CPSC 213
Introduction to Computer Systems

Unit 1c: Instance Variables and Dynamic Allocation

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

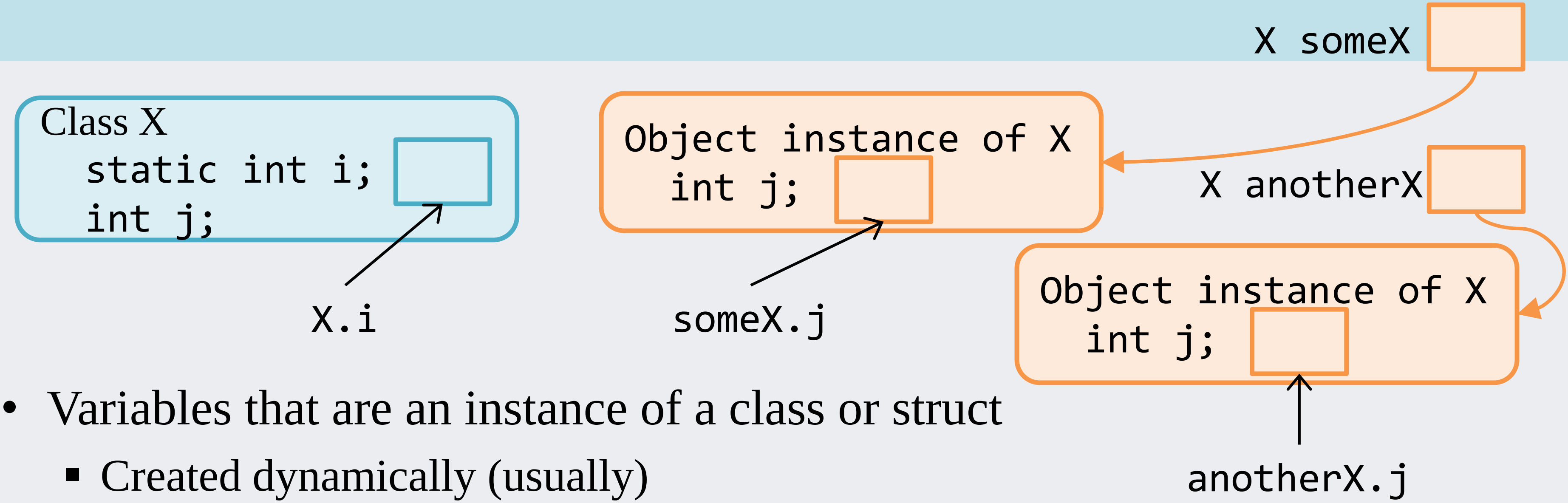
- Reading
  - Companion: 2.4.4-2.5
- Reference
  - Textbook: 3.9.1, 9.9, 3.10
- Learning objectives
  - read and write C code that includes structs
  - compare Java classes/objects with C structs
  - explain the difference between static and non-static variables in Java and C
  - explain why ISAs have both base-plus-offset and indexed addressing modes by showing how each is useful for implementing specific features of programming languages like C and Java
  - distinguish static and dynamic computation for access to members of a static struct variable in C
  - distinguish static and dynamic computation for access to members of a non-static struct variable in C
  - translate C struct-access code into assembly language
  - count memory references required to access struct elements
  - compare Java dynamic allocation the C's explicit-delete approach by identifying relative strengths and weaknesses
  - identify and correct dangling-pointer and memory leak bugs in C caused by improper use of free()
  - write C code that uses techniques that avoid dynamic allocation as a way to minimize memory-allocation bugs
  - write C code that uses reference counting as a way to minimize memory-allocation bugs
  - explain why Java code does not have dangling-reference errors but can have memory-leak errors

Variable types seen so far…

The simplest variables in any language

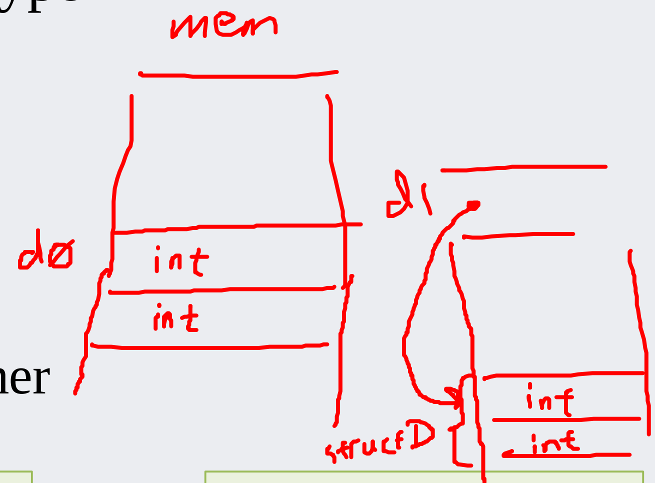
- Static variables
  - the compiler allocates them; i.e. their memory address is a constant
- Scalars and arrays
  - a scalar variable stores a single value
  - an array stores multiple values named by a single variable and an index
    - array data can be allocated either statically or dynamically
    - array variable can *be* the array (static) or store a pointer to it (dynamic)
    - index is generally a dynamic value

Instance variables



- Variables that are an instance of a class or struct
  - Created dynamically (usually)
  - many instances of the same class/struct can coexist
- Java vs C
  - Java: objects are instances of non-static variables of a class
  - C: structs are named variable groups, instance also called a struct
- Accessing an instance variable
  - requires a reference/pointer to a particular object/struct
  - then variable name chooses a variable in that object/struct

- A struct is a collection of variables of arbitrary type
  - allocated and accessed together



- Declaration
  - similar to declaring a Java class without methods
  - name is "struct" plus name provided by programmer
    - static: `struct D d0;`
    - dynamic: `struct D* d1;`

type

```
struct D {
  int e;
  int f;
};
```

≈

```
class D {
  public int e;
  public int f;
};
```

- Access
  - static: $\overbrace{\texttt{d0}}^{\text{struct D}}$`.e = d0.f;`
  - dynamic: $\underbrace{\texttt{d1}}_{\text{struct D*}}$`->e = d1->f;`

can also access variables by dereferencing pointer and using "."

$$d1 \to e \equiv (*d1).e$$

$$\underbrace{(*\underbrace{d1}_{\text{struct D*}})}_{\text{struct D}}.e$$

Struct allocation

```
struct D {
  int e;
  int f;
};
```

- Static structs are allocated by the compiler

```
struct D d0;
```

Static memory layout

```
0x1000: value of d0.e
0x1004: value of d0.f
```

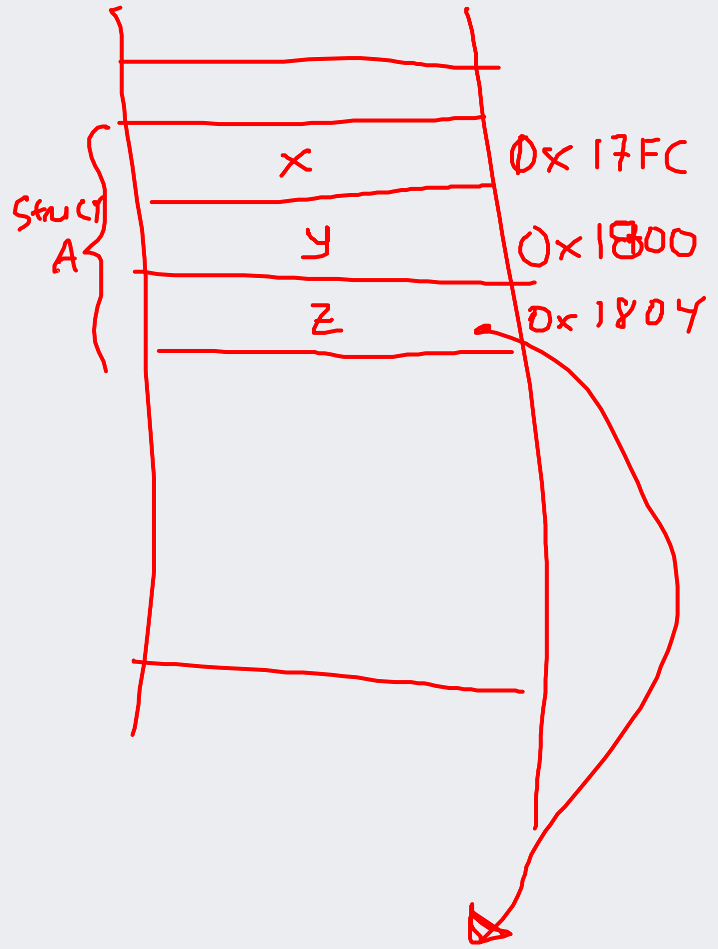
- Dynamic structs are allocated at runtime
  - the variable that stores the struct pointer may be static or dynamic
  - the struct itself is allocated when the program calls `malloc`

```
struct D* d1;
```

Static memory layout

```
0x1000: value of d1 (address of struct)
```

```
struct A {
    int x ;   4 bytes
    int y ;   4 bytes
    int* z ;  4 bytes
} ;
```



Struct allocation

Dynamic allocation

```
struct D {
  int e;
  int f;
};
```

- Runtime allocation of dynamic struct

.pos 0x1000

d1: .long 0

(static)

```
struct D* d1;
```

```
void foo() {
  d1 = malloc( sizeof(struct D) );
};
```

- assuming that the code above allocates the struct at address 0x2000:

d1:

```
0x1000: 0x2000 (address of struct)
...
0x2000: value of d1->e
0x2004: value of d1->f
```

Struct access

in assembly

```
struct D {
  int e;
  int f;
};
```

- Struct members can be accessed using offset from base address
  - offset to each member/variable from base of struct is static
- As before, static and dynamic differ by an extra memory access

(d0 is static)

```
d0.e = d0.f;
```

```
0x1000: value of d0.e
0x1004: value of d0.f
```

```
m[0x1000] ← m[0x1004]
```

```
r[0] ← 0x1000          # r0 = base address

r[1] ← m[r[0] + 4]     # r1 = d0.f
m[r[0]] ← r1           # d0.e = r1
```

??

```
d1->e = d1->f;
```

d1

```
0x1000: 0x2000
...
0x2000: value of d1->e
0x2004: value of d1->f
```

```
m[m[0x1000]] ← m[m[0x1000] + 4]
```

```
r[0] ← 0x1000          address of d1 pointer
r[1] ← m[r[0]]         r1: base address of struct
r[2] ← m[r[1] + 4]
m[r[1]] ← r[2]
```

```
struct D {
  int e;
  int f;
};
```

Struct access

in assembly

```
d0.e = d0.f;
```

```
0x1000: value of d0.e
0x1004: value of d0.f
```

```
m[0x1000] ← m[0x1004]
```

```
r[0] ← 0x1000

r[1] ← m[r[0] + 4]
m[r[0]] ← r[1]
```

```
ld $0x1000, r0 # r0 = address of d0

ld 4(r0), r1    # r1 = d0.f
st r1, (r0)     # d0.e = d0.f
```

```
d1->e = d1->f;
```

```
0x1000: 0x2000
...
0x2000: value of d1->e
0x2004: value of d1->f
```

```
m[m[0x1000]] ← m[m[0x1000] + 4]
```

```
r[0] ← 0x1000
r[1] ← m[r[0]]
r[2] ← m[r[1] + 4]
m[r[1]] ← r[2]
```

```
ld $0x1000, r0 # r0 = address of d1
ld (r0), r1     # r1 = address of struct
ld 4(r1), r2    # r2 = d1->f
st r2, (r1)     # d1->e = d1->f
```

- Introducing… the load/store base plus offset instruction
  - dynamic base address in a register, plus a static offset (displacement)

```
ld 4(r1), r2
```

SM213 ISA

Revised to include offsets

- Machine format for base + offset:
  - offset in our case will always be a multiple of 4
  - we have only 4 bits to store the offset
  - so, we will store $p = {}^{o}/_{4}$ in the instruction

Name	Semantics	Assembly	Machine
load immediate	`r[d] ← v`	`ld $v, rd`	`0d-- vvvvvvvv`
load base + offset	`r[d] ← m[r[s]+o]`	`ld o(rs), rd`	`1psd`
load indexed	`r[d] ← m[r[s]+4*r[i]]`	`ld (rs, ri, 4), rd`	`2sid`
store base + offset	`m[r[d]+o] ← r[s]`	`st rs, o(rd)`	`3spd`
store indexed	`m[r[d]+4*r[i]] ← r[s]`	`st rs, (rd, ri, 4)`	`4sdi`

Memory addressing modes reviewed

- Scalars `i = a;`
  - address in register (e.g. `r1`)
  - access memory at address in register `ld (r1), r0`

- Arrays `i = a[j];`
  - base address in register (e.g. `r1` stores address of `a`)
  - dynamic index in register (e.g. `r2` stores value of `j`) `ld (r1, r2, 4), r0`
  - access memory at base + index * element size (e.g. 4)

- Struct members (instance variables) `i = a.j;` `i = b->k;`
  - base address in register (e.g. `r1` stores starting address of struct)
  - static (constant) offset (e.g. `X` is offset to `j`/`k` from beginning of struct)
  - access memory at base plus offset `ld X(r1), r0`
  - equivalent to array access with static index

Variations in struct declaration

- struct variables can be declared inside other structs

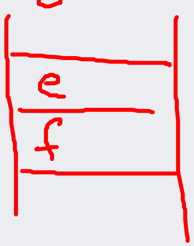
```
struct D {
  int e;
  int f;
};
```

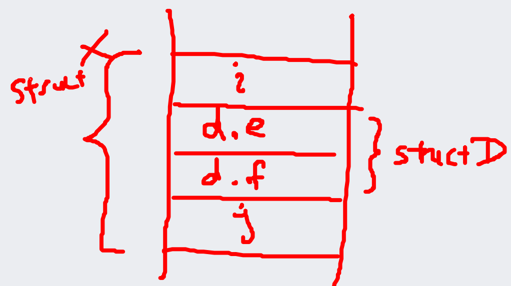
```
struct X {
  int i;
  struct D d;
  int j;
};
```

```
struct Y {
  int i;
  struct D* d;
  int j;
};
```

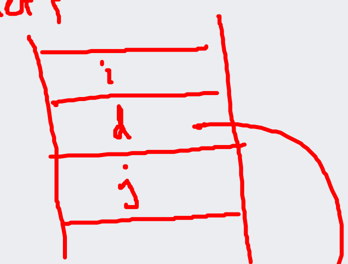
```
struct Z {
  int i;
  struct Z* z;
  int j;
};
```

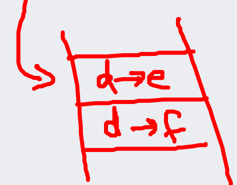
struct D layout:





struct Y





- struct members can be arrays or pointers

W contains the actual array

```
struct W {
  int i;
  int a[10];
  int j;
};
```

```
struct U {
  int i;
  int* a;
  int j;
};
```

array lives somewhere else

- What is the offset (in bytes) of member `j` in `struct X` below?

```c
struct D {
  int e;
  int f;
};
```

```c
struct X {
  int i;
  struct D d;
  int j;
};
```

struct D

0	e
4	f

struct X

0	i
4	d.e
8	d.f
12	j

A. 0

B. 4

C. 8

D. 12

E. something else

iClicker 1c.2

- What is the offset (in bytes) of member `j` in `struct X` below? Assume 4-byte addresses.

```
struct D {
  int e;
  int f;
};
```

```
struct X {
0 int i;
4 struct D* d;
8 int j;
};
```

A. 0

B. 4

C. 8

D. 12

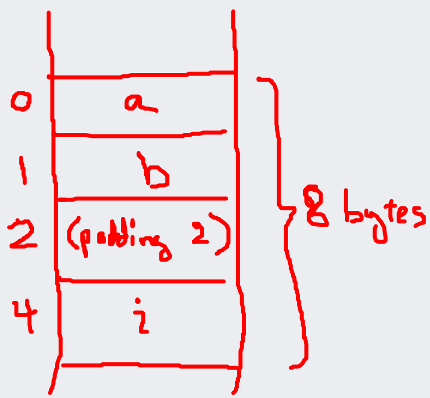
E. something else

Struct size and alignment

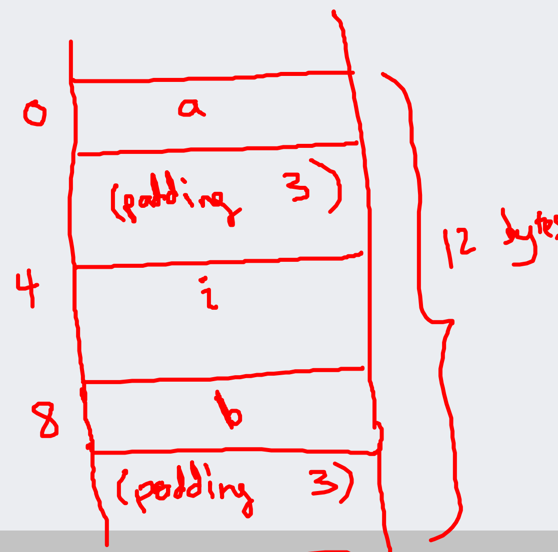
- Alignment rules apply inside of a struct
  - Each instance variable will be aligned within the struct according to its type size
  - structs are aligned according to their largest instance variable type

  - Structs can mix types (padding might be added in or after the members)
    - What are the *sizes* of the structs below? (i.e. `sizeof(struct X)`)
    - what are the *offsets* of each of their members?

```
struct X {
  char a;
  char b;
  int i;
};
```




```
struct Y {
  char a;
  int i;
  char b;
};
```



Arrays of structs

- We can declare arrays of structs (or of struct pointers)

```
struct X {
  int i;
  int j;
  int k;
};
```

```
struct X a[10];
struct X* b[10];

...

a[3].i = b[5]->j;
```

array of struct X

one struct X

array of pointers

one pointer

iClicker 1c.3

- What is the offset (in bytes) of member `j` in `struct X` below?

```
struct D {
  int e;
  int f;
};
```

4
4
8 bytes

```
struct X {
  int i;
  struct D d[10];
  int j;
};
```

4
80 bytes
84

Ans. 84

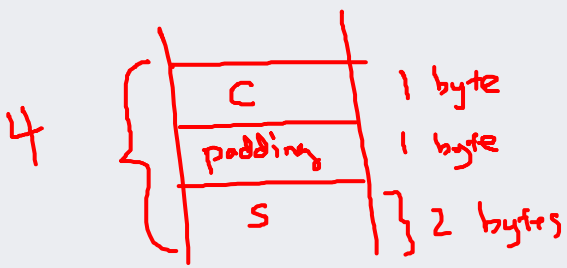
short: 2 bytes

- What is the offset (in bytes) of member `j` in `struct X` below?

```
struct Y {
  char c;
  short s;
};
```

```
struct X {
  int i;   4
  struct Y y[10]; 40
44 int j;
};
```

struct Y



Ans 44

iClicker 1c.5

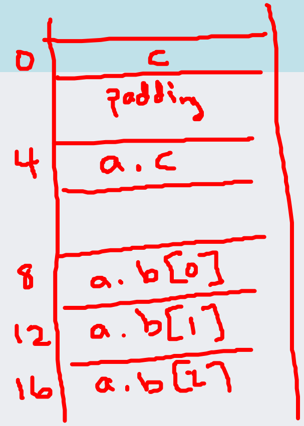
- Given

44 bytes

```
struct A {
  char a;
  int b[10];
};
```

1+3 padding

```
struct X {
  char c;
  struct A a;
  int j;
};
```




```
struct X s; // a global variable
```

```
ld $s, r0
```

- Which of the following loads `s.a.b[2]` into `r1`?

A. `ld 10(r0), r1`

B. `ld 12(r0), r1`

C. `ld 16(r0), r1`

D. `ld 20(r0), r1`

E. `ld 24(r0), r1`

iClicker 1c.6

- Given

```
struct A {
  char a;
  int b[10];
  int* p;
};
```

```
struct X {
  char c;
  struct A a;
  int j;
};
```

```
struct X s; // a global variable
```

```
ld $s, r0
```

- Which of the following loads `s.a.b[2]` into `r1`?

A. `ld 10(r0), r1`

B. `ld 12(r0), r1`

C. `ld 16(r0), r1`

D. `ld 20(r0), r1`

E. `ld 24(r0), r1`

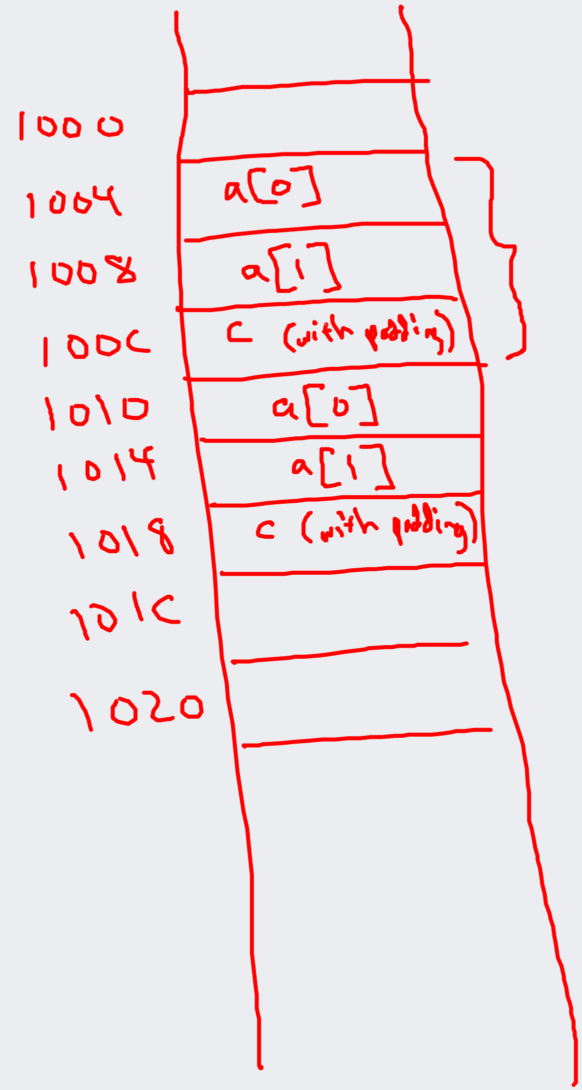
iClicker 1c.7

- Given

```
struct X {
  int a[2];
  char c;
};
```

2×4
1
+3 padding

```
struct X s; // a global variable
```



- What is the value of `sizeof(s)`?

A. 3

B. 9

C. 12

D. 16

iClicker 1c.8

- Given

```
struct A {
  char a;
4 int* b;
  int* p;
};
```

```
struct X {
  char c;
  struct A* a;
  int j;
};
```

```
struct X* s; // a global variable
```

```
ld $s, r0        # r0: address of s pointer
ld (r0), r0      # r0: address of the struct X
ld 4(r0), r0     # r0: value of s->a
                 #     (address of struct A)
ld 4(r0), r0     # r0: value of s->a->b
                 #     (address of 'int array')
ld 8(r0), r1     # r1: s->a->b[2]
```

access memory

- How many memory reads to load `s->a->b[2]` into `r1`?

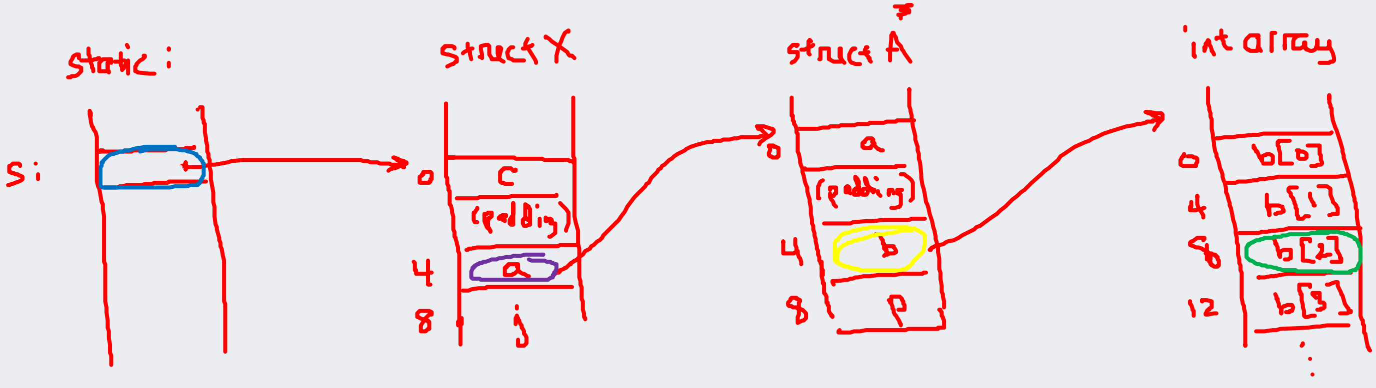
A. 2

B. 3

C. 4

D. 5

E. 6



iCliker 1c.9

- Given

```
struct A {
  char a;
  int b[10];
  int* p;
};
```

3 bytes padding

```
struct X {
  char c;
  struct A a;
  int j;
};
```

3 bytes padding

```
struct X s; // a global variable
```

static:

s: 0	c
	(padding)
4	a.a
	(padding)
8	a.b[0]
12	a.b[1]
16	a.b[2]
20	a.b[3]

- How many memory reads to load `s.a.b[2]` into `r1`?

A. 1

B. 2

C. 3

D. 4

E. 5

ld $s, r0 # r0: base address of s

ld 16(r0), r1 # r1: s.a.b[2]

- Given

```
struct A {
  char a;
  int* b;
  int* p;
};
```

3 bytes padding

```
struct X {
  char c;
  struct A a;
  int j;
};
```

3 bytes padding

ld $s, r0
- ld 8(r0), r0
- ld 8(r0), r1

```
struct X s; // a global variable
```

- How many memory reads to load `s.a.b[2]` into `r1`?

A. 1

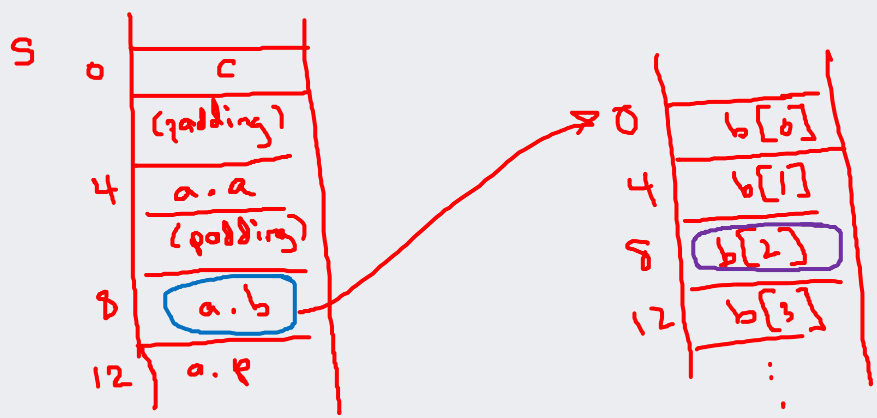
B. 2

C. 3

D. 4

E. 5

Static



Dynamic allocation in C and Java

- Programs can allocate memory dynamically
  - allocation reserves a range of memory for a purpose
- In Java, instances of classes are allocated by the `new` keyword
- In C, byte ranges are allocated by call to `malloc` procedure
  - these bytes can be used for any type that can fit in them

Dynamic allocation in C

- Memory allocation: `void* malloc(unsigned long n);`
  - n is the number of bytes to allocate
  - return type is void*
    - a pointer to anything (no specific type assigned) "just an address"
    - can be cast to/from any other pointer type
    - cannot be dereferenced directly

allocation (and the number of bytes) is associated with the base address

- Use `sizeof` to determine the number of bytes to allocate
  - `sizeof(x)` statically computes the # of bytes in a type or variable
  - caution: `sizeof(pointer)` gives the size of a *pointer*, not what it *points to*

```
struct Foo* f = malloc(sizeof(*f));
```

possible, but dangerous!

void* p = malloc(sizeof(struct Foo));   *((int*) p) = 4;

void* q = malloc(sizeof double);

Memory deallocation

- Wise management of memory requires deallocation
  - memory is a finite resource
  - deallocation frees previously allocated memory for re-use
- In Java:
  - automatic garbage collection: requires keeping track of every reference to an object
- In C:
  - Dynamic memory must be deallocated explicitly by calling `free`
  - `free` deallocates memory immediately; does not check to see if it is still in use
- Memory leak:
  - when dynamically allocated data is not deallocated when it is no longer needed
  - size of program gradually increases; available memory leaks away

Memory heap

- The heap is a large section of memory from which `malloc` allocates objects
  - `malloc` finds unused space in the heap, marks the chunk of bytes as used, and returns the address of the first byte
  - `free` marks the previously allocated chunk of bytes as unused

- In Java: all objects are stored on the heap

malloc (number of bytes) : returns address of first byte

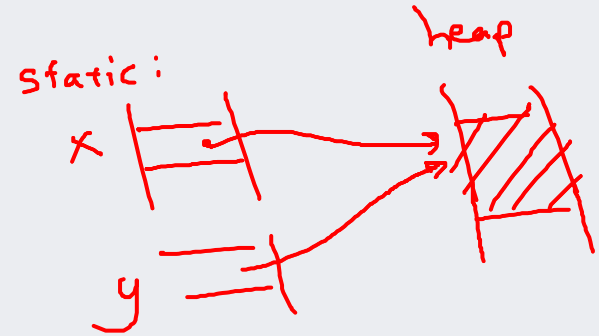
free (address of an allocation)

Issues with explicit deallocation

must be an address associated with a malloc call

- What `free(x)` does
  - deallocates "object" at address `x` (x is a pointer)
  - this memory can be reused by subsequent call to malloc

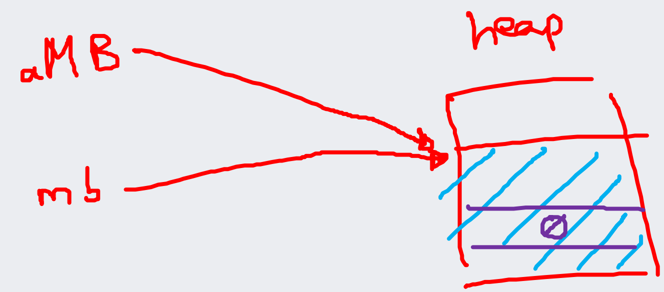
- What `free(x)` does not do
  - it does not change the value of `x`
  - other variables may still point there too
  - the binary data stored at address `x` is not erased



Issues with explicit deallocation

```
struct buffer* create() {
  struct buffer* buf = malloc(sizeof(*buf));
  ...
  return buf;
}

void foo() {
  struct buffer* mb = create();
  bar(mb);
  free(mb);
}

struct buffer* aMB;

void bar(struct buffer* mb) {
  aMB = mb;
}

void bat() {
  aMB->x = 0;
}
```

malloc (sizeof (struct buffer)) ;



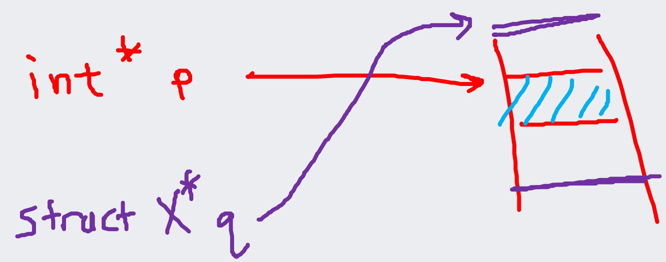
- what might happen in `bar()`, and
- why a subsequent call to `bat()` would expose a serious bug?

Dangling pointers

"use after free"

- A dangling pointer is a pointer to an object that has been freed
  - or, could point to unallocated memory or to another object

- Why they are a problem
  - program thinks it is writing to object of type X, but isn't actually
  - it may be writing to an object of type Y



Avoiding dangling pointers in C

- Understand the problem
  - when allocation and free appear in different places in your code
  - e.g. when a procedure returns a pointer to something it allocates
- Avoid the problem cases, if possible
  - restrict dynamic allocation/free to single procedure
  - don't write procedures that return pointers
  - use local variables instead
    - we'll see later that local variables are automatically allocated on call and freed on return
- Engineer for memory management
  - define rules for which procedure is responsible for deallocation
  - implement explicit reference counting if multiple potential deallocators
  - define rules for which pointers can be stored in data structures
  - use coding conventions and documentation to ensure rules are followed

Co-locate allocation and deallocation

- If a procedure returns value of dynamically allocated object
  - allocate that object in its caller, and pass pointer to it to the procedure (callee)
  - good if caller does both `malloc` / `free` itself

```
struct buffer* receive() {
  struct buffer* buf = malloc( sizeof(*buf) );
  ...
  return buf;
}

void foo() {
  struct buffer* mb = receive();
  free(mb);
}
```

```
void receive(struct buffer* buf) {
  ...
}

void foo() {
  struct buffer* mb = malloc( sizeof(*mb) );
  receive(mb);
  free(mb);
}
```

Delegate problem to caller

Transfer `malloc` to foo

Use local variables instead

- If a procedure does both malloc and free
  - use a "static" local variable instead of malloc
    - and don't give another procedure a pointer to the local variable
  - local variables are allocated on call and deallocated on return

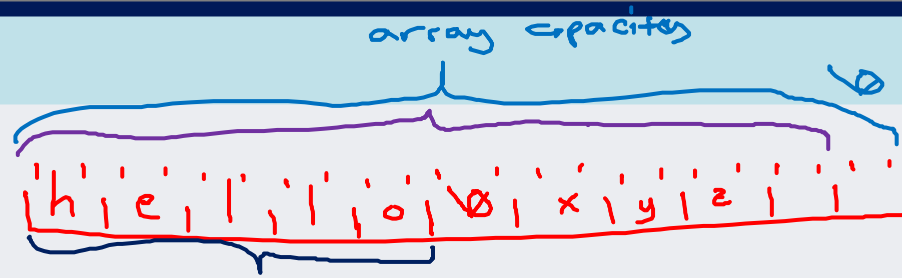
```
struct buffer* receive() {
  struct buffer* buf = malloc( sizeof(*buf) );
  ...
  return buf;
}

void foo() {
  struct buffer* mb = receive();
  free(mb);
}
```

```
void receive(struct buffer* buf) {
  ...
}

void foo() {
  struct buffer mb;
  receive(&mb);
}
```

local struct buffer variable

automatically allocated on call, passed by reference to `receive`

Example – C strings



- A string `"like this"` in C:
  - is an array of `char`s
  - has its end indicated by the first null `'\0'` in the array
  - so, every string has a
    - maximum length (the capacity of the array – 1), and
    - a printable length determined by the position of the first null character

- The standard C library (in `string.h`) has many operations on strings
  - e.g. `strlen(s)` returns the (printable) length of a string

char*

- Let's consider:
  - how to create a new string that is a copy of an existing string

String copy

Version 1

```
char* copy(char* s) {
  int len = strlen(s);
  char* d = malloc(len + 1);

  for (int i = 0; i <= len; i++)
    d[i] = s[i];

  return d;
}
```

```
// in your application program
void foo(char* s) {
  char* d = copy(s);
  printf("%s\n", d);

  free(d);
}

void bar() {
  foo("Hello, World!");
}
```

C library equivalent: `strdup`

- What can be improved in this code?

- delegate allocation, so that malloc and free are in the same procedure

String copy

Version 2

```
void copy(char* d, char* s) {
  int len = strlen(s);

  for (int i = 0; i <= len; i++)
    d[i] = s[i];
}
```

length for copying

```
// in your application program
void foo(char* s) {
  int len = strlen(s);
  char* d = malloc(len + 1);
  copy(d, s);
  printf("%s\n", d);

  free(d);
}
```

length for allocation

C library equivalent: `strcpy`

- Pros and cons of this version?

what if d is much shorter than s?

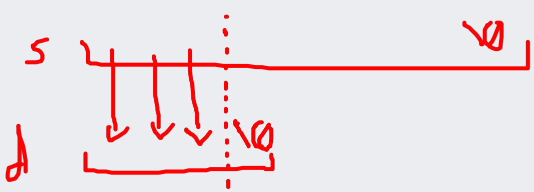
String copy

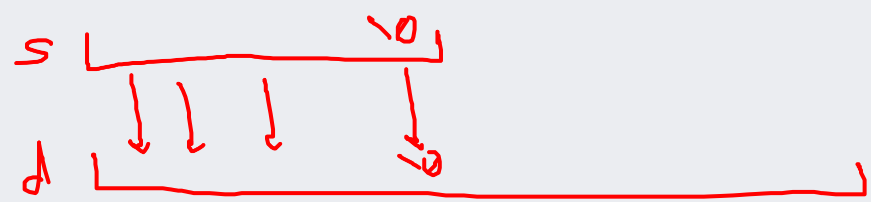
Version 3

```
void copy(char* d, int dsize, char* s) {
  int len = strlen(s);

  if (len > dsize-1) len = dsize-1;

  for (int i = 0; i < len; i++)
    d[i] = s[i];

  d[len] = '\0';
}
```

C library equivalent: `strncpy`

```
// in your application program
void foo(char* s) {
  int len = strlen(s);
  char* d = malloc(len + 1);
  copy(d, len+1, s);
  printf("%s\n", d);

  free(d);
}
```





"…if possible…"

… use local variables, don't return pointers, malloc/free in same procedure

- Sometimes, it's not possible
- If a reference needs to be passed among multiple modules
  - each module itself knows when it needs and doesn't need the reference
  - but these modules may not communicate that information between themselves

- What do we need to know in order to correctly `free` an object?
  - `free` should only happen if the object is still in use
  - you need to keep track of all its uses
    - when it starts being used
    - when it stops being used

Reference counting

- We can use reference counting to track object use
  - any procedure that stores a reference (starts using the object) increments the count
  - any procedure that discards a reference (stops using the object) decrements the count
  - the object is `free`d when the count goes to zero

Reference counting

One possible approach

- Reference counter can be part of a `struct`
  - updates to reference counter are done through special methods
  - no longer explicitly call `malloc`/`free` directly when creating the struct instances

```
struct buffer* malloc_buf() {
  struct buffer* buf = malloc(sizeof *buf);
  buf->ref_count = 1;
  return buf;
}
```

```
struct buffer {
  ...
  // the actual data attributes
  ...
  int ref_count;
}
```

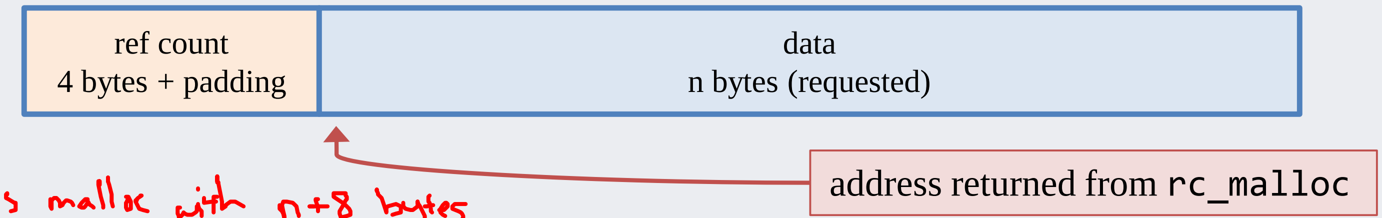
```
void add_reference(struct buffer* buf) {
  buf->ref_count++;
}
```

```
void free_reference(struct buffer* buf) {
  buf->ref_count--;
  if (buf->ref_count == 0)
    free(buf);
}
```

Reference counting

Another approach

address returned by malloc



rc_malloc calls malloc with n+8 bytes

- Sample code on Canvas: ref-count-grades-example.zip
  - refcount.c: malloc/free wrapper with ref-count implementation
  - map.c: hashmap where values are ref-counted
  - intlist.c: ref-counted list of integers
  - grades.c: implements a map from student ID to grade list

- Key memory management challenge
  - map.c and grades.c have references to grade lists
  - lists may be released in different points of the code
    - mapping may change from one value to another
    - marks may still be in use when mapping changes

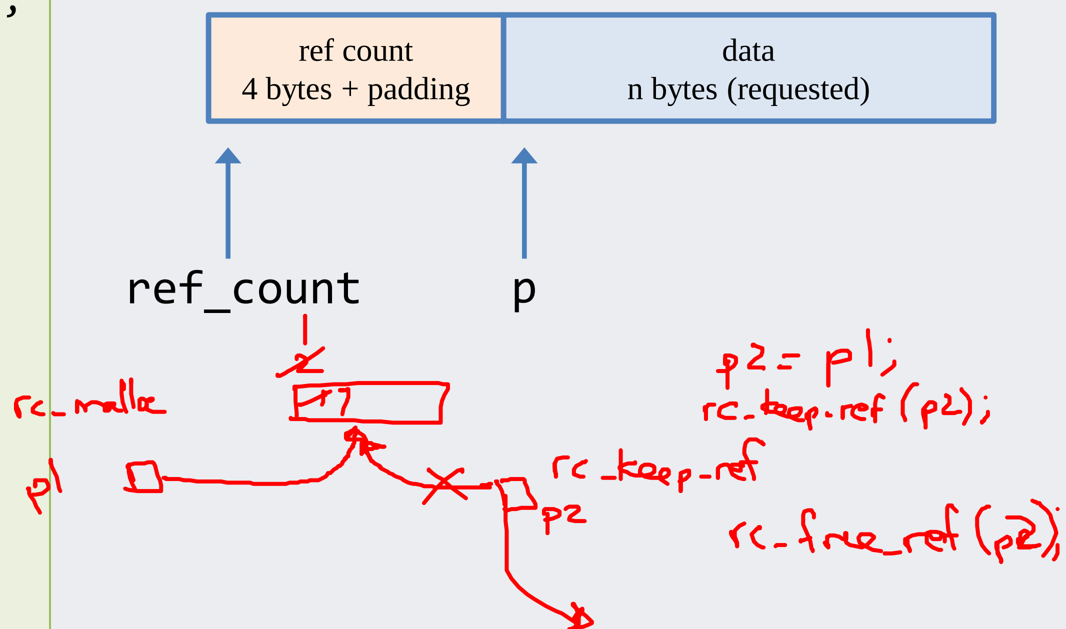
Implementing reference counting

refcount.h

```
void* rc_malloc(int nbytes);  // alternative malloc with room for ref_count
void rc_keep_ref(void* p);    // call when reference added
void rc_free_ref(void* p);    // call when reference removed
```

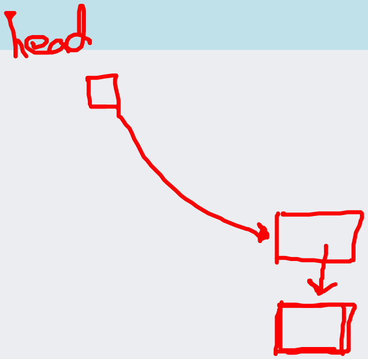
refcount.c

```
void* rc_malloc(int nbytes) {
  int* ref_count = malloc(nbytes + 8);
  *ref_count = 1;
  return ((void*) ref_count) + 8;
}

void rc_keep_ref(void* p) {
  int* ref_count = p - 8;
  (*ref_count)++;
}

void rc_free_ref(void* p) {
  int* ref_count = p - 8;
  (*ref_count)--;
  if (*ref_count == 0)
    free(ref_count);
}
```



Reference counting demonstration

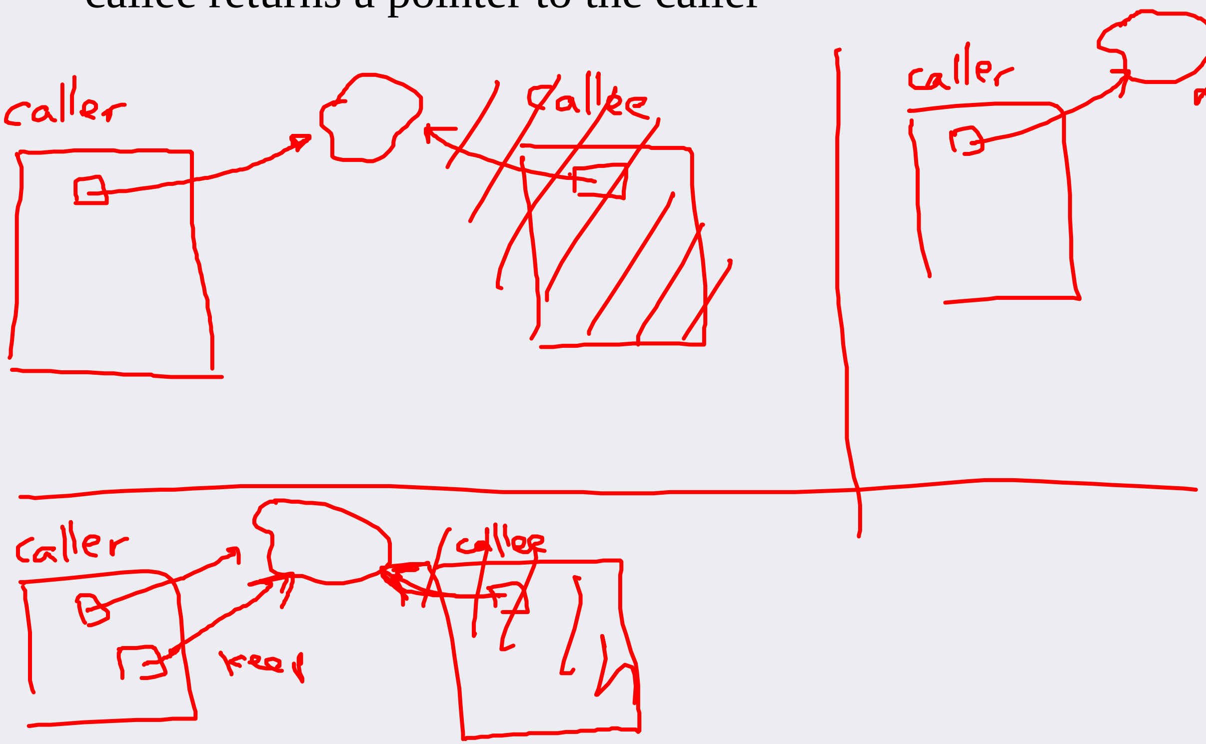
stack.zip, on Canvas



Reference counting and procedure calls

refcount_cases.c, on Canvas

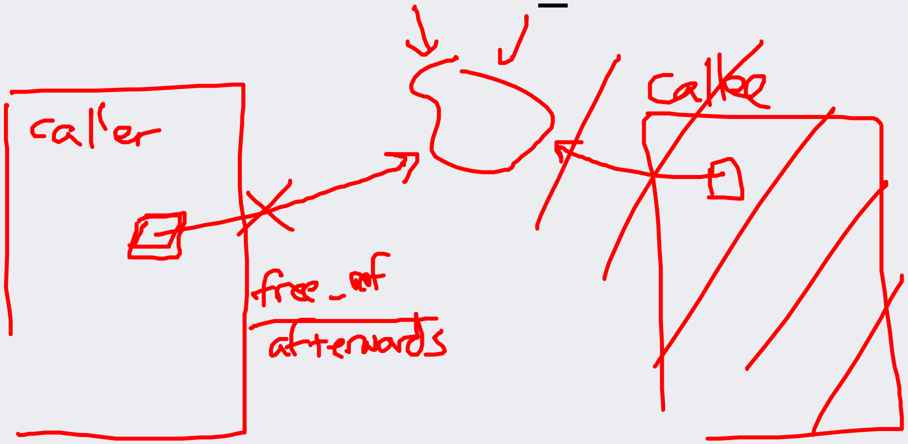
- When a reference is a parameter
  - the caller has a reference and maintains it for the duration of the call
  - so, the callee need not perform a `keep_ref` UNLESS
    - callee saves the pointer someplace that will outlast the call (e.g. global variable)
    - callee returns a pointer to the caller



Reference counting and procedure calls

refcount_cases.c, on Canvas

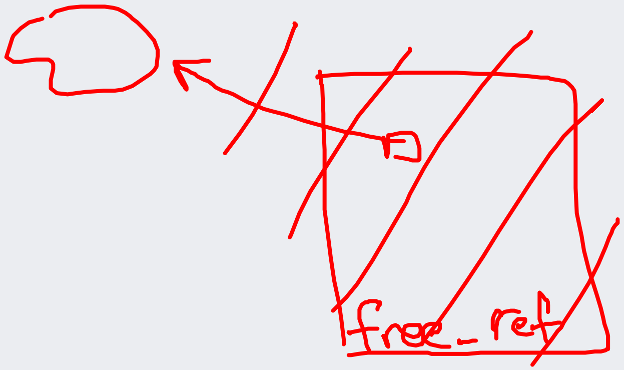
- When a reference is a return value
  - the callee must have a reference to the value
  - it passes that reference to the caller
    - the callee implicitly gives up its reference as part of the return
    - the reference is transferred to the caller
  - the caller must call `free_ref` when it no longer stores the reference



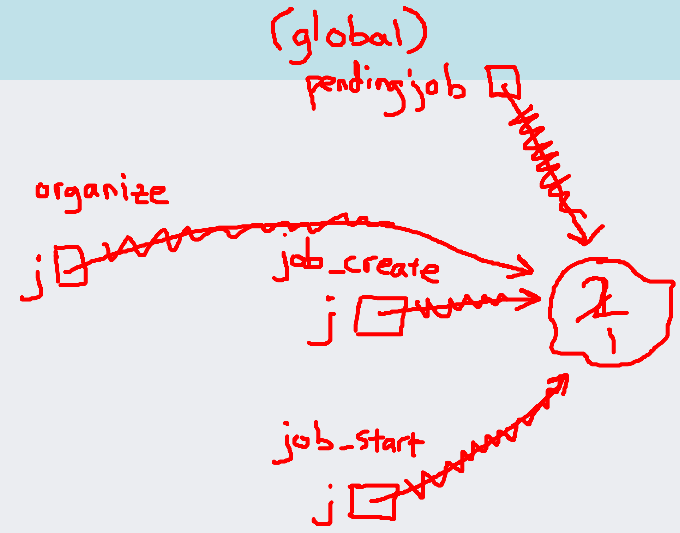
Reference counting and procedure calls

refcount_cases.c, on Canvas

- When a reference is stored in a local variable
  - that variable goes away implicitly when the procedure returns
  - and so the procedure must call `free_ref` before it returns



job.zip on Canvas



- Select ALL of the following that are true.
- Line numbers refer to the original program

1. `job.c`, between lines 26 and 27 B
2. `job.c`, between lines 32 and 33 A
3. `job.c`, between lines 35 and 36 D
4. Replace in `organize.c`, line 8 C

A. add `rc_keep_ref`
B. add `rc_free_ref`
C. replace line(s)
D. no change

Be careful with references

quiz.zip, on Canvas

- Consider this code:

```
int main(int argc, char** argv) {
  char* name = strdup("Quiz X");
  quiz_t quizzes[4];
  for (int i = 0; i < 4; i++) {
    name[5] = i + 49; // ASCII numeral
    quizzes[i] = quiz_create(name);
  }
  for (int i = 0; i < 4; i++) {
    printf("%d %s\n", i, quiz_get_name(quizzes[i]));
  }
}

quiz_t quiz_create(char* name) {
  struct quiz* quiz = malloc(sizeof(*quiz));
  quiz->name = name;
  return quiz;
}
```

main

name

Quiz ~~X~~ 4

quiz create

name

It prints:

```
0 Quiz 4
1 Quiz 4
2 Quiz 4
3 Quiz 4
```

Why?

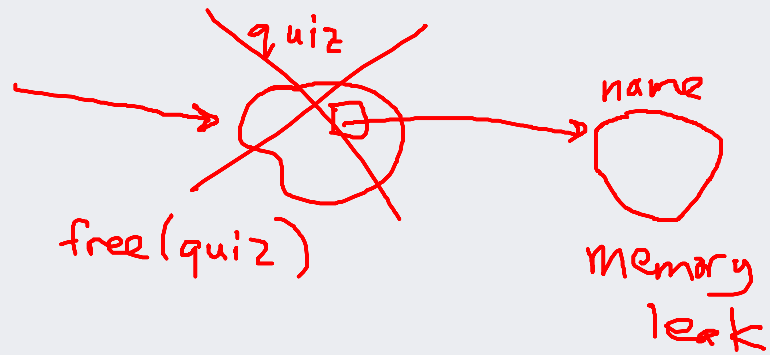
Nesting and reference counting

quiz.zip, continued

- How does this impact reference counting?

```
quiz_t quiz_create(char* name) {
  struct quiz* quiz = malloc(sizeof(*quiz));
  quiz->name = strdup(name);
  return quiz;
}

void quiz_delete(quiz_t quiz) {
  free(quiz->name);
  free(quiz);
}
```

What must change to use reference counting?

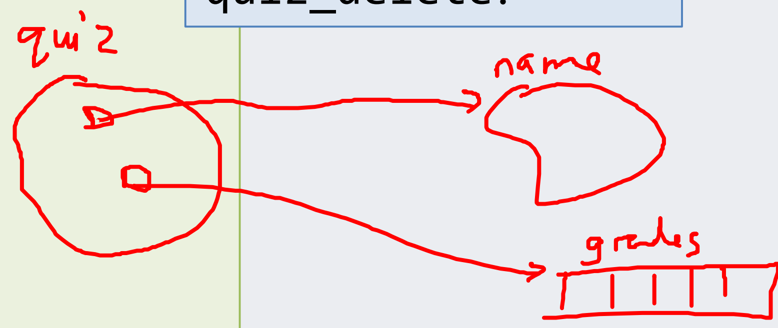


Nesting and reference counting

What if there is more to the quiz structure?

```c
struct quiz {
  char* name;
  int* grades;
  int num_grades;
};

quiz_t quiz_create(char* name) {
  quiz_t quiz = rc_malloc(sizeof(*quiz));
  int name_size = strlen(name) + 1;
  quiz->name = rc_malloc(name_size);
  strlcpy(quiz->name, name, name_size);
  quiz->grades = NULL;
  return quiz;
}

void addGrades(quiz_t quiz, int* grades, int numGrades) {
  quiz->grades = rc_malloc(sizeof(*quiz->grades) * numGrades);
  for (int i = 0; i < numGrades; i++)
    quiz->grades[i] = grades[i];
}

void quiz_delete(quiz_t quiz) {
  rc_free_ref(quiz->name);
  rc_free_ref(quiz);
}
```

How should we fix `quiz_delete`?



Detecting memory problems

Valgrind

- Valgrind is a program that performs dynamic analysis of the runtime of a program
  - e.g. run using `valgrind ./my_program`

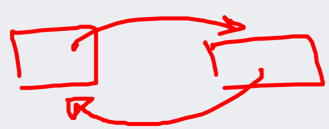
- It runs your program and monitors dynamic allocation and deallocation
- It can tell if your program has:
  - memory leaks
  - use after free (dangling pointers)

Garbage collection

Deallocation in Java

- In Java, objects are deallocated implicitly
  - the program never says `free`
  - the runtime system tracks every object reference
  - a garbage collector runs periodically to deallocate unreachable objects

- Advantage compared to explicit `free` (C)
  - No dangling pointers
  - (almost) no memory leaks
  - reference cycles are not a problem
    - e.g. object a has a pointer to b, object b has a pointer to a
    - these would cause memory leak when using reference counting alone



Discussion

- What are the advantages of explicit free?

  fast, efficient, makes us careful with memory management

- What are the advantages of reference counting?

  reduces inter-module coupling

- What are the advantages of garbage collection?

  automatic, handles most cases

- Should we ignore deallocation in Java programs?

  NO e.g. HashMap

Memory management in Java

- Memory "leaks" can still occur
  - when garbage collector fails to reclaim unneeded objects
  - still a significant problem for long-running programs where garbage accumulates

- Why would an object not be reclaimed?
  - garbage collector only reclaims unreachable objects
  - Unneeded/unused objects that keep references are not reclaimed
  - Collections and maps may maintain old unneeded objects