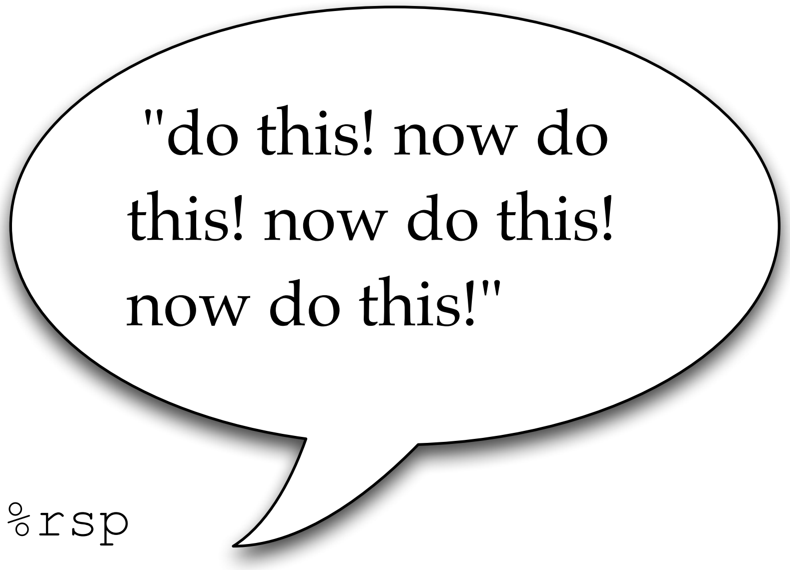
C++ Classes

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



```
__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!"

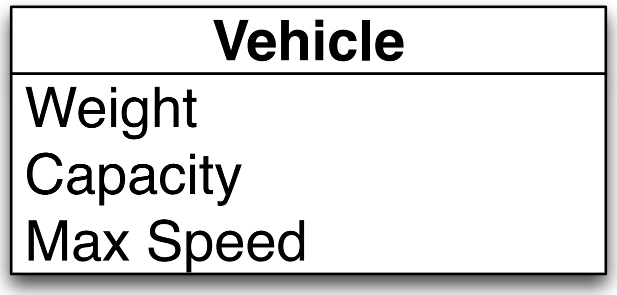
Other programming paradigms include:

Declarative	Say *what* you want, not how
Object-Oriented	*Classes* and *subclasses* with encapsulation
Functional	No side effects
Reactive	Constraints/rules maintained implicitly
Dataflow	Computation as a directed graph

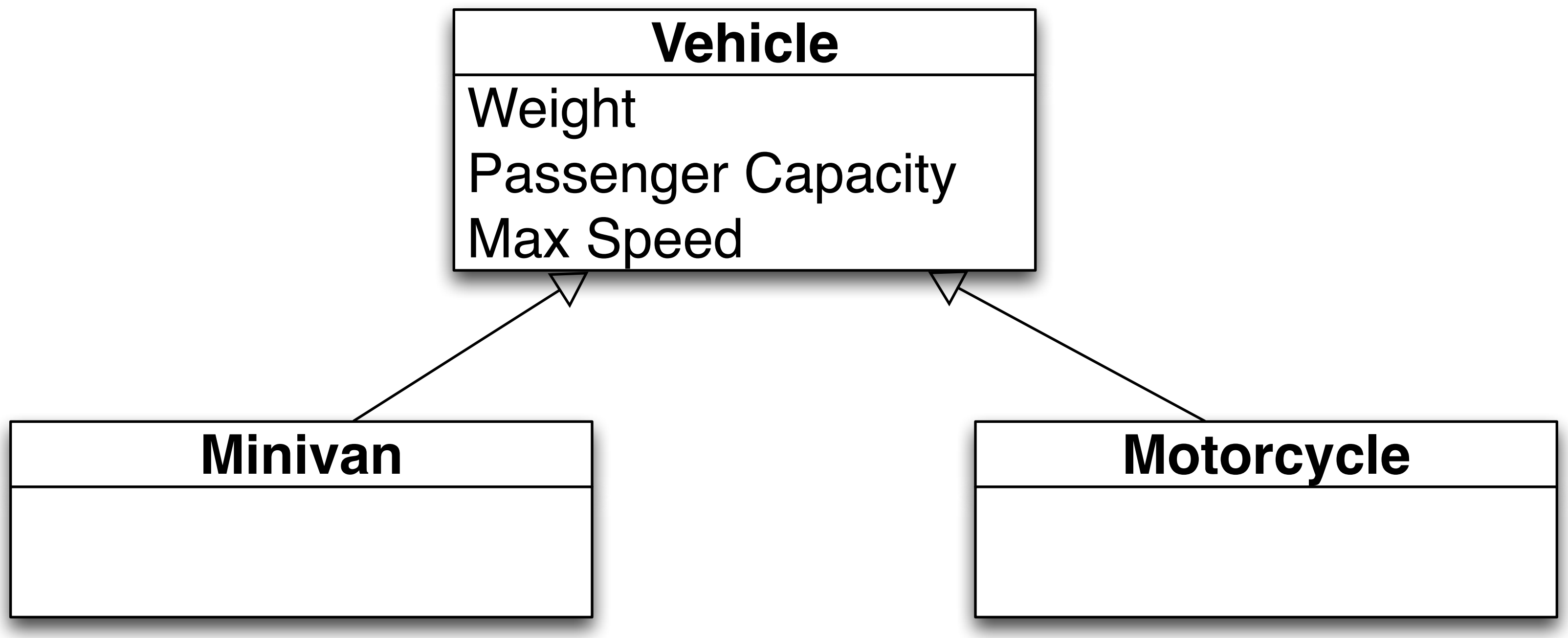
There are many (many many) more.

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.

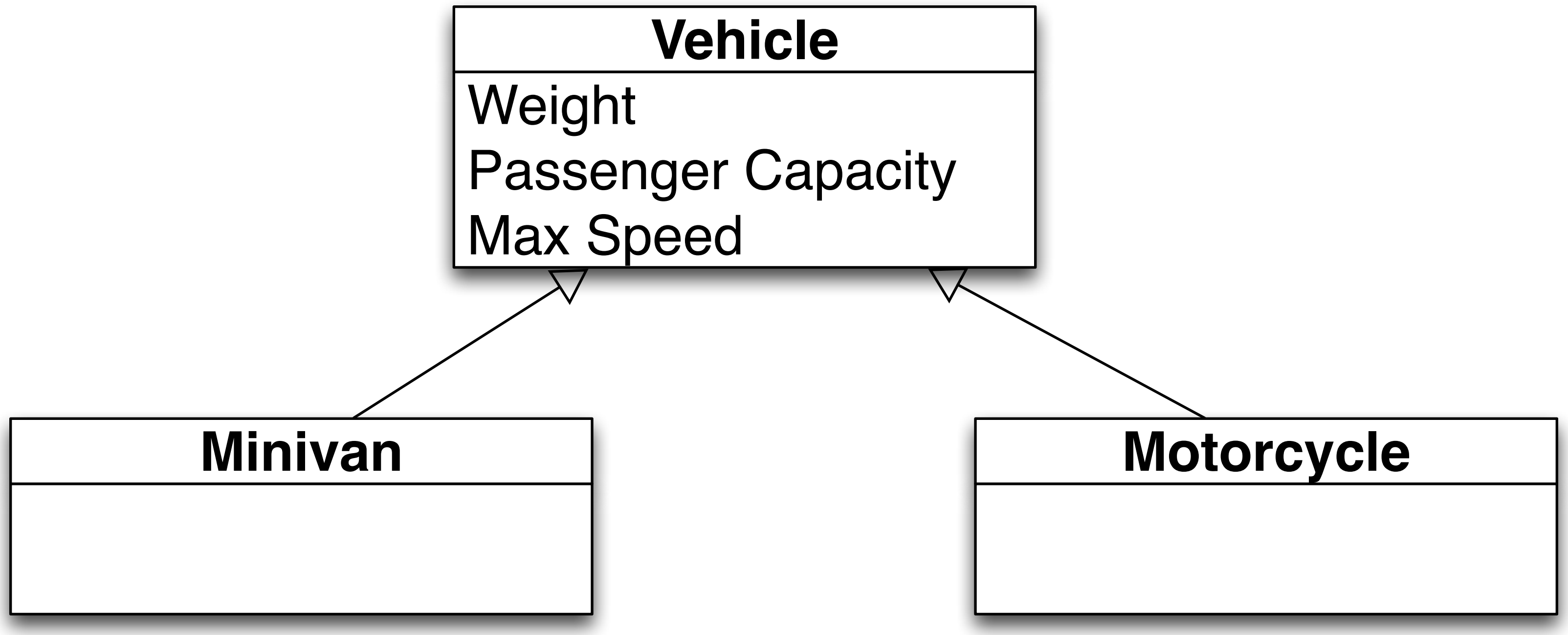


This is called a *class hierarchy*, and if you do any application programming it is a near certainty you'll need to understand how this works. It describes "isa" relationships:

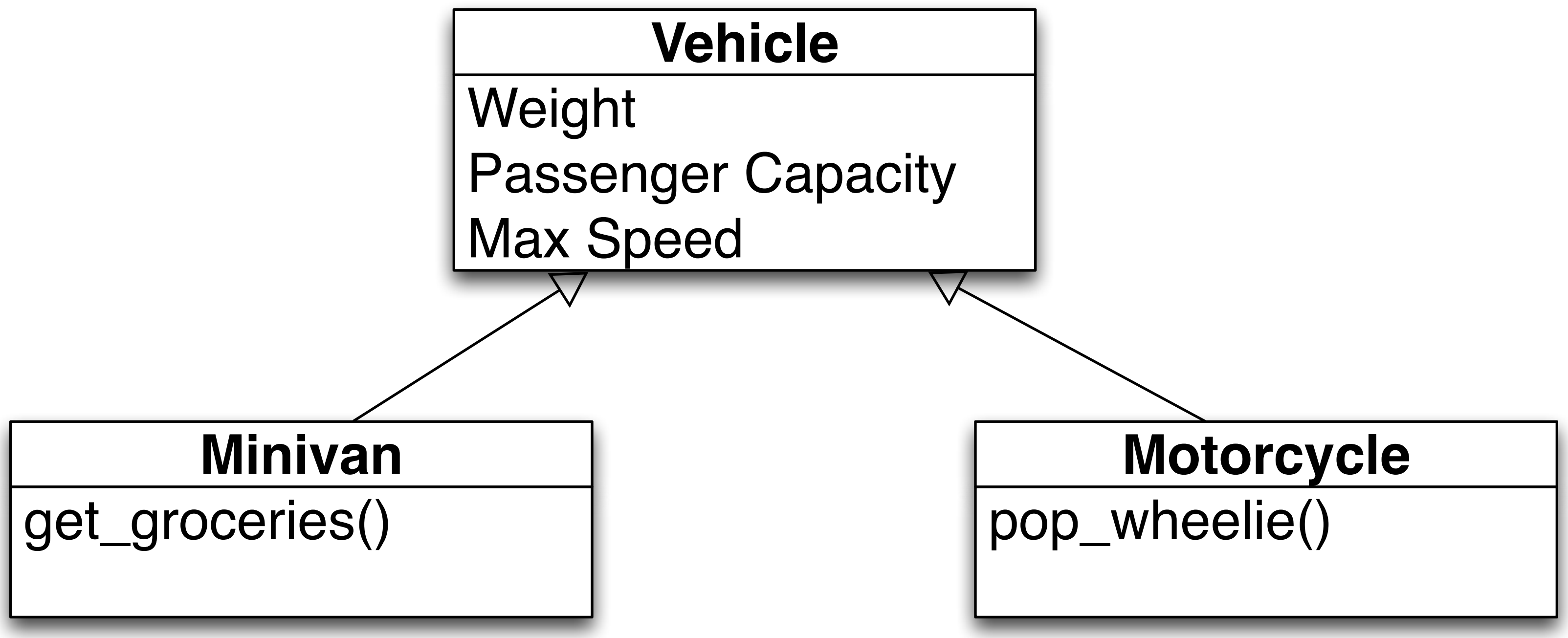
A minivan *is a* vehicle.

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 FSM homework, so we'll focus on this.

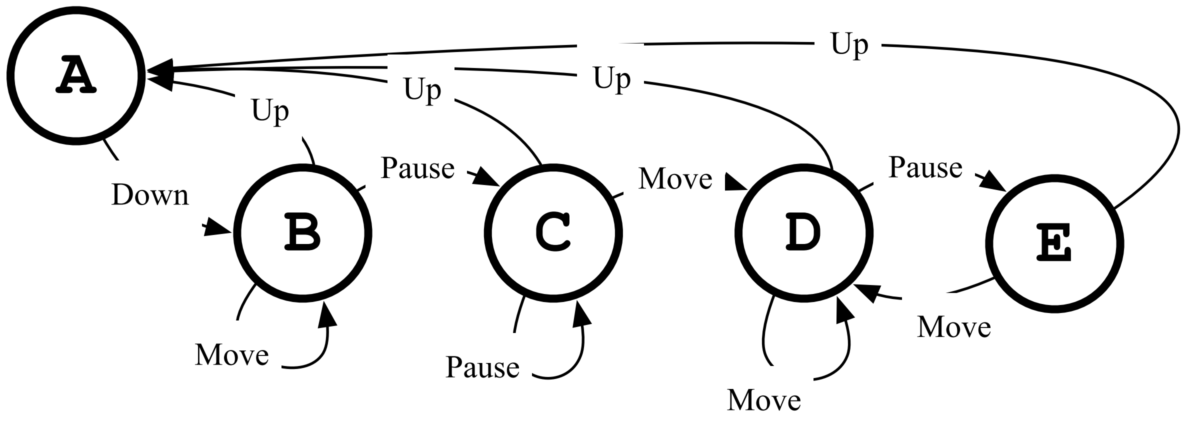
```
class FSM {
private:
  vector<State*> states;
  vector<Transition*> transitions;
  … (more)
};
```

```
class Transition {
public:
  string label;
  int signal;
  int next_state;
};
```

```
class State {
public:
  bool accept;
  string label;
  int failure_trans;
  vector<int> trans;
};
```

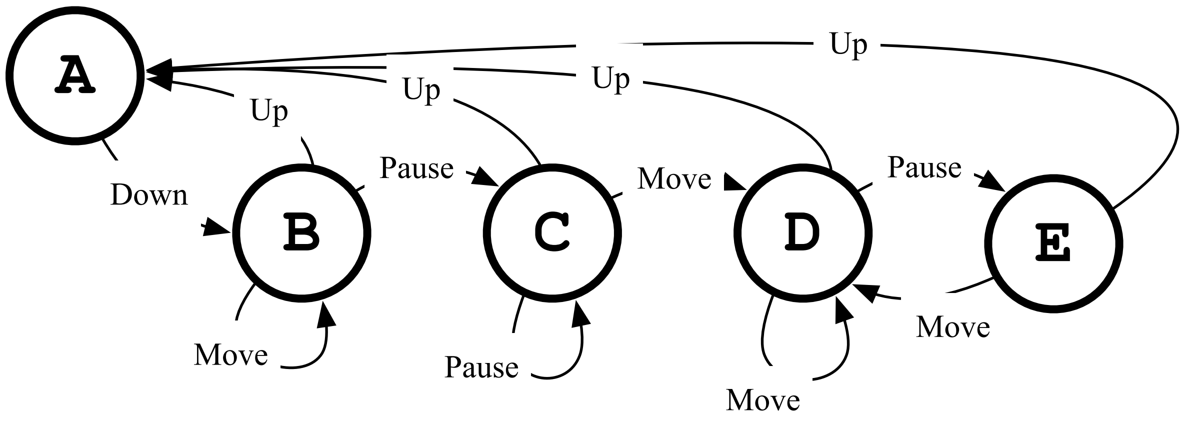
```
class FSM {
private:
  vector<State*> states;
  vector<Transition*> transitions;
};
```

The FSM below has five states: A, B, C, D, and E. Pointers to these State objects are stored in the 'states' vector. *That's the only place we retain State objects*.



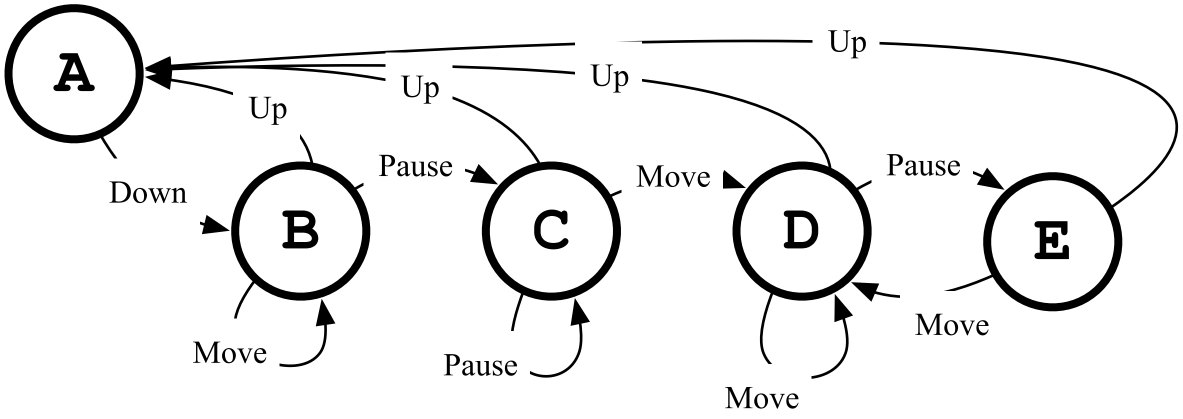
```
class FSM {
private:
  vector<State*> states;
  vector<Transition*> transitions;
};
```

It also has 12 transitions. Pointers to Transition objects are kept in the aptly-named *transitions* vector. Again, this is the only place where we persistently store Transitions.



```
class State {
public:
  bool accept;
  string label;
  int failure_trans;
  vector<int> trans;
};
```

If we have a reference to a single State object, we *do not have access to the FSM's data*. State objects only have access to the four variables listed above. Importantly: this means the State object doesn't contain Transition objects, directly or by pointer. They have transition *ids,* but not Transition objects.



So how do we know what data we have? And how do we gain access to the data we need? Consider this fragment of the FSM class definition:

```
class FSM {
private:
  vector<State*> states;
  vector<Transition*> transitions;
  int state;
  int default_state;
public:
  int addState(string label, bool is_accept_state);
};
```

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

```
int FSM::addState(string label,
                  bool is_accept_state) {
  // do some stuff, then return an integer
}
```

The return type is listed first.

```
int FSM::addState(string label,
                  bool is_accept_state) {
  // do some stuff, then return an integer
}
```

The namespace is listed second. This tells the compiler that we are implementing the FSM's function called addState. This is to resolve conflicts if there happen to be several functions called 'addState'. It also ensures that the compiler knows this is a member function of the FSM class.

The syntax is just the namespace followed by two colons.

```
int FSM::addState(string label,
                  bool is_accept_state) {
  // do some stuff, then return an integer
}
```

Next is the name of the function.

```
int FSM::addState(string label,
                  bool is_accept_state) {
  // do some stuff, then return an integer
}
```

Next is the formal parameter list. These are the variables that are given to us explicitly. We should be used to this by now: this is how we did things for all the homeworks up to the Graph one. That's where we started using C++ classes, and where lots of people started getting confused.

So here we have 'label' and 'is_accept_state' as explicit variables that we can read or write.

```
int FSM::addState(string label,
                  bool is_accept_state) {
  // do some stuff, then return an integer
}
```

But what about the FSM's other variables? Remember from the FSM class definition we had four variables:

```
class FSM {
private:
  vector<State*> states;
  vector<Transition*> transitions;
  int state;
  int default_state;
  … more …
};
```

How do we use these?

```
int FSM::addState(string label,
                  bool is_accept_state,
                  FSM* this,
                  vector<State*> states,
                  vector<Transition*> transitions,
                  int state,
                  int default_state) {
  // do some stuff, then return an integer
}
```

We have *implicit access* to the other member variables of the FSM, 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.

```
int FSM::addState(string label,
                  bool is_accept_state) {
  int id = states.size();
  State* st = new State;
  st->accept = is_accept_state;
  st->label = label;
  st->failure_trans = -1;
  states.push_back(st);
  if (id == 0) {
    default_state = id;
    setState(id);
  }
  return id;
}
```

Here is how I implemented the FSM::addState function. Explicitly provided variables are bold and black, implicit ones are italic and red.

```
int FSM::addState(string label,
                  bool is_accept_state) {
  int id = states.size();
  State* st = new State;
  st->accept = is_accept_state;
  st->label = label;
  st->failure_trans = -1;
  states.push_back(st);
  if (id == 0) {
    default_state = id;
    setState(id);
  }
  return id;
}
```

Notice how not all the implicitly available members are accessed. There is no mention of 'transitions', or 'state'.

```
int FSM::addState(string label,
                  bool is_accept_state) {
  int id = states.size();
  State* st = new State;
  st->accept = is_accept_state;
  st->label = label;
  st->failure_trans = -1;
  states.push_back(st);
  if (id == 0) {
    default_state = id;
    setState(id);
  }
  return id;
}
```

setState is called on '*this*', which is of type FSM*, and refers to the FSM that holds all the other implicit variables.

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
...
```