Polymorphism

Back to Procedure Calls

- Static Method Invocations and Procedure Calls
  - target method/procedure address is known statically

  - in Java
    - static methods are class methods
      - invoked by naming the class, not an object
      ```java
      public class A {
        static void ping () {} 
      }
      public class Foo {
        static void foo () {
          A.ping ();
        }
      }
      ```

    - in C
      - specify procedure name
      ```c
      void ping () {} 
      void foo () {
        ping ();
      }
      ```

Readings for Next Two Lectures

- Text
  - Switch Statements, Understanding Pointers
    - 2nd ed: 3.6.7, 3.10
    - 1st ed: 3.6.6, 3.11
Polymorphism

- Invoking a method on an object in Java
  - variable that stores the object has a static type
  - object reference is dynamic and so is its type
    - object's type must implement the type of the referring variable
    - but object's type may override methods of this base type

Polymorphic Dispatch

- target method address depends on the type of the referenced object
- one call site can invoke different methods at different times

```java
class A {
  void ping () {}
  void pong () {}
}
class B extends A {
  void ping () {}
  void wiff () {}
}
```

```java
static void foo (A a) {
  a.ping ();
  a.pong ();
}
static void bar () {
  foo (new A());
  foo (new B());
}
```

Which ping gets called?

Dynamic Jumps in C

- Function pointer
  - a variable that stores a pointer to a procedure
  - declared
    - `<return-type> (*<variable-name>)(<formal-argument-list>);`
  - used to make dynamic call
    - `<variable-name> (<actual-argument-list>);`
- Example

```c
void ping () {}
void foo () {
  void (*aFunc) ();
  
  aFunc = ping;
  aFunc ();
}
calls ping
```
Use a struct to store jump table
• drawing on previous example of A ...

Declaration of A's jump table and code

```c
struct A {
  void (*ping)();
  void (*pong)();
};

void A_ping () { printf("A_ping\n"); }
void A_pong () { printf("A_pong\n"); }

struct A* new_A () {
  struct A* a = (struct A*) malloc(sizeof(struct A));
  a->ping = A_ping;
  a->pong = A_pong;
  return a;
}
```

Create an instance of A's jump table

```c
struct B {
  void (*ping)();
  void (*pong)();
  void (*wiff)();
};

void B_ping () { printf("B_ping\n"); }
void B_wiff () { printf("B_wiff\n"); }

struct B* new_B () {
  struct B* b = (struct B*) malloc(sizeof(struct B));
  b->ping = B_ping;
  b->pong = A_pong;
  b->wiff = B_wiff;
  return b;
}
```

Create an instance of B's jump table

* invoking ping and pong on an A and a B ...

```c
void foo (struct A* a) {
  a->ping ();
  a->pong ();
}
void bar () {
  foo (new_A ());
  foo ((struct A*) new_B ());
}
```

Dispatch Diagram for C (data layout)
Dispatch Diagram for C (the dispatch)

ISA for Polymorphic Dispatch

‣ How do we compile
• a->ping () ?

‣ Pseudo code
• pc ← m[r[1]+0*4]

‣ Current jumps supported by ISA

‣ We will benefit from a new instruction in the ISA
• that jumps to an address that is stored in memory

Switch Statements

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>jump absolute</td>
<td>pc ← a</td>
<td>j a</td>
<td>b------aaaaaaa</td>
</tr>
<tr>
<td>indirect</td>
<td>pc ← r[t] + (o==pp*2)</td>
<td>j o(rt)</td>
<td>ctpp</td>
</tr>
<tr>
<td>dbl-ind jump b+o</td>
<td>pc ← m[r[t] + (o==pp*2)]</td>
<td>j *o(rt)</td>
<td>dtpp</td>
</tr>
</tbody>
</table>
Switch Statement

```c
int i;
int j;

void foo () {
    switch (i) {
        case 0: j=10; break;
        case 1: j=11; break;
        case 2: j=12; break;
        case 3: j=13; break;
        default: j=14; break;
    }
}
```

Human vs Compiler

- Benefits for humans
  - the syntax models a common idiom: choosing one computation from a set
- But, switch statements have interesting restrictions
  - case labels must be static, cardinal values
    - a cardinal value is a number that specifies a position relative to the beginning of an ordered set
    - for example, integers are cardinal values, but strings are not
  - case labels must be compared for equality to a single dynamic expression
    - some languages permit the expression to be an inequality
- Do these restrictions benefit humans?
  - have you ever wanted to do something like this?

```c
switch (i,j) {
    case i>0:
    case i==0 & j>a:
    case i<0 & j==a:
    default:
}
```

Happy Compilers mean Happy People

- Computation can be much more efficient
  - compare the running time to if-based alternative
- But, could it all go horribly wrong?
  - construct a switch statement where this implementation technique is a really bad idea
- Guidelines for writing efficient switch statements

```c
label jumpTable[4] = { L0, L1, L2, L3 };
if (i > 3) goto DEFAULT;
goto jumpTable[i];
L0: j = 10;
goto CONT;
L1: j = 11;
goto CONT;
L2: j = 12;
goto CONT;
L3: j = 13;
goto CONT;
DEFAULT:
j = 14;
goto CONT;
CONT:
```
The basic implementation strategy

- General form of a switch statement

```
switch (<cond>) {
  case <label_i>: <code_i>        repeated 0 or more times
  default: <code_default>  optional
}
```

- Naive implementation strategy

```
goto address of code_default if cond > max_label_value
goto jumptable[label_i]
```

statically: jumptable[label_i] = address of code_i forall label_i

- But there are two additional considerations

  - case labels are not always contiguous
  - the lowest case label is not always 0

Refining the implementation strategy

- Naive strategy

- Non-contiguous case labels

  - what is the problem
  - what is the solution

```
switch (i) {
  case 0: j=10; break;
  case 3: j=13; break;
  default: j=14; break;
}
```

- Case labels not starting at 0

  - what is the problem
  - what is the solution

```
switch (i) {
  case 1000: j=10; break;
  case 1001: j=11; break;
  case 1002: j=12; break;
  case 1003: j=13; break;
  default: j=14; break;
}
```

Implementing Switch Statements

- Choose strategy

  - use jump-table unless case labels are sparse or there are very few of them
  - use nested-if-statements otherwise

- Jump-table strategy

  - statically
    - build jump table for all label values between lowest and highest
  - generate code to
    - goto default if condition is less than minimum case label or greater than maximum
    - normalize condition to lowest case label
  - use jumptable to go directly to code selected case arm

```
goto address of code_default if cond < min_label_value
goto address of code_default if cond > max_label_value
goto jumptable[cond-min_label_value]
```

statically: jumptable[cond-min_label_value] = address of code_i forall i: min_label_value <= i <= max_label_value

Snippet B: In template form

```
switch (i) {
  case 20:  j=10; break;
  case 21:  j=11; break;
  case 22:  j=12; break;
  case 23:  j=13; break;
  default:  j=14; break;
}
```

```
label jumpTable[4] = { L20, L21, L22, L23 };
if (i < 20) goto DEFAULT;
if (i > 23) goto DEFAULT;
goto jumptable[i-20];
L20:
  j = 10;
goto CONT;
L21:
  j = 11;
goto CONT;
L22:
  j = 12;
goto CONT;
L23:
  j = 13;
goto CONT;
default:  j=14; break;
}
```

CONT:
### Dynamic Jumps

- **Indirect Jump**
  - Jump target address stored in a register
  - We already introduced this instruction, but used it for static procedure calls

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>indirect</td>
<td>pc ← r[t] + (o==pp*2)</td>
<td>j o(t)</td>
<td>ctpp</td>
</tr>
</tbody>
</table>

- **Double indirect jumps**
  - Jump target address stored in memory
  - Base-plus-displacement and indexed modes for memory access

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbl-ind jump b+o</td>
<td>pc ← m[r[t] + (o==pp*2)]</td>
<td>j o(t)</td>
<td>dtpp</td>
</tr>
<tr>
<td>dbl-ind jump indexed</td>
<td>pc ← m[r[t] + r[i]*4]</td>
<td>j *(r,t,i,4)</td>
<td>eti</td>
</tr>
</tbody>
</table>

### Static and Dynamic Control Flow

- **Jump instructions**
  - specify a target address and a jump-taken condition
  - target address can be static or dynamic
  - jump-target condition can be static (unconditional) or dynamic (conditional)

- **Static jumps**
  - jump target address is static
  - compiler hard-codes this address into instruction

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch</td>
<td>pc ← (a==pc+oo*2)</td>
<td>br a</td>
<td>8-oo</td>
</tr>
<tr>
<td>branch if equal</td>
<td>pc ← (a==pc+oo*2) if r[c]==0</td>
<td>beg a</td>
<td>9-oo</td>
</tr>
<tr>
<td>branch if greater</td>
<td>pc ← (a==pc+oo*2) if r[c]&gt;0</td>
<td>bgt a</td>
<td>a-oo</td>
</tr>
<tr>
<td>jump</td>
<td>pc ← a</td>
<td>j a</td>
<td>b-aaaaaa</td>
</tr>
</tbody>
</table>

- **Dynamic jumps**
  - jump target address is dynamic

### Static vs Dynamic Flow Control

- static if jump target is known by compiler
- dynamic for polymorphic dispatch, function pointers, and switch statements

### Polymorphic Dispatch in Java

- invoking a method on an object in java
- method address depends on object’s type, which is not know statically
- object has pointer to class object; class object contains method jump table
- procedure call is a double-indirect jump – i.e., target address in memory

### Function Pointers in C

- a variable that stores the address of a procedure
- used to implement dynamic procedure call, similar to polymorphic dispatch

### Switch Statements

- syntax restricted so that they can be implemented with jump table
- jump-table implementation running time is independent of the number of case labels
- but, only works if case label values are reasonably dense