CPSC 213
Introduction to Computer Systems

Unit 1f
Dynamic control flow

All slides adapted from materials by Mike Feeley, Jonatan Schroeder, Robert Xiao, and Jordon Johnson

Announcements

- Google doc for lecture questions
  - See Piazza for link, section 102
  - https://docs.google.com/document/d/1G6hkekQS7mT9lFpP8AVftYao8vLRujIrRLAvOuX_07w/edit
  - Add your question anonymously (at the top)
  - Help answer questions too!

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

Review: static 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

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

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

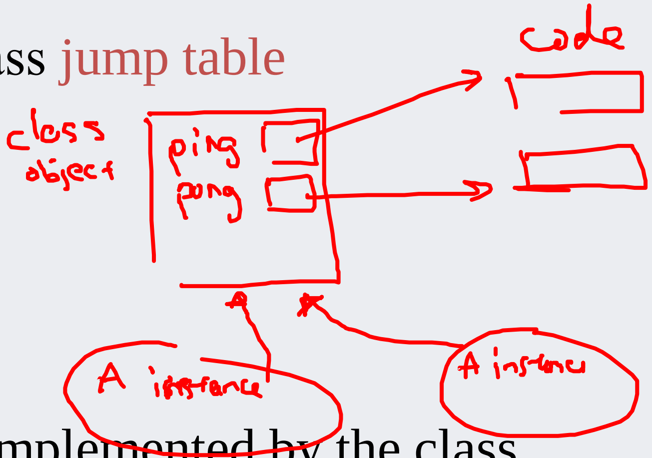
```java
static void foo(A a) {
  a.ping();
  a.pong();
}

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

which ping is called?

Polymorphic dispatch

```java
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



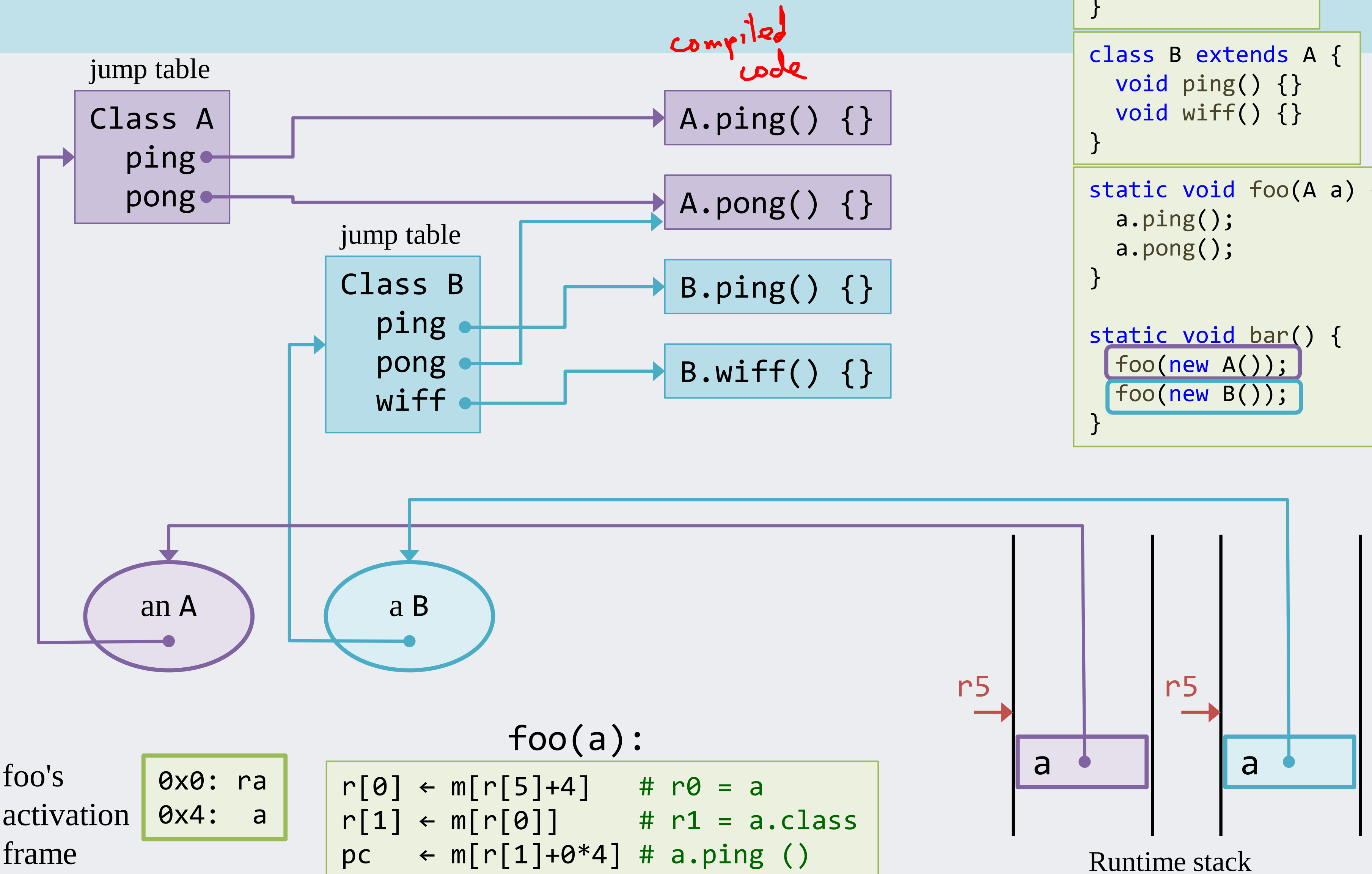
- Static and dynamic of method invocation
  - address of jump table is determined dynamically
    - objects of different actual types will have different jump tables
  - method’s offset into jump table is determined statically

Java dispatch example

```
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's activation frame

```
0x0: ra
0x4:  a
```

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 as:
    - `<return-type> (*<variable-name>)(<formal-argument-list>);`
  - used to make dynamic call
    - `<variable-name> (<actual-argument-list>);`
- Example

```
void ping() {}
```

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

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

assign using procedure name without ()
call using variable name with ()

iClicker 1f.1

- What is the difference between these two C snippets?

void baz() { .... }

(1)
```c
void foo() { printf("foo\n"); }

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

void bat() {
  go(foo);
} go(baz);
```

pointer to a procedure with a void return type and no arguments

(2)
```c
void foo() { printf("foo\n"); }

void go() {
  foo();
}

void bat() {
  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

Exercise 1f.2

PrairieLearn – IC_11_03

- (a) write a function that
  - EITHER adds two numbers or multiplies them
  - depending on a parameter
  - using function pointers
- (b) write code that uses this function

Polymorphism in C

- Use a `struct` to store jump table
  - Declaration of class:

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

If we want something like this:

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

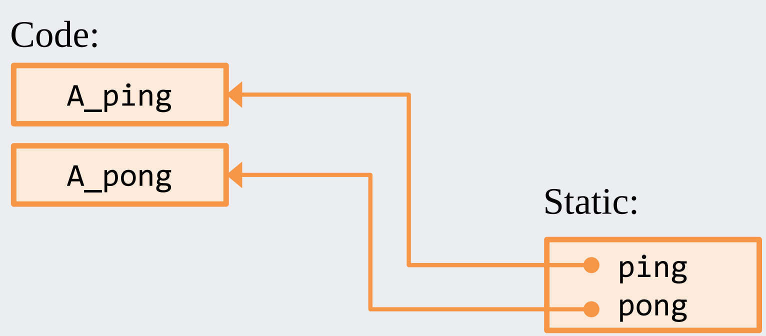
  - Declaration of instance methods:

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

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

  - Static allocation and initialization of class object (i.e. class jump table)

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



Polymorphism in C

(continued)

- Object (instance of class)
  - Object template

```c
struct A {
  struct A_class* class; // pointer to class object
  int i;                 // instance's attribute
};
```

  - Constructor method

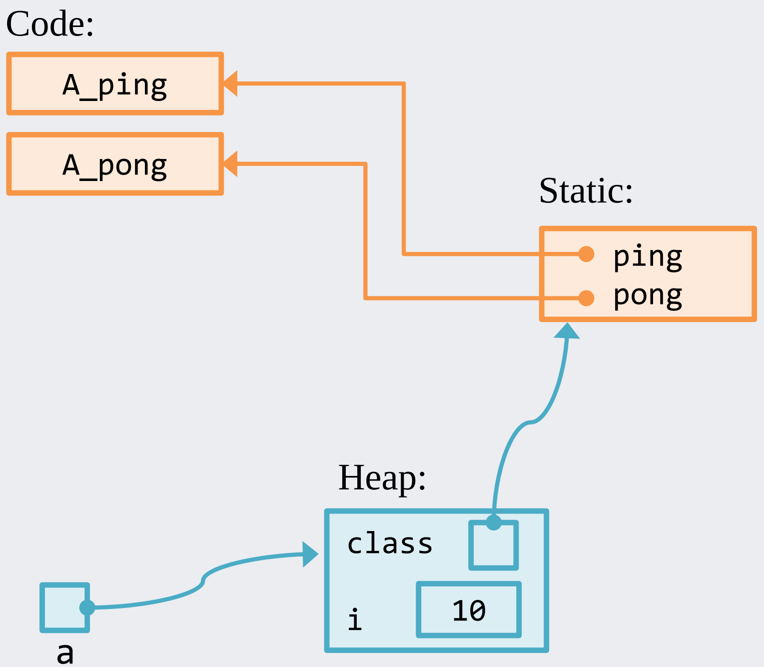
```c
struct A* new_A(int i) {
  struct A* obj = malloc(sizeof(struct A));
  obj->class = &A_class_table;
  obj->i = i;
}; return obj;
```

  - Allocating an instance

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

  - Calling instance methods

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



Polymorphism in C

mimicking `class B extends A`

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

- B's class struct (jump table) is a super-set of A's

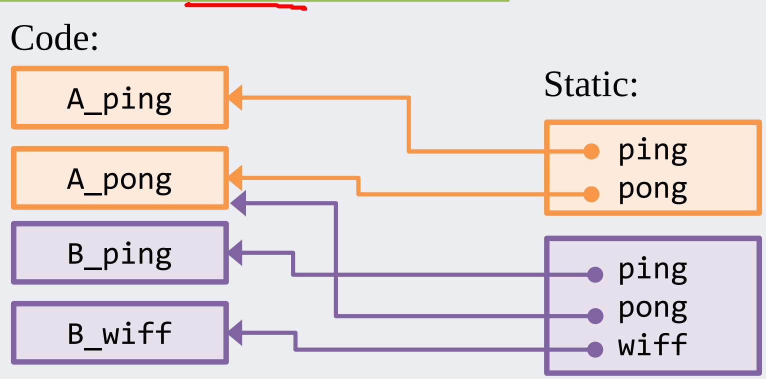
```
struct B_class {
  void (*ping) (void*);
  void (*pong) (void*);
  void (*wiff) (void*);
};
```

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

- B's method declarations and class object (static) allocation

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

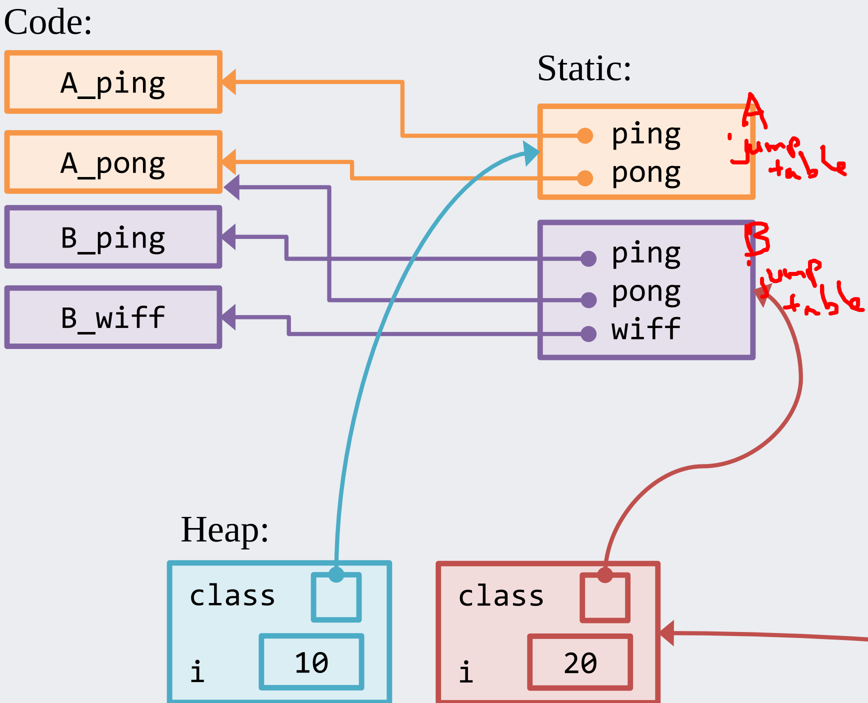
```
struct B_class B_class_table = {B_ping, A_pong, B_wiff};
```



And we will construct/allocate `struct B` instances similarly

Polymorphism in C

Dispatch diagram for C polymorphism



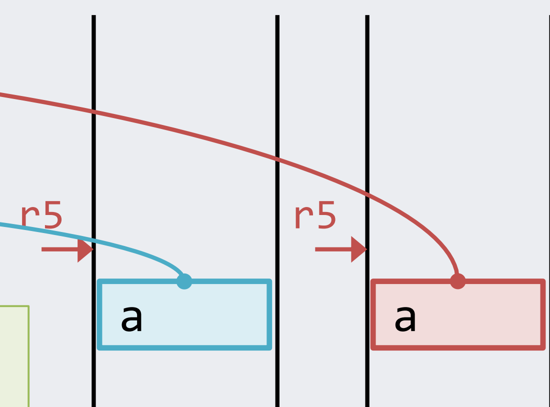

```c
void foo(void* av) {
  struct A* a = av;
  a->class->ping(a);
  a->class->pong(a);
}

void bar() {
  foo(new_A(10));
  foo(new_B(20));
}
```

foo's activation frame

```
0x0:  a
0x4: ra
0x8: av
```

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

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



Runtime stack

ISA for polymorphic dispatch

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

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

- How do we compile `a->class->ping()` ?
- Pseudocode:
  - `pc ← m[r[1]+0*4]`
- Jumps currently supported by SM213 ISA (so far):

Name	Semantics	Assembly	Machine
jump	`pc ← a`	`j a`	`b--- aaaaaaaa`
indirect jump	`pc ← r[t] + o (or r[t] + pp*2)`	`j o(rt)`	`ctpp`

- We will benefit from a new instruction in our ISA
  - that jumps to an address that is stored in memory (not just in a register)

ISA for polymorphic dispatch

Double-indirect jump instruction (b+o)

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

Name	Semantics	Assembly	Machine
jump	pc ← a	j a	b--- aaaaaaaa
indirect jump	pc ← r[t] + o (or r[t] + pp*2)	j o(rt)	ctpp
dbl-ind jump b+o	pc ← m[r[t] + o] (or m[r[t] + pp*2])	j *o(rt)	dtpp

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

```
ld   0(r5), r0   # r0 = a
ld   (r0), r1    # r1 = a->class
j    *0(r1)      # a->class->ping()
j    *4(r1)      # a->class->pong()
```

```
ld   0(r5), r0  # r0 = a
ld   (r0), r1   # r1 = a->class
deca r5         # allocate frame
st   r0, (r5)   # put a on stack
gpc  $2, r6     # get return address
j    *0(r1)     # a->class->ping(a)
inca r5         # deallocate frame
deca r5         # allocate frame
st   r0 (r5)    # put a on stack
gpc  $2, r6     # get return address
j    *4(r1)     # a->class->pong(a)
inca r5         # deallocate frame
```

Other uses of function pointers

Consider: 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);
}
```

Quicksort

- The logic/code for `Quicksort/partition` is *mostly* type-independent
  - change parameter to sort anything
    - actually, like Java, array parameter will be 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

- array parameter – easy to deal with
- `pivotValue` type – easy to deal with
- comparison – ???

Type-independent Quicksort, in Java

```
int partition (Comparable<T> array[], int left, int right, 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;
  }
}
```

Type-independent, parameterized Quicksort in C

Using a comparator function pointer

int left, int right, int pivotIndex

```
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;
}
```

formerly,
`array[i] <= pivotValue`

Side note: avoiding `void*` confusion

```
void sort (void** array, int n, int (*cmp) (void*, void*) ) { ...
```

```
int cmpIntegers (void* av, void* bv) {
  int* a = av;
  int* b = bv; ...
```

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

- What is `void*`?
  - It is a pointer type that cannot be dereferenced (directly)
    - "just an address", sometimes called an *opaque* pointer
  - in C we used it to represent a pointer to *anything*
  - before using, you must cast it to another pointer type
    - there is no type checking on this type case
    - casting can be done implicitly; explicit cast is not needed
  - code with `void*` can be confusing, especially when encountering `void**`
- C's `typedef` statement:
  - creates a new type name for a type expression
  - leads to more readable code
  - common convention: end type names with a "`_t`"
    - e.g. `typedef void* element_t;`

Sorting with `typedef`

Using `void*`:

```
int partition ( void** array, ..., int (*cmp) (void*, void*)) {
  void* pivotValue, *t;
  ...
}
```

Using new type defined by `typedef`:

```
typedef void* element_t;

int partition ( element_t* array, ..., int (*cmp) (element_t, element_t)) {
  element_t pivotValue, t;
  ...
}
```

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}; // addresses of a's elements
sort ((void**) pa, 3, cmpIntegers);
```

- To sort strings (with `typedef`):

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

or simply:

```
return strcmp(a, b);
```

```
char* array[] = {"Psyduck", "Bulbasaur", "Meowth", ..., "Eevee"};
sort ((element_t*) array, sizeof (array) / sizeof (array[0]), cmpStrings);
```

Exercise 1f.3

`pair_sort.zip` on Canvas

- Sort a list of integer pairs, in ascending order by their product

```
struct pair {
  int x;
  int y;
};
```

```
sort( void** list, int len, int (*cmp)(void*, void*));
```

Reference counting with function pointers

- Recall A6: `value` needs the same refcount as the element
  - because `value` can't automatically be freed otherwise

```
struct element {
  int num;
  char* value;
};

struct element* element_new(int num, char* value) {
  struct element* e = rc_malloc(sizeof(*e));
  e->value = rc_malloc(strlen(value) + 1);
  ...
}

void element_keep_ref(struct element* e) {
  rc_keep_ref(e->value);
  rc_keep_ref(e);
}

void element_free_ref(struct element* e) {
  rc_free_ref(e->value);
  rc_free_ref(e);
}
```

Reference counting with function pointers

Introducing: finalizers!

- This is better

```
struct element {
  int num;
  char* value;
};

void element_finalizer(void* ev) {
  struct element* e = ev;
  free(e->value);
}

struct element* element_new(int num, char* value) {
  struct element* e = rc_malloc(sizeof(*e));
  e->value = malloc(strlen(value) + 1);
  ...
}

void element_keep_ref(struct element* e) {
  rc_keep_ref(e);
}

void element_free_ref(struct element* e) {
  rc_free_ref(e);
}
```

But how/when does this finalizer get called?
How do we "automate" it?

Reference counting with function pointers

Fixing the refcount library

```
struct rc_metadata {
  int ref_count;
  void (*finalizer) (void* ev); // finalizer associated with each allocation
};

void* rc_malloc(int size, void (*ef) (void* ev)) {
  struct rc_metadata* md = malloc(size + sizeof(struct rc_metadata));
  md->ref_count = 1;
  md->finalizer = ef;

  return md + 1; // address of payload
}

void rc_free_ref(void* p) { // p is a payload address
  struct rc_metadata* md = p;
  md -= 1; // set md to start of metadata
  (md->refcount) -= 1;
  if (md->refcount == 0) {
    md->finalizer(p); // call the allocation-specified finalizer
    free (ref_count); // free the entire allocation
  }
}
```

We should actually check that the provided finalizer is not NULL before calling it

Reference counting with function pointers

Usage of the modified `rc_malloc`, etc.

```
struct element {
  int num;
  char* value;
};

void element_finalizer(void* ev) {
  struct element* e = ev;
  free(e->value);
}

struct element* element_new(int num, char* value) {
  struct element* e = rc_malloc(sizeof(*e), element_finalizer);
  e->value = malloc(strlen(value) + 1);
  ...
}

void element_keep_ref(struct element* e) {
  rc_keep_ref(e);
}

void element_free_ref(struct element* e) {
  rc_free_ref(e);
}
```

Just use `rc_keep_ref(e)` and `rc_free_ref(e)`, for all allocations

Switch statements

It's super effective!

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;
  }
}
```

```c
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;
}
```

- Semantically, the same as simplified nested/cascading 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 including this language feature?
  - is it for humans, to facilitate ease of writing and reading of code?
  - is it for compilers, permitting a more efficient implementation?
- Implementing switch statements
  - we know how to implement conditionals; is there anything more to consider?

Switch statements

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 (pokemonType) {
  case "water": ...
  case "grass": ...
  case "psychic": ...
}
```

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

Why compilers like switch statements

- Notice what we have (in a valid switch statement):
  - 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
- Example:

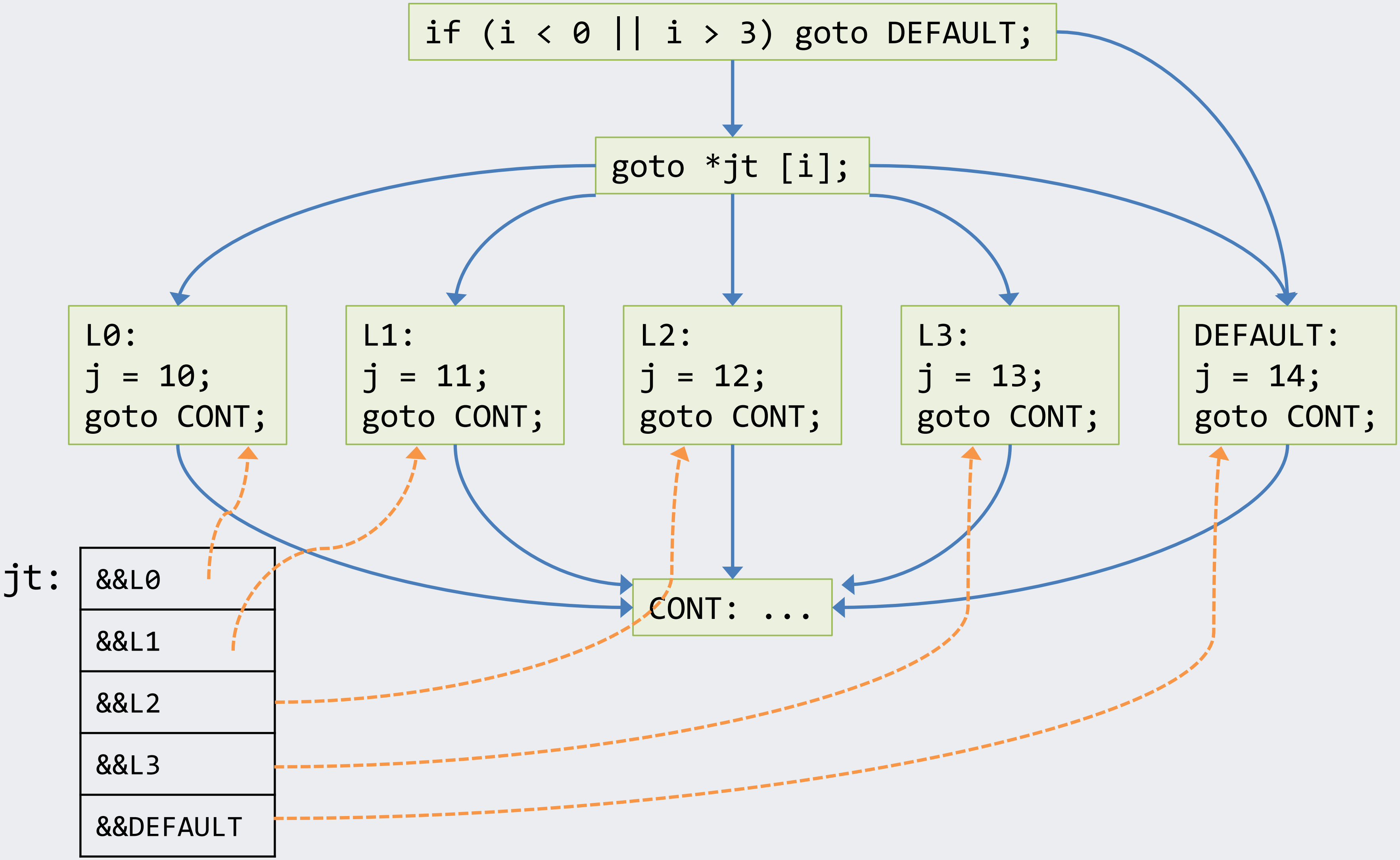
a jump table!

static const

```
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; // "break"
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; // "break"
L1: j = 11;
    goto CONT;
L2: j = 12;
    goto CONT;
L3: j = 13;
    goto CONT;
DEFAULT:
    j = 14;
    goto CONT;
CONT:
```

```
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;
```

- Computation can be much more efficient
  - compare the running time to `if`-based alternative
- Could this all go horribly wrong?
  - construct a switch statement where this implementation technique is a really bad idea

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
}
```

- Naïve 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

- Naïve 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

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

  - what is the problem?

    indices in range, but not "valid"

  - what is the solution?

    send over to default (jump table)

- Case labels not starting at 0

i-1000

```
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;
}
```

  - what is the problem?

    large portions of jump table unused

  - what is the solution?

    normalize to 0

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; // "break"
L21: j = 11;
     goto CONT;
L23: j = 13;
     goto CONT;
DEFAULT:
     j = 14;
     goto CONT;
CONT:
```

Snippet B: in assembly form

```
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
          j    *(r1, r0, 4)       # goto jmptable[i-20]
```

r1: jump table address
r0: index within jump table

```
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
```

switch cases (code)

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

statically allocated

Review: 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
branch	`pc ← a (or pc + p*2)`	`br a`	`8-pp`
branch if equal	`pc ← a (or pc + p*2) if r[c] == 0`	`beq rc, a`	`9cpp`
branch if greater	`pc ← a (or pc + p*2) if r[c] > 0`	`bgt rc, a`	`acpp`
jump	`pc ← a`	`j a`	`b--- aaaaaaaa`

- Dynamic jumps
  - jump target address is dynamic

Review: dynamic jumps

plus a new double-indirect jump

- Indirect jump
  - jump target address stored in a register
  - already introduced, but we used it for static procedure calls

Name	Semantics	Assembly	Machine
indirect jump	`pc ← r[t] + o (or r[t] + p*2)`	`j o(rt)`	`ctpp`

- Double indirect jumps
  - jump target address stored in memory
  - base-plus-displacement and indexed modes for memory access

Name	Semantics	Assembly	Machine
dbl-ind jump b+o	`pc ← m[r[t] + o] (or m[r[t] + pp*2])`	`j *o(rt)`	`dtpp`
dbl-ind jump indexed	`pc ← m[r[t] + r[i]*4]`	`j *(rt, ri, 4)`	`eti-`

new!

iClicker 1f.4

- 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])(void) = {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

iClicker 1f.5

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

proc

0	foo
1	bar
2	bat

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

iClicker 1f.6

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

A.

```
ld      (r5), r0        input
ld      $proc, r1       jump table
deca    r5
mov     r0, r2
inc     r2              input + 1
st      r2, (r5)
gpc     $2, r6
j       *(r1, r0, 4)
```

B.

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

```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);}

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

int main (int argc, char* argv[]) {
  int input;  ← local variable of main
  if (argc == 2) {
    input = atoi (argv [1]);
    if (input>=0 && input <=2)
     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.