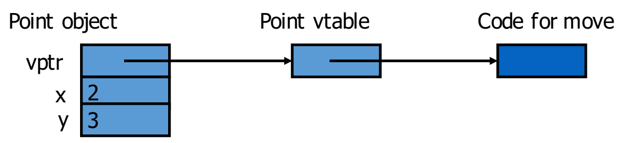
OO: optimizing with static types, code-sharing mechanisms

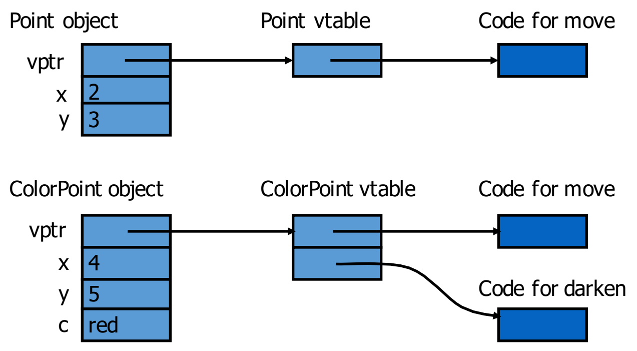
1. Prelude: C++-style representation
2. Multiple inheritance
3. Interfaces
4. Traits/mixins

Uses slides by Steve Freund

C++ Run-Time Representation

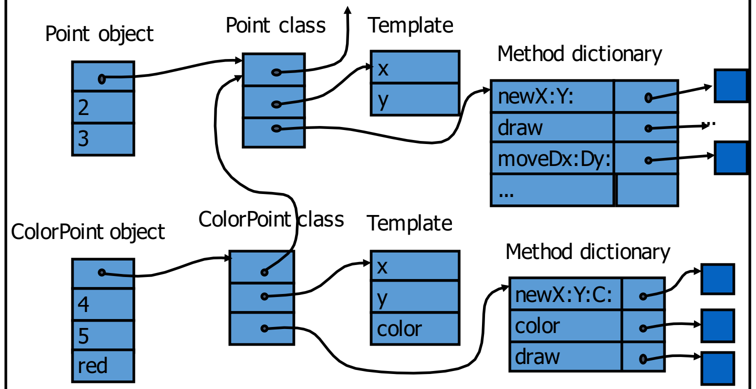


C++ Run-Time Representation



Data at same offset **Function pointers at same offset**

Contrast with Smalltalk representation

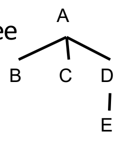


C++ goals

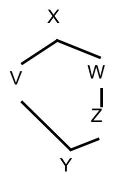
- Zero-cost abstraction
- Pay as you go
- …
- High level of control over representation/performance.

Single v. Multiple Inheritance

- Single inheritance: tree
  - one superclass
  - Linear code reuse



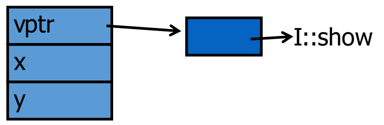
- Multiple inheritance: DAG
  - multiple superclasses
  - Compositional code reuse



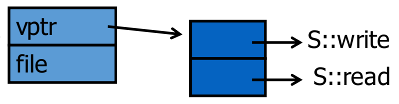
8

Multiple Inheritance

```
class Image {
  int x,y;
  virtual void show();
};
```



```
class Serializable {
  string file;
  virtual void write();
  virtual void read();
};
```



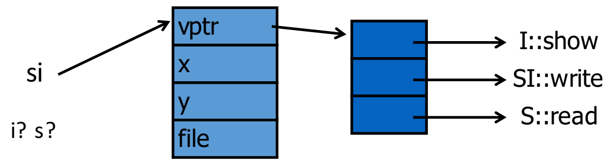
```
class SerialImage: public Image, public Serializable {
  virtual void write();
  virtual void show();
};
```

Multiple Inheritance

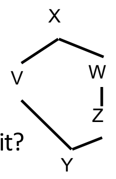
```
SerialImage  *si = new SerialImage();
si -> show();
si -> write();
si -> read();

Image *i = si;
i -> show();

Serializable  *s = si;
s -> write();
s -> read();
```



Downside: too many choices



If *V* and *Z* both define method **`m`**, what does *Y* inherit?
What does **`super`** mean?

What if *X* defines a method **`m`** that *Z* but not *V* overrides?

If *X* defines fields, should *Y* have one or tow **`f`**s?
Is the answer different if V and W both define field **`g`**?

…

C++ approach: support all combinations of possibilities!

11

ArtistCowboys

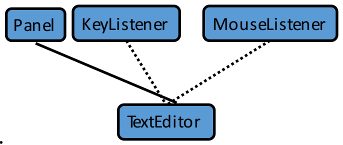
```
class PocketWearer {
  var pocket …
}
class Artist extends PocketWearer {
  def draw(): Unit =
    … // get brush from pocket and draw with it
}
class Cowboy extends PocketWearer {
  def draw(): Unit =
    … // draw pistol from pocket and aim
}

class ArtistCowboy extends Artist, Cowboy {
   // not Scala!
}

new ArtistCowboy.draw // ?????????
```

12

Java Interfaces



```
interface  KeyListener  {
  void keyPressed(KeyEvent  e);
  void keyReleased(KeyEvent  e);
  void keyTyped(KeyEvent  e);
}

interface  MouseListener  {
  void buttonClicked(MouseEvent  e);
}

class  TextEditor  extends  Panel
                 implements  KeyListener,MouseListener  {
  void keyPressed(KeyEvent  e)  {  /* code  */ }
  void keyReleased(KeyEvent  e)  {  /* code  */ }
  void keyTyped(KeyEvent  e)  {  /* code  */ }
  void buttonClicked(MouseEvent  e)  {  /* code  */ }
 }
```

Interfaces vs. Multiple Inheritance

```
class DefaultKeyListener {
  // default is to do nothing
  void keyPressed(KeyEvent e)  { }
  void keyReleased(KeyEvent e) { }
  void keyTyped(KeyEvent e) { }
}

class TextField : public Panel, DefaultKeyListener {
  void keyTyped(KeyEvent e) { /* code here */ }
  ...
}

class TerminalWindow : public Panel, DefaultKeyListener {
  void keyTyped(KeyEvent e) { /* code here */ }
  ...
}
```

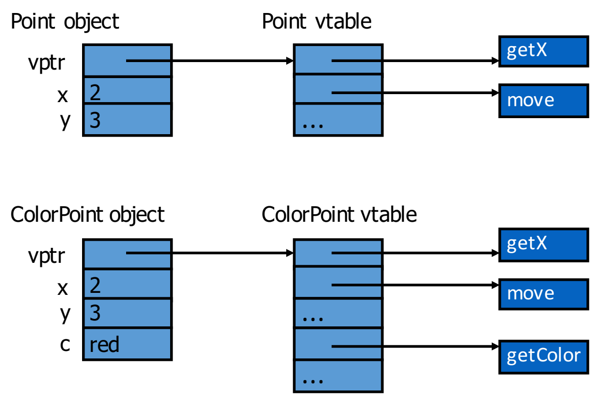
Interfaces vs. Multiple Inheritance

```
interface KeyListener {
  void keyPressed(KeyEvent e);
  void keyReleased(KeyEvent e);
  void keyTyped(KeyEvent e);
}

class TextField extends Panel implements KeyListener {
  void keyPressed(KeyEvent e) { }
  void keyTyped(KeyEvent e) { /* code here */ }
  void keyReleased(KeyEvent e) { }
}

class TerminalWindow extends Panel implements KeyListener {
  void keyPressed(KeyEvent e) { }
  void keyTyped(KeyEvent e) { /* code here */ }
  void keyReleased(KeyEvent e) { }
}
```

Dispatching interface method calls?



Scala Traits

- Completely Abstract

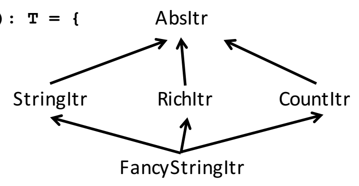
```
trait  AbsIterator[T]   {
  def  hasNext():  boolean;
  def  next():  T;
}
```

- Partially Implemented

```
trait RichIterator[T] extends AbsIterator[T] {
  def foreach(f: T => Unit): Unit = {
    while (hasNext()) f(next())
  }
}
```

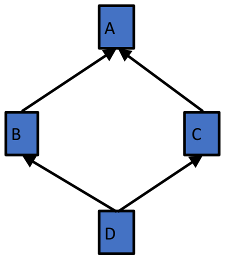
Scala Traits

```
trait CountingIterator[T] extends AbsIterator[T] {
  var count = 0;
  abstract override def next(): T = {
    count = count + 1;
    super.next();
  }
  def count() = count;
}
```



```
class FancyStringIterator(s: String)
       extends StringIterator(s)
       with RichIterator[Char]
       with CountingIterator[Char] { ... }
```

Name Resolution via Linearization



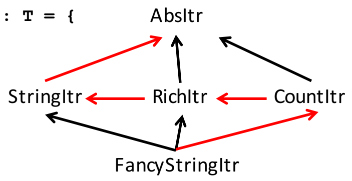
```
trait  A { }
trait  B extends  A { }
trait  C extends  A { }
class  D extends  B with  C { }
```

Right-first depth-first search: [D,C,A,B,A]

Eliminate all but last occurrence: [D,C,B,A]

Scala Traits

```
trait CountingIterator[T] extends AbsIterator[T] {
  var count = 0;
  abstract override def next(): T = {
    count = count + 1;
    super.next();
  }
  def count() = count;
}
```



```
class FancyStringIterator(s: String)
       extends StringIterator(s)
       with RichIterator[Char]
       with CountingIterator[Char] { ... }
```

http://en.wikipedia.org/wiki/Mixin