CSE 401 – Compilers

Code Shape II – Objects & Classes
Hal Perkins
Autumn 2010
Agenda

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

(As before, more generality than we strictly 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());
}
Object Representation

- The naïve explanation is that an object contains
  - Fields declared in its class and in all superclasses
    - Redeclaration of a field hides superclass instance
      - But the hidden fields are still present and accessible in superclass methods
  - Methods declared in its class and all superclasses
    - Redeclaration of a method overrides (replaces)
      - But overridden methods can still be accessed by super....

- When a method is called, the method “inside” that particular object is called
  - (But we really don’t want to copy all those methods, do we?)
Actual representation

- Each object contains
  - An entry for each field (variable)
  - A pointer to a runtime data structure describing the class
    - Key component: method dispatch table
- Basically a C struct
- Fields hidden by declarations in extended classes are *still* allocated in the object and are accessible from superclass methods
Method Dispatch Tables

- Often known as “vtables”
- One pointer per method - points to beginning of method code
- Dispatch table offsets fixed at compile time
- One instance of this per class, not per object
Method Tables and Inheritance

- Simple implementation
  - Method table for extended class has pointers to methods declared in it
  - Method table also contains a pointer to parent class method table
- Method dispatch
  - Look in current table and use if method declared locally
  - Look in parent class table if not local
  - Repeat
- Actually used in some dynamic systems (e.g. SmallTalk, Ruby, etc.)
O(1) Method Dispatch

- Idea: First part of method table for extended class has pointers for same methods in same order as parent class
  - BUT pointers actually refer to overriding methods if these exist
  - \therefore Method dispatch is indirect using fixed offsets known at compile time – O(1)
    - In C: *(object->vtbl[offset])(parameters)
- Pointers to additional methods in extended class are included in the table following inherited/overridden ones
Method Dispatch Footnotes

- Still want pointer to parent class method table for other purposes
  - Casts and instanceof

- Multiple inheritance requires more complex mechanisms
  - Also true for multiple interfaces
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); }
    }

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
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 when appropriate
- Use first word of object for pointer to method table/class information
- Objects are allocated on the heap
  - No actual bits in the generated code
Local Variable Field Access

- **Source**

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

- **X86**

  Assuming that obj is a local variable in the current method

  ```
  mov eax,[ebp+offset_obj] ; load obj ptr
  mov eax,[eax+offset_fld] ; load fld
  mov [ebp+offset_n],eax    ; store n
  ```
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 present explicitly
- Mechanism: a reference to the current object is an implicit parameter to every method
  - Can be in a register or on the stack
Source Level View

- When you write:

```c
void setIt(int it) {
    this.it = it;
}
...
obj.setIt(42);
```

- You really get:

```c
void setIt(ObjType this, int it) {
    this.it = it;
}
...
setIt(obj, 42);
```
x86 Conventions (C++)

- ecx is traditionally used as “this”
- Add to method call
  
  ```
  mov ecx, receivingObject ; ptr to object
  ```

  - Do this after arguments are evaluated and pushed, right before dynamic dispatch code that actually calls the method
  - Need to save ecx in a temporary or on the stack in methods that call other non-static methods
    - One possibility: add to prologue
    - Following examples aren’t careful about this
x86 Local Field Access

- **Source**
  ```
  int n = fld;  or  int n = this.fld;
  ```

- **X86**
  ```
  mov   eax,[ecx+offset_fld]    ; load fld
  mov   [ebp+offset_n],eax      ; store n
  ```
x86 Method Tables (vtbls)

- Generate these as initialized data in the assembly language source program
- Need to pick a naming convention for method labels; suggest:
  - For methods, classname$methodname
    - Would need something more sophisticated for overloading
  - For the vtables themselves, classname$$
- First method table entry points to superclass table
- Also useful: second entry points to default (0-argument) constructor (if you have constructors)
Method Tables For Perverse Example (Intel/Microsoft asm)

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

class Two extends One {
    void setTag() { … }  
    int getThat() { … }  
    void resetIt() { … }  
}

.data
One$$   dd  0         ; no superclass
        dd  One$One
        dd  One$setTag
        dd  One$getTag
        dd  One$setIt
        dd  One$getIt
Two$$   dd  One$$          ; parent
        dd  Two$Two
        dd  Two$setTag
        dd  One$getTag
        dd  One$setIt
        dd  One$getIt
        dd  Two$getThat
        dd  Two$resetIt
Method Table Footnotes

- Key point: First four non-constructor method entries in Two’s method table are pointers to methods declared in One in *exactly the same order*

  ∴ Compiler knows correct offset for a particular method pointer regardless of whether that method is overridden and regardless of the actual (dynamic) type of the object
Object Creation – new

Steps needed

- Call storage manager (malloc or similar) to get the raw bits
- Store pointer to method table in the first 4 bytes of the object
- Call a constructor (with pointer to the new object, `this`, in `ecx`)
- Result of new is pointer to the constructed object
Object Creation

- **Source**
  
  ```java
  One one = new One(...);
  ```

- **X86**
  
  ```assembly
  push nBytesNeeded ; obj size + 4
  call mallocEquiv ; addr of bits returned in eax
  add esp,4 ; pop nBytesNeeded argument
  lea edx,One$$ ; get method table address
  mov [eax],edx ; store vtab ptr at beginning of object
  mov ecx,eax ; set up “this” for constructor
  push ecx ; save ecx (constructor might clobber it)
  <push constructor arguments> ; arguments (if needed)
  call One$One ; call constructor (no vtab lookup needed)
  <pop constructor arguments> ; (if needed)
  pop eax ; recover ptr to object
  mov [ebp+offset_one],eax ; store object reference in variable one
  ```
Constructor

- Only special issue here is generating call to superclass constructor
  - Same issues as super.method(...) calls – we know the superclass name, so just generate a direct call to the appropriate method.
Method Calls

Steps needed

- Push arguments as usual
- Load pointer to object in ecx (this)
- Get pointer to method table from first 4 bytes of object
- Jump indirectly through method table
- Restore ecx to point to current object (if needed after method returns)
  - Useful hack: push ecx in the function prologue so it is always in the stack frame at a known location & reload when needed if it might be clobbered
Method Call

- **Source**
  
  \[ \text{obj.meth(...);} \]

- **X86**
  
  \(<\text{push arguments from right to left}> \quad \text{; (as needed)}\)
  
  \(\text{mov \ } \text{ecx,}[\text{ebp+offset}_{\text{obj}}] \quad \text{; get pointer to object}\)
  
  \(\text{mov \ } \text{eax,}[\text{ecx}] \quad \text{; get pointer to method table}\)
  
  \(\text{call \ } \text{dword ptr}[\text{eax+offset}_{\text{meth}}] \quad \text{; call indirect via method tbl}\)
  
  \(<\text{pop arguments}> \quad \text{; (if needed)}\)
  
  \(\text{mov \ } \text{ecx,}[\text{ebp+offset}_{\text{ecx}_\text{temp}}] \quad \text{; (if needed)}\)
Runtime Type Checking

- Use the method table for the class as a “runtime representation” of the class
- 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 how you represent things)
    - If no match by the top of the chain, result is “false”
- Same test as part of check for legal downcast
Coming Attractions

- Code generation: register allocation, instruction selection & scheduling
  - Industrial-strength versions plus a simpler “get it to work” scheme for our project
- Code optimization