C++ Classes

A procedural language like C strongly reflects the imperative nature of machine code or assembly language:

"do this! now do this! now do this! now do this!"

```
__Z11fsm_moonmanv:
0000000100004d10 pushq  %rbp
0000000100004d11 movq %rsp,%rbp
0000000100004d14 subq $0x000003e0,%rsp
0000000100004d1b movq %rdi,%rax
0000000100004d1e movb $0x00,0xff(%rbp)
0000000100004d22 movq %rdi,0xfffffe20(%rbp)
0000000100004d29 movq %rax,0xfffffe18(%rbp)
0000000100004d30 callq  0x100001420
0000000100004d35 leaq 0xe8(%rbp),%rax
0000000100004d39 movq %rax,%rdi
...
```

Plain C code exposes the brutal and/or beautiful truth of how a computer really works: there's a CPU, and some memory, and it has input and output devices. Everything else are just details.

It makes sense that the first languages would start there, but there's no reason why we have to *stay* there.

Languages like Fortran and C are thin abstractions of assembly language. They are *way easier* to program, but fundamentally they are "do this! now do this! now do this!"

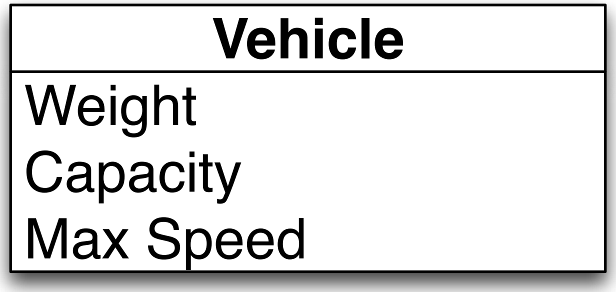
Other programming paradigms include:

Logical	Compute truth based on predicates
Object-Oriented	*Classes* and *subclasses* with encapsulation
Functional	No side effects
Reactive	Constraints/rules maintained implicitly
Dataflow	Computation as a directed graph

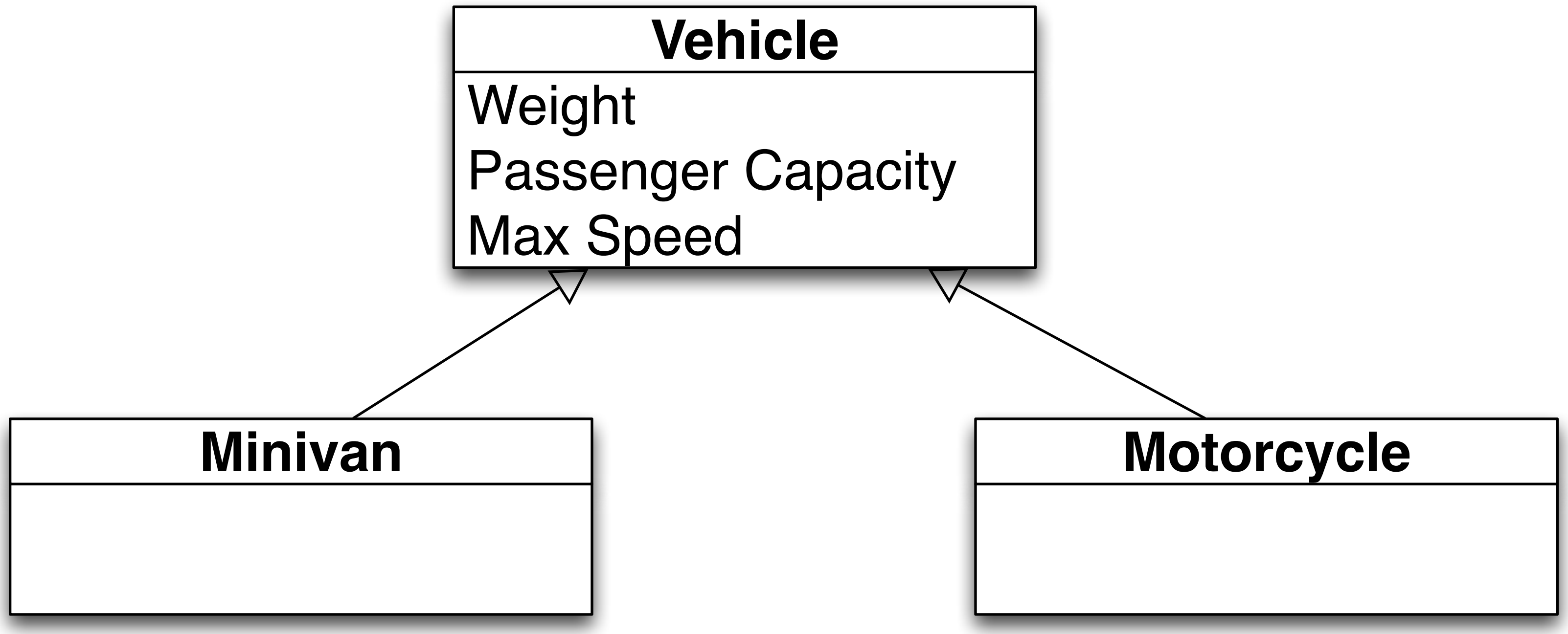
There are many (many many) more.

Object-oriented programming is usually presented as a way of modeling relationships among different kinds of data called *classes*.

We might have a specific class called Vehicle that specifies a weight, passenger capacity, and maximum speed.



Classes can be related in a few ways. One ways is by *inheritance*: one class is a superclass of another. For example, a Minivan is a special kind of vehicle. So is a Motorcycle.

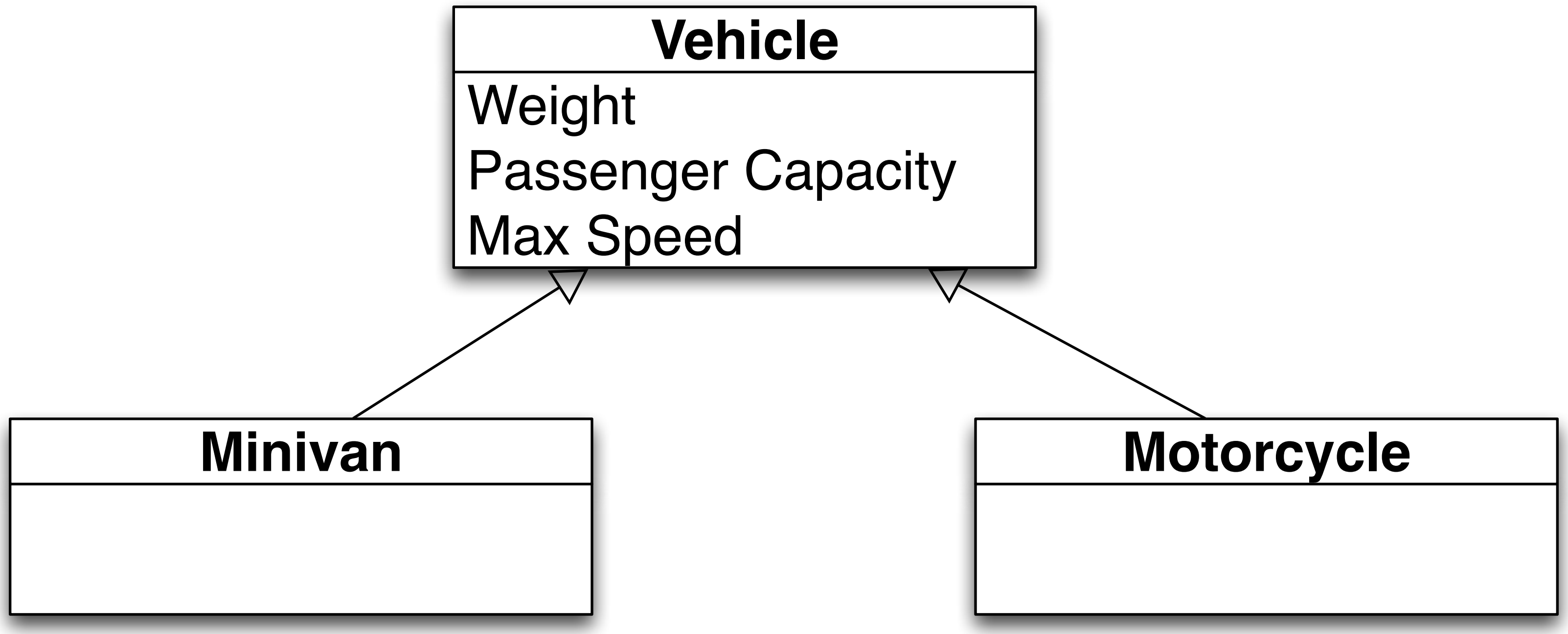


This is called a *class hierarchy*, and if you do any application programming it is a near certainty you'll need to understand how this works. It describes "isa" relationships:

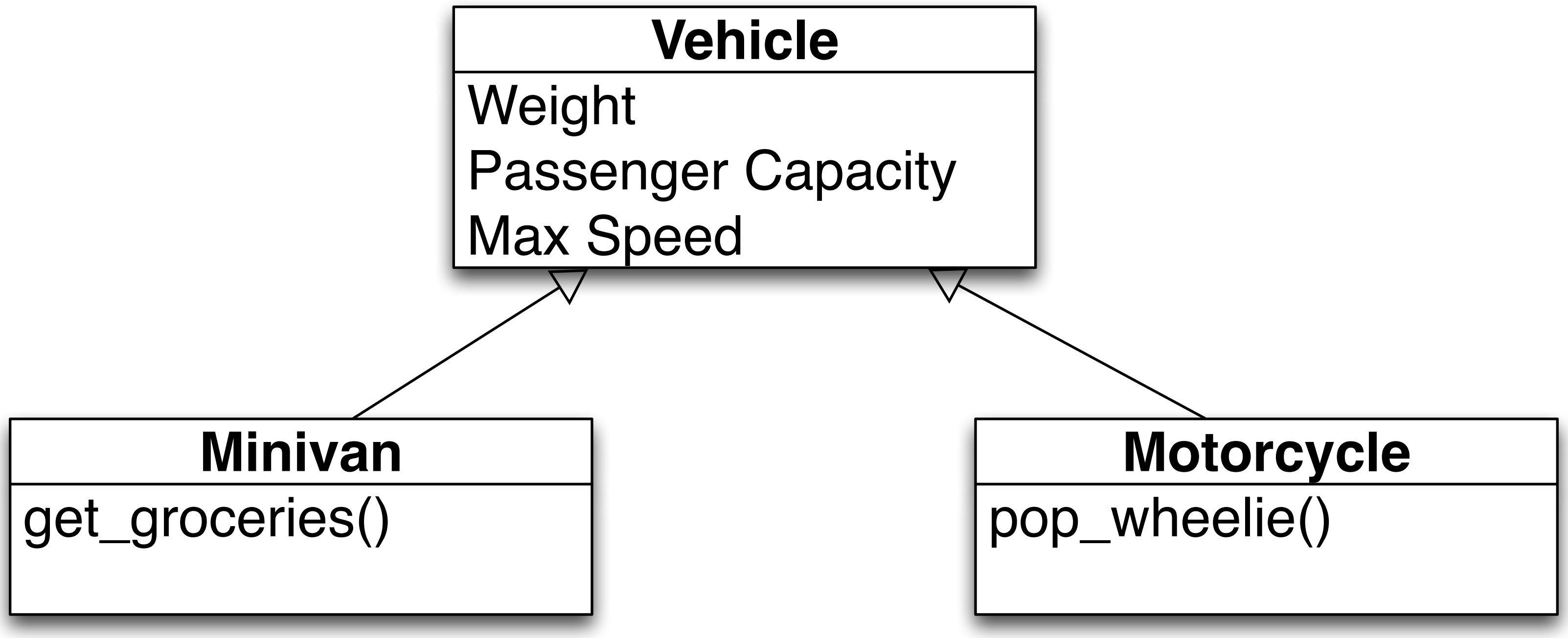
A minivan *is a* vehicle.

A motorcycle *is a* vehicle.

And so on.



The subclasses may add new members, including *data* and *behaviors*. For example, it makes sense for a motorcycle to be able to pop a wheelie, but not a minivan. We might have behavior for the minivan to bring home groceries, but this is unbecoming for a motorcycle. So: classes *specialize* and *differentiate* based on data and behavior.



A subclass has access to all the public and protected members of the parent class. What's that mean? Classes let you specify a *protection level* for all members. Many object-oriented languages have this idea. In C++, they are:

public	Members accessed from anywhere
protected	Members accessed by subclasses only
private	Members accessed by class (no subclasses)

We aren't actually going to use any of the inheritance stuff in this class, and you won't need to understand it for the final. Just showing it to let you know what those keywords mean. Rule of thumb: make variables *protected* and functions *public* (if they are the interface you want others to use) or *protected or private* (if they are helper functions).

Another way classes can be related is via *composition*. One class refers to another. We see this in the Linked List homework, where we have a LinkedList class and a Node class.

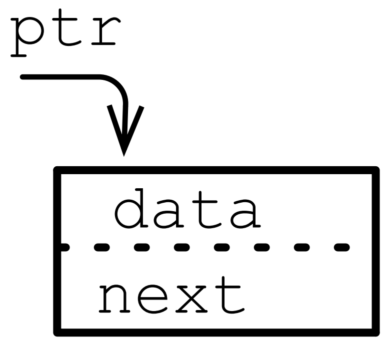
```
class LinkedList {
private:
  Node* head;
public:
  int size();
  // many more
};
```

```
class Node {
public:
  int data;
  Node* next;
};
```

```
class Node {
public:
  int data;
  Node* next;
};
```

We draw a node object as above: the data field is first, then a 'next' field.

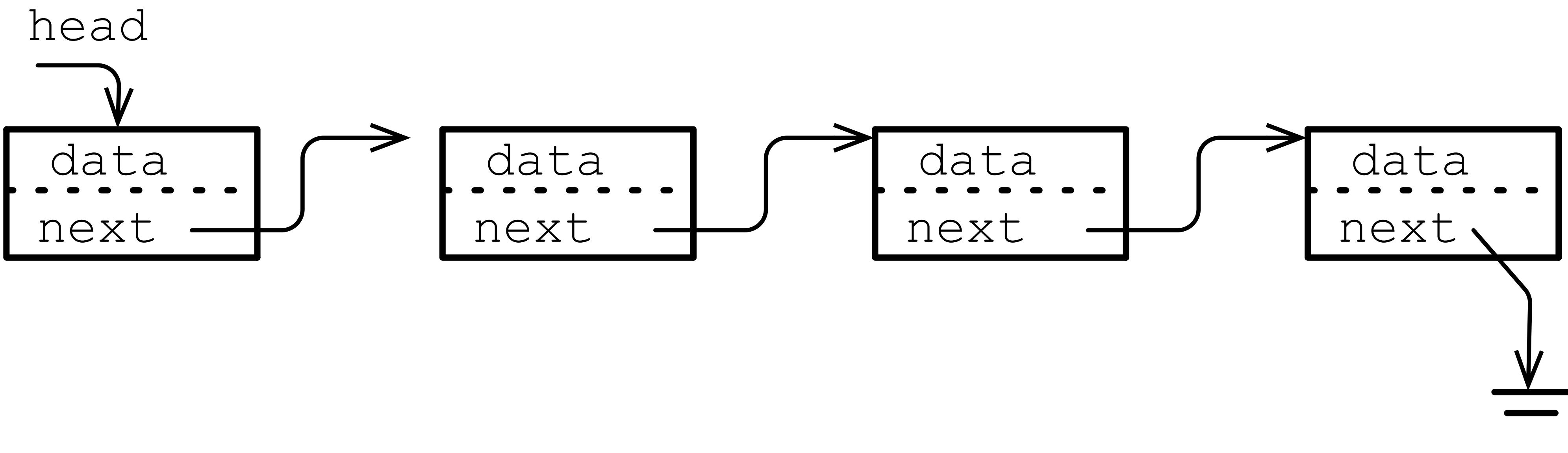
When we have a Node* (a pointer to a Node) we represent that with an arrow terminating at one of those box things, like this:



The LinkedList class has a single state variable:

Node* head;

But by using this data structure we can have lists with much more than one item:



So how do we know what data we have? And how do we gain access to the data we need? Consider the LinkedList class definition:

```
class LinkedList {
private:
  Node* head;
public:
  bool contains(int val);
  // many more
};
```

We're going to implement 'contains(val)', paying close attention to the data that is explicitly given as parameters, and *also the implicit data available from the object instance*.

```
bool LinkedList::contains(int val){
  // implement me
}
```

The return type is listed first.

```
bool LinkedList::contains(int val){
  // implement me
}
```

The namespace is listed second. This tells the compiler that we are implementing the LinkedList function called *contains*. This is to resolve conflicts if there happen to be several functions with that name. It also ensures that the compiler knows this is a member function of the LinkedList class.

The syntax is just the namespace followed by two colons.

```
bool LinkedList::contains(int val){
   // implement me
}
```

Next is the name of the function.

```
bool LinkedList::contains(int val){
   // implement me
}
```

Next is the formal parameter list. These are the variables that are given to us explicitly. This is how we did things for the pervious C++ homework assignments.

Here the 'val' param is declared as an integer. That's the number we are looking for somewhere in our LinkedList.

```
bool LinkedList::contains(int val){
  // implement me
}
```

But what about the LinkedLists's other functions? Here are some of the other functions in the LinkedList class:

```
class LinkedList {
private:
  Node* head;
public:
  bool contains(int val);
  int size();
  void append_data(int num);
  // many more
};
```

How do we use these?

```
bool LinkedList::contains(int val,
                          Node* head,
                          LinkedList* this){
   // implement me
}
```

We have *implicit access* to all the member variables of the class, as well as a reference to the specific object in question (called *this*). It is almost as though those variables are sent along as parameters to the function.

They aren't actually parameters, so all that bold red text up there is just to spur your imagination. But they *are* present, almost exactly as though they were sent along as parameters.

```
bool LinkedList::contains(int val){
  bool ret = false;
  Node* cursor = head;
  while (cursor != NULL) {
    if (cursor->data == val) {
      ret = true;
      break;
    }
    cursor = cursor-> next;
  }
  return ret;
}
```

Here is how I implemented the LinkedList::contains function. The explicitly provided variable is bold and black, the implicit one is italic and red.

Recall how at the beginning I talked about the imperative nature of machine code. When we use C++ classes, we ultimately get this kind of code. Our implementation of addState didn't explicitly ask for certain variables like states or default_state. The C++ compiler is 'smart' enough to know which variables should be made available, and issues the machine code for doing this.

"do this! now do this! now do this! now do this!"

```
__Z11fsm_moonmanv:
0000000100004d10 pushq  %rbp
0000000100004d11 movq   %rsp,%rbp
0000000100004d14 subq   $0x000003e0,%rsp
0000000100004d1b movq   %rdi,%rax
0000000100004d1e movb   $0x00,0xff(%rbp)
0000000100004d22 movq   %rdi,0xfffffe20(%rbp)
0000000100004d29 movq   %rax,0xfffffe18(%rbp)
0000000100004d30 callq  0x100001420
0000000100004d35 leaq   0xe8(%rbp),%rax
0000000100004d39 movq   %rax,%rdi
...
```