

# CSE P 501 – Compilers

Code Shape II – Objects & Classes

Hal Perkins

Autumn 2021

# Agenda

- Object representation and layout
- Field access
- What is **this**?
- Object creation - **new**
- Method calls
  - Dynamic dispatch
  - Method tables
  - Super
- Runtime type information

(As before, more generality than we actually need for the project)

# What does this program print?

```
class One {
    int tag;
    int it;
    void setTag()      { tag = 1; }
    int getTag()      { return tag; }
    void setIt(int it) { this.it = it; }
    int getIt()       { return it; }
}

class Two extends One {
    int it;
    void setTag() {
        tag = 2; it = 3;
    }
    int getThat() { return it; }
    void resetIt() { super.setIt(42); }
}

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

    one.setTag();
    System.out.println(one.getTag());

    one.setIt(17);
    two.setTag();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
    two.resetIt();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
}
```

# Your Answer Here

# Object Representation

- The naïve explanation is that an object contains:
  - Fields declared in its class and in all superclasses
    - Redeclaration of a field hides (shadows) superclass instance – but the superclass field is still there and is in scope for, and accessed by, superclass methods
  - All methods declared in its class and all superclasses
    - Redeclaration of a method overrides (replaces) – but overridden methods can still be accessed by super. , and all relevant methods are part of the object’s “behavior”
- When a method is called, the appropriate method “inside” that particular object is called
  - Regardless of the static (compile-time) type of the variable that points to the object
    - (But we really don’t want to copy/duplicate all those methods, do we?)

# Actual representation

- 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 its class
    - Key component: method dispatch table (vtable, next slide)
- An object is basically a C struct
- Fields hidden (shadowed) by declarations in subclasses are *still* allocated in the object and are accessible from superclass methods (using offsets assigned as part of superclass object layout)
  - Subclass methods access new fields using offsets assigned when subclass fields appended to superclass struct layout

# Method Dispatch Tables

- One of these per class, not per object
- Often called “vtable”, “vtbl”, or “vtab”
  - (virtual function table – term from C++; standard term in all languages with dynamic dispatch)
- One pointer for each method in the vtable – points to beginning of compiled method code

# Method Tables and Inheritance

- A naïve, really simple implementation – dictionaries!
  - One method table for each class containing names of methods declared locally in that class (keys), with pointers to compiled code for each method (values)
  - Method table also contains a pointer to parent class method table
  - Method dispatch:
    - Look in table for object’s class and use if method found
    - Look in parent class table if not local
    - Repeat
    - “Message not understood” if you can’t find it after search
  - Actually used in typical implementations of some dynamic languages (e.g. Ruby, SmallTalk, etc.)



# Better: $O(1)$ Method Dispatch

- Idea: Method table for extended class has pointers to *all* inherited and local methods for that class
- First part of method table for extended class has pointers for the same methods in the same order as the parent class
  - BUT pointers actually refer to overriding methods if any
  - So, dispatch for a method can be done with an indirect jump using a fixed method offset known at compile time, regardless of whether this points to an overriding method
    - In C: `*(object->vtbl[offset])(parameters)`
- Pointers to additional methods declared (added) in subclass are included in the vtable after pointers to inherited or overridden superclass methods

# Perverse Example Revisited

```
class One {
    int tag;
    int it;
    void setTag() { tag = 1; }
    int getTag() { return tag; }
    void setIt(int it) {this.it = it;}
    int getIt() { return it; }
}
class Two extends One {
    int it;
    void setTag() {
        tag = 2; it = 3;
    }
    int getThat() { return it; }
    void resetIt() { super.setIt(42); }
}
```

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

    one.setTag();
    System.out.println(one.getTag());

    one.setIt(17);
    two.setTag();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
    two.resetIt();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
}
```

# Implementation

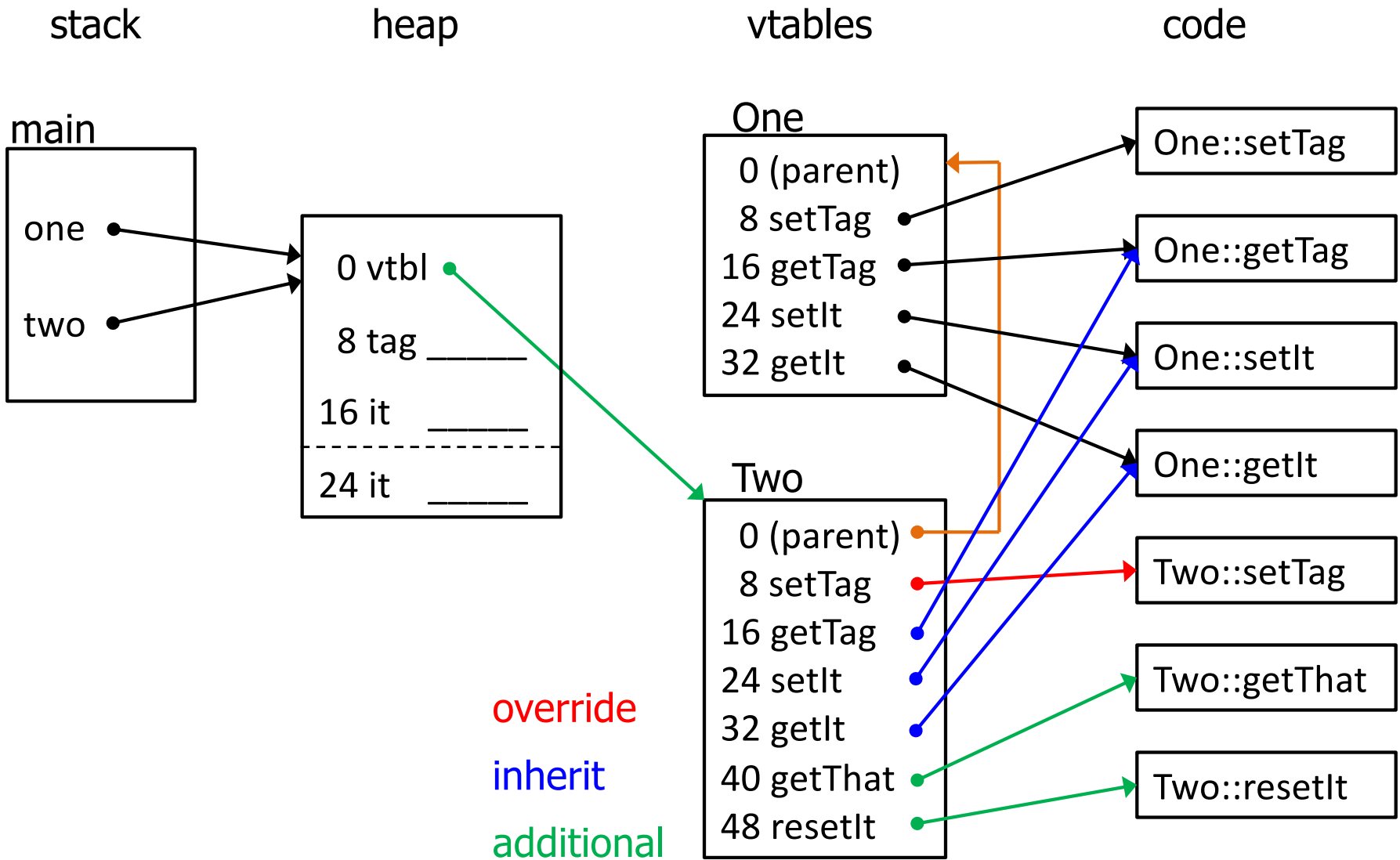
stack

heap

vtables

code

# Implementation



# Method Dispatch Footnotes

- Don't need a pointer to parent class vtable to implement method calls, but often useful for other purposes
  - Casts and instanceof
- Multiple inheritance requires more complex mechanisms
  - Also true for multiple interfaces

# Now What?

- Need to explore
  - Object layout in memory
  - Compiling field references
    - Implicit and explicit use of “this”
  - Representation of vtables
  - Object creation – new
  - Code for dynamic dispatch
  - Runtime type information – instanceof and casts

# Object Layout

- Typically, allocate fields sequentially
- Follow processor/OS alignment conventions for structs/objects when appropriate/available
  - Include padding bytes for alignment as needed
- Use first word of object to hold pointer to method table (vtable)
- Objects are allocated on the heap (in Java)
  - Unlike C++ where objects can also be on stack
  - No bytes reserved for object data in generated code – use either heap or stack as appropriate

# Object Field Access

- Source

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

- x86-64

- Assuming that obj is a local variable in the current method's stack frame

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

- Same idea used to reference fields of “this”

- Use implicit “this” parameter passed to method instead of a local variable to get object address



# Local Fields

- A method can refer to fields in the receiving object either explicitly as “this.f” or implicitly as “f”
  - Both compile to the same code – an implicit “this.” is assumed if not written explicitly
  - A pointer to the object (i.e., “this”) is an implicit, hidden parameter to all methods

# Source Level View

What you write:

```
int getIt() {  
    return it;  
}  
void setIt(int it) {  
    this.it = it;  
}  
...  
obj.setIt(42);  
k = obj.getIt();
```

What compiler really does:

```
int getIt(Objtype this) {  
    return this.it;  
}  
void setIt(ObjType this, int it) {  
    this.it = it;  
}  
...  
setIt(obj, 42);  
k = getIt(obj);
```

# x86-64 “this” Convention (C++)

- “this” is an implicit first parameter to every non-static method
- Address of object (“this”) placed in %rdi for every non-static method call
- Remaining parameters (if any) in %rsi, etc.
- We’ll use this convention in our project

# MiniJava Method Tables (vtbls)

- Generate these as initialized data in the assembly language source program
- Need to pick a naming convention for assembly language labels. This will work for us:
  - For methods, classname\$methodname
    - Need something more sophisticated for overloading
  - For the vtables themselves, classname\$\$
- First method table entry points to superclass table (we might not use it in our project, but is helpful if you add instanceof or type cast checks)

# Method Tables For Perverse Example (gcc/as syntax)

```
class One {  
    void setTag() { ... }  
    int getTag() { ... }  
    void setIt(int it) {...}  
    int getIt() { ... }  
}
```

```
class Two extends One {  
    void setTag() { ... } // override  
    int getThat() { ... } // additional  
    void resetIt() { ... }  
}
```

```
One$$: .data  
       .quad 0          # no superclass  
       .quad One$setTag  
       .quad One$getTag  
       .quad One$setIt  
       .quad One$getIt
```

```
Two$$: .quad One$$    # superclass  
       .quad Two$setTag  
       .quad One$getTag  
       .quad One$setIt  
       .quad One$getIt  
       .quad Two$getThat  
       .quad Two$resetIt
```

# Method Table Layout

Key point: First entries in Two's method table are pointers to methods in *exactly the same order* as in One's method table

- Actual pointers reference method appropriate for objects of each class (inherited or overridden)

∴ Compiler knows correct offset for a particular method pointer *regardless of whether that method is overridden* and *regardless of the actual type* (dynamic) or subclass of the object

# Object Creation – new

## Steps needed

- Call storage manager (malloc or equivalent) to get the raw bytes
- Initialize bytes to 0 (for Java, not in e.g., C++ \*)
- Store pointer to method table (vtbl) in the first 8 bytes of the object
- Call a constructor with “this” pointer to the new object in %rdi and other parameters as needed
  - (Not in MiniJava since we don’t have constructors)
- Result of new is a pointer to the new object

\*Recent versions of C++ have new strange and wonderful rules about default initialization. Left as an exercise for aspiring programming language lawyers.

# Object Creation

- Source

```
One one = new One(...);
```

- x86-64

```
movq    $nBytesNeeded,%rdi    # obj size + 8 (include space for vtbl ptr)
call    mallocEquiv           # addr of allocated bytes returned in %rax
<zero out allocated object, or use calloc instead of malloc to get the bytes>
leaq    One$$(%rip),%rdx      # get method table address
movq    %rdx,0(%rax)          # store vtbl ptr at beginning of object
movq    %rax,%rdi             # set up "this" for constructor
movq    %rax,offset_temp(%rbp) # save "this" for later (or maybe pushq)
<load constructor arguments>  # arguments (if needed)
call    One$One               # call ctor if we have one (no vtbl lookup)
movq    offset_temp(%rbp),%rax # recover ptr to object
movq    %rax,offset_one(%rbp) # store object reference in variable one
```



# Constructor

- Why don't we need a vtable lookup to find the right constructor to call?
- Because at compile time we know the actual class (it says so right after "new"), so we can generate a call instruction to a known label
  - Same with super.method(...) or superclass constructor calls – at compile time we know all of the superclasses (need superclass details to compile subclass and construct method tables), so we know statically which class "super.method" belongs to

# Method Calls

- Steps needed
  - Parameter passing: just like an ordinary C function, except load a pointer to the object in `%rdi` as the first (“this”) argument
  - Get a pointer to the object’s method table from the first 8 bytes of the object
  - Jump indirectly through the method table

# Method Call

- Source

`obj.method(...);`

- x86-64

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

\*Can get same effect with:

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

or with:

```
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 one vtable at a unique address
- The test for “o instanceof C” is:
  - Is o’s method table pointer == &C\$\$ ?
    - If so, result is “true”
  - 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 whether there is a ultimate superclass of everything)
    - If no match by the top of the chain, result is “false”
- Same test as part of check for legal downcast (e.g., how to check for ClassCastException in (type)obj cast)

# Coming (& past) Attractions

- Other IRs besides ASTs
- Code analysis and optimization
- Industrial-strength back end (register allocation, instruction selection & scheduling)
- Other topics as time allows
  - GC? Dynamic languages? JVM? What else?
- And simple code generation for the project