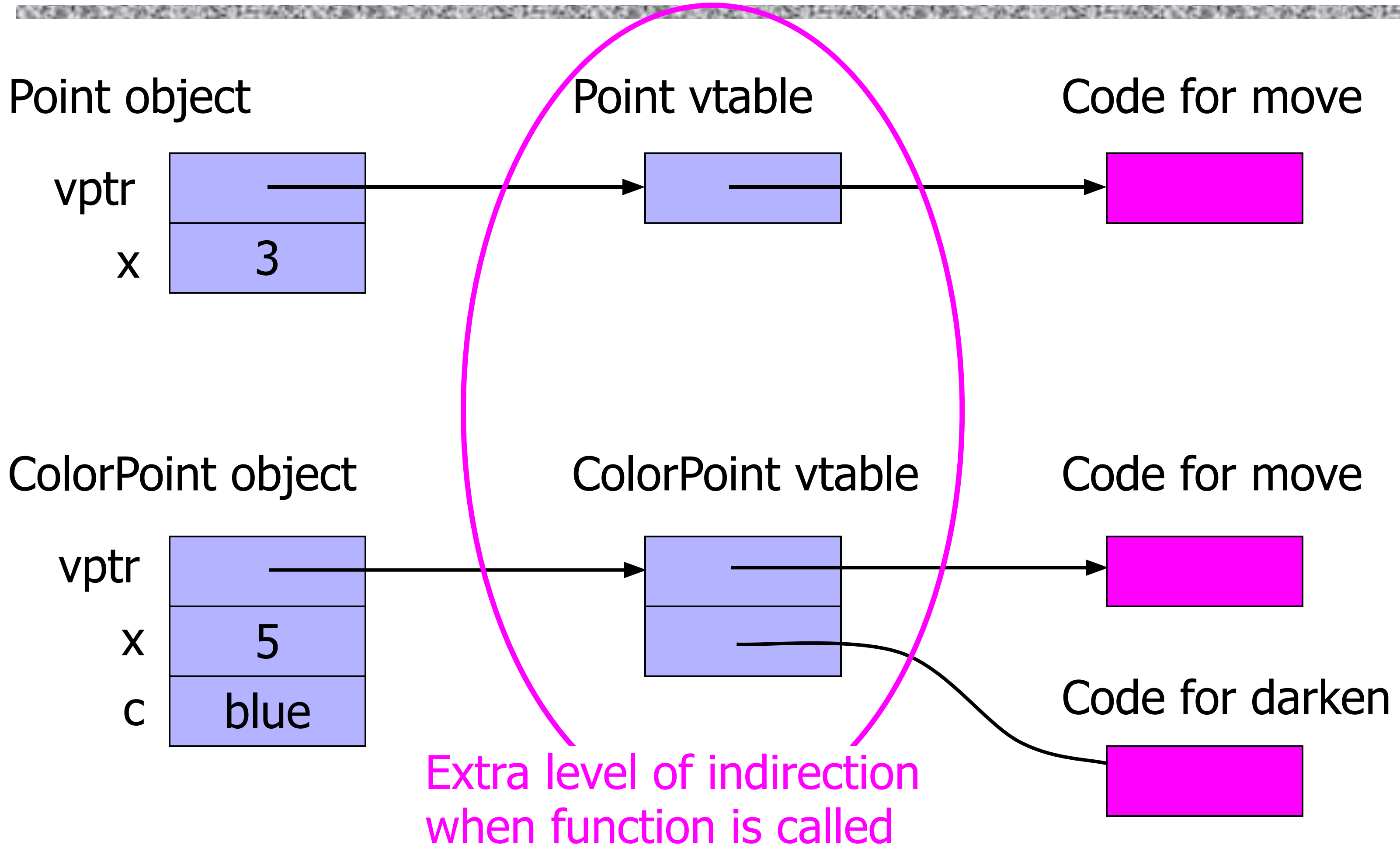
- Virtual member functions
  - Accessed by indirection through pointer in object
  - May be redefined in derived subclasses
  - The exact function to call determined <u>dynamically</u>
- Non-virtual functions are ordinary functions
  - Cannot redefine in subclasses (but can overload)
- Member functions are virtual if explicitly declared or inherited as virtual, else non-virtual
- Pay overhead only if you use virtual functions

Sample Derived Class: Color Point

```
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; };
```

- `class ColorPt: public Pt {` — Public base class gives supertype
- `ColorPt(int xv,int cv);`, `ColorPt(Pt* pv,int cv);`, `ColorPt(ColorPt* cp);` — Overloaded constructor
- `int getColor();` — Non-virtual function
- `virtual void move(int dx);`, `virtual void darken(int tint);` — Virtual functions
- `void setColor(int cv);` — Protected write access
- `int color; };` — Private member data

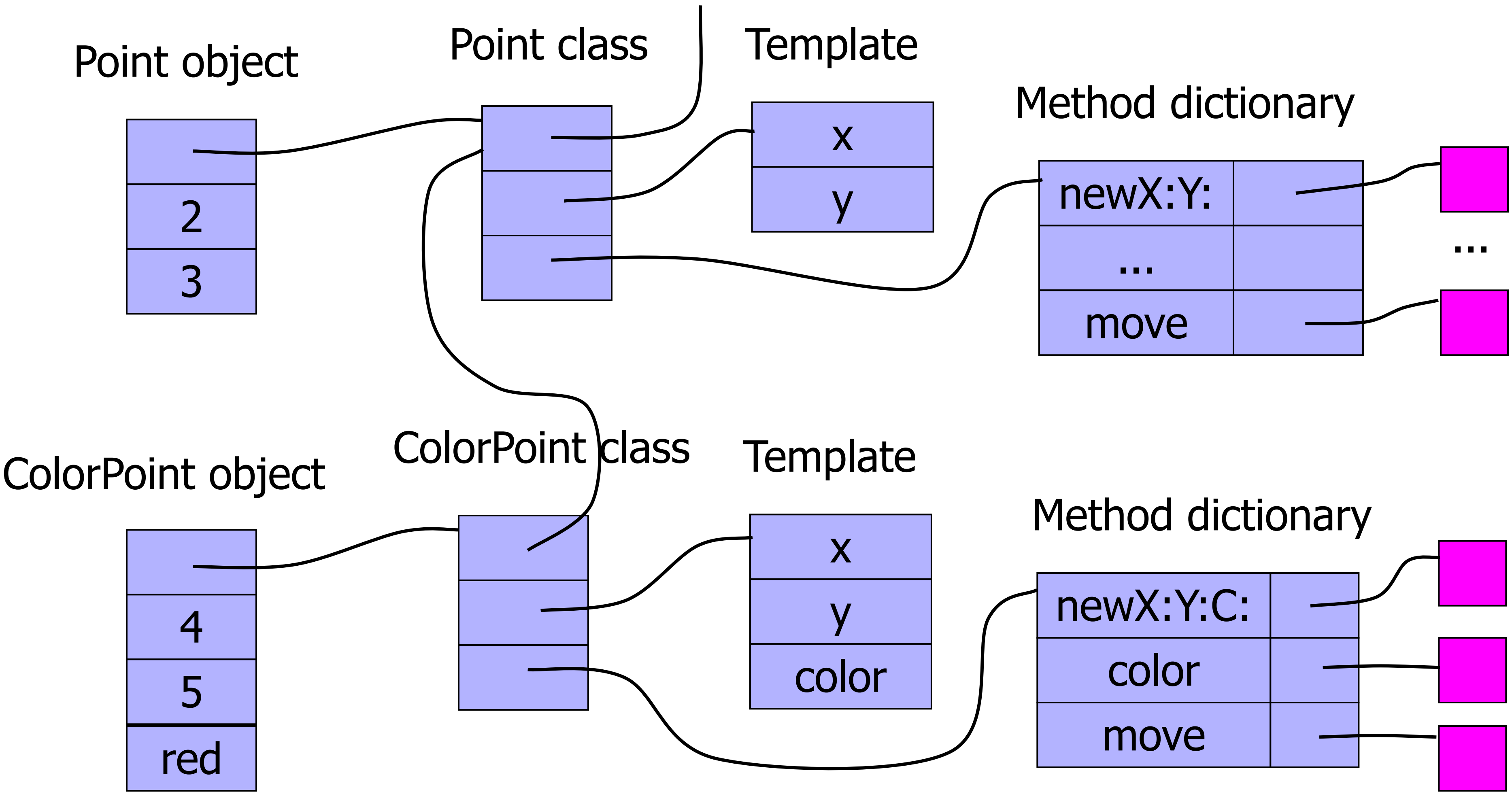
Run-Time Representation



Data at same offset

Function pointers at same offset

Compare to Smalltalk



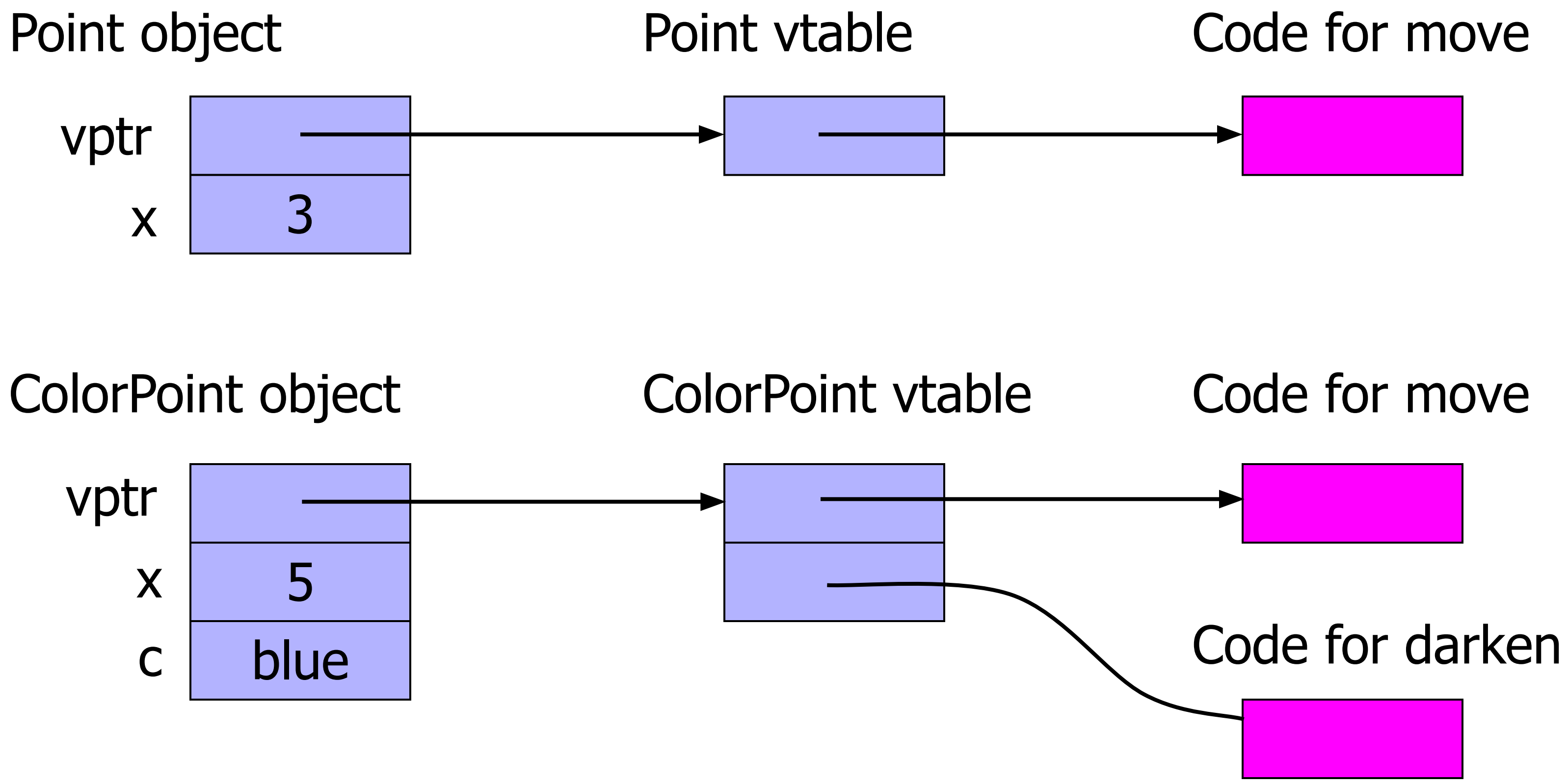
Why Is C++ Lookup Simpler?

- Smalltalk has no static type system
  - Code p message:params could refer to any object
  - Need to find method using pointer from object
  - Different classes will put methods at different places in the method dictionary
- C++ type gives compiler some superclass (how?)
  - Offset of data, function pointers is the same in subclass and superclass, thus known at compile-time
  - Code p->move(x) compiles to equivalent of

  (*(p->vptr[0]))(p,x) if move is first function in vtable

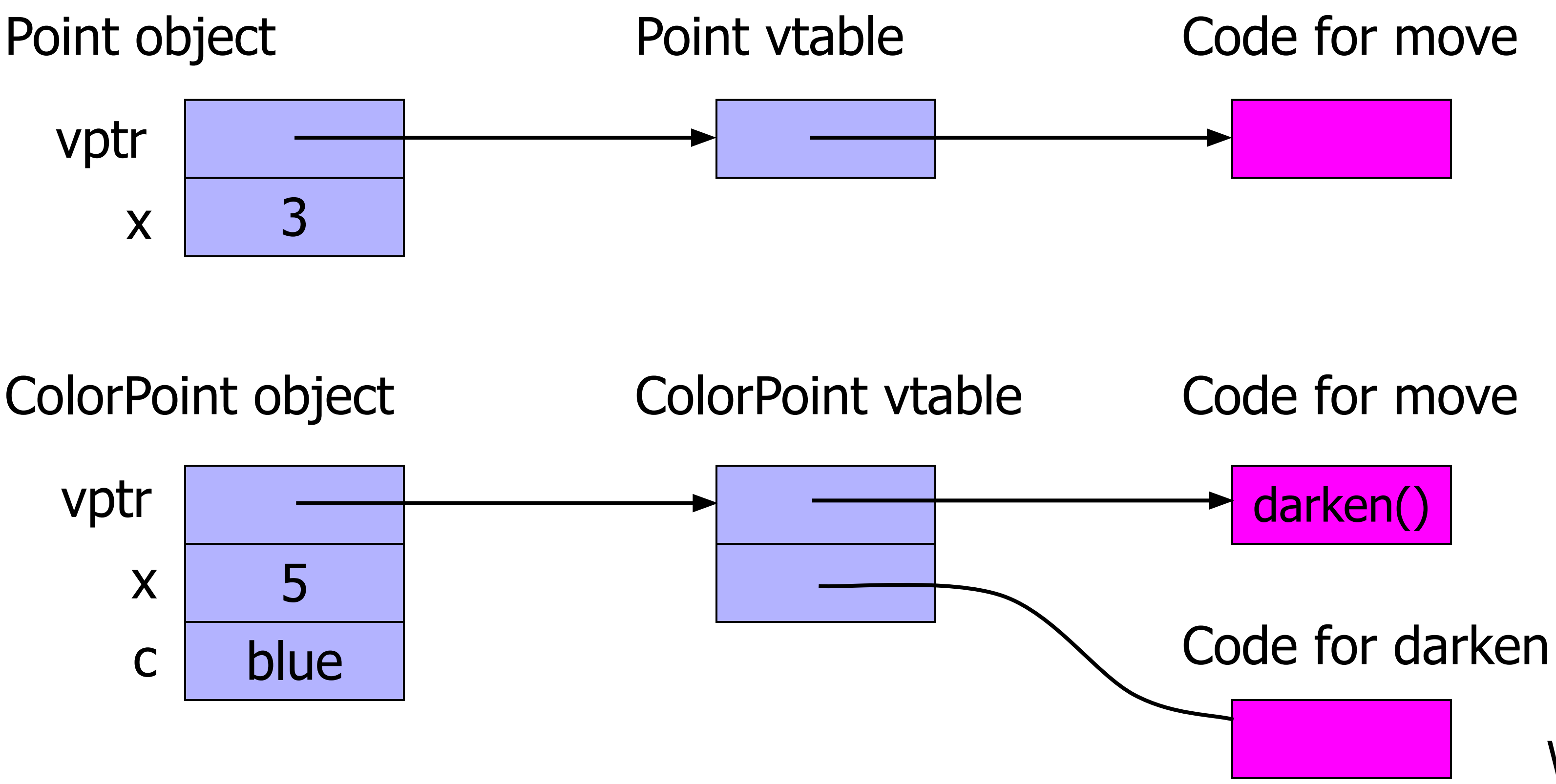
data passed to member function

Looking Up Methods (1)




```
Point p = new Pt(3);
p->move(2);          // Compiles to equivalent of (*(p->vptr[0]))(p,2)
```

Looking Up Methods (2)



Point cp = new ColorPt(5,blue);

cp->move(2);   // Compiles to equivalent of (*(cp->vptr[0]))(cp,2)

What is this for?

Calls to Virtual Functions

- One member function may call another

```
class A {
    public:
        virtual  int  f (int x);
        virtual  int  g(int y);
};
int A::f(int x) { … g(i) …;}
int A::g(int y) { … f(j) …;}
```

- How does body of f call the <u>right</u> g?
  - If g is redefined in derived class B, then inherited f must call B::g

“This” Pointer

- Code is compiled so that member function takes object itself as first argument

  Code int A::f(int x) { … g(i) …;}

  compiled as int A::f(A *this, int x) { … this->g(i) …;}

- “this” pointer may be used in member function
  - Can be used to return pointer to object itself, pass pointer to object itself to another function, etc.
- Analogous to “self” in Smalltalk

Non-Virtual Functions

- How is code for non-virtual function found?
- Same way as ordinary functions:
  - Compiler generates function code and assigns address
  - Address of code is placed in symbol table
  - At call site, address is taken from symbol table and placed in compiled code
  - But some special scoping rules for classes
- Overloading
  - Remember: overloading is resolved at compile time
  - Different from run-time lookup of virtual function

Scope Rules in C++

- Scope qualifiers: binary :: operator, ->, and .
  - class::member, ptr->member, object.member
- A name outside a function or class, not prefixed by unary :: and not qualified refers to global object, function, enumerator or type
- A name after X::, ptr-> or obj. refers to a member of class X or a base class of X
  - Assume ptr is pointer to class X and obj is an object of class X

Virtual vs. Overloaded Functions

```
class parent { public:
    void printclass() {printf("p ");};
    virtual void printvirtual() {printf("p ");};   };
class child : public parent { public:
    void printclass() {printf("c ");};
    virtual void printvirtual() {printf("c ");};    };
main() {
    parent p;  child c;  parent *q;
    p.printclass(); p.printvirtual(); c.printclass(); c.printvirtual();
    q = &p;  q->printclass(); q->printvirtual();
    q = &c;  q->printclass(); q->printvirtual();
}
```

Output: p p c c p p p c

Subtyping

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

```
Point:        int getX();
              void move(int);           } Public members
ColorPoint:   int getX();
              int getColor();
              void move(int);           } Public members
              void darken(int tint);
```

- C++: A <: B if class A has public base class B
  - This is weaker than necessary (why?)

No Subtyping Without Inheritance

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

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

- C++ does not treat ColorPoint <: Point (as written)
  - Unlike Smalltalk!
  - Need public inheritance ColorPoint : public Point (why?)

Why This Design Choice?

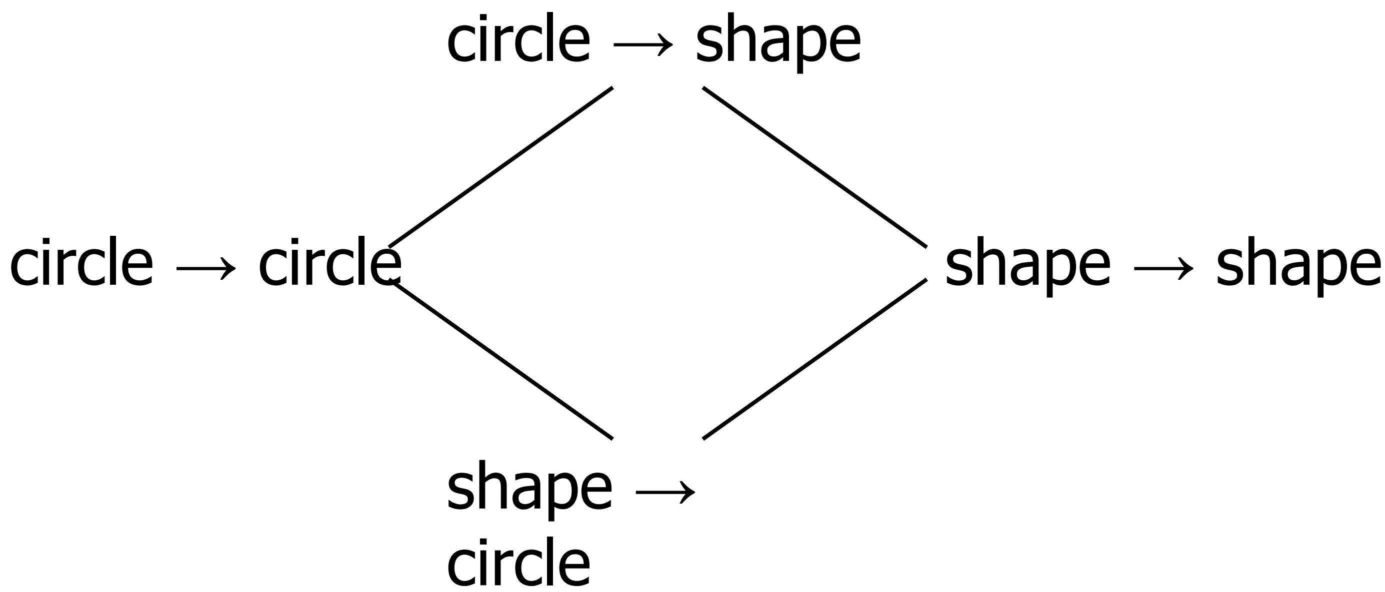
- Client code depends only on public interface
  - In principle, if ColorPoint interface contains Point interface, then any client could use ColorPoint in place of Point (like Smalltalk)
  - But offset in virtual function table may differ, thus lose implementation efficiency (like Smalltalk)
- Without 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

- 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 $C \rightarrow A <: C \rightarrow B$
  - Covariant: A <: B implies F(A) <: F(B)
- Subtyping for function arguments
  - If A <: B, then $B \rightarrow C <: A \rightarrow C$
  - Contravariant: A <: B implies F(B) <: F(A)

Examples

- If circle <: shape, then



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:      ...
};
```

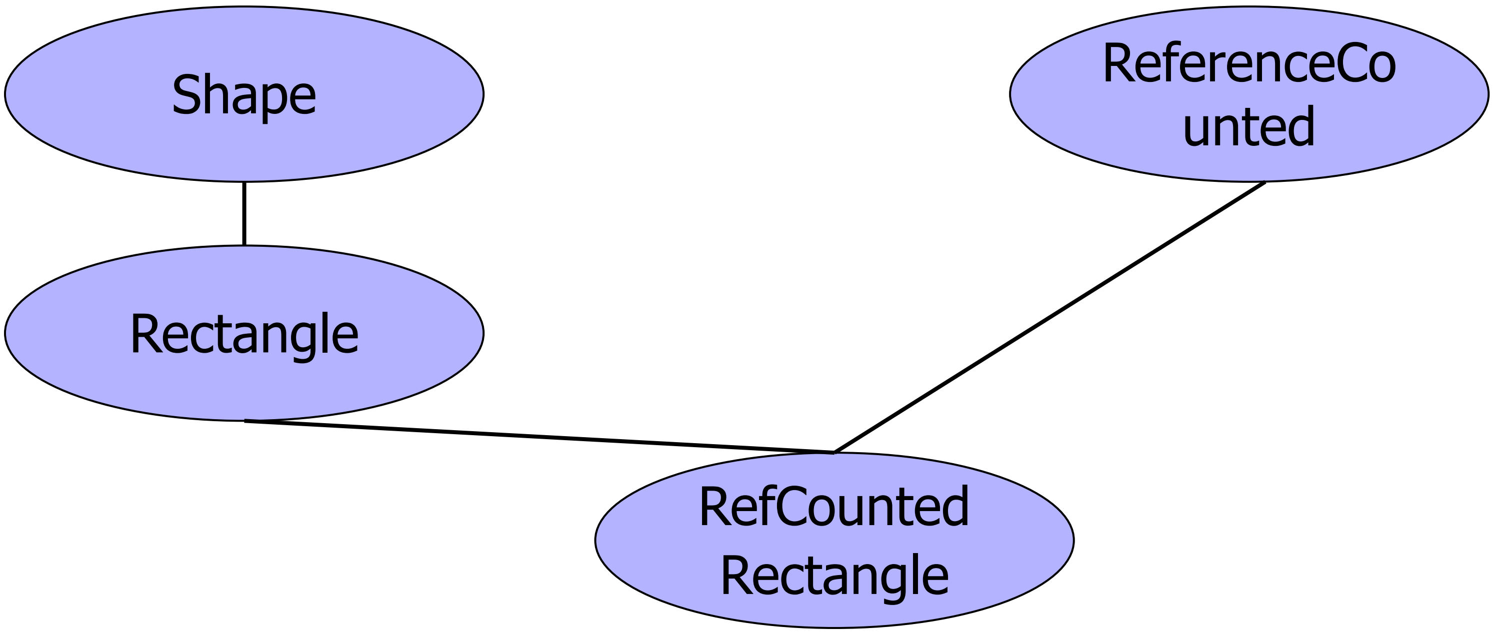
Inherited, but repeated here for clarity

- In principle: can have ColorPoint <: Point
- In practice: some compilers allow, others not
  - This is covariant case; contravariance is another story

Abstract Classes

- Abstract class: a class without complete implementation
- Declared by =0   (what a great syntax! ☺)
- 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)

Multiple Inheritance



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 two base classes

Solving Name Clashes

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

Explicit Resolution of Name Clashes

- 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();
        }
```

- Eliminates ambiguity
- Preserves dependence on A
  - Changes to A::f will change C::f

vtable for Multiple Inheritance

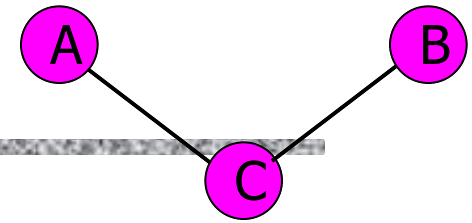
```
class A {
   public:
      int x;
      virtual void f();
};
class B {
   public:
      int y;
      virtual void g();
      virtual void f();
};
```

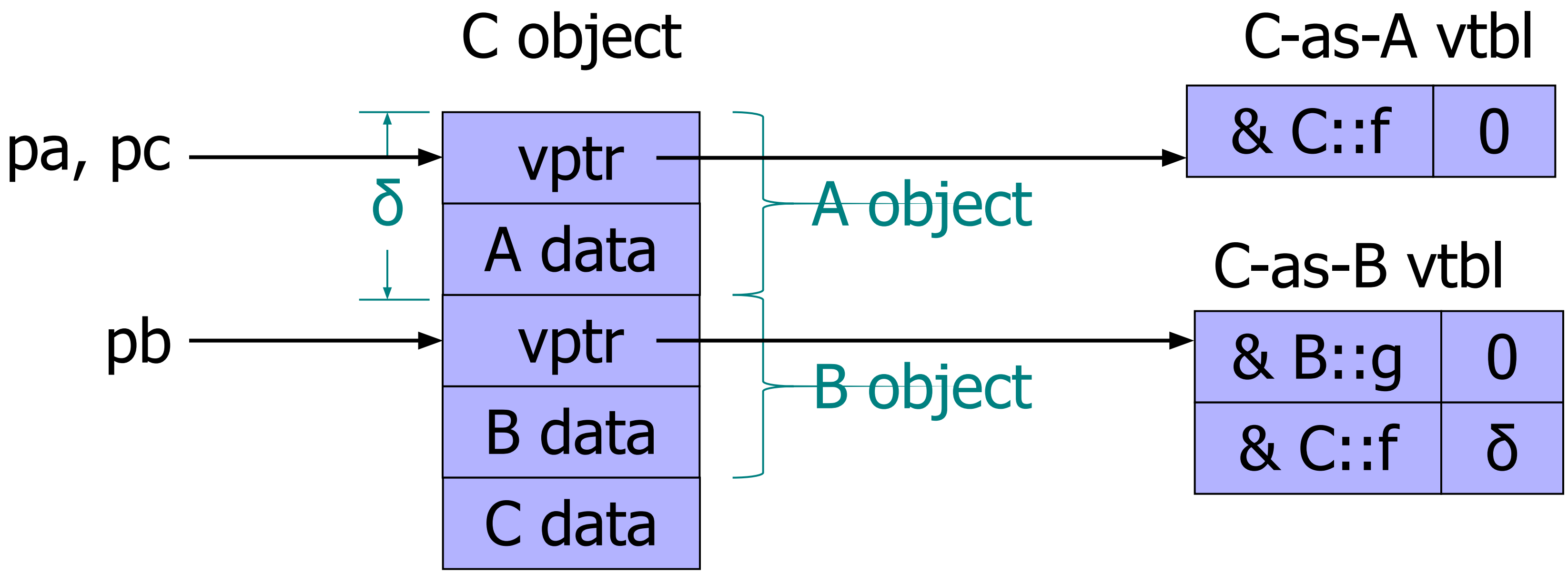
```
class C: public A, public B {
   public:
      int z;
      virtual void f();
};

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

Three pointers to same object, but different static types.

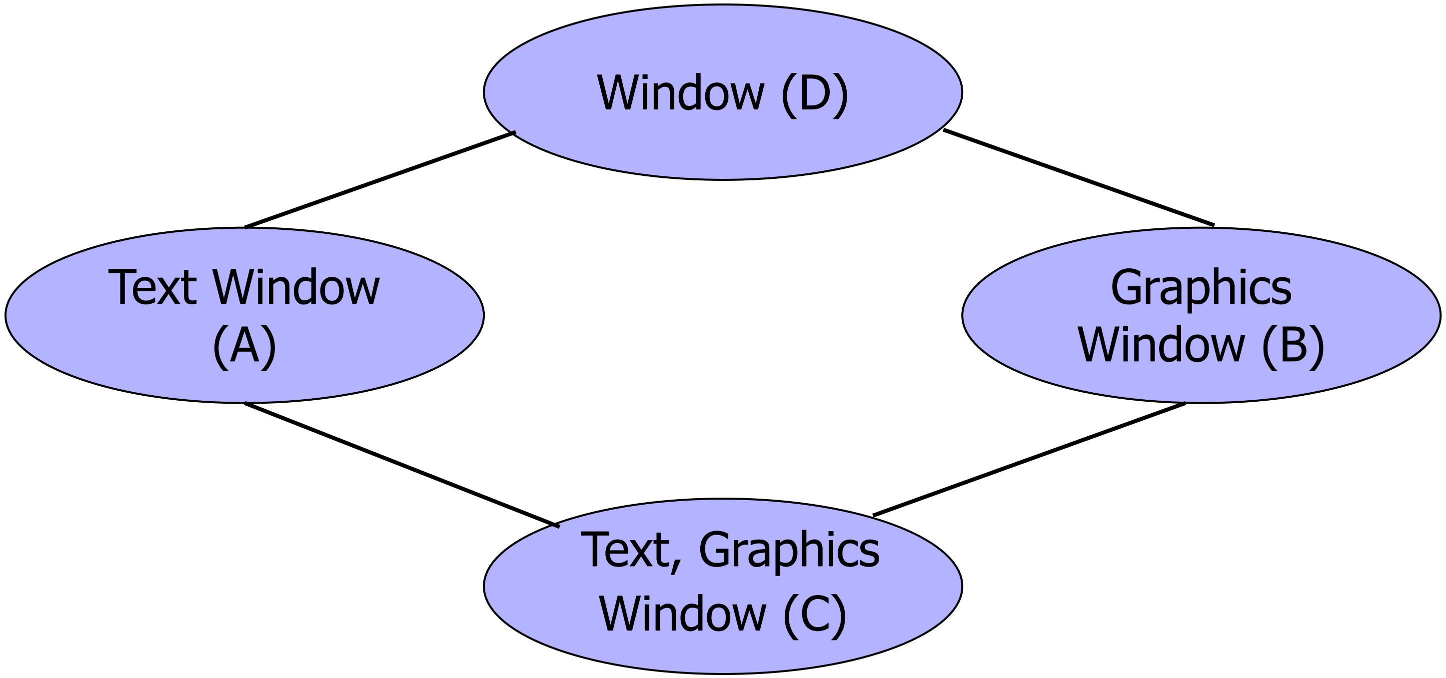
Object Layout





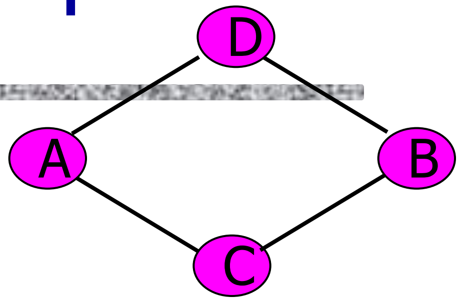
- Offset δ in vtbl is used in call to pb->f, since C::f may refer to A data that is above the pointer pb
- Call to pc->g can proceed through C-as-B vtbl

Multiple Inheritance "Diamond"



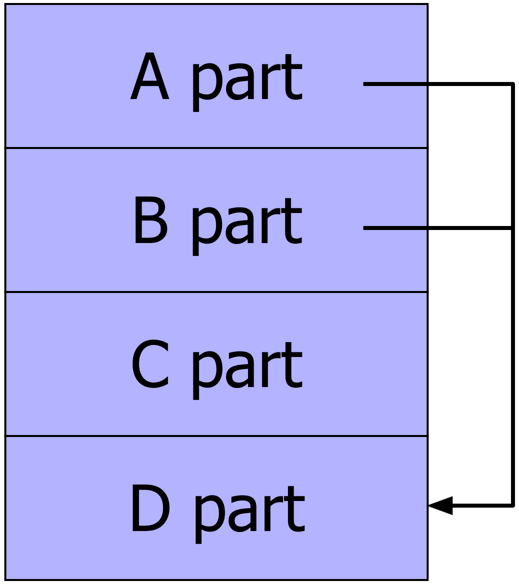
- Is interface or implementation inherited twice?
- What if definitions conflict?

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 part because of desire to maintain efficient lookup

C++ Summary

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