

# CPSC 213

## Introduction to Computer Systems

*Unit 1f*

***Dynamic Control Flow***

***Polymorphism and Switch Statements***

# Reading

## ▶ Companion

- 2.7.4, 2.7.7-2.7.8

## ▶ Text

- *Switch Statements, Understanding Pointers*
- 2ed: 3.6.7, 3.10
  - yup, 3.10 again - mainly "Function pointers" box
- 1ed: 3.6.6, 3.11

# 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

```
public class A {  
    static void ping () {}  
}  
  
public class Foo {  
    static void foo () {  
        A.ping ();  
    }  
}
```

## ▶ in C

- specify procedure name

```
void ping () {}  
  
void foo () {  
    ping ();  
}
```

# 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

```
class A {  
    void ping () {}  
    void pong () {}  
}
```

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

```
static void foo (A a) {  
    a.ping ();  
    a.pong ();  
}
```

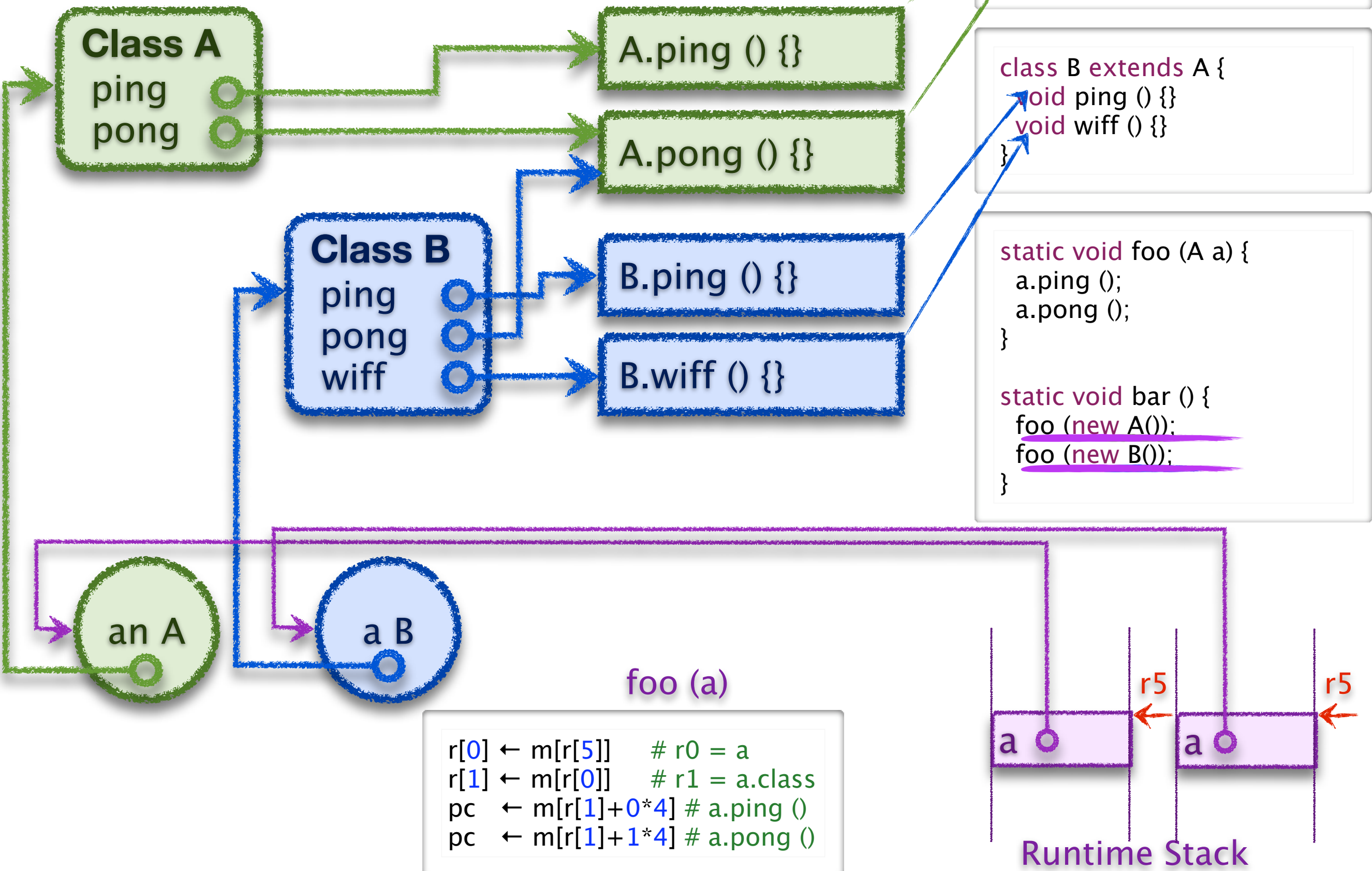
```
static void bar () {  
    foo (new A());  
    foo (new B());  
}
```

Which ping gets called?

# Polymorphic Dispatch

- ▶ **Method address is determined dynamically**
  - compiler can not hardcode target address in procedure call
  - instead, compiler generates code to lookup procedure address at runtime
  - address is stored in memory in the object's class *jump table*
- ▶ **Class Jump table**
  - every class is represented by class object
  - the class object stores the class's jump table
  - the jump table stores the address of every method implemented by the class
  - objects store a pointer to their class object
- ▶ **Static and dynamic of method invocation**
  - address of jump table is determined dynamically
  - method's offset into jump table is determined statically

# Example of Java Dispatch

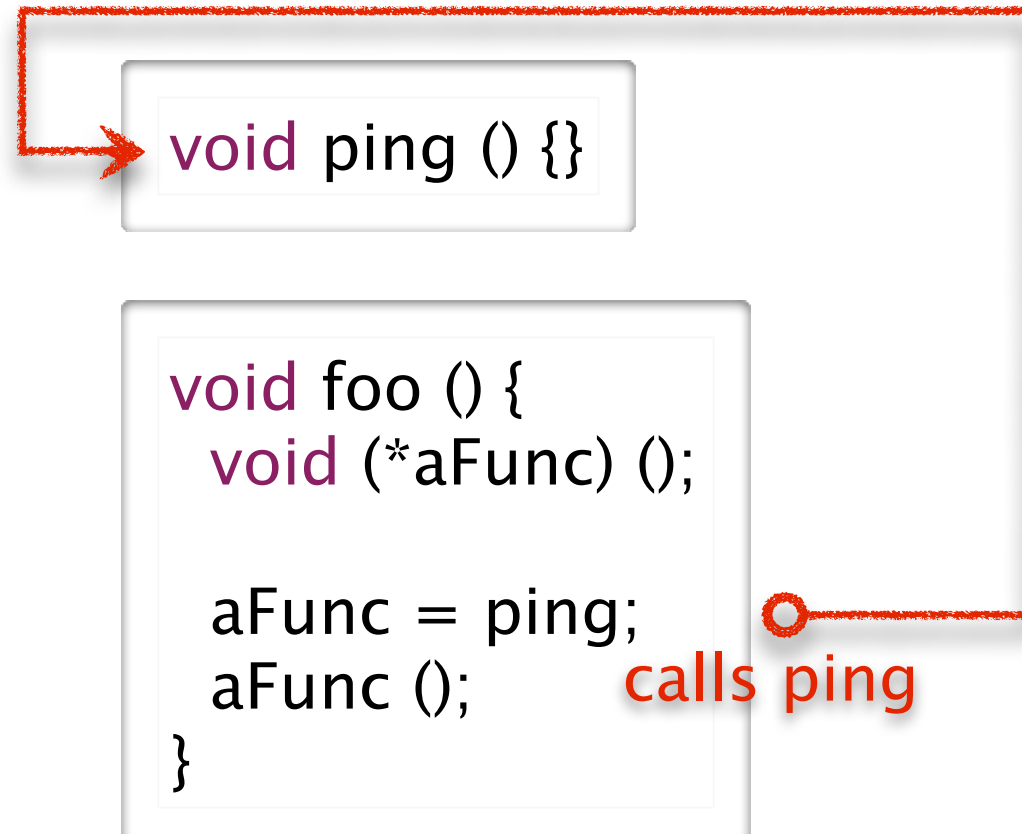


# Dynamic Jumps in C

## ▶ 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





# Simplified Polymorphism in C (SA-dynamic-call.c)

## ▶ Use a struct to store jump table

- drawing on previous example of A ...

Declaration of A's jump table and code

```
struct A {  
    void (*ping) ();  
    void (*pong) ();  
};  
  
void A_ping () { printf ("A_ping\n"); }  
void A_pong () { printf ("A_pong\n"); }
```

Create an instance of A's jump table

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

- and B ...

## Declaration of B's jump table and code

```
struct B {  
    void (*ping)();  
    void (*pong)();  
    void (*wiff)();  
};  
  
void B_ping () { printf ("B_ping\n"); }  
void B_wiff () { printf ("B_wiff\n"); }
```

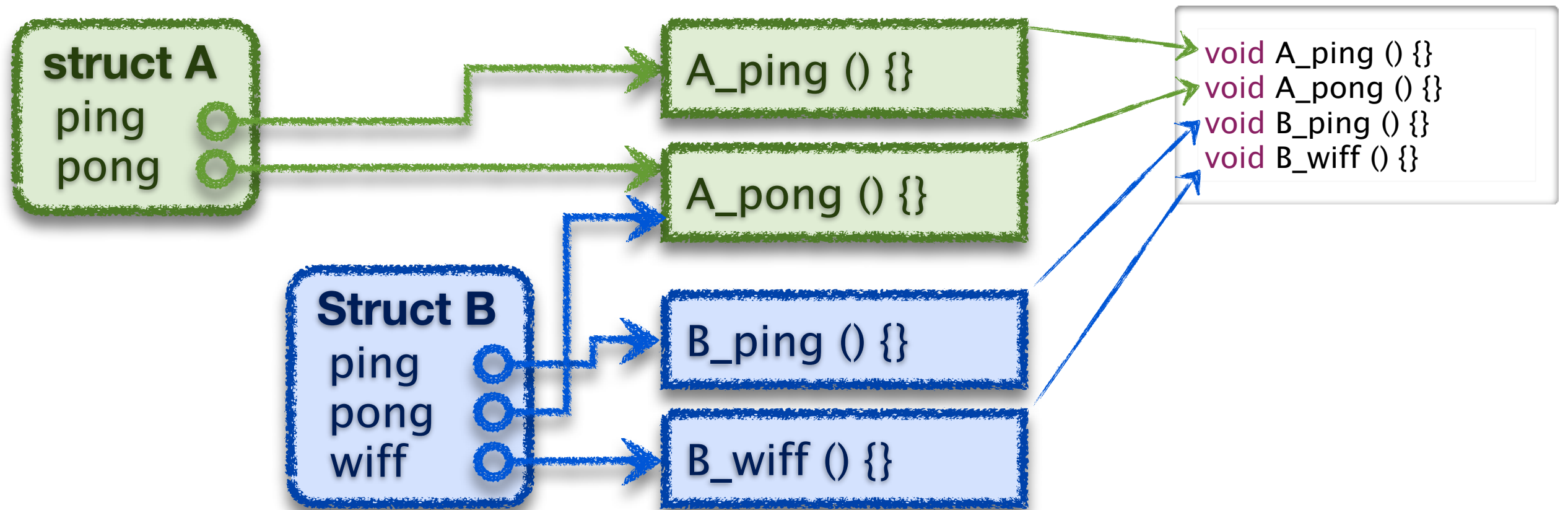
## Create an instance of B's jump table

```
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;  
}
```

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

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

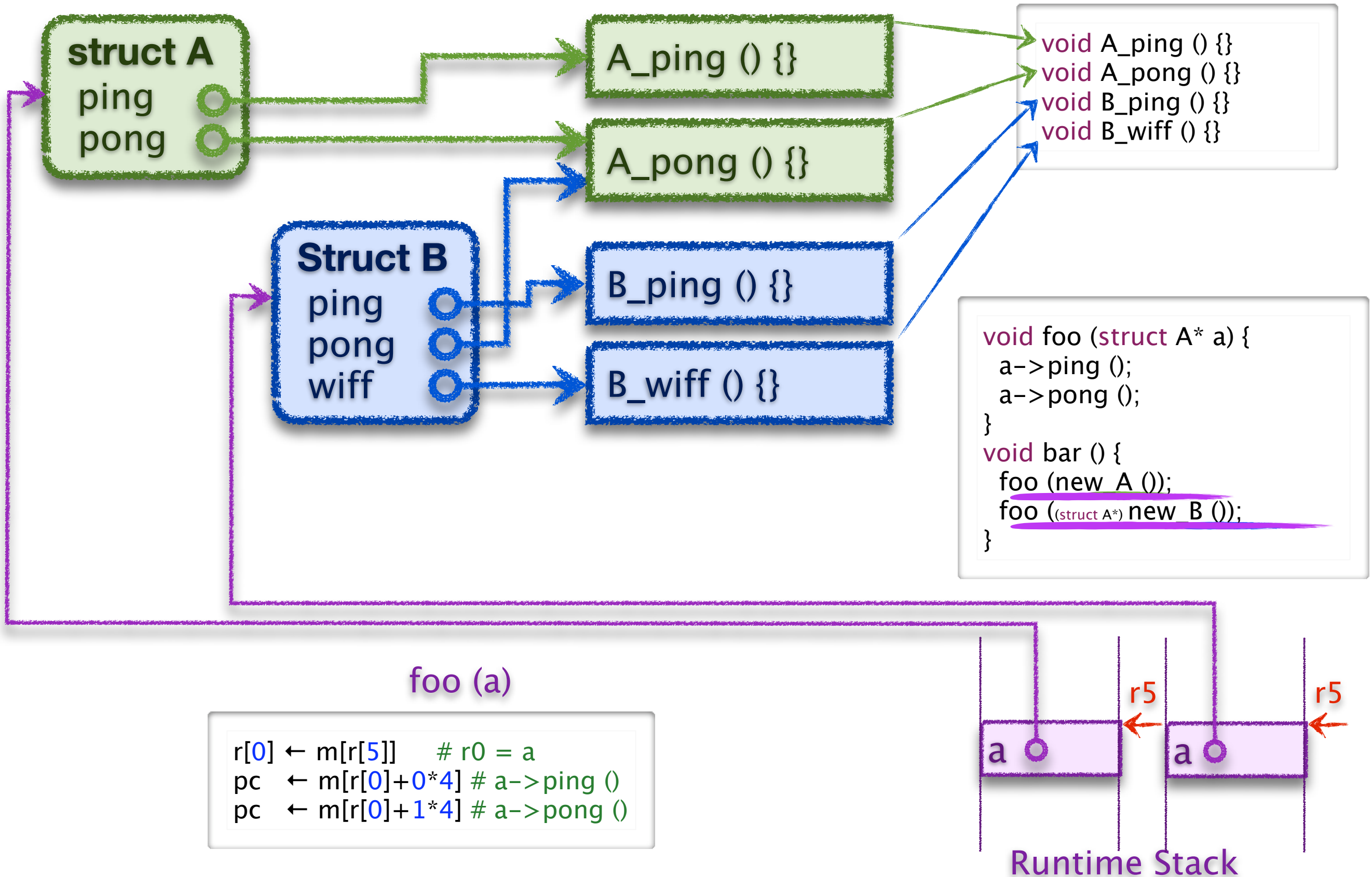
# Dispatch Diagram for C (data layout)



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

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

# Dispatch Diagram for C (the dispatch)



# ISA for Polymorphic Dispatch

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

```
r[0] ← m[r[5]]    # r0 = a  
pc ← m[r[1]+0*4] # a->ping ()  
pc ← m[r[1]+1*4] # a->pong ()
```

## ▶ How do we compile

- a->ping () ?

## ▶ Pseudo code

- $pc \leftarrow m[r[1]+0*4]$

## ▶ Current jumps supported by ISA

Name	Semantics	Assembly	Machine
<i>jump absolute</i>	$pc \leftarrow a$	j a	<b>b</b> ---- aaaaaaaaa
<i>indirect jump</i>	$pc \leftarrow r[s] + (o == pp * 2)$	j o(rs)	<b>c</b> spp

## ▶ We will benefit from a new instruction in the ISA

- that jumps to an address that is stored in memory

## ▶ Double-indirect jump instruction (b+o)

- jump to address stored in memory using base+offset addressing

Name	Semantics	Assembly	Machine
<i>jump absolute</i>	$pc \leftarrow a$	j a	<b>b</b> ---- <b>aaaaaaaa</b>
<i>indirect jump</i>	$pc \leftarrow r[s] + (o == pp * 2)$	j <b>o</b> (rs)	<b>cspp</b>
<i>dbl-ind jump b+o</i>	$pc \leftarrow m[r[s] + (o == pp * 4)]$	j * <b>o</b> (rs)	<b>dspp</b>

# Question 1

- ▶ What is the difference between these two C snippets?

(1)

```
void foo () {printf ("foo \n");}

void go(void (*proc)()) {
    proc();
}

go (foo);
```

(2)

```
void foo () {printf ("foo \n");}

void go() {
    foo();
}

go();
```

- [A] (2) calls foo, but (1) does not
- [B] (1) is not valid C
- [C] (1) jumps to foo using a dynamic address and (2) a static address
- [D] They both call foo using dynamic addresses
- [E] They both call foo using static addresses

Now, implement [proc\(\)](#) and [foo\(\)](#) assembly code



# Switch Statements

# Switch Statement

```
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;
    }
}
```

```
void bar () {
    if (i==0)
        j=10;
    else if (i==1)
        j = 11;
    else if (i==2)
        j = 12;
    else if (i==3)
        j = 13;
    else
        j = 14;
}
```

## ▶ Semantics the same as simplified nested if statements

- where condition of each *if* tests the same variable
- unless you leave the *break* the end of the case block

## ▶ So, why bother putting this in the language?

- is it for humans, facilitate writing and reading of code?
- is it for compilers, permitting a more efficient implementation?

## ▶ Implementing switch statements

- we already know how to implement if statements; is there anything more to consider?

# 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?

```
switch (treeName) {  
  case "larch":  
  case "cedar":  
  case "hemlock":  
}
```

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

# Why Compilers like Switch Statements

## ▶ Notice what we have

- switch condition evaluates to a number
- each case arm has a distinct number

## ▶ And so, the implementation has a simplified form

- build a table with the address of every case arm, indexed by case value
- switch by indexing into this table and jumping to matching case arm

## ▶ For example

```
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;  
}
```

```
label jumpTable[4] = { L0, L1, L2, L3 };  
if (i < 0 || 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:
```

# Happy Compilers mean Happy People

```
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;  
}
```

```
label jumpTable[4] = { L0, L1, L2, L3 };  
if (i < 0 || 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:
```

- ▶ 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

```
if (i==0)  
  j=10;  
else if (i==1)  
  j = 11;  
else if (i==2)  
  j = 12;  
else if (i==3)  
  j = 13;  
else  
  j = 14;
```

# 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

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

statically: jumptable[label\_i] = address of code\_i for all label\_i

## ▶ 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[i-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;  
  goto CONT;  
CONT:
```

# Snippet B: In Assembly Code

```
foo:  ld  $i, r0      # r0 = &i
      ld  0x0(r0), r0  # r0 = i
      ld  $0xffffffff, r1 # r1 = -1
      add r0, r1      # r0 = i-1
      bgt r1, l0      # goto l0 if i>1
      br  default     # goto default if i<20
l0:   ld  $0xffffffe9, r1 # r1 = -23
      add r0, r1      # r1 = i-23
      bgt r1, default  # goto default if i>23
      ld  $0xfffffec, r1 # r1 = -20
      add r1, r0      # r0 = i-20
      ld  $jmpltable, r1 # r1 = &jmpltable
      j  *(r1, r0, 4)  # goto jmptable[i-20]
```

```
case20: ld  $0xa, r1    # r1 = 10
        br  done       # goto done
...
default: ld  $0xe, r1   # r1 = 14
         br  done       # goto done
done:    ld  $j, r0     # r0 = &j
         st  r1, 0x0(r0) # j = r1
         br  cont       # goto cont
```

```
jmpable: .long 0x00000140 # &(case 20)
         .long 0x00000148 # &(case 21)
         .long 0x00000150 # &(case 22)
         .long 0x00000158 # &(case 23)
```

Simulator ...

# 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

Name	Semantics	Assembly	Machine
<i>branch</i>	$pc \leftarrow (a=pc+pp*2)$	br a	<b>8-pp</b>
<i>branch if equal</i>	$pc \leftarrow (a=pc+pp*2)$ if $r[s]==0$	beq a	<b>9spp</b>
<i>branch if greater</i>	$pc \leftarrow (a=pc+pp*2)$ if $r[s]>0$	bgt a	<b>aspp</b>
<i>jump</i>	$pc \leftarrow a$	j a	<b>b---</b> aaaaaaaaa

## ▶ Dynamic jumps

- jump target address is dynamic

# Dynamic Jumps

## ▶ Indirect Jump

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

Name	Semantics	Assembly	Machine
<i>indirect jump</i>	$pc \leftarrow r[s] + (o == pp * 2)$	<code>j o(rs)</code>	<code>cspp</code>

## ▶ Double indirect jumps

- Jump target address stored in memory
- Base-plus-displacement and indexed modes for memory access

Name	Semantics	Assembly	Machine
<i>dbl-ind jump b+o</i>	$pc \leftarrow m[r[s] + (o = pp * 4)]$	<code>j *o(rs)</code>	<code>dspp</code>
<i>dbl-ind jump indexed</i>	$pc \leftarrow m[r[s] + r[i] * 4]$	<code>j *(rs,ri,4)</code>	<code>esi-</code>

# Question 2

- ▶ What happens when this code is compiled and run?

```
void foo (int i) {printf ("foo %d\n", i);}
void bar (int i) {printf ("bar %d\n", i);}
void bat (int i) {printf ("bat %d\n", i);}

void (*proc[3])(int) = {foo, bar, bat};

int main (int argv, char** argc) {
    int input;
    if (argv==2) {
        input = atoi (argc[1]);
        proc[input] (input+1);
    }
}
```

- [A] It does not compile
- [B] For any value of input it generates an error
- [C] If input is 1 it prints “bat 1” and it does other things for other values
- [D] If input is 1 it prints “bar 2” and it does other things for other values

# Question 3

▶ Which implements **proc[input] (input+1);**

• [A]

```
ld (r5), r0
ld $proc, r1
deca r5
mov r0, r2
inc r2
st r2, (r5)
gpc $2, r6
j *(r1, r0, 4)
```

• [B]

```
ld (r5), r0
ld $proc, r1
deca r5
mov r0, r2
inc r2
st r2, (r5)
gpc $6, r6
j bar
```

```
void foo (int i) {printf ("foo %d\n", i);}
void bar (int i) {printf ("bar %d\n", i);}
void bat (int i) {printf ("bat %d\n", i);}
```

```
void (*proc[3])(int) = {foo, bar, bat};
```

```
int main (int argc, char** argv) {
    int input;
    if (argc==2) {
        input = atoi (argv[1]);
        proc[input] (input+1);
    }
}
```

- [C] I think I understand this, but I can't really read the assembly code.
- [D] Are you serious? I have no idea.

# Summary

## ▶ 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 known 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