Objects, Encapsulation, Inheritance (2)

Reading (two lectures)

Chapter 10, except section 10.4

Chapter 11, sections 11.1, 11.2, 11.3.1 and 11.4., 11.5, 11.6 only

Chapter 12, sections 12.1, 12.2, 12.3 and 12.5 only

Chapter 13, sections 13.1 and 13.2 only

Outline

- Central concepts in object-oriented languages
  - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records
  - Simula – implementation as activation records with static scope
- Pure dynamically-typed object-oriented languages
  - Object implementation and run-time lookup
  - Class-based languages (Smalltalk)
  - Prototype-based languages (Self, JavaScript)
- Statically-typed object-oriented languages (this lecture)
  - C++ – using static typing to eliminate search
    - problems with C++ multiple inheritance
  - Java – using Interfaces to avoid multiple inheritance

C++ Background

- C++ is an object-oriented extension of C
- C was designed by Dennis Ritchie at Bell Labs
  - used to write Unix, based on BCPL
- C++ designed by Bjarne Stroustrup at Bell Labs
  - His original interest at Bell was research on simulation
  - Early extensions to C are based primarily on Simula
  - Called “C with classes” in early 1980’s
  - Popularity increased in late 1980’s and early 1990’s
  - Features were added incrementally

    Classes, templates, exceptions, multiple inheritance, type tests...

C++ 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?

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

Significant constraints

- C has specific machine model
  - Access to underlying architecture
- No garbage collection
  - Consistent with 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

C++ Object System

- 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

Some (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
  - Better control of subtyping than without private base classes

Some problem areas

- Casts
  - Sometimes no-op, sometimes not (e.g., multiple inheritance)
- Lack of garbage collection
  - Memory management is error prone
    - Constructors, destructors are helpful // smart pointers?
- Objects allocated on stack
  - Better efficiency, interaction with exceptions
  - But assignment works badly, possible dangling pointers
- Overloading
  - Too many code selection mechanisms?
- Multiple inheritance
  - Emphasis on efficiency leads to complicated behavior

Sample class: one-dimen. points

```
class Pt {
  public:
     Pt(int xv);
     Pt(Pt* pv);
     int getX();
     virtual void move(int dx);
  protected:
      void setX(int xv);
  private:
      int x;
  };
```

Overloaded constructor

Public read access to private data

Virtual function

Protected write access L

Private data

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

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;
 };
```

Public base class gives supertype

Overloaded constructor

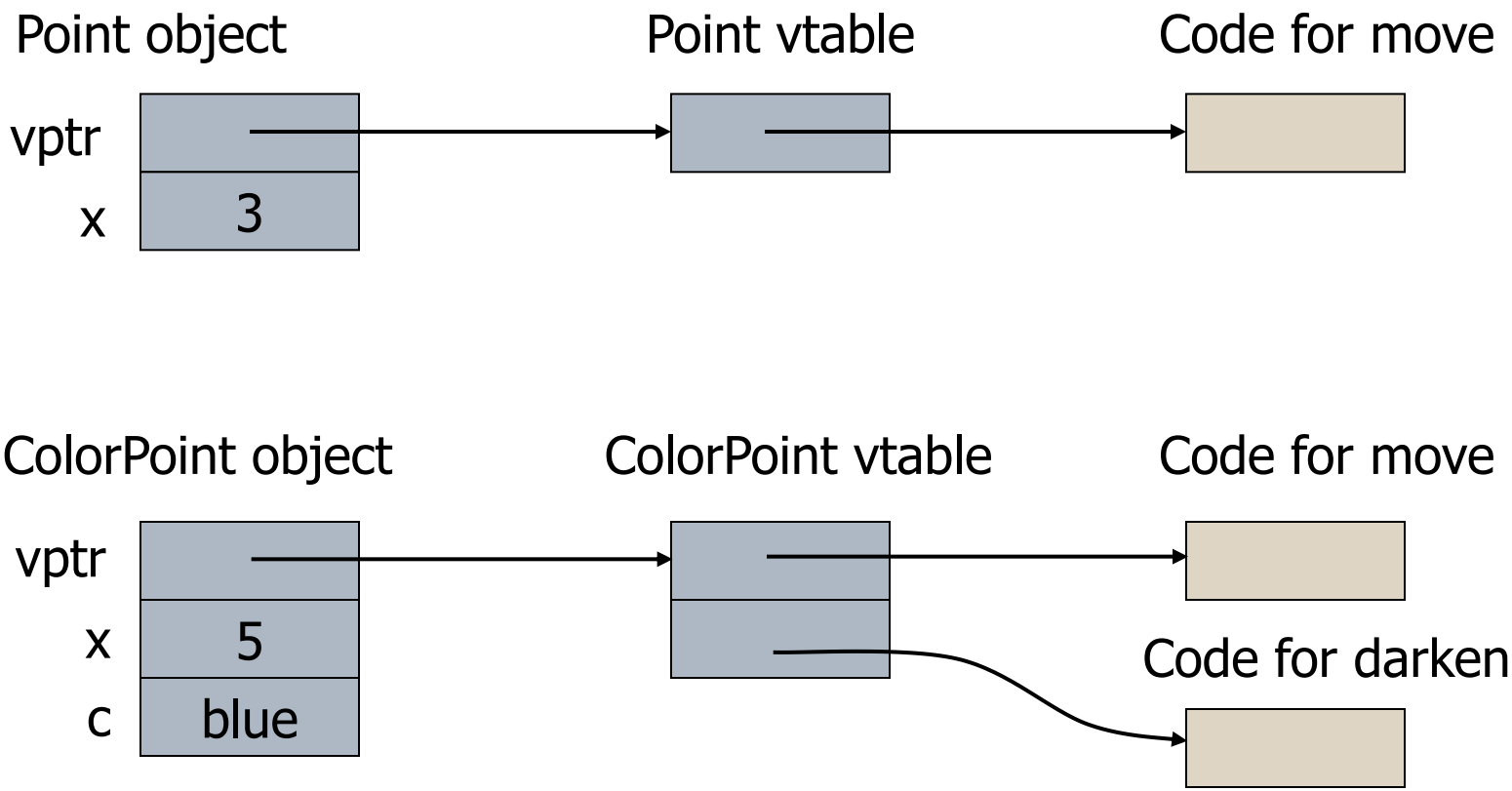
Non-virtual function

Virtual functions

Protected write access

Private data

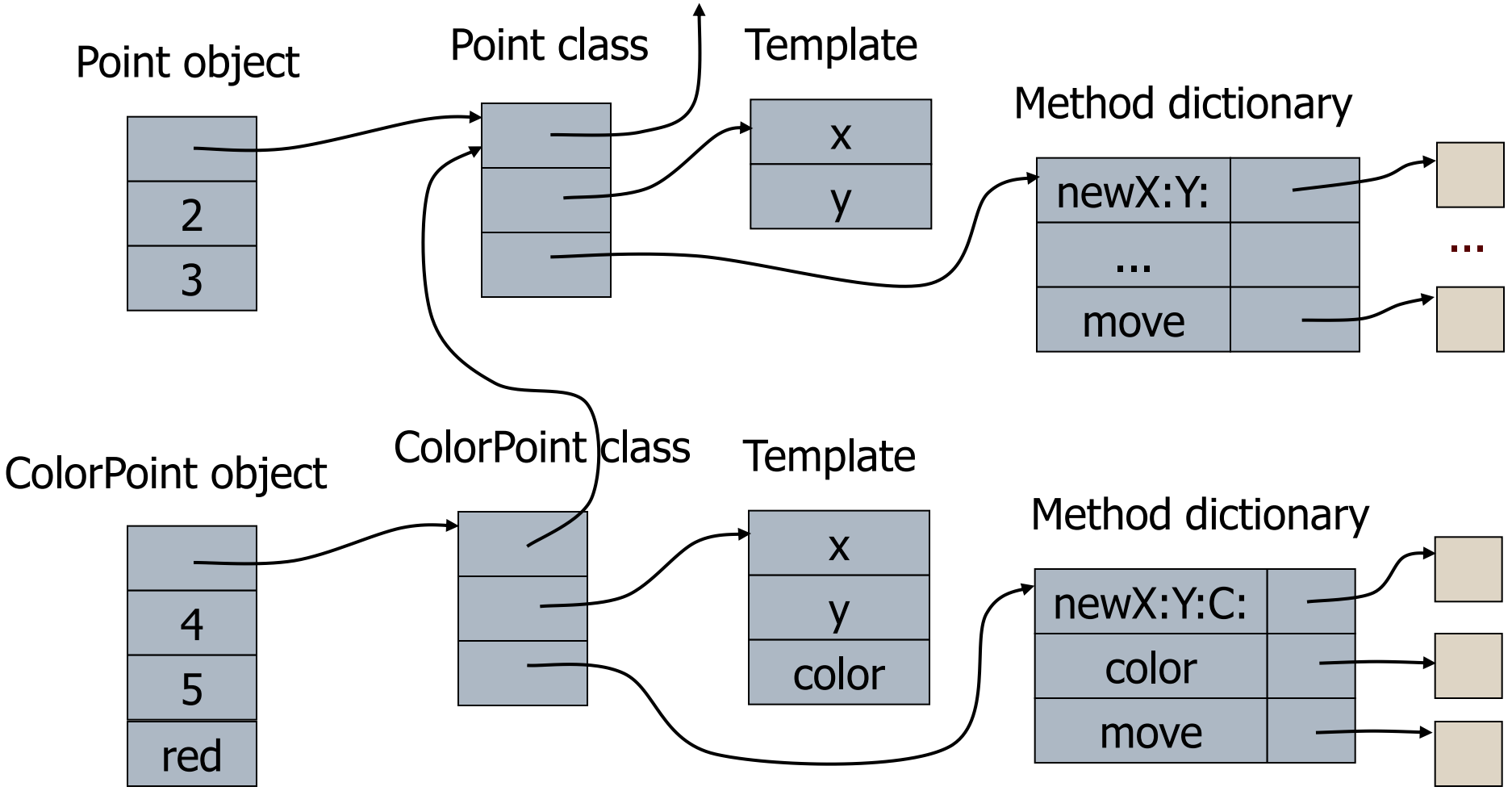
Run-time representation



Data at same offset

Function pointers at same offset

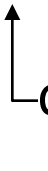
Compare to Smalltalk/JavaScript



Why is C++ lookup simpler?

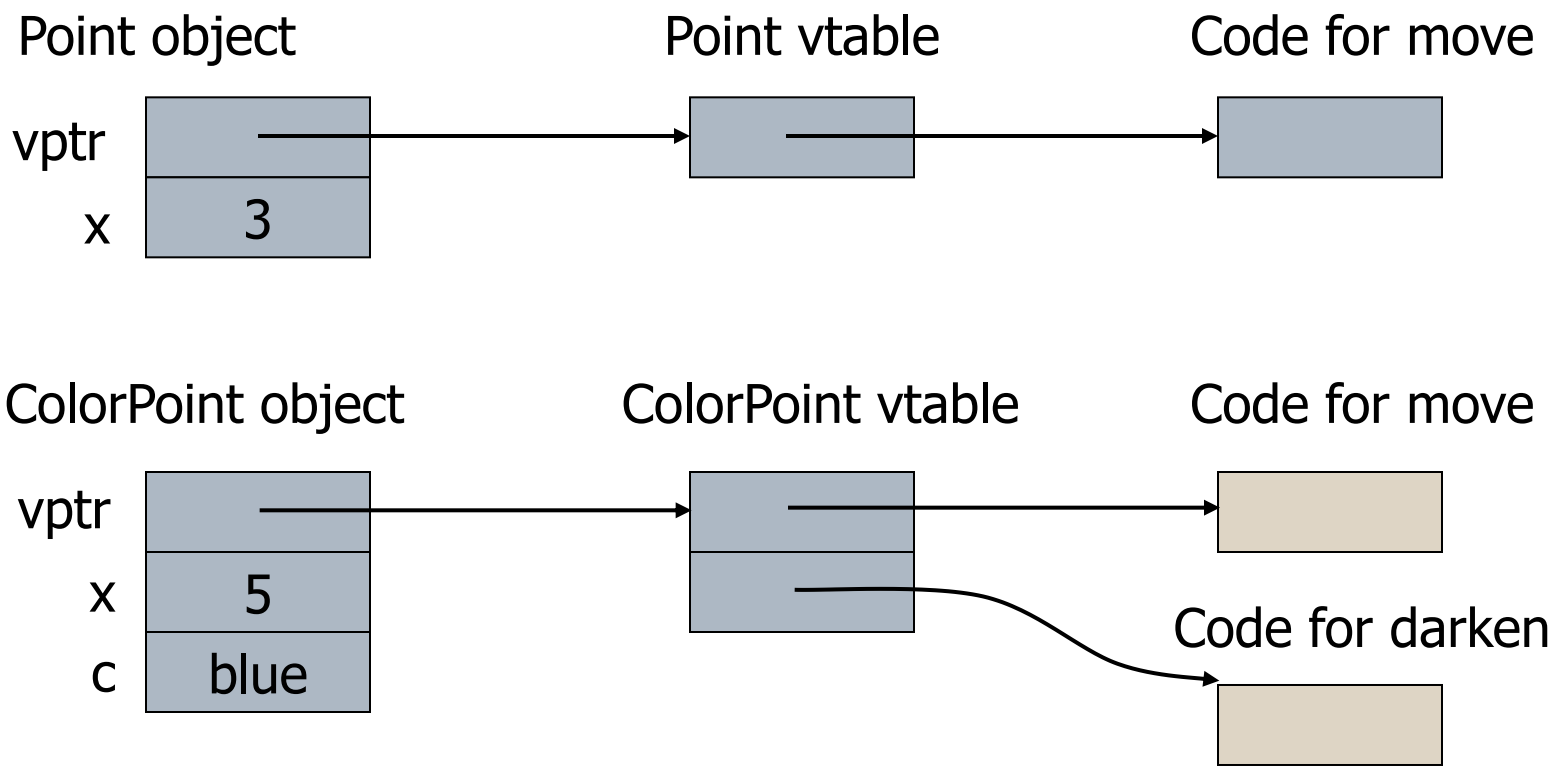
- Smalltalk/JavaScript have no static type system
  - Code obj.operation(parms) could refer to any object
  - Need to find method using pointer from object
  - Different classes will put methods at different place in method dictionary
- C++ type gives compiler some superclass
  - Offset of data, fctn ptr same in subclass and superclass
  - Offset of data and function ptr known at compile time
  - Code p->move(x) compiles to equivalent of

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



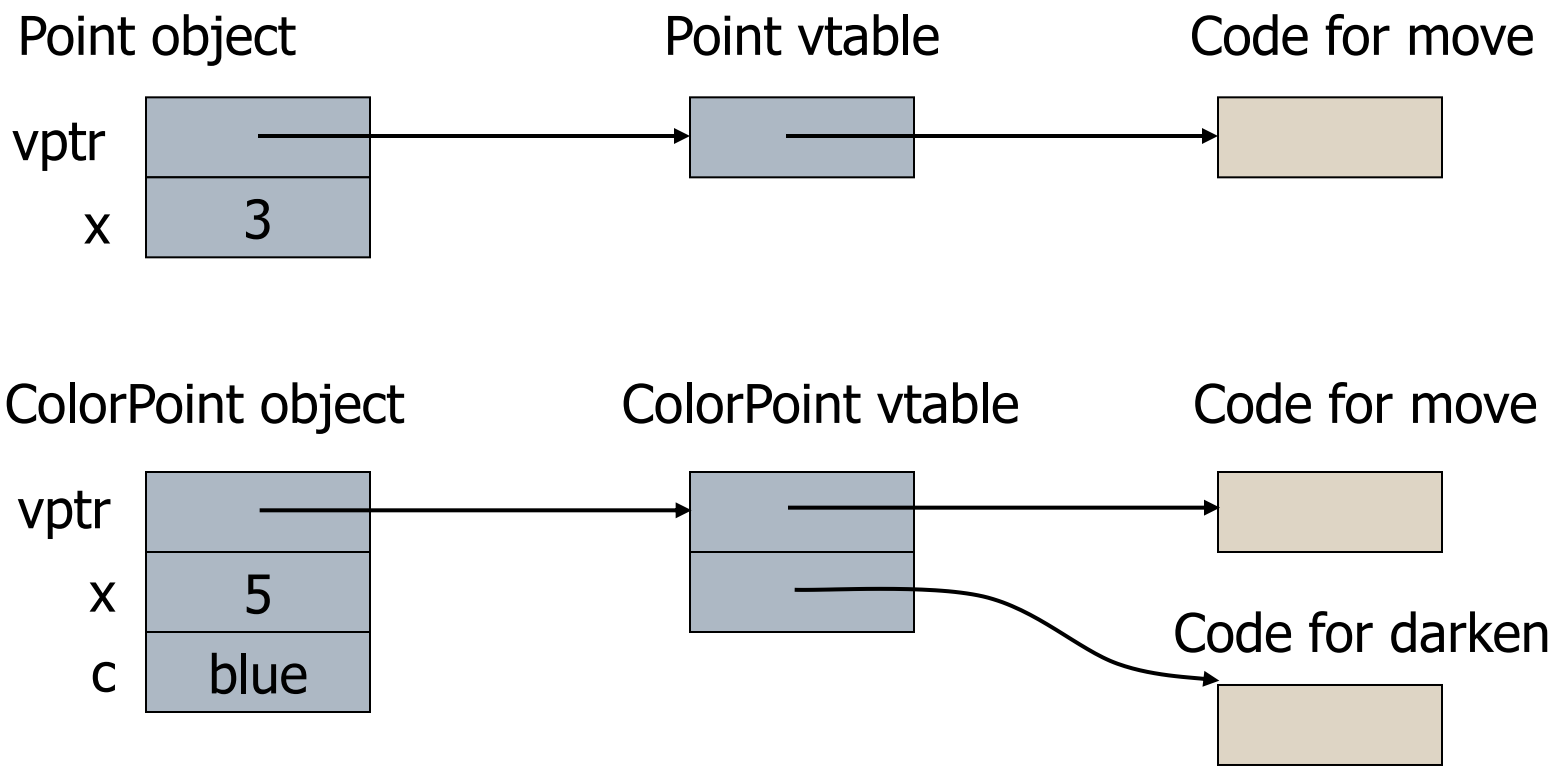
data passed to member function; see next slides

Looking up methods




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

Looking up methods, part 2




```
Point cp = new ColorPt(5,blue);
cp->move(2);          // (*(cp->vptr[0]))(cp,2)
```

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 right g?
  - If g is redefined in derived class B, then inherited f must call B::g

“This” pointer (*self* in Smalltalk)

- 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, ...

Non-virtual functions

- How is code for non-virtual function found?
- Same way as ordinary “non-member” 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
- Overloading
  - Remember: overloading is resolved at compile time
  - This is different from run-time lookup of virtual function

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();
}
```

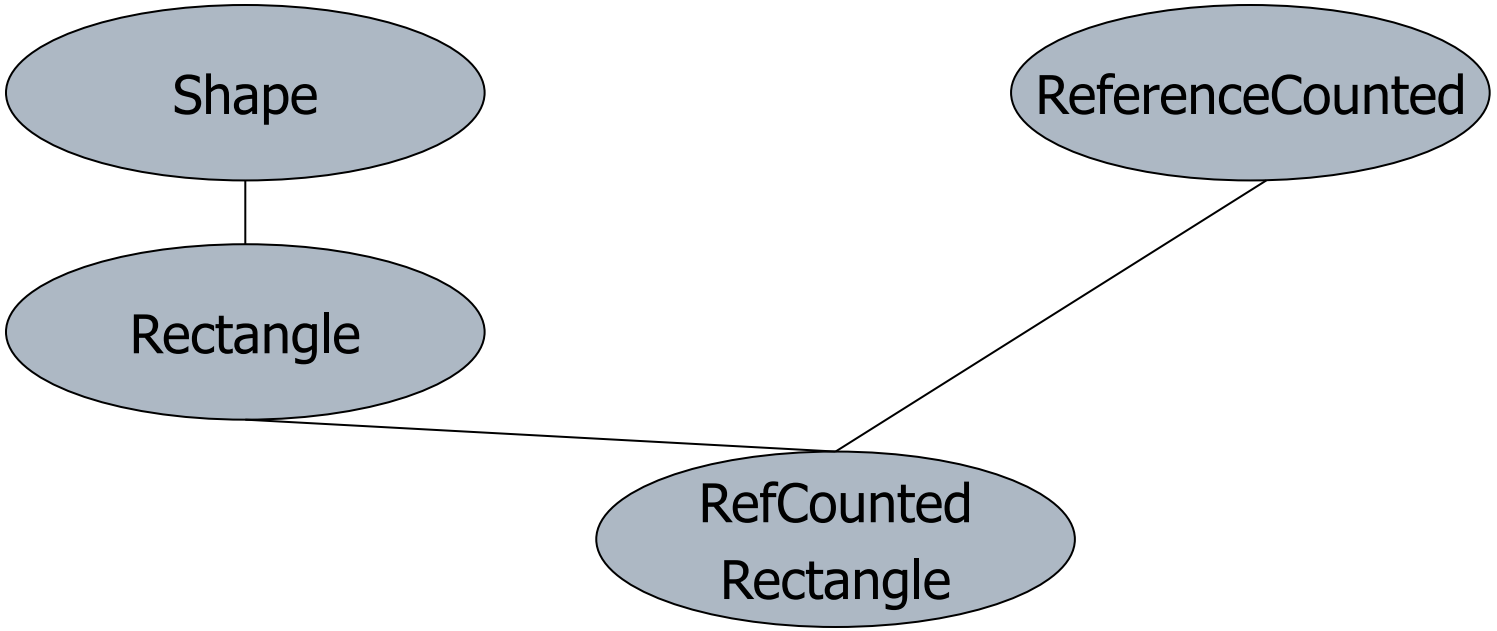
Output: p p c c p p ? ?

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

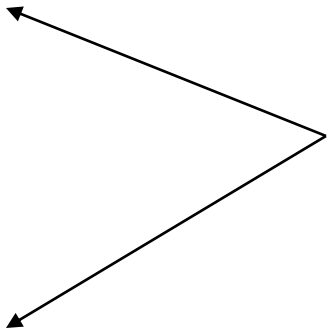
Multiple Inheritance



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 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

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
    - Changes to A::f will change C::f

vtable for Multiple Inheritance

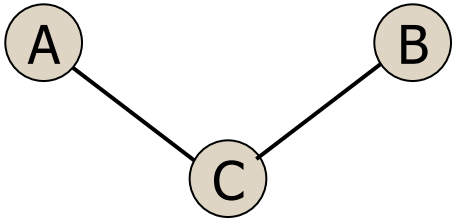
```
class A {
   public:
      int x;
      virtual void f();
};
class B {
  public:
      int y;
      virtual void g();
      virtual void f();
};
```

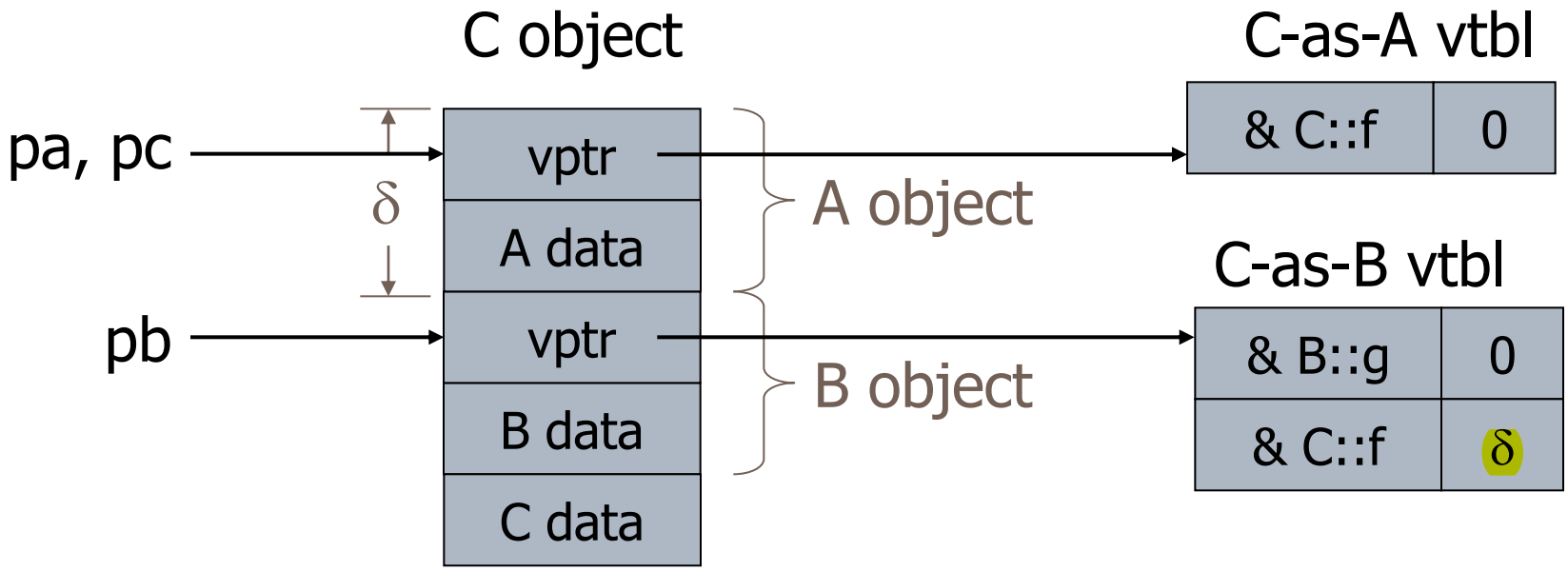
```
class C: public A, public B {
   public:
      int z;
      virtual void f();
};

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

Three pointers to same object, but different static types.

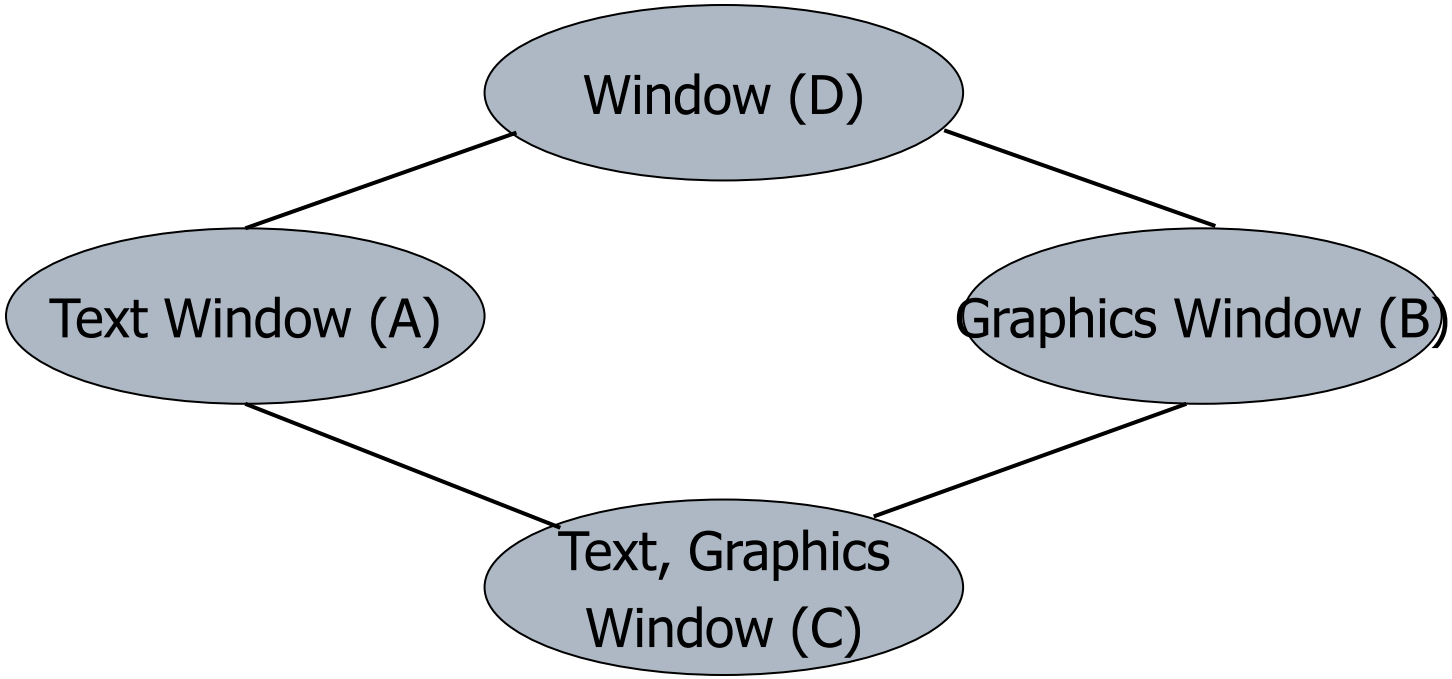
Object and classes





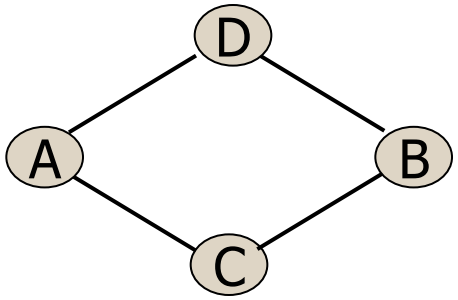
- Offset δ in vtbl is used in call to pb->f, since C::f may refer to A data that is above the pointer pb
- Call to pc->g can proceed through C-as-B vtbl

Multiple Inheritance “Diamond”



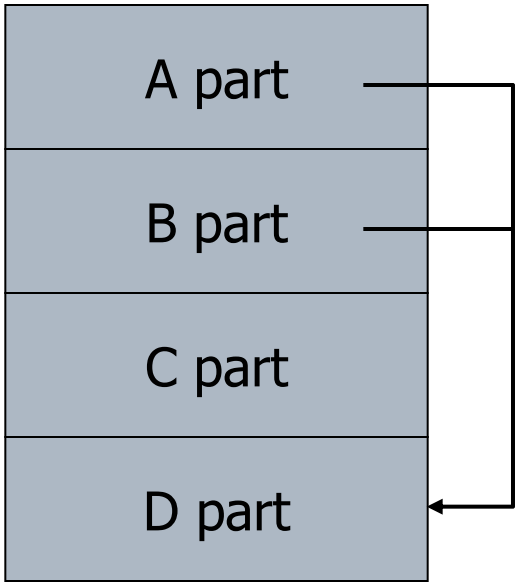
- Is interface or implementation inherited twice?
- What if definitions conflict?

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

because of desire to maintain efficient lookup

Outline

- Central concepts in object-oriented languages
  - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records
  - Simula – implementation as activation records with static scope
- Pure dynamically-typed object-oriented languages
  - Object implementation and run-time lookup
  - Class-based languages (Smalltalk)
  - Prototype-based languages (Self, JavaScript)
- Statically-typed object-oriented languages
  - C++ – using static typing to eliminate search
    - problems with C++ multiple inheritance
  - Java – using Interfaces to avoid multiple inheritance

Java language background

- James Gosling and others at Sun, 1990 - 95
- Oak language for “set-top box”
  - small networked device with television display
    - graphics
    - execution of simple programs
    - communication between local program and remote site
    - no “expert programmer” to deal with crash, etc.
- Internet applications
  - simple language for writing programs that can be transmitted over network
  - not an integrated web scripting language like JavaScript

Design Goals

- Portability
  - Internet-wide distribution: PC, Unix, Mac
- Reliability
  - Avoid program crashes and error messages
- Safety
  - Programmer may be malicious
- Simplicity and familiarity
  - Appeal to average programmer; less complex than C++
- Efficiency
  - Important but secondary

General design decisions

- Simplicity
  - Almost everything is an object
  - All objects on heap, accessed through pointers
  - No functions, no multiple inheritance, no go to, no operator overloading, few automatic coercions
- Portability and network transfer
  - Bytecode interpreter on many platforms
- Reliability and Safety
  - Typed source and typed bytecode language
  - Run-time type and bounds checks
  - Garbage collection

Language Terminology

- Class, object - as in other languages
- Field – data member
- Method - member function
- Static members - class fields and methods
- this - self
- Package - set of classes in shared namespace
- Native method - method compiled from in another language, often C

Java Classes and Objects

- 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++

Point Class

```
class Point {
   private int x;
   protected void setX (int y)  {x = y;}
   public int  getX()     {return x;}
   Point(int xval) {x = xval;}      // constructor
};
```

– Visibility similar to C++, but not exactly (later slide)

Object initialization

- Java guarantees constructor call for each object
  - Memory allocated
  - Constructor called to initialize memory
  - Some interesting issues related to inheritance

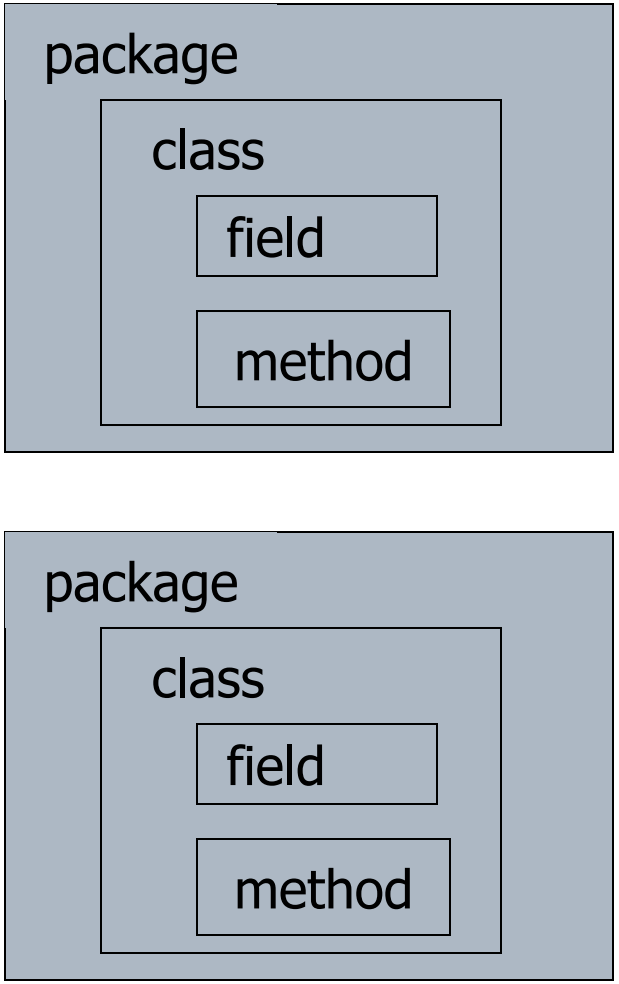
    We'll discuss later …
- Cannot do this (would be bad C++ style anyway):
  - Obj* obj = (Obj*)malloc(sizeof(Obj));
- Static fields of class initialized at class load time
  - Talk about class loading later

Garbage Collection and Finalize

- Objects are garbage collected
  - No explicit *free*
  - Avoids dangling pointers and resulting type errors
- Problem
  - What if object has opened file or holds lock?
- Solution
  - *finalize* method, called by the garbage collector
    - Before space is reclaimed, or when virtual machine exits
    - Space overflow is not really the right condition to trigger finalization when an object holds a lock...)
  - Important convention: call super.finalize

Encapsulation and packages

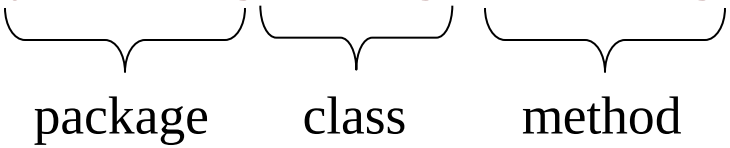
- Every field, method belongs to a class
- Every class is part of some package
  - Can be unnamed default package
  - File declares which package code belongs to



Visibility and access

- Four visibility distinctions
  - public, private, protected, package
- Method can refer to
  - private members of class it belongs to
  - non-private members of all classes in same package
  - protected members of superclasses (in diff package)
  - public members of classes in visible packages

    Visibility determined by files system, etc. (outside language)
- Qualified names (or use import)
  - java.lang.String.substring()



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

Example subclass

```
class ColorPoint extends Point {
  // Additional fields and methods
   private Color c;
   protected void setC (Color d)  {c = d;}
   public Color  getC()     {return c;}
  // Define constructor
   ColorPoint(int xval, Color cval) {
       super(xval);   // call Point constructor
       c = cval;  }     // initialize ColorPoint field
};
```

Class *Object*

- Every class extends another class
  - Superclass is *Object* if no other class named
- Methods of class *Object*
  - GetClass – return the Class object representing class of the object
  - ToString – returns string representation of object
  - equals – default object equality (not ptr equality)
  - hashCode
  - Clone – makes a duplicate of an object
  - wait, notify, notifyAll – used with concurrency
  - finalize

Constructors and Super

- Java guarantees constructor call for each object
- This must be preserved by inheritance
  - Subclass constructor must call super constructor
    - If first statement is not call to super, then call super() inserted automatically by compiler
    - If superclass does not have a constructor with no args, then this causes compiler error (yuck)
    - Exception to rule: if one constructor invokes another, then it is responsibility of second constructor to call super, e.g.,

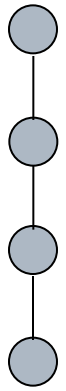
      ColorPoint() { ColorPoint(0,blue);}

      is compiled without inserting call to super
- Different conventions for finalize and super
    - Compiler does not force call to super finalize

Final classes and methods

- Restrict inheritance
  - Final classes and methods cannot be redefined
- Example

  java.lang.String
- Reasons for this feature
  - Important for security
    - Programmer controls behavior of all subclasses
    - Critical because subclasses produce subtypes
  - Compare to C++ virtual/non-virtual
    - Method is “virtual” until it becomes final



Java Interfaces (by example)

```
interface Shape {
    public float center();
   public void rotate(float degrees);
}
interface Drawable {
    public void setColor(Color c);
   public void draw();
}
class Circle implements Shape, Drawable {
    // does not inherit any implementation
   // but must define Shape, Drawable methods
}
```

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
    - For now, think of Java implementation as Smalltalk/JavaScript implementation, although we will see that the Java type system supports some optimizations

Outline

- Central concepts in object-oriented languages
  - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records
  - Simula – implementation as activation records with static scope
- Pure dynamically-typed object-oriented languages
  - Object implementation and run-time lookup
  - Class-based languages (Smalltalk)
  - Prototype-based languages (Self, JavaScript)
- Statically-typed object-oriented languages
  - C++ – using static typing to eliminate search
    - problems with C++ multiple inheritance
  - Java – using Interfaces to avoid multiple inheritance