CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

Unit 1f – Jul 23, 25

Dynamic Control Flow

Overview

- Reading
  - Companion: 2.7.4, 2.7.7-2.7.8
- Reference
  - Text: 3.6.7, 3.10
- Learning Goals
  - Write C programs that use function pointers
  - Explain how Java implements polymorphism
  - Identify the number of memory references that occur when a static method is called in Java and when an instance method is called
  - Convert Java instance-method call into equivalent C code that uses function pointers
  - Convert C programs that use function pointers into assembly code
  - Explain why switch statements in C (and Java until version 1.7) restrict case labels to cardinal types (i.e, things that map to natural numbers)
  - Convert C switch statement into equivalent C statement using gotos and an array of label pointers (a gcc extension to C)
  - Convert C switch statement into equivalent assembly language that uses a jump table
  - Determine whether a given switch statement would be better implemented using if statements or a jump table and explain the tradeoffs involved

Dynamic Control Flow

- Function Call/Return

```
foo:    gpc   $6, r6
        j     ping
        …
ping:   j     (r6)
```

- Return is dynamic control flow!
- Calling different functions

```
        beq r0, L2
L1:     gpc $6, r6     # run func1 if r0 != 0
        j func1
        br L3
L2:     gpc $6, r6     # run func2 if r0 == 0
        j func2
L3:     …
```

Dynamic Control Flow

- Calling different functions

```
        beq r0, L2
L1:     gpc $6, r6     # run func1 if r0 != 0
        j func1
        br L3
L2:     gpc $6, r6     # run func2 if r0 == 0
        j func2
L3:     …
```

- Can we do better?

```
        beq r0, L2
L1:     ld $func1, r1 # run func1 if r0 != 0
        br L3
L2:     ld $func2, r1 # run func2 if r0 == 0
L3:     gpc $2, r6
        j (r1)
```

Dynamic Control Flow

- Dynamic Control Flow!

```
        beq r0, L2
L1:     ld $func1, r1 # run func1 if r0 != 0
        br L3
L2:     ld $func2, r1 # run func2 if r0 == 0
L3:     gpc $2, r6
        j (r1)
```

- Functions have addresses - can make *pointers to functions*
- **Lots** of uses for dynamic control flow
  - Polymorphism
  - Function Arguments
  - Jump Tables/Switch Statements

Return vs. Dynamic Call

- Return is usually at the *end* of a function
- By convention, *only* use r6 when returning

```
ld   (r5), r6
inca r5
j    (r6)
```

- Dynamic call is in the middle of a function
- gpc to set return address

```
gpc  $2, r6
j    (r1)
```

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 (apparent type)
  - the object reference is dynamic and so is its type
    - object's *actual type* must be a subtype of the *apparent type* of the referring variable
    - but object's actual type may override methods of the apparent 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 is called?

Polymorphic Dispatch

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

- 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 a class object
  - objects store a pointer to their class object
  - the class object stores the class's jump table
  - the jump table stores the address of methods implemented by the class
- Static and dynamic of method invocation
  - address of jump table is determined dynamically
    - objects of different actual types will have different jump tables
  - method's offset into jump table is determined statically

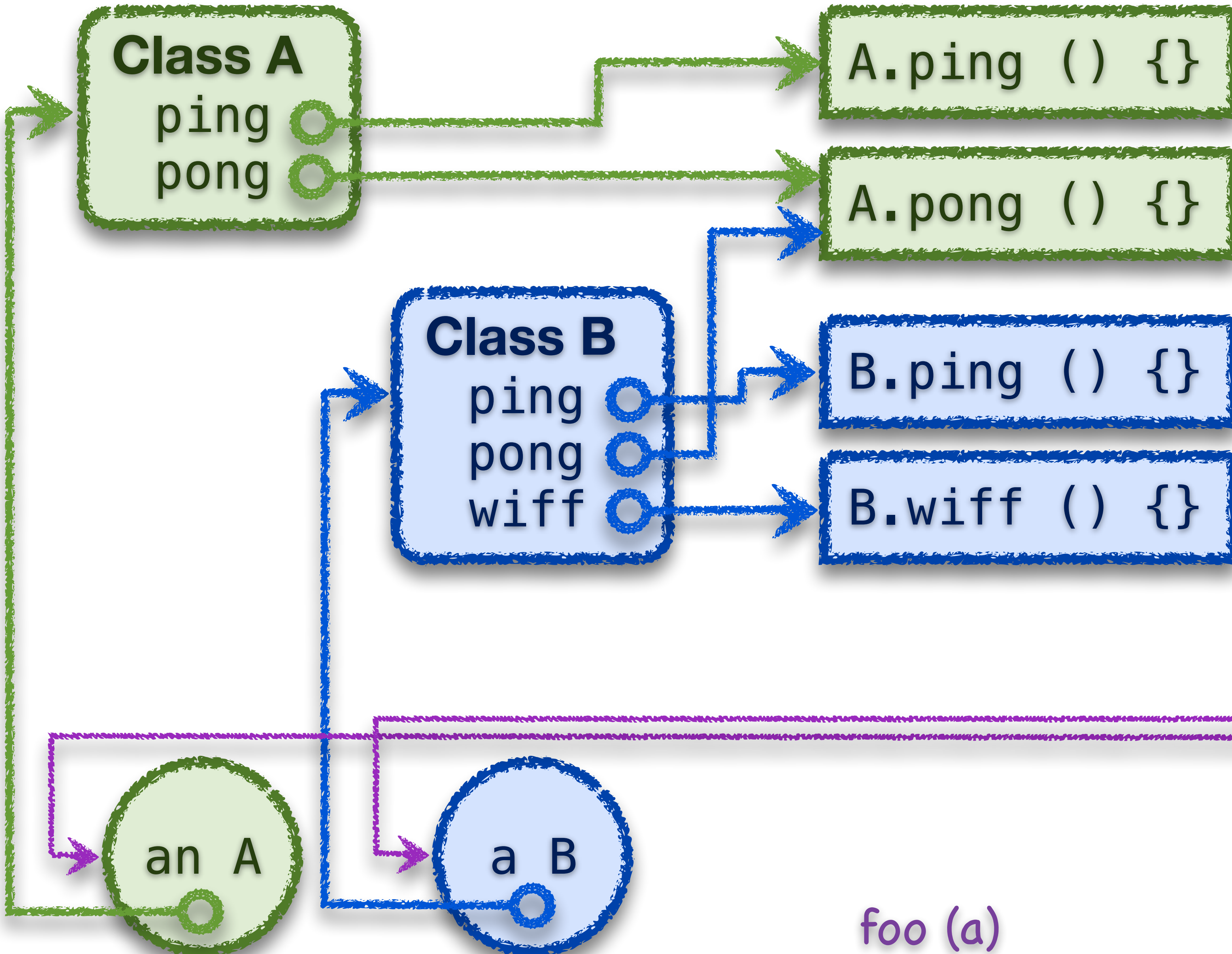
Example of Java Dispatch

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

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

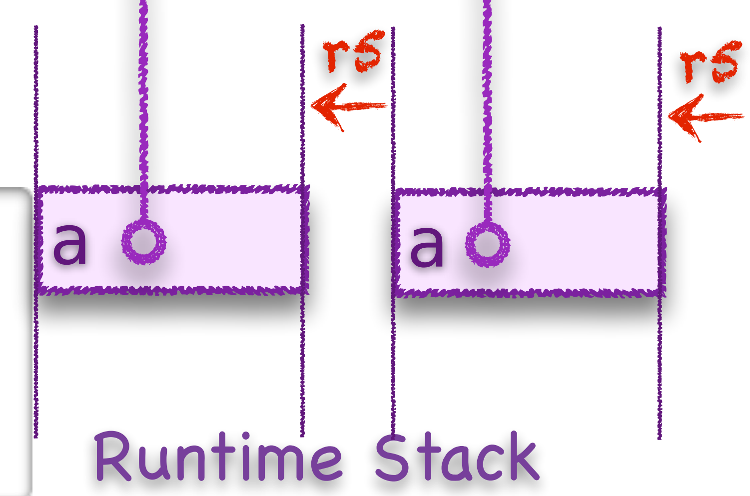
```
static void foo (A a) {
  a.ping ();
  a.pong ();
}

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



foo (a)

```
r[0] ← m[r[5]+4]   # r0 = a
r[1] ← m[r[0]]     # r1 = a.class
pc   ← m[r[1]+0*4] # a.ping ()
pc   ← m[r[1]+1*4] # a.pong ()
```



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() {}
```

```c
void foo() {
  void (*aFunc)();

  aFunc = ping;
  aFunc();
}
```

calls ping

procedure name **without** ()

variable name **with** ()

Aside: Function Pointer Syntax

- Function pointer syntax is *ugly!*

```
void (*fptr)(int);
```

- How do you return a function pointer?

```
void (*)(int) return_fptr() { … }
```

```
void (*return_fptr())(int) { … }
```

😱😱😱

- How do you return a function pointer *sanely*?
- Typedef:

```
typedef void (*fptr_t)(int);
```

```
fptr_t return_fptr() { … }
```

🥳🥳🥳

CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

Unit 1f – Jul 23, 25

Dynamic Control Flow

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

- Use a struct to store jump table

```
class A {
  void ping() {…}
  void pong() {…}
  int  i;
}
```

  - Declaration of Class

```
struct A_class {
  void (*ping)(void*);
  void (*pong)(void*);
};
```

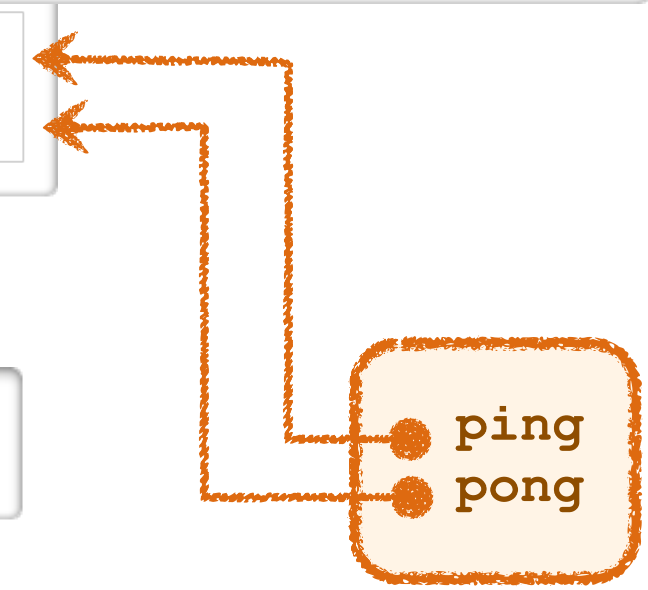
```
void A_ping(void *thisv) {
  struct A *this = thisv;
  printf("A_ping %d\n", this->i);
}
```

  - Declaration of Instance Methods

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

  - Static Allocation and Initialization of Class Object

```
struct A_class A_class_table = {A_ping, A_pong};
```



Object (Instance of Class)

- Object Template

```
struct A {
  struct A_class *class;
  int i;
};
```

- Constructor Method

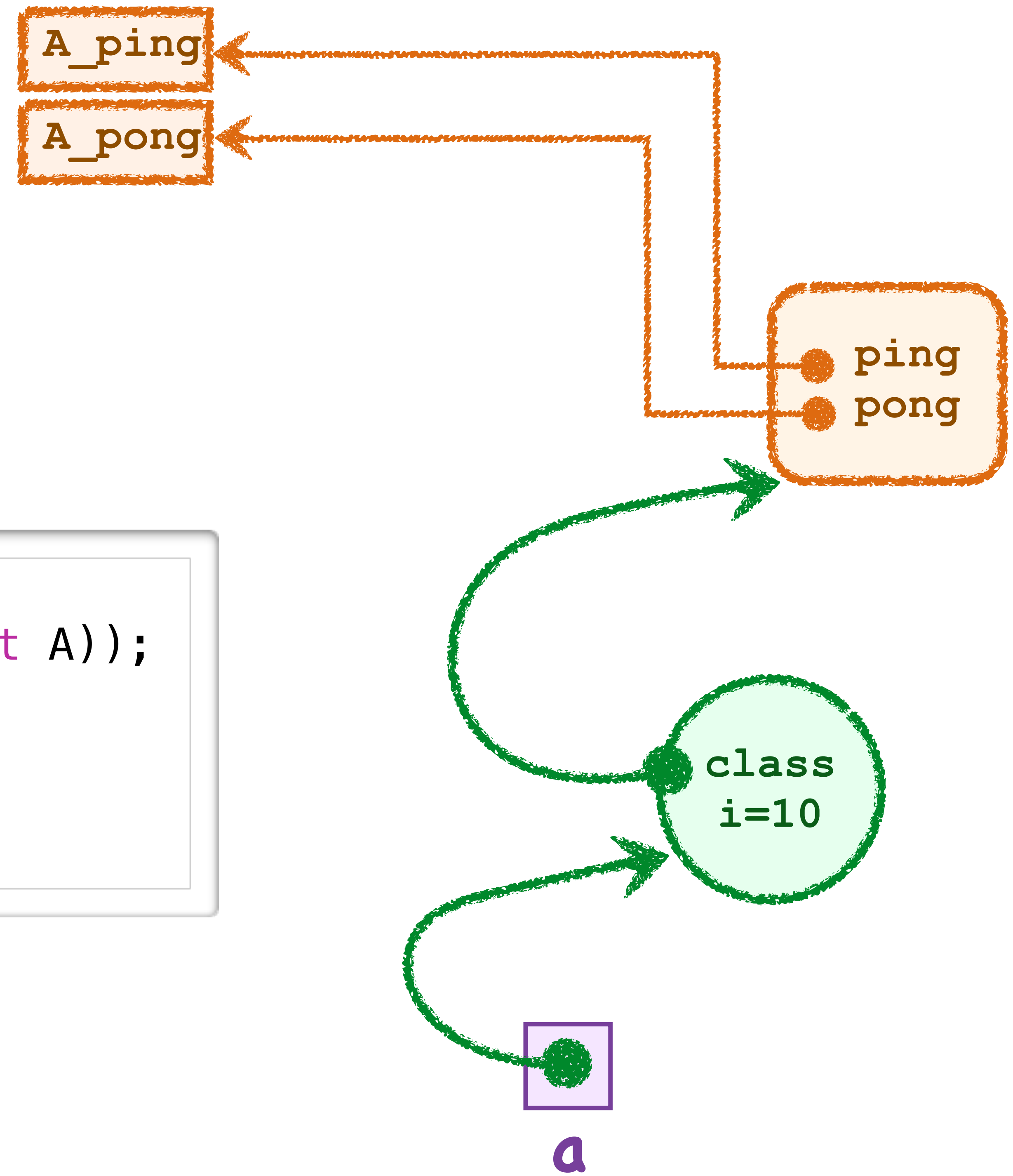
```
struct A *new_A(int i) {
  struct A *obj = malloc(sizeof(struct A));
  obj->class = &A_class_table;
  obj->i     = i;
  return obj;
}
```

- Allocating an Instance

```
struct A *a = new_A(10);
```

- Calling Instance Methods

```
a->class->ping(a);
a->class->pong(a);
```



The same thing for *class B extends A*

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

- The class struct is a super set of A's

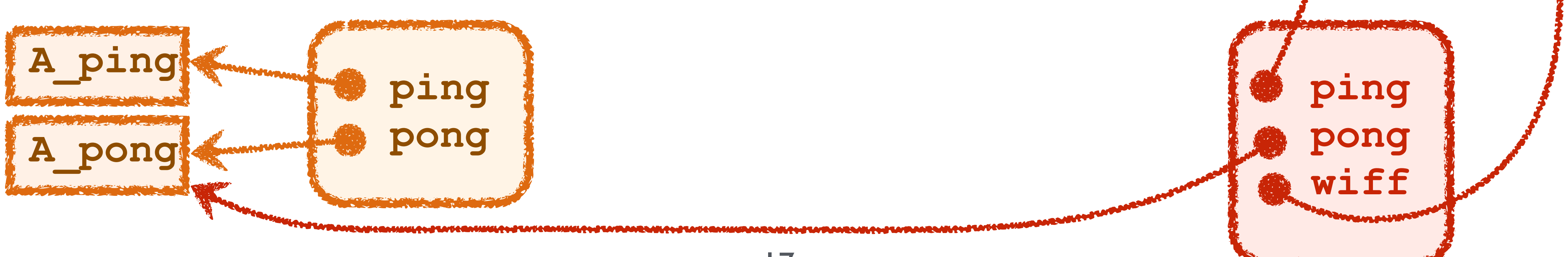
```
struct B_class {
  void (*ping)(void*);
  void (*pong)(void*);
  void (*wiff)(void*);
};
```

```
struct A_class {
  void (*ping)(void*);
  void (*pong)(void*);
};
```

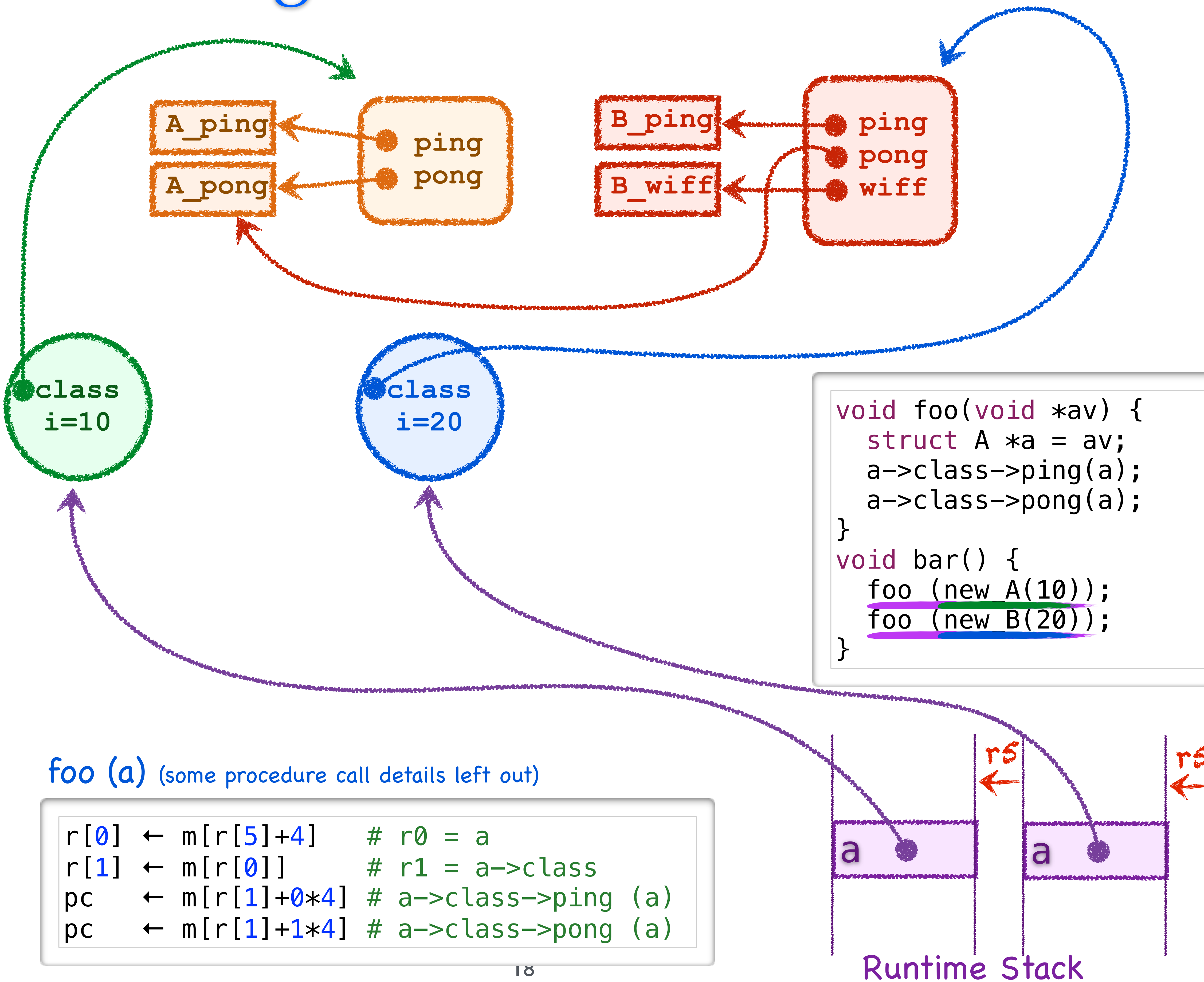
- B's methods and class object

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

```
struct B_class B_class_table = {B_ping, A_pong, B_wiff};
```



Dispatch Diagram for C




```
void foo(void *av) {
  struct A *a = av;
  a->class->ping(a);
  a->class->pong(a);
}
void bar() {
  foo (new A(10));
  foo (new B(20));
}
```

foo (a) (some procedure call details left out)

```
r[0] ← m[r[5]+4]    # r0 = a
r[1] ← m[r[0]]      # r1 = a->class
pc   ← m[r[1]+0*4] # a->class->ping (a)
pc   ← m[r[1]+1*4] # a->class->pong (a)
```

Other Uses of Function Pointers

Example: Quicksort

- Consider, for example, writing Quicksort to sort integers

```
int partition (int* array, int left, int right, int pivotIndex) {
  int pivotValue, t;
  int storeIndex, i;

  pivotValue         = array [pivotIndex];
  array [pivotIndex] = array [right];
  array [right]      = pivotValue;
  storeIndex = left;
  for (i=left; i<right; i++)
    if (array [i] <= pivotValue) {
      t                  = array [i];
      array [i]          = array [storeIndex];
      array [storeIndex] = t;
      storeIndex += 1;
    }
  t                  = array [storeIndex];
  array [storeIndex] = array [right];
  array [right]      = t;
  return storeIndex;
}

void quicksort (int* array, int left, int right) {
  int pivotIndex;

  if (left < right) {
    pivotIndex = partition (array, left, right, left + (right-left)/2);
    quicksort (array, left,           pivotIndex - 1);
    quicksort (array, pivotIndex + 1, right         );
  }
}

void sort (int* array, int n) {
  quicksort (array, 0, n-1);
}
```

The Code is *Mostly* Type Independent

- Parameterize to sort anything
  - actually, like Java, a pointer to anything (or anything the same size as a pointer)

```
int partition (void** array, int left, int right, int pivotIndex) {
  void* pivotValue, *t;
  int  storeIndex,  i;

  pivotValue          = array [pivotIndex];
  array [pivotIndex] = array [right];
  array [right]       = pivotValue;
  storeIndex = left;
  for (i=left; i<right; i++)
    if (array [i] <= pivotValue) {
      t                   = array [i];
      array [i]           = array [storeIndex];
      array [storeIndex] = t;
      storeIndex += 1;
    }
  t                   = array [storeIndex];
  array [storeIndex] = array [right];
  array [right]       = t;
  return storeIndex;
}
```

Actually, only 3 parts of the code are type dependent. And two are easy to deal with.

But, what about the comparison?

Type-Independent, Parameterized Sort

- Using a comparator function pointer

```
int partition (void** array, … , int (*cmp) (void*, void*)) {
  void *pivotValue, *t;
  int  storeIndex, i;

  pivotValue          = array [pivotIndex];
  array [pivotIndex] = array [right];
  array [right]       = pivotValue;
  storeIndex = left;
  for (i=left; i<right; i++)
    if (cmp (array [i], pivotValue) <= 0) {
      t                   = array [i];
      array [i]           = array [storeIndex];
      array [storeIndex] = t;
      storeIndex += 1;
    }
  t                   = array [storeIndex];
  array [storeIndex] = array [right];
  array [right]       = t;
  return storeIndex;
}
```

was: `array [i] <= pivotValue`

Compared to Java

```
int partition (Comparable<T> array[], …, int pivotIndex) {
  Comparable<T> pivotValue, t;
  int storeIndex, i;

  pivotValue         = array [pivotIndex];
  array [pivotIndex] = array [right];
  array [right]      = pivotValue;
  storeIndex = left;
  for (i=left; i<right; i++)
    if (array [i] .compareTo (pivotValue)) <= 0) {
      …
}
```

```
class ComparableInteger<Integer> extends Integer {
  @Override
  int compareTo(Integer i) {
    return intValue() < i.intValue()? -1: intValue == i.intValue()? 0: 1;
  }
}
```

Genericity

- C library qsort function

```c
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

- *Full* genericity through *width* parameter

<table>
<tr><td colspan="4">struct</td><td colspan="4">struct</td><td colspan="4">struct</td><td colspan="4">struct</td></tr>
<tr><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td><td>int</td></tr>
</table>

- Can we make *everything* variable width in C?

Genericity

- Limits to genericity: object sizes must be known
- In practice, easier to work with arrays of pointers

int *	int *	short *	char *	struct *	long *	void(*)()	void *

- The pointer to "anything" is the *opaque pointer* **`void *`**
- Can't dereference it directly (would get "void" which is invalid); cast it to a real pointer type instead
  - `void *x = ...`
  - `int *pi = x;`
  - `char *pc = x;`
  - `void **pv = x;`
- In A8, we **`typedef void *element_t`** and make arrays of them

Using the Parameterized Quicksort

To sort integers

```
int cmpIntegers(void *av, void *bv) {
  int *a = av;
  int *b = bv;
  return *a < *b? -1: *a == *b? 0 : 1;
}
```

```
int  a  [] = {3, 8, 1};
int* pa [] = {a, a+1, a+2};
sort ((void**) pa, 3, cmpIntegers);
```

To sort strings

```
int cmpStrings(void *av, void *bv) {
  char *a = av;
  char *b = bv;

  while(*a != 0 && *b != 0 && *a == *b) {
    a++;
    b++;
  }
  return *a < *b? -1 : *a == *b? 0 : 1;
}
```

```
char *array[] = {"Mike", "Ben", "Liam", …, "Ace"};
sort(array, sizeof(array) / sizeof(array[0]), cmpStrings);
```

Higher-Order Functions

Remember Dr Racket

- Map – (map f lst …)
  - (**map** + (list 1 4 3) (list 7 2 5)) => (list 8 6 8)
    - (map (lambda (a b) (+ a b)) (list 1 4 3) (list 7 2 5))
  - (**map** *max* (list 1 4 3) (list 7 2 5)) => (list 7 4 5)
- Other list iterators
  - foldl, filter, …

```
(foldl + 0 (list 1 2 3))
(filter (lambda (a) (> a 3)) (list 1 2 3 4 5))
```

- Other languages
  - python, javascript and Java …

Implementing map in C

- Map – (map f lst ..)
  - (**map** + (list 1 4 3) (list 7 2 5))   => (list 8 6 8)
    - (map (lambda (a b) (+ a b)) (list 1 4 3) (list 7 2 5))
  - (**map** *max* (list 1 4 3) (list 7 2 5)) => (list 7 4 5)
- In C
  - We can do it with an int array
  - But to generalize, we need `void *`
-
```
void map(void (*f)(void*, void*, void**), int n, void** s0, void** s1, void** d)
```

How about *foldl* – aka *reduce* ?

Consider

- using (foldl f init lst …) to compute sum, min, max, count of an array
- (foldl + 0 (list 1 2 3 4))

In C

```
void foldl (void (*f)(void*, void**), int n, void** v, void** a) {
  for (int i=0; i<n; i++)
    f (v, a[i]);
}
```

```
int  a[]  = {1,2,3,4,5};
int* ap[] = {&a[0], &a[1], &a[2], &a[3], &a[4]};
int  s = 0;
int* sp = &s;
foldl (add, sizeof(a) / sizeof (a[0]), (void**) &sp, (void**) ap);
```

Implement *add*

- *see foldl-starter.c*

Or to concatenate strings

```c
void concat(void** vv, void* av) {
  char **v = (char**) vv, *a = av;
  *v = realloc(*v, strlen(*v) + strlen(a) + 1);
  strcat(*v, a);
}
```

```c
char* ss[] = {"Hello", " ", "World", "!"};
char* ds    = malloc(1);
*ds         = 0;
foldl(concat, sizeof(ss) / sizeof(char*), (void**) &ds,
(void**) &ss);
printf("%s\n", ds);
free(ds);
```

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 out the *break* at the end of a 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

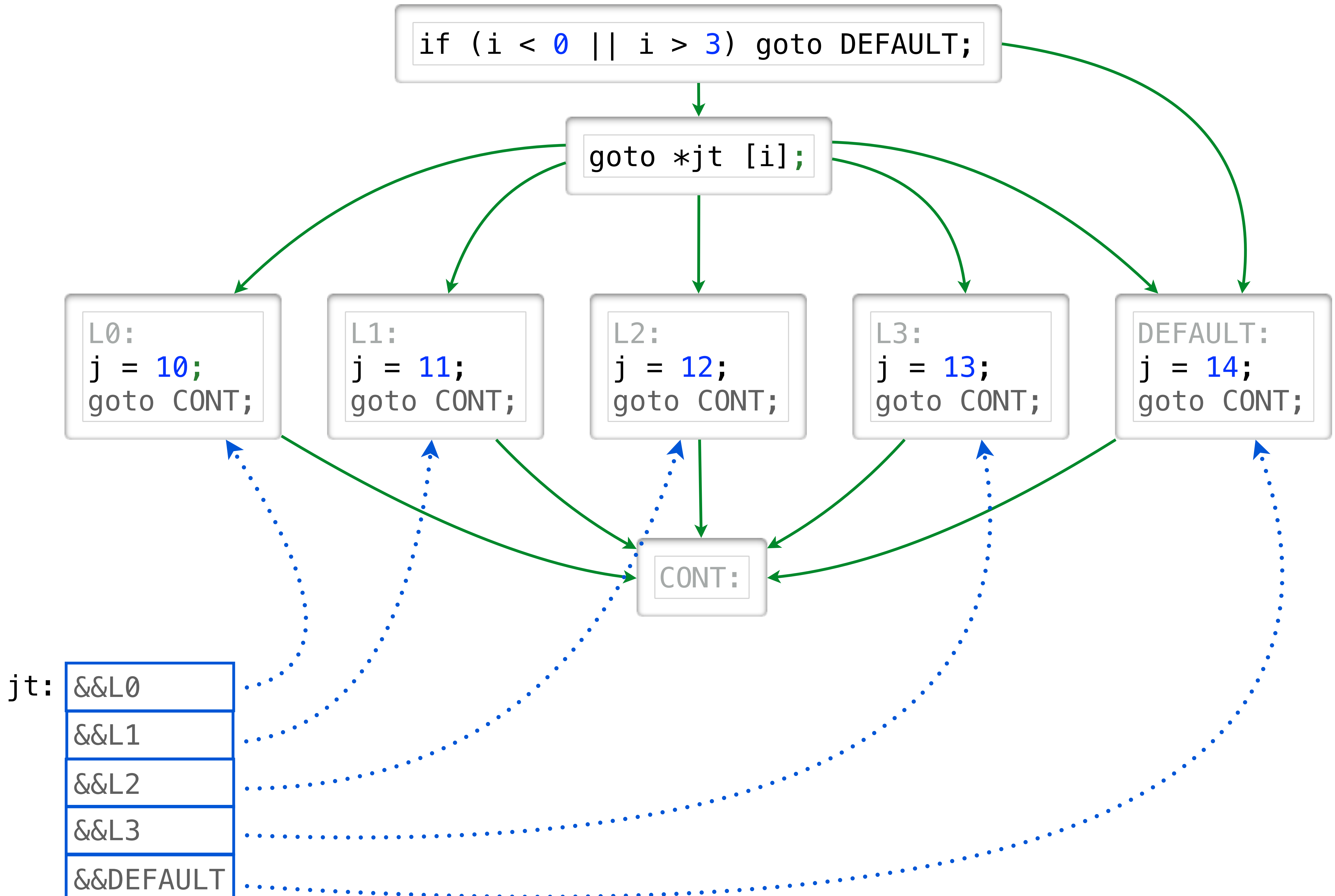
- Notice what we have
  - switch condition evaluates to a number
  - each case arm has a distinct number
- And so, the implementation has a simplified form
  - build a table with the address of every case arm, indexed by case value
  - switch by indexing into this table and jumping to matching case arm
- 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;
}
```

static const

```
    void* jt[4] = { &&L0, &&L1, &&L2, &&L3 };
    if (i < 0 || i > 3) goto DEFAULT;
    goto *jt [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:
```

Switch Statement Control Flow with Jump Table



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;
}
```

```
    void* jt[4] = { &&L0, &&L1, &&L2, &&L3 };
    if (i < 0 || i > 3) goto DEFAULT;
    goto *jt [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 address in 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 address in jumptable [label_i]

statically: jumptable [label_i] = address of code_i forall 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 value to lowest case label
    - use jump table 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 address in 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 jump-table form

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

```
     static const void* jt[4] = { &&L20, &&L21, &&DEFAULT, &&L23 };
     if (i < 20 || i > 23) goto DEFAULT;
     goto *jt [i-20];
L20: j = 10;
     goto CONT;
L21: j = 11;
     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   $0xffffffed, r1     # r1 = -19
          add  r0, r1              # r0 = i-19
          bgt  r1, l0              # goto l0 if i>19
          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   $0xffffffec, r1     # r1 = -20
          add  r1, r0              # r0 = i-20
          ld   $jmptable, r1       # r1 = &jmptable
          ld   (r1, r0, 4), r1     # r1 = jmptable[i-20]
          j    (r1)                # 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
```

```
jmptable: .long case20            # & (case 20)
          .long case21            # & (case 21)
          .long default           # & (case 22)
          .long case23            # & (case 23)
```

Simulator ...

Question 1f.3

- 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);}

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

int main(int argc, char **argv) {
  int input;
  if(argc == 2) {
    input = atoi(argv[1]);
    if(input >= 0 && input <= 2)
     proc[input](input+1);
  }
}
```

A. It does not compile

B. For any value of input it generates a runtime error

C. If input is 1 it prints “bat 2” 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 1f.4

- Which implements `proc[input](input+1);`
  - [A]

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

  - [B]

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

```c
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);}

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

int main(int argc, char **argv) {
  int input;
  if(argc == 2) {
    input = atoi(argv[1]);
    if(input >= 0 || input <= 2)
      proc[input](input+1);
  }
}
```

  - [C] Neither snippet.
  - [D] Both snippets.
  - [E] I think I understand this, but I can’t really read the assembly code.

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 know statically
  - object has pointer to class object; class object contains method jump table
  - procedure call is this 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