Reminders/
Announcements

• Midterms are graded!
  – I’ll stop a few minutes early so you can pick them
    up at the end of class
• Project part 3 due Friday, March 1 (1 week)
• Part 4 will be due on Friday, March 15 (last
day of class). I will put the assignment out
next week – likely before part 3 is due, in case
anyone wants to get a head start.
Agenda

- Finish last topic from Wednesday – 2D arrays
- Then, object-oriented code generation (may spill into Monday):
  - Object representation and layout
  - Field access
  - What is this?
  - Object creation - new
  - Method calls
    - Dynamic dispatch
    - Method tables
    - Super
  - Runtime type information

2-D Arrays

- C, etc. use row-major order
  - E.g., an array with 3 rows and 2 columns is stored in this sequence: a[0][0], a[0][1], a[1][0], a[1][1], a[2][0], a[2][1]
- Fortran uses column-major order (and indexed from 1)
  - So, a(1,1), a(2,1), a(3,1), a(1,2), a(2,2), a(3,2)
  - Can’t naively pass multidimensional array references between C and Fortran
- Java does not have “real” 2-D arrays. A Java 2-D array is a pointer to a list of pointers to the rows
a[i][j] in C/C++/etc.

• To find a[i][j], we need to know
  – Values of i and j
  – How many columns the array has
• Location of a[row][column] is (assuming 0-indexed)
  – Base of a + (row*(#of columns) + column)*elementSize
• Can factor to pull out any constant part and evaluate that at compile or link time – avoids recalculating at runtime
  – E.g., a[1][5] becomes a + 15 if a has 10 columns and byte-sized elements.

Agenda

• Finish last topic from Wednesday – 2D arrays
• Then, object-oriented code generation (may spill into Monday):
  – Object representation and layout
  – Field access
  – What is this?
  – Object creation - new
  – Method calls
    • Dynamic dispatch
    • Method tables
    • Super
  – Runtime type information
Motivating Exercise: 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());
}

Winter 2013
UW CSE 401 (Michael Ringenburg)
Motivating Exercise: 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(); // tag<-2, Two::it<-3
    System.out.println(one.getTag()); // 2
    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());
}

Winter 2013
UW CSE 401 (Michael Ringenburg)
Motivating Exercise: 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();  // tag<-2, Two::it<-3
    System.out.println(one.getTag());  // 2
    one.setIt(17);  // One::it<-17
    two.setTag();  // tag<-2, Two::it<-3
    System.out.println(two.getTag());
    two.resetIt();
    System.out.println(two.getTag());
    System.out.println(two.getThat());
    System.out.println(two.getIt());
    System.out.println(two.getTag());
}
Motivating Exercise: What does this program print?

```java
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(); // tag<-2, Two::it<-3
    System.out.println(one.getTag()); // 2
    one.setTag(); // One::it<-17
    System.out.println(one.getTag()); // 17
    System.out.println(two.getIt()); // 3
    two.setTag();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
    two.resetIt();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
}
```

Winter 2013  
UW CSE 401 (Michael Ringenburg)
Motivating Exercise: What does this program print?

```java
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(); // tag<2, Two::it<3
    System.out.println(one.getTag()); // 2
    one.setTag(17); // One::it<17
    two.setTag(); // tag<2, Two::it<3
    System.out.println(two.getTag()); // 17
    System.out.println(two.getIt()); // 3
    two.resetIt(); // One::it<42
    System.out.println(two.getIt()); // 42
    System.out.println(two.getThat());
}
```

Winter 2013  UW CSE 401 (Michael Ringenburg)
Object Representation

- How do we represent objects so that we can achieve this behavior?
- The naive explanation is that an object contains
  - Fields declared in its class and in all superclasses
    - Redeclaration of a field hides superclass instance – but the superclass field is still there somewhere...
  - 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 (instance variable)
  - A pointer to a runtime data structure describing the class
    - Key component: method dispatch table
- Basically looks like a C struct with an extra pointer
- Fields hidden by declarations in extended classes are still allocated in the object and are still accessible from superclass methods
Method Dispatch Tables

• What are these tables?
  – Collection of pointers to methods
  – One pointer per method – points to beginning of
    method code

• One of these per class, not per object
  – Each object contains a pointer to the table for its
    runtime class, to ensure correct dynamic dispatch.

• Often known as “vtables” (virtual method tables)

• The table structure is fixed at compile time

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
    • Otherwise, look in parent class table
    • Repeat (grandparent, great-grandparent, etc.)
  – Actually used in typical implementations of some
    dynamic languages (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
  - Means method dispatch uses a fixed table offsets known at compile time, regardless of the runtime type – therefore, O(1)
    - In C, object->foo(params) becomes:
      
      *(object->vtbl[foo_OFFSET])(params)
    
- 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 for other purposes
  - Anytime you need runtime type information, e.g., casts and instanceof
- Multiple inheritance requires more complex mechanisms
  - Also true for multiple interfaces
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

One
- vtable
- tag
- it

Two
- vtable
- One::tag
- One::it
- it

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); }
}
Implementation

```java
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());
}
```

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 struct/object alignment conventions when appropriate/available
- Use first word of object for pointer to method table/class information

Local Variable Field Access

- Source
  \[
  \text{int } n = \text{obj.fld};
  \]
- X86
  - Assuming that obj is a local variable in the current method
    \[
    \begin{align*}
    &\text{mov } \text{eax},[\text{ebp+}\text{offset}_{\text{obj}}]; \text{ load obj ptr} \\
    &\text{mov } \text{eax},[\text{eax+}\text{offset}_{\text{fld}}]; \text{ load fld} \\
    &\text{mov } [\text{ebp+}\text{offset}_{n}],\text{eax}; \text{ store n (from assignment)}
    \end{align*}
    \]
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:
  ```java
  int it;
  void setIt(int i) {
    it = i;
  }
  ...
  obj.setIt(42);
  ```

• You really get:
  ```java
  int it;
  void setIt(ObjType this, int i) {
    this.it = i;
  }
  ...
  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: push or save in method 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
  ```

- Notice that if we have this stored in a register (e.g., ecx), we do one less load from memory than a standard field access.
x86 Method Tables

- Generate these as initialized data in the assembly language source program
- Need to pick a naming convention for method labels; one possibility:
  - For methods, `classname$methodname`
    - Would need something more sophisticated for overloading
  - For the vtables themselves, `classname$`
- By convention, 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 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() { ... }
}

.class One
    .data
    .data
    .data
    One$$ dd 0 ; no superclass
    dd OneSOne
    dd OneSsetTag
    dd OneSgetTag
    dd OneSsetIt
    dd OneSgetIt

.class Two
    .data
    .data
    .data
    Two$$ dd One$$ ; parent
    dd TwoSTwo
    dd TwoSsetTag
    dd OneSgetTag
    dd OneSsetIt
    dd OneSgetIt
    dd TwoSgetThat
    dd TwoSresetIt
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
  - Makes dynamic dispatch easy

Object Creation – new

- Steps needed
  - Call storage manager (malloc or similar) to get the memory
  - 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, with constructors

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

- **X86**
  
  ```
  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 eax ; in case constructor clobbers eax
  <push constructor arguments> ; arguments (if needed)
  call One$One ; call constructor (no vtable 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 at compile time, 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
  
  obj.meth(...);

• X86
  
  <push arguments from right to left>  ; (as needed)
  mov  ecx,[ebp+offset_obj]  ; get pointer to object
  mov  eax,[ecx]  ; get pointer to method table
  call  [eax+offset_meth]  ; call indirect via method tbl
  <pop arguments>  ; (if needed)
  mov  ecx,[ebp+offset_ecxtemp]  ; (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

- x86-64: what changes; what doesn’t
- Simple code generation for project