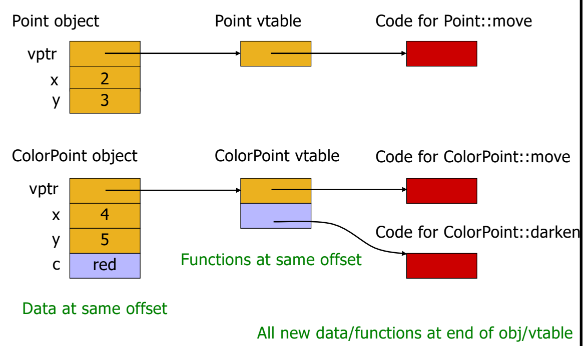
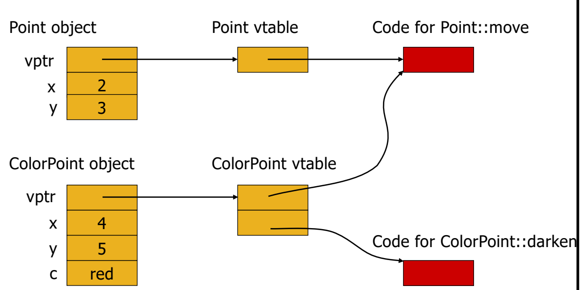
C++ Part 2

CSCI 334
Stephen Freund

C++ Run-Time Representation



C++ Run-Time Representation



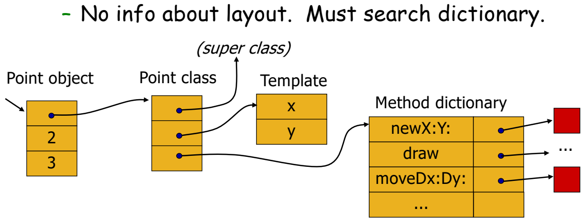
Smalltalk vs. C++ Dynamic Dispatch

- C++:
  - `Point *pt = ...;`
    `pt -> move(3,10);`
  - pt must be a Point or a subclass of Point
  - All subclasses of Point use same vtable order for Point methods.
  - Compiler can hard-code in vtable offset:

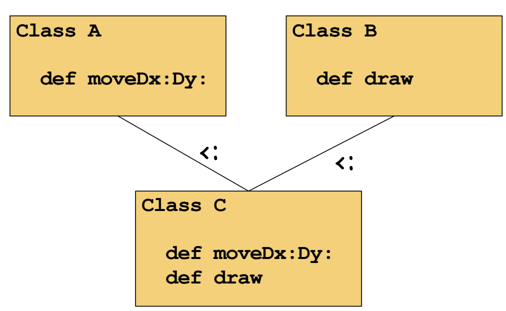
call (pt->vtable[0])(pt, 3, 10)

Smalltalk vs. C++ Dynamic Dispatch

- Smalltalk:
  - `pt moveDx: 3 Dy: 10`
  - `pt` could be any object with `moveDx:Dy:` in protocol
  - No info about layout. Must search dictionary.



Could we put methods at same offset?



Smalltalk vs. C++ Dynamic Dispatch

- Static type info about object layout
  - more efficient lookup
  - but less expressive
    - A classes not related by inheritance cannot be subtypes
- Things to think about:
  - How do Java interface declarations fit into this picture?
  - How do Scala traits?

Non-Virtual Methods

```
class Point {
    int getX();
    virtual void move(int dx, int dy);
};

Point *pt = ...;
pt -> getX();
```

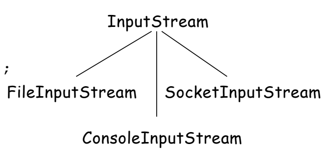
- Compiler hard-codes in direct jump:

  call Point::getX(pt, 3, 10)

- Why useful?

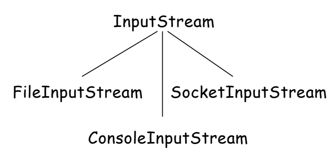
Stream Class Hierarchy

```
class InputStream {
  protected:
    virtual int read();
  public:
    virtual String readLine();
    virtual int readInt();
    virtual long readLong();
};

String InputStream::readLine() {
  String result = "";
  while (true) {
    int ch = read();
    if (ch == '\n') break;
    result += ch;
  }
  return result;
}
```



Stream Class Hierarchy

```
class InputStream {
  protected:
    virtual int read();
  public:
    String readLine();
    int readInt();
    long readLong();
};

String InputStream::readLine() {
  String result = "";
  while (true) {
    int ch = read();
    if (ch == '\n') break;
    result += ch;
  }
  return result;
}
```



Stream Class Hierarchy

```
class FileInputStream : public InputStream {
  protected:
    virtual int read() { ... }
};

class SocketInputStream : public InputStream {
  protected:
    virtual int read() { ... }
};
```

- two primary uses for non-virtual methods:
  - guaranteed behavior
  - efficiency

Overloading vs. Virtual Methods

```
class Point {
  void printClass() { cout << "Point"; }
  virtual void printClassVirtual() { cout << "Point"; }
};

class ColorPoint : public Point {
  void printClass() { cout << "ColorPoint"; }
  virtual void printClassVirtual() { cout << "ColorPoint"; }
};

  Point *p = new Point();
  p->printClassVirtual();          Point
  p->printClass();                 Point

  ColorPoint *cp = new ColorPoint();
  cp->printClassVirtual();         ColorPoint
  cp->printClass();                ColorPoint
```

Overloading vs. Virtual Methods

```
class Point {
  void printClass() { cout << "Point"; }
  virtual void printClassVirtual() { cout << "Point"; }
};

class ColorPoint : public Point {
  void printClass() { cout << "ColorPoint"; }
  virtual void printClassVirtual() { cout << "ColorPoint"; }
};

  Point *p = new ColorPoint();

  p->printClassVirtual();           ColorPoint
  p->printClass();                  Point
```

Overloading Based on Arguments

```
class OutputStream {
    void println(int v);
    void println(double d);
    void println(String s);
    void println(Object o);
    ...
}

OutputStream *out = ...;
out->println(3);
out->println("moo");

Object *obj = "cow";      // String <: Object
out->println(obj);
```

Overriding vs. Overloading

```
class Point {
      virtual boolean equals(Point *p)        // 1
        { ... }
};

class ColorPoint : public Point {
      virtual boolean equals(Point *p)        // 2
        { ... }

      virtual boolean equals(ColorPoint *cp)  // 3
        { ... }
};
```

```
Point *p1 = new Point();
Point *p2 = new ColorPoint();
ColorPoint *cp = new ColorPoint();

p1->equals(p1);
p1->equals(p2);
p1->equals(cp);

p2->equals(p1);
p2->equals(p2);
p2->equals(cp);

cp->equals(p1);
cp->equals(p2);
cp->equals(cp);
```

```
Point *p1 = new Point();
Point *p2 = new ColorPoint();
ColorPoint *cp = new ColorPoint();

p1->equals(p1);     Point::equals(Point *p)
p1->equals(p2);     Point::equals(Point *p)
p1->equals(cp);     Point::equals(Point *p)

p2->equals(p1);     ColorPoint::equals(Point *p)
p2->equals(p2);     ColorPoint::equals(Point *p)
p2->equals(cp);     ColorPoint::equals(Point *p)

cp->equals(p1);     ColorPoint::equals(Point *p)
cp->equals(p2);     ColorPoint::equals(Point *p)
cp->equals(cp);     ColorPoint::equals(ColorPoint *cp)
```

Different Notions of Subtyping

- Smalltalk protocol conformance
  - A <: B if protocol of B $\subseteq$ protocol of A
- Java
  - A <: B if A `extends` B
  - A <: I if A `implements` I
- C++
  - A <: B if B is *public* base class of A
  - `class A : public B { ... }` ➔ A <: B
  - `class A : private B { ... }` ➔ A ~~<:~~ B

Private vs Public Base Classes

```
class DEBuffer {
  public:
    void addFirst(int v);
    void addLast(int v);
    int removeFirst();
    int removeLast();
  protected:
    int elems[];
    int size;
}
```

+ reuse code
+ don't expose full interface

```
class Stack : private DEBuffer {
  public:
    void push(int v) {
      addLast(v);
    }

    int pop() {
      return removeLast();
    }

    int sum() { ... }
}

class Queue : private DEBuffer {
  public:
    void enqueue(int v) { ... }
    int dequeue()       { ... }
  }
}
```

Why Not Just Use Delegation?

```
class DEBuffer {
  public:
    void addFirst(int v);
    void addLast(int v);
    int removeFirst();
    int removeLast();
  protected:
    int elems[];
    int size;
}
```

+ access to internal rep enables more efficient code

```
class Stack {

  private:
    DEBuffer *delegate;

  public:
    void push(int v) {
      delegate->addLast(v);
    }

    int pop() {
      return
         delegate->removeLast();
    }

    int sum() { ... }
}
```

Abstract Base Classes

```
class Expr {
  private: Type t;
  virtual int eval() = 0;
  virtual int children() { return 0; }
}

class Plus : public Expr {
  private:
    Expr *left;
    Expr *right;
  public:
    virtual int eval() {
        return left->eval() + right->eval();
    }
    virtual int children() { return 2; }
}
```

+ introduce type for general concept
+ defines layout for all subclasses
+ can provide partial impl.

Is This Okay?

```
class Point {
  virtual Point *copy() { ... }
}

class ColorPoint : public Point {
  virtual ColorPoint *copy() { ... }
}


Point *pt;
...
Point *copy = pt->copy();
```

Subtyping Principle for Function Types

- Given two functions:
  - f1 : C → D
  - f2 : A → B
- When can we replace f1 with f2 and not introduce potential type errors?

...	...
y = f1(x);	y = f2(x);
...	...

Subtyping Principle for Function Types

- Given two functions:
  - f1 : C → D
  - f2 : A → B
- When can we replace f1 with f2 and not introduce potential type errors?

...	...
y = f1(x);	y = f2(x);
...	...

- f1 : Point → Point
- f2 : Point → ColorPoint ✔

Subtyping Principle for Function Types

- Given two functions:
  - $f1 : C \to D$
  - $f2 : A \to B$
- When can we replace f1 with f2 and not introduce potential type errors?

```
...                     ...
y = f1(x);              y = f2(x);
...                     ...
```

- $f1 : Point \to ColorPoint$
- $f2 : Point \to Point$

Subtyping Principle for Function Types

- Given two functions:
  - $f1 : C \to D$
  - $f2 : A \to B$
- When can we replace f1 with f2 and not introduce potential type errors?

```
...                     ...
y = f1(x);              y = f2(x);
...                     ...
```

- $f1 : Point \to Point$
- $f2 : ColorPoint \to Point$

Subtyping Principle for Function Types

- Given two functions:
  - $f1 : C \to D$
  - $f2 : A \to B$
- When can we replace f1 with f2 and not introduce potential type errors?

```
...                     ...
y = f1(x);              y = f2(x);
...                     ...
```

- $f1 : ColorPoint \to Point$
- $f2 : Point \to Point$

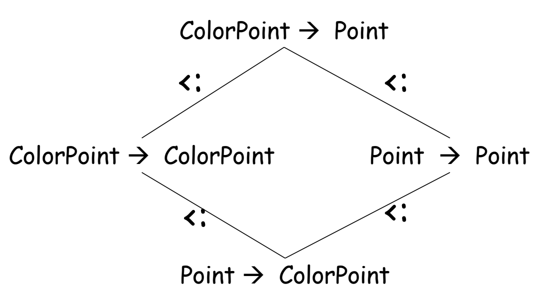
Subtyping Principle for Function Types

- Given two functions:
  - $f1 : C \to D$
  - $f2 : A \to B$
- When can we replace f1 with f2 and not introduce potential type errors?

```
...                     ...
y = f1(x);              y = f2(x);
...                     ...
```

- If $C <: A$ and $B <: D$ then $A \to B <: C \to D$

Subtyping Principle for Function Types



Is This Okay?

```
class Point {
  virtual Point *copy() { ... }
}

class ColorPoint : public Point {
  virtual ColorPoint *copy() { ... }
}

Point *pt;
...
Point *copy = pt->copy();
```

yes

Is This Okay? (Binary Method Problem)

```
class Point {
  virtual boolean equals(Point *pt) { ... }
}

class ColorPoint : public Point {
  virtual boolean equals(ColorPoint *cpt) {...}
}

Point *pt;
ColorPoint *cpt;
Point *pt2 = cpt;
...
if (pt2 -> equals(pt)) { ... }
```

Assume this overrides def from superclass

no

Stack Allocated Objects

```
void m() {
  Point *ref = new Point(1,1);
  Point pt(2,3);
  ColorPoint cpt(4,5,red);

  ref->move(5,5);
  pt.move(10,10);
  cpt.getColor();

  delete ref;
}
```