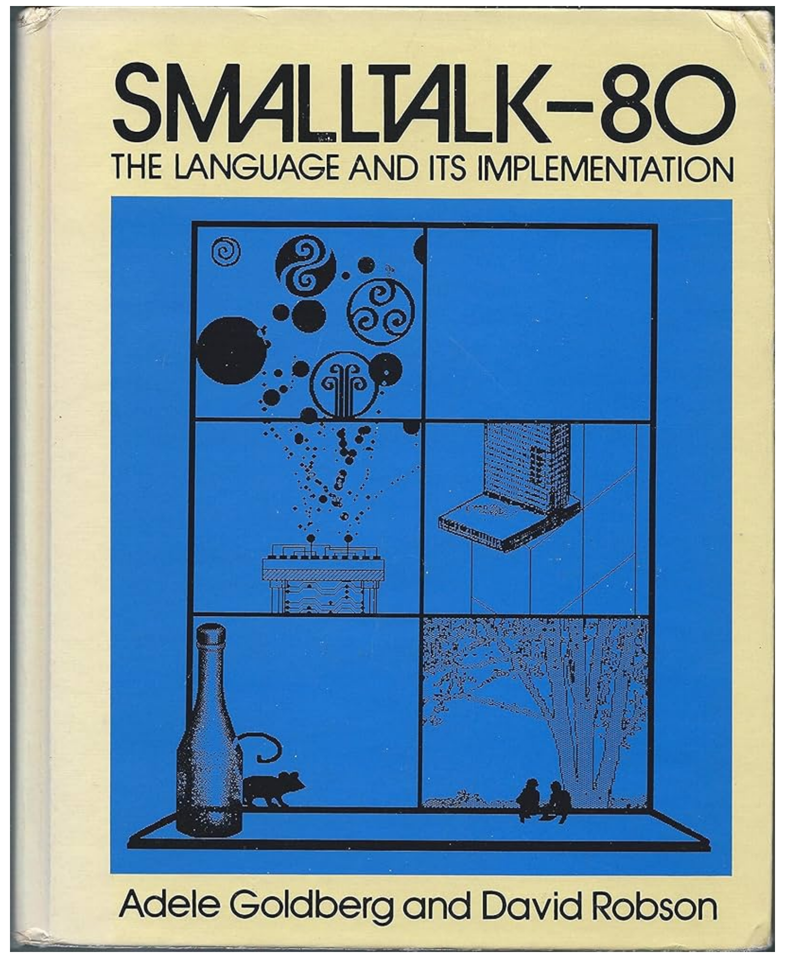
Object-Oriented Programming

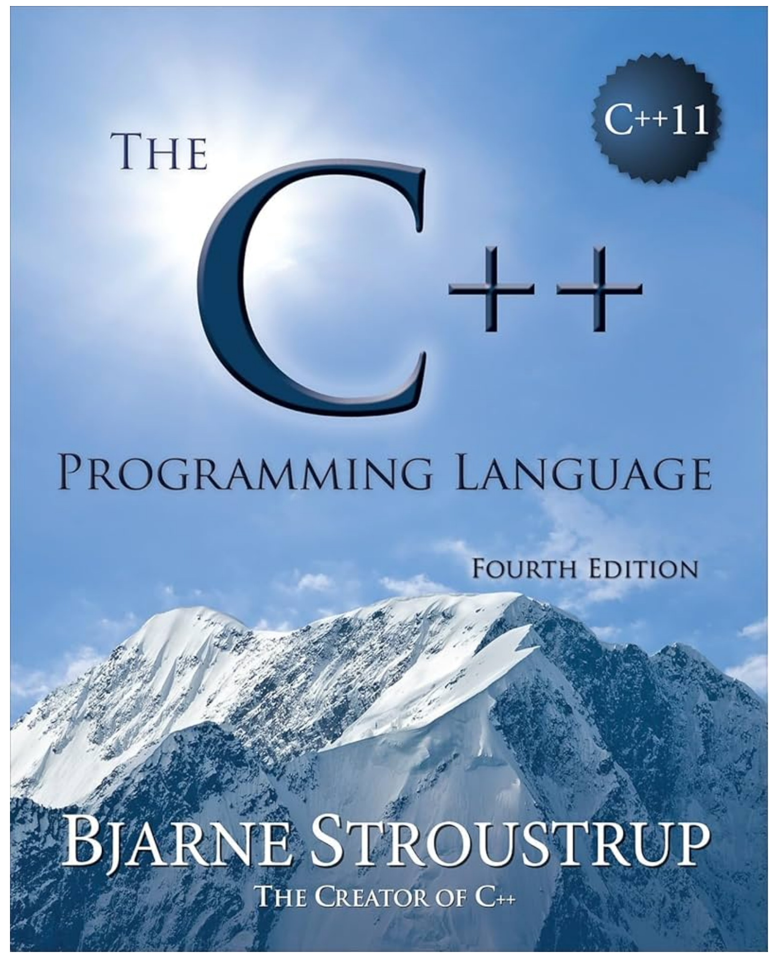
Kristopher Micinski

Syracuse University

CIS531, Fall 2025

- Object-Oriented Programming (OOP) is an extremely popular software development methodology
  - Center design around ***objects*** which exchange ***messages***
  - Original ideas date back to CLU (Liskov), Smalltalk, etc.
- Became very widely popular with C++
  - "C with classes" extension to C, by Stroustrup, Java, etc.
- Also, prototype-based polymorphism from JavaScript, etc.
  - Also involves objects, but implementation quite different than C++, Java, etc.





```
int add1(int x) { return x+1; }
```

In stored-program machines, all code sits somewhere in memory.

In C/C++, you can obtain pointers to functions at run-time, and invoke them! The pointer for `add1` can be obtained with:

```
&add1
```

A function pointer, `cmp`, passed to `sort` as an argument.

```
int sort(int* x, int len, bool (*cmp)(int,int))
{
    // …

        // …
            if ((*cmp)(*x,*y))
            {
                swap(*x,*y);

                // …
            }

    // …
}
```

The function pointer, `cmp`, dereferenced and invoked.

```
{
    // …

    sort(buff, length, &lessthan);

    // …
}
```

A pointer to function `lessthan` **is passed into** `sort`**.**

```
int add1(int x) { return x+1; }


int main()
{
    int (*f)(int) = &add1;

    // …

    int four = (*f)(3);
}
```

Objects

- In OOP, everything is an ***object***
  - Even primitive types—but in practice, we might want to work with an unboxed type; we can see this as kind of an optimization.
- Objects have ***methods*** which respond to ***messages***
- The object is created via a ***constructor*** which initializes private fields
  - Information hiding: can't see fields directly, only via methods
  - Enforced via static type system which checks correctness of field usage

C++ Classes

- In C++ (and other ***class-based*** OO langs), an object is built from a ***class***
  - At runtime, each object is an instance of some class
  - A class is a ***blueprint*** for an object, and can either be ***abstract*** or ***concrete***
  - A class definition specifies both (a) fields, and (b) methods
    - Fields are data elements, which will be held by each object
    - Methods are ***functions*** which may reference fields
      - All of the fields are in scope within every method
  - A class also specifies a ***constructor*** which is called when object is created

```
class Point {
private:
    int x, y; // private fields
public:
    Point(int a, int b) : x(a), y(b) {} // Constructor, initializes x/y
    void print() { cout << "(" << x << ", " << y << ")\n"; }
};
```

Inheritance

- In OOP, we have ***inheritance***, which means that we can define ***subclasses***, which extend the definition of parent classes to add more features / complexity
- An abstract class is a class which has some undefined methods
  - You cannot instantiate an abstract class, only a concrete class
- But you can always treat a concrete class as ***more abstract***
  - I can always soundly treat a Square as Shape, but not vice-versa…
- Can ***omit*** a method and—if defined in superclass—it will get used automatically

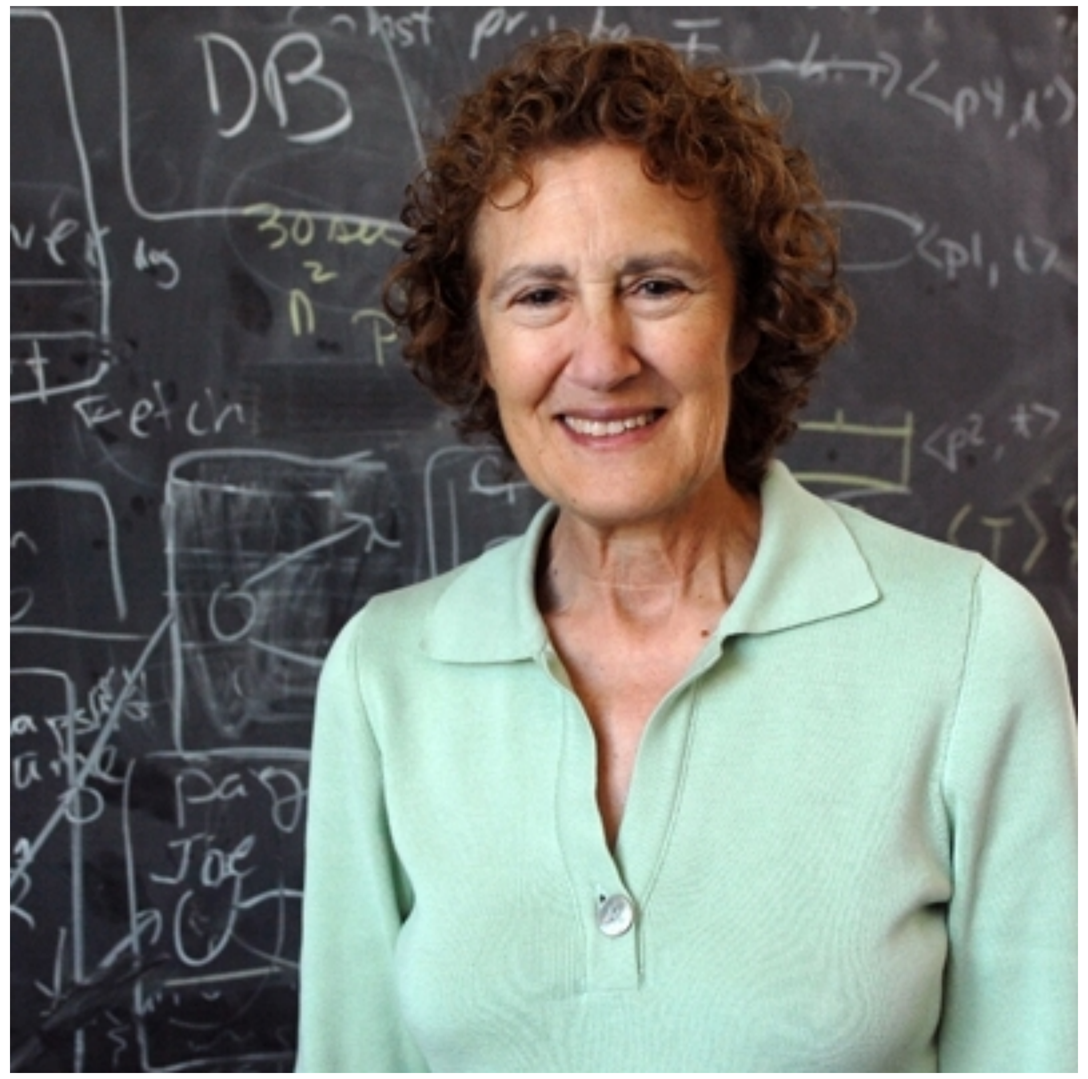
```
class Shape {
  virtual uint area() = 0;
}
```

```
class Square : Shape {
private:
  uint length;
public:
  Square(uint l) { length = l; }
  virtual uint area() { return length*length; }
}
```

```
class Rectangle : Shape {
private:
  uint length, width;
public:
  Rectangle(uint l, uint w) { length=l; width=w; }
  virtual uint area() { return length*width; }
}
```

Liskov Substitution Principle

- Directly addresses the question of ***subtyping***, by defining ***semantic subtyping***
  - If $S \sqsubseteq T$, i.e., if S is a subtype of T, then every possible property that holds of T also has to hold of S
  - I.e., for all properties ϕ, $S \sqsubseteq T \Rightarrow \forall x{:}T.\ \phi(x) \rightarrow \forall y{:}S.\ \phi(y)$
- This says that I can always **soundly upcast**
  - *I.e.*, if I have an S, I can always treat it as a T
    - If I have a Square, I can always treat it as a shape
  - <u>Can't</u> safely downcast, in general (must check dynamic type!)
- From a superficial level, this means that all subclasses respond to the same messages (implement same methods)
- But it also guides how I write my code
  - Need to be careful to respect LSP!



Polymorphism in OOP

- Roughly, polymorphism is the idea that I can write ***one*** block of code that will handle multiple different shapes of data
  - “Polymorphism” is an extremely overloaded term in programming languages!
  - In the OO context, we can use virtual methods to enable overloading

- Superclasses / abstract classes specify interfaces, and these define the high-level communication / behavioral patterns that your app speaks
  - Then, specific implementations of those classes implement the logic
  - Encourages code reuse, program against the interface:
    - E.g., LinkedList vs. ArrayList vs. …: tune data structures to workload
    - Ideal for very “wide” (and “deep”) hierarchies of relationships
      - Enterprise software: the original slop

```
class B
{
    virtual int f() { return 1; }
};
class A : public B
{
    virtual int f() { return 2; }
};


B* a = new A(); // Get a pointer to an A obj
std::cout << a->f() << std::endl;
```

// Which value is printed out?   ANSWER: 2

C++ dynamic dispatch: class polymorphism

```
class Cmp
{
    virtual bool cmp(int x, int y) = 0;
};
class LessThan : public Cmp
{
    virtual bool cmp(int x, int y)
    { return x < y; }
};
class GreaterThan : public Cmp
{
    virtual bool cmp(int x, int y)
    { return x > y; }
};
```

An instance of type `Cmp`, `cmp`, has overloaded method `cmp`.

```
int sort(int* x, int len, const Cmp& cmp)
{
        // …
            if (cmp.cmp(*x,*y))
            {
                swap(*x,*y);
                // …
```

Pass in object `lessthan` by reference to polymorphic type Cmp supporting the Cmp::cmp(int, int) member.

```
int main()
{
    // …
    LessThan lessthan;
    sort(buff, length, lessthan);
```

```
class Animal
{
    virtual const char* name() = 0;
    virtual int weight() const = 0;
    virtual void eat(Animal* prey)
    {
        if (this->weight()
               < 2 * prey->weight())
            return;
        delete prey;
        std::cout << prey->name()
                  << “ was eaten!\n”;
    }
};
```

```
class Mouse : public Animal
{
    int grams;

    Mouse(int grams)
        : grams(grams) {}

    virtual const char* name()
    {
        return “Mouse”;
    }

    virtual int weight() const
    {
        return this->grams;
    }
};
```

```
class Cat : public Animal
{
    Cat() {}

    virtual const char* name()
    {
        return “Cat”;
    }

    virtual int weight() const
    {
        return 4260;
    }
};
```

```
class Giraffe : public Animal
{
    virtual const char* name()
    {
        return “Giraffe”;
    }
    virtual int weight() const
    {
        return 1570000;
    }
    virtual void eat(Animal* prey)
    {
        std::cout << this->name()
                  << “ wont eat that.\n”;
    }
};
```

Now, the question becomes…

"How do we actually lay out these classes / objects in memory?"

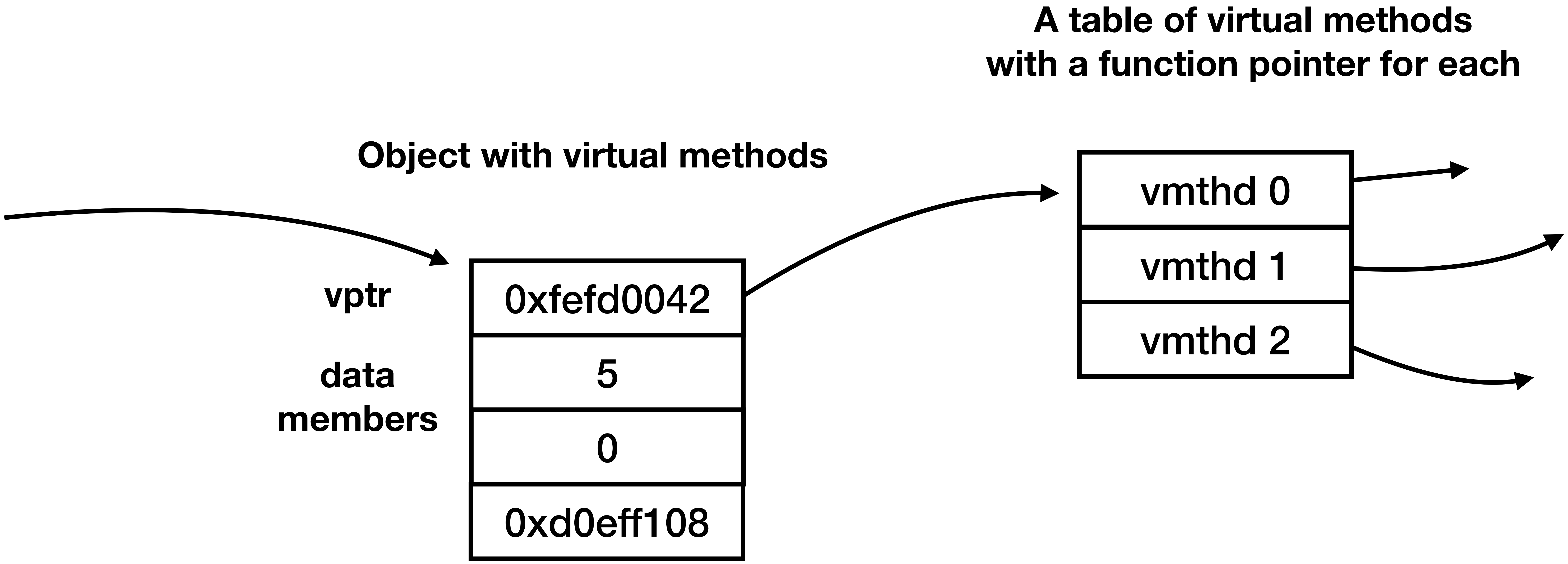
Object Layout...

- In general, object consists of methods and fields
  - Subclasses might **add** methods / fields, but can never ***subtract*** them (LSP)

- How to lay out object?
  - Class statically enumerates the **fields**, also methods, use virtual method table
  - Then, store fields

- Class-based languages use ***virtual method tables***:
  - Each class has vtable, which maps method names to function pointers
  - Each object has a pointer to the vtable
    - What are the trade-offs here? Consider: cache, object space, etc.

Virtual Method Tables



```
// vtable struct for Animal subclasses
struct AnimalVTable
{
    const char* (*name)(void*);
    int (*weight)(const void*);
    void (*eat)(void*,void*);

    AnimalVTable(const char* (*name)(void*),
                 int (*weight)(const void*),
                 void (*eat)(void*,void*))
      : name(name), weight(weight), eat(eat)
    {}
};

// Allocate a vtable for each concrete Animal
AnimalVTable mouse_vtable(&nameMouse,
                          &weightMouse,
                          &eatAnimal);
```

```
// Class Mouse compiled to a struct
struct Mouse
{
   AnimalVTable* vptr;
   int grams;
};


// An allocator/constructor for Mouse
Mouse* newMouse(int grams)
{
    Mouse* m = (Mouse*)malloc(sizeof(Mouse));
    m->vptr = &mouse_vtable;
    m->grams = grams;
    return m;
}
```

```
// A name method for Mouse instances
const char* nameMouse(void* _ths)
{
    return “Mouse”;
}


// A weight method for Mouse instances
int weightMouse(const void* _ths)
{
    const Mouse* ths = (const Mouse*)_ths;
    return ths->grams;
}
```

```
// Looks up the vtable for an object
VTable* vtable(void* obj)
{
    return (VTable*)((void**) obj)[0];
}



{
    // To call a member function f:
    // e.g., obj->f(arg0, arg1, …);

    vtable(obj)->f(obj, arg0, arg1, …);
}
```

```
// Looks up the vtable for an Animal object
AnimalVTable* vtable(void* obj)
{
    return (AnimalVTable*)((void**) obj)[0];
}

// A default eat method for Animals
void eatAnimal(void* ths, void* prey)
{
    if (vtable(ths)->weight(ths)
           < 2 * vtable(prey)->weight(prey))
        return;
    delete prey; // vtable(prey)->~Animal…
    std::cout << vtable(prey)->name(prey)
              << “ was eaten!\n”;
}
```

Try an example:

How do you define the constructor for Giraffe?

```
// Class Giraffe compiled to a struct
struct Giraffe
{
   AnimalVTable* vptr;
   // No data members
};

AnimalVTable giraffe_vtable(&nameGiraffe,
                            &weightGiraffe,
                            &eatGiraffe);

// An allocator/constructor for Giraffe
Giraffe* newGiraffe()
{
    Giraffe* g = new Giraffe();
    g->vptr = giraffe_vtable;
    return g;
}
```

```
const char* nameGiraffe(void* _ths)
{
    return “Giraffe”;
}

int weightGiraffe(const void* _ths)
{
    return 1570000;
}

void eatGiraffe(void* _ths)
{
    Giraffe* ths = (Giraffe*)_ths;
    std::cout << vtable(ths)->name(ths)
              << “ wont eat that.\n”;
}
```

Algebraic Data + Pattern Matching + Recursion (i.e., FP) vs. OOP

- IMO, OOP is oppositional to pure FP
  - FP is pure matching+recursion+algebraic data (lists)
  - In OOP, need to use something like ***visitor pattern***
  - Scatters handling of AST cases across lots of different methods, separates implementations…
  - More complex AST processing means that there is hidden state / order dependence when writing visitor / processing passes…
- For small things, algebraic data + pattern matching is a ***clear, obvious win*** to me (Kris) personally
  - For big APIs, with a huge number of cases, a giant pattern match would become unwieldy…
  - (Each visitor is complex, lots of code, vs. Racket)
  - In these cases, visitors clearly win!
  - LLVM / etc: lots of visitors

```java
interface Expr { <R> R accept(ExprVisitor<R> v); }

interface ExprVisitor<R> {
    R visitNum(Num n);
    R visitVar(Var v);
    R visitAdd(Add a);
    R visitMul(Mul m);
}

final class Num implements Expr {
    final int value;
    Num(int value) { this.value = value; }
    public <R> R accept(ExprVisitor<R> v) { return v.visitNum(this); }
}

final class Var implements Expr {
    final String name;
    Var(String name) { this.name = name; }
    public <R> R accept(ExprVisitor<R> v) { return v.visitVar(this); }
}

final class Add implements Expr {
    final Expr left, right;
    Add(Expr left, Expr right) { this.left = left; this.right = right; }
    public <R> R accept(ExprVisitor<R> v) { return v.visitAdd(this); }
}

final class Mul implements Expr {
    final Expr left, right;
    Mul(Expr left, Expr right) { this.left = left; this.right = right; }
    public <R> R accept(ExprVisitor<R> v) { return v.visitMul(this); }
}

final class Eval implements ExprVisitor<Integer> {
    private final Map<String,Integer> env;
    Eval(Map<String,Integer> env) { this.env = env; }
    public Integer visitNum(Num n) { return n.value; }
    public Integer visitVar(Var v) { return env.getOrDefault(v.name, 0); }
    public Integer visitAdd(Add a) { return a.left.accept(this) + a.right.accept(this); }
    public Integer visitMul(Mul m) { return a(m.left) * a(m.right); }
    private int a(Expr e){ return e.accept(this); }
}

final class Print implements ExprVisitor<String> {
    public String visitNum(Num n) { return Integer.toString(n.value); }
    public String visitVar(Var v) { return v.name; }
    public String visitAdd(Add a) { return "(" + a.left.accept(this) + " + " + a.right.accept(this) + ")"; }
    public String visitMul(Mul m) { return "(" + m.left.accept(this) + " * " + m.right.accept(this) + ")"; }
}
```

Base class for instruction visitors. More...

`#include "llvm/IR/InstVisitor.h"`

Inheritance diagram for llvm::InstVisitor< SubClass, RetTy >: