Lecture L:

Code Shape II — OOP

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- Midterm Results — Should be out
  - ✦ Again, if you did not do as well as you'd hoped, take the opportunity to reflect and figure out why. This is only a fraction of your grade. Remember we're here to help.
- Parser/AST feedback — if you have questions, feedback or regrade requests, please email cse401-staff@cs and cc your partner on all emails. It's good to fix serious problems before moving forward.
- Checking is DUE next Tuesday
  - ✦ Make sure to come (w/partner) to sections this Thursday

Plan for This Week + a bit

- I was wrong! Lectures are not in a ratio of 1:2, but more like 1:1, or even 2:1.
- So, we're cutting up Wednesday in half
- No worries!

Outline

Dynamic Method Dispatch

Object Layout & Field Access

Virtual Tables, Methods, & the Rest

Outline

Dynamic Method Dispatch

Object Layout & Field Access

Virtual Tables, Methods, & the Rest

What does this program print?

```
class One {
  int foo;
  int bar;
  void setFoo() {
    foo = 1;
  }
  int getFoo() {
    return foo;
  }
  void setBar(int bar) {
    this.bar = bar;
  }
  int getBar() {
    return bar;
  }
}

class Two extends One {
  int bar;
  void setFoo() {
    foo = 2; bar = 3;
  }
  int getIt()    { return bar; }
  void resetIt() { super.setBar(42); }
}
```

```
public class Main {
  public static void main(String[] args) {
    Two two = new Two();
    One one = two;

    one.setFoo();
    System.out.println(one.getFoo());

    one.setBar(17);
    two.setFoo();
    System.out.println(two.getBar());
    System.out.println(two.getIt());

    two.resetIt();
    System.out.println(two.getBar());
    System.out.println(two.getIt());
  }
}
```

Your Answer Here!

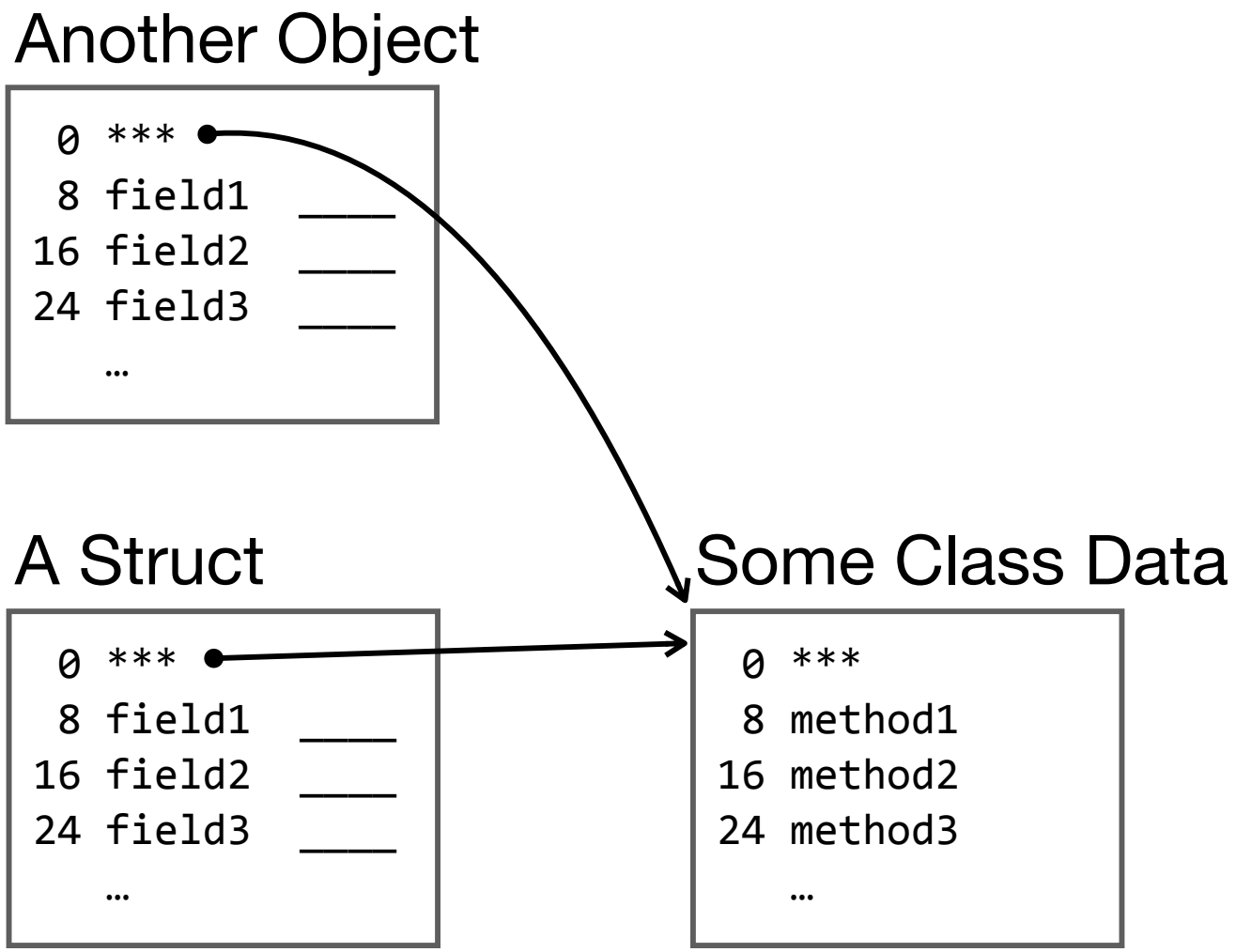
Naive View of Objects

- Objects have field data and methods
- When you inherit and override data or methods, they are no longer accessible
  - ✦ Think of it as a "map" (like the symbol table) from field names to data and from method names to functions
- This is subtly wrong
  - ✦ The object ***still has*** the overriden fields (and methods)
  - ✦ They can still be accessed indirectly, and so must be represented
- When a field or method is accessed, we need to resolve (dynamically or statically) ***which class*** we are talking about

Representation of Objects

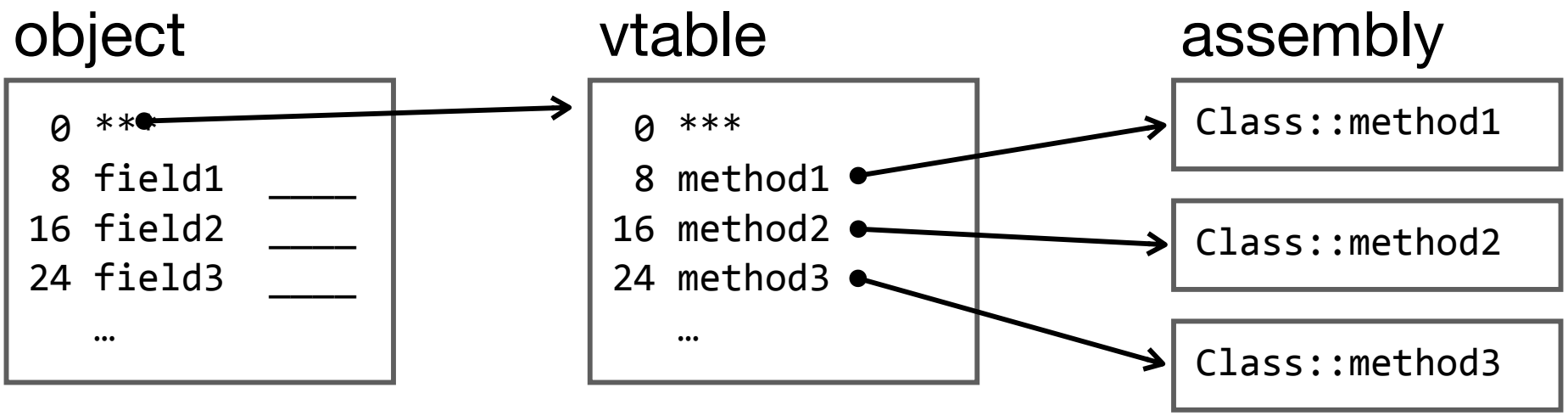
- Each object contains
  - ✦ storage for every field (instance variable)
    - including all inherited fields (public or private or …)
  - ✦ A pointer to a runtime data structure for the class
    - i.e. a method dispatch table (vtable, see next slide)
- An object is a struct (equipped with a pointer to a vtable)
- **Crucial!** — Fields are allocated regardless of whether they are shadowed. Superclass methods can still access those fields!

A Struct + Some Class Data



Method Dispatch Tables

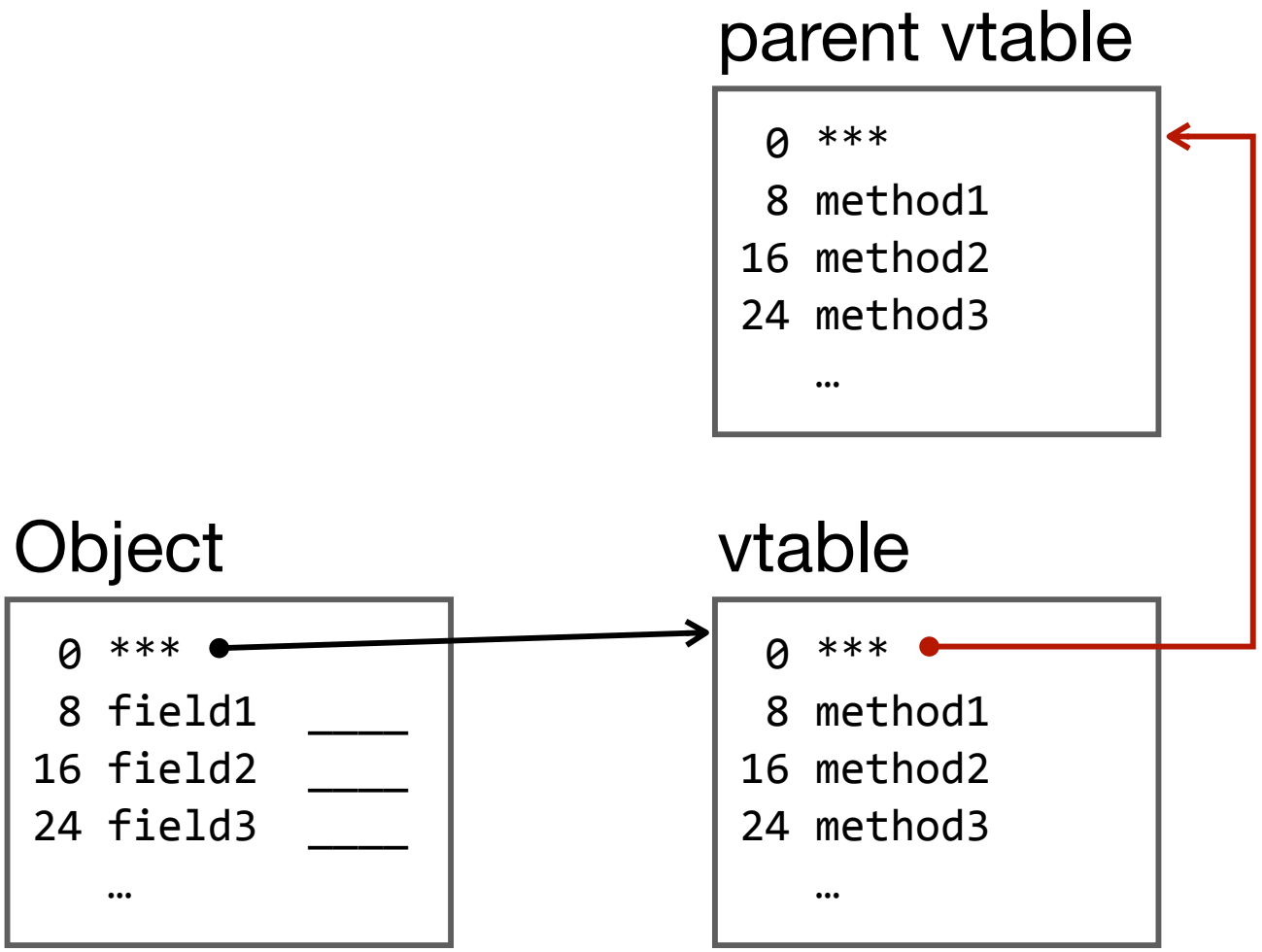
- One of these *per class*, not *per object*
- Often called a **virtual table**, a.k.a. vtable, (or vtbl or vtab)
  - The name *virtual function table* is the term from C++
- There is one pointer in the vtable for each method
  - points to the beginning of the compiled method code



Method Tables and Inheritance

- What about if a method is in some super-class of an object?
  - ✦ We need to “redirect” lookups to super-classes
- What does this remind you of?
  - ✦ Symbol tables — can’t we just use symbol tables at runtime? (conceptually yes, but…)
- Method Dispatch
  - ✦ look in table for object’s class for method
  - ✦ if not found there, look in the parent class table
  - ✦ if not found there, repeat (one class up)
  - ✦ error if we run out of parents
- The above method is really used for dynamic languages! (e.g. Python, Ruby, SmallTalk, Lua, JavaScript, Self, …)

Parent Class Pointers



Better — O(1) Method Dispatch

- The method table for an extended class has pointers to ***all*** inherited and local methods for that class
- The first part of the method table for the subclass has pointers for methods in the same order as for the superclass.
  - ***But***, pointers refer to the overriding method (if any)
  - Or, pointers refer to the original method if not overriden
  - So, dispatch can just be done with an indirect jump, regardless of the dynamic type of an object. The entry for method `foo()` is always at exactly the same offset
    - C code equiv: `(*(obj->vtbl[offset]))(params)`

Perverse Example — Revisited

```
class One {
  int foo;
  int bar;
  void setFoo() {
    foo = 1;
  }
  int getFoo() {
    return foo;
  }
  void setBar(int bar) {
    this.bar = bar;
  }
  int getBar() {
    return bar;
  }
}

class Two extends One {
  int bar;
  void setFoo() {
    foo = 2; bar = 3;
  }
  int getIt()    { return bar; }
  void resetIt() { super.setBar(42); }
}
```

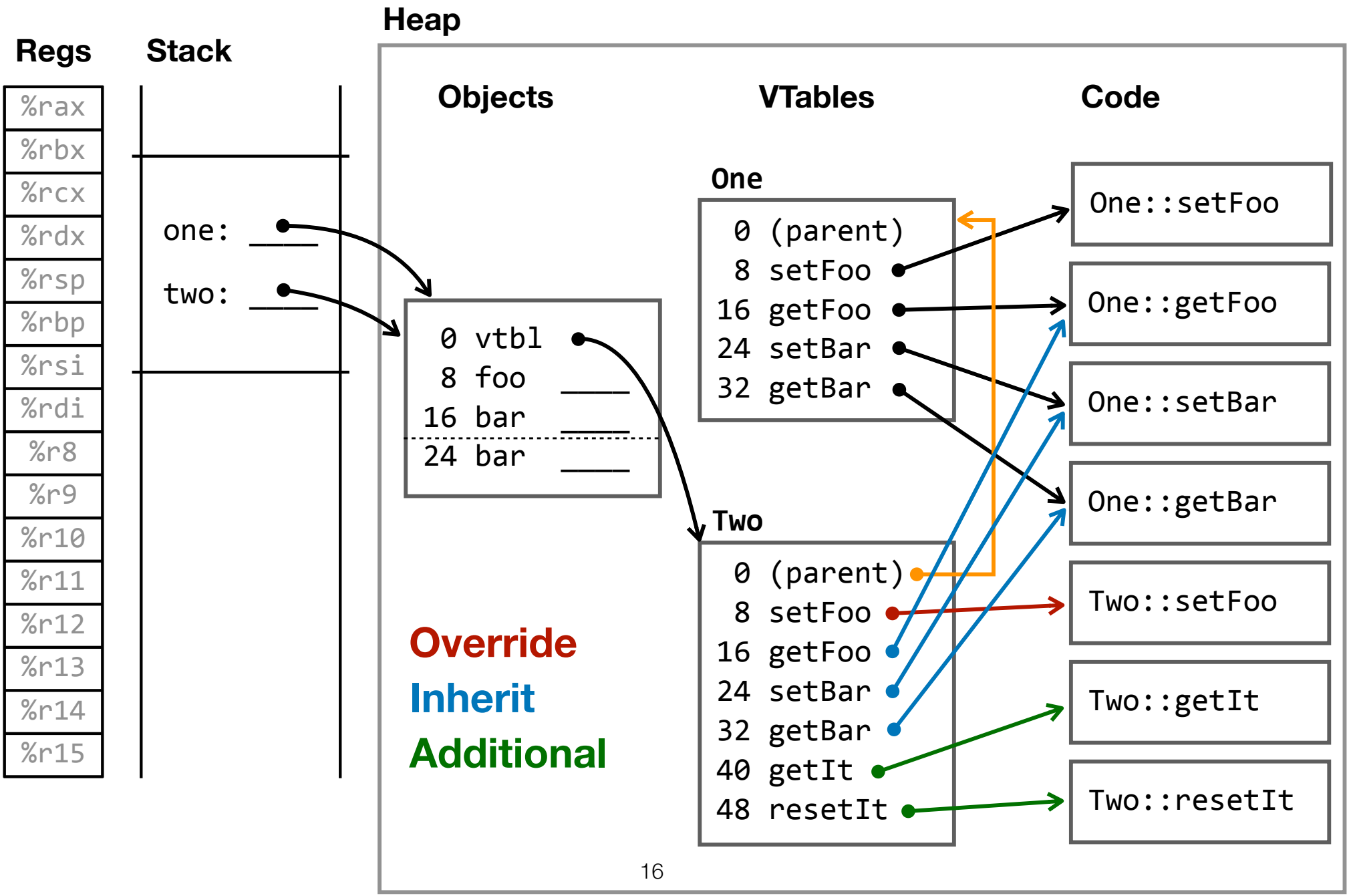
```
public class Main {
  public static void main(String[] args) {
    Two two = new Two();
    One one = two;

    one.setFoo();
    System.out.println(one.getFoo());

    one.setBar(17);
    two.setFoo();
    System.out.println(two.getBar());
    System.out.println(two.getIt());

    two.resetIt();
    System.out.println(two.getBar());
    System.out.println(two.getIt());
  }
}
```

Implementation



Method Dispatch Footnotes

- We don't need a pointer to the parent class vtable to implement method calls any more, but it is often still useful and important
  - e.g. checking *casting* and *instanceof*
- Multiple inheritance requires more complicated mechanisms
  - This is also true for classes that implement multiple interfaces.

Outline

Dynamic Method Dispatch

Object Layout & Field Access

Virtual Tables, Methods, & the Rest

Object Layout

- Typically, we allocate fields sequentially
- We need to follow any processor/OS/ABI conventions on the (address) alignment of structs/objects if present
  - ✦ *so, we may need to include padding bytes*
- Use the first (quad)-word of the object to hold a pointer to the method table (vtable)
- All objects are allocated on the heap (in Java)
  - ✦ In C++ objects can also be allocated on the stack or as globals

Object Field Access

- Example Source Code

```
int n = obj.fld
```

- x86-64

```
movq offset_obj(%rbp),%rax # load obj ptr
movq offset_fld(%rax),%rax # load fld, using obj ptr
movq %rax,offset_n(%rbp)   # store n (assignment)
```

- Same idea for references to `this.`*`fld`*
  - ✦ use the implicit `this` parameter passed to the method instead of a local variable to get the object address

Local Fields

- In Java or MiniJava, a method can refer to a field "`f`" of the object/class as either "`f`" or "`this.f`"
  - ✦ Both of these compile to the same code, using an implicit "`this`"
  - ✦ How does this work? There is a hidden, implicit "`this`" argument to all methods

Source-Level View

What you write

```
class One {
  …
  int getBar() {
    return bar;
  }
  void setBar(int bar) {
    this.bar = bar;
  }
}
…
  {
    obj.setBar(42);
    k = obj.getBar();
  }
```

What you really get

```
…
  int getBar(One this) {
    return this.bar;
  }
  void setBar(One this,
              int bar) {
    this.bar = bar;
  }
…
{
  setBar(obj,42);
  k = getBar(obj);
}
```

Who Decides “this”?

- Is “this” the responsibility of the ISA? the ABI? the Programming Language? the compiler?
  - ✦ Hmmm… if our ABI is language-specific (usually we assume the *C ABI*) then the ABI. If (like for real Java) there is a runtime/JIT compiler, then the compiler doesn’t need to share…
- For the MiniJava project, we’ll place “this” in %rdi (the first argument register) for every non-static method call
- *(further arguments go in remaining argument registers)*

Outline

Dynamic Method Dispatch

Object Layout & Field Access

Virtual Tables, Methods, & the Rest

MiniJava Method Tables (vtables)

- Generate these as initialized data in the assembly language source program (not on heap or stack)
  - *recall — these are essentially global constants*
- We need a ***naming convention*** that will work for the assembly language labels assigned to methods/classes
  - Methods — `classname$methodname`
  - VTables — `classname$$`
- The first entry in the method table points to the superclass method table. We shouldn't need to use it in the project, but allows for dynamic casts, checks, and can help you with debugging.

Aside: Overloading vs. Overriding

- Recall
  - ✦ Overloading — two methods with the same name (maybe in the same class) but with different numbers of and types of arguments (i.e. different signatures)
  - ✦ Overriding — a method with the same signature in a sub-class that shadows the super-class method.
- Overloading is more complicated to handle
  - ✦ (conveniently omitted from MiniJava)
- I will generally avoid talking about overloading in this class for the sake of simplicity

Perverse Example — VTables

```
class One {
  void setFoo() {…}
  int getFoo() {…}
  void setBar(int bar) {…}
  int getBar() {…}
}

class Two extends One {
  // override
  void setFoo() {…}
  // additional
  int getIt() {…}
  void resetIt() {…}
}
```

```
                .data
                # 0 = no superclass
One$$:          .quad 0
                .quad One$setFoo
                .quad One$getFoo
                .quad One$setBar
                .quad One$getBar
                #     superclass
Two$$:          .quad One$$
                .quad Two$setFoo
                .quad One$getFoo
                .quad One$setBar
                .quad One$getBar
                .quad Two$getIt
                .quad Two$resetIt
```

Method Table Layout

- The first 4 entries in `class Two`'s method table are pointers to methods in *exactly the same order* and *same offset* as in `One`'s method table
- **Thus**, the compiler knows the correct offset for a particular method pointer regardless of whether or not that method is overridden, and regardless of the actual (dynamic) type of the object on which the method is being invoked.
- This helps enable highly efficient method dispatch

Object Creation — `new`

- Steps required
  - ✦ Call the storage manager (e.g. malloc) to get raw bytes
  - ✦ Initialize the bytes to 0 (for Java, not in C++)
  - ✦ Store pointer to the class method table in the first 8 bytes of the object
  - ✦ Call the appropriate constructor function for the class, passing the new object as the "`this`" pointer (in `%rdi`)
    - note in MiniJava there are no constructors
  - ✦ The result of `new` is a pointer to the new object

Object Creation (assembly)

- Example Source Code

```
One one = new One(…);
```

- x86-64

```
movq   $nBytesNeeded,%rdi    # obj size + 8 for vtbl ptr
call   mallocEquiv           # addr of bytes returned in %rax
<zero out allocated obj, or use calloc>
leaq   One$$(%rip),%rdx      # get method table address
movq   %rdx,0(%rax)          # store vtbl ptr at beginning of obj
movq   %rax,%rdi             # set "this" argument
movq   %rax,offset_tmp(%rbp) # save "this" for after the call
<load constructor arguments>
call   One$One               # call constructor (if used)
movq   offset_tmp(%rbp),%rax # recover ptr to new object
movq   %rax,offset_one(%rbp) # store to variable "this"
```

A Very Weird Addressing Mode

- On the preceding slide you may find…

```
leaq  One$$(%rip)
```

  - ✦ what the heck?
- "I thought you're not allowed to reference %rip in assembly code!"
- This particular addressing mode is "PC-relative" addressing — somewhat obscure, not worth worrying about the precise meaning.
  - ✦ Important for a feature called "position independent code"
  - ✦ It's ok to just "do it this way" for the project (x86 🤦)

Constructor — vtable?

- Why don't we need a vtable lookup to find the right constructor to call?

- At compile time we know the actual class. It must be exactly the class name following `new`! So, we can generate a call instruction to the specific label statically.
  - *(same with any invocation of super.method(…) inside some piece of code. This particular method dispatch can be resolved statically)*

Method Calls

- Parameter passing — just like an ordinary C function, except we load a “this” pointer into the first argument %rdi
- We can get a pointer to the object’s method table from the first 8 bytes of the object
- We then jump indirectly through the method table

Method Call (assembly)

- Example Source Code

```
obj.method(…);
```

- x86-64

```
<load arguments into the registers as usual & needed>
movq   offset_obj(%rbp),%rdi   # the first arg is obj ptr ("this")
movq   0(%rdi),%rax            # load vtbl address into %rax
call   *offset_method(%rax)    # call the function whose addr
                               # is at the specified vtbl offset
```

- We can get the same effect as the last line with

```
addq   offset_method,%rax
call   *(%rax)
```

  - ✦ or

```
movq   offset_method(%rax),%rax
call   *%rax
```

Runtime Type Checking

- We can use the method table for the class as a "runtime representation" of the class — each class has a unique vtbl at a unique address
- The test for "`o instance of C`" is
  - Is o's method table pointer `&C$$`?
  - Recursively, get pointer to superclass method table from the method table and check that
  - Stop when you reach Object (or a null pointer, depending on Java vs. MiniJava)
- The same idea covers checking whether a downcast is legal or not

Next Time…

- Next Monday, we cover Optimization
- Next Wednesday (and Section) we cover the last bits needed for the compiler project
- Next Wednesday and Friday, we cover Dataflow program anlayses
  - ✦ We'll cover some very basic analyses, but this is a **very deep** topic that's very important for all kinds of tools that analyze programs (not just compilers)
- Beyond!
  - ✦ SSA (IR), and Backend Compiler Optimizations!