C

TA: Sasha Singh

Some Basics

- C is:
  - **Weakly typed** -you can cast anything into anything
  - **Pass by value** - if you pass something into a function, it will be copied
    - For passing references, use pointers (more into that later)
  - **Compiled**
    - Meaning significantly better runtime
  - Sometimes annoying :(

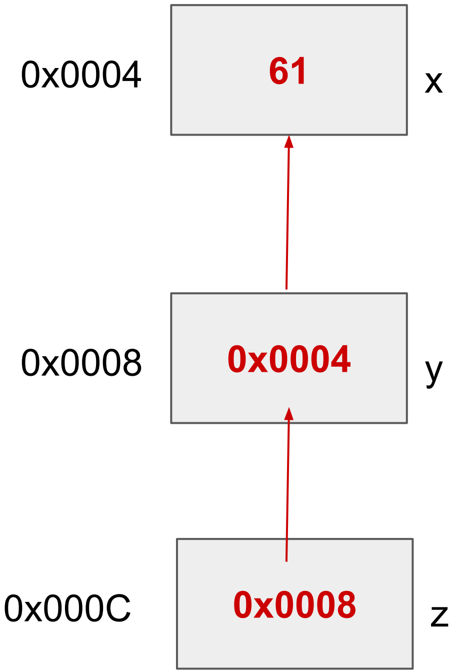
Variable and Pointer in C

- Memory is storage with address labels
- Explain the following expressions in C:
  - **int x;**
    - Variable **x** holds 4 bytes, representing an integer
  - **int* z;**
    - Variable **z** holds 4 bytes, representing a memory location
      - **Memory location = address = pointer**
    - This location holds an integer
  - **int a = *z;**
    - z holds a location
    - ***z** gets what's in the location (an integer)
    - Put the integer value in variable a

Pointer Diagram

- **int x = 61;**
- **int *y = &x**
- **int **z = &y**

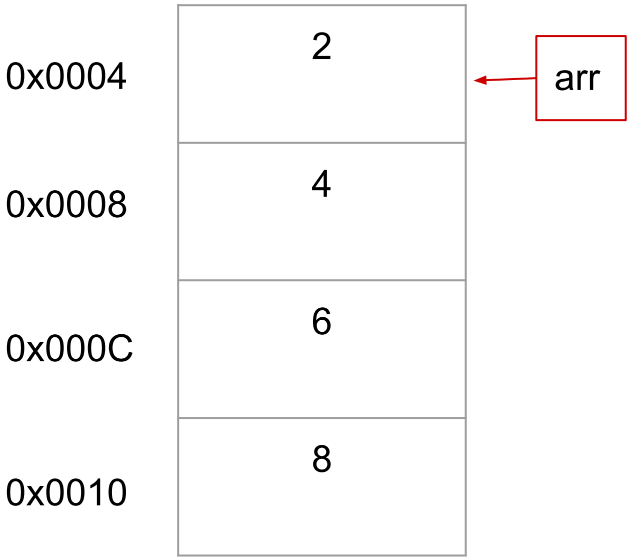
- What are the values of the following?:
  - X is equal to… 61
  - &x is equal to… 0x0004
  - Y is equal to… 0x0004
  - &y is equal to… 0x0008
  - *y is equal to… 61
  - **z is equal to… 61



Arrays and Pointer Math

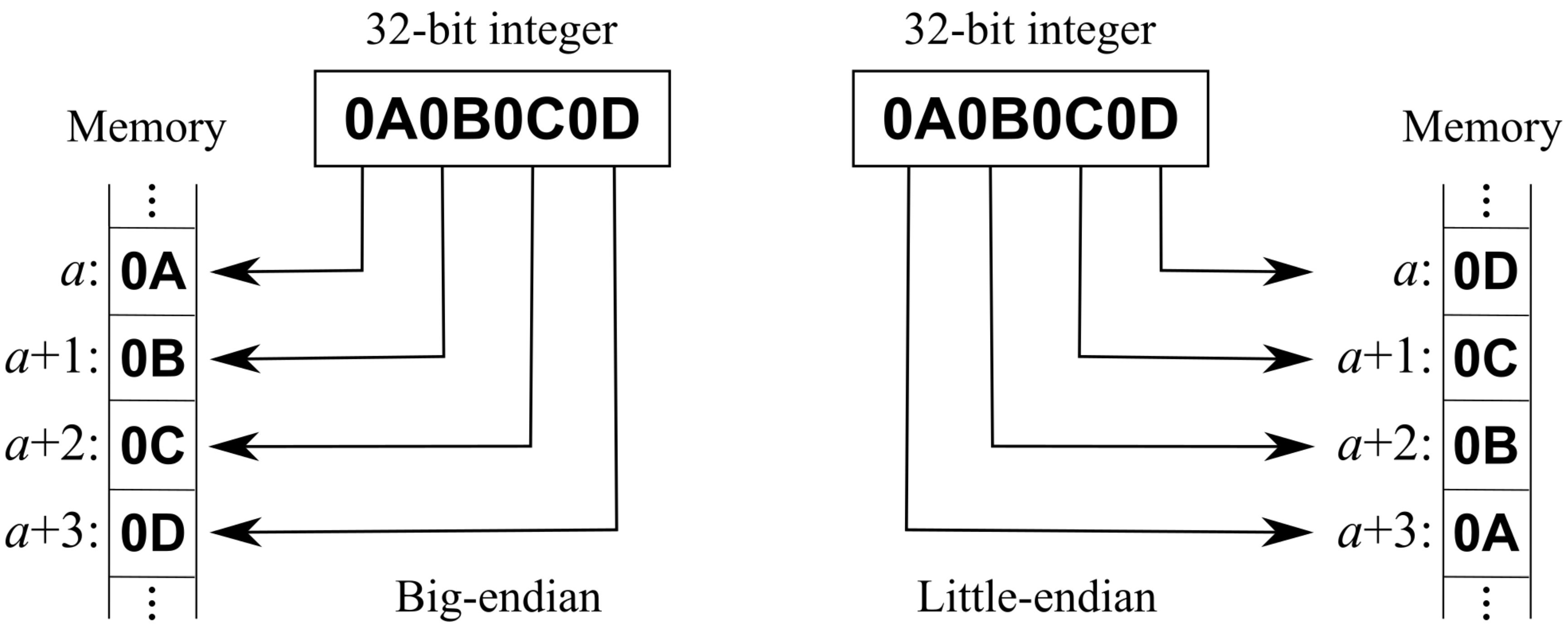
Each int = 4 bytes

- Arrays are pointers
  - Where do we point? The first element in the array
- int arr[4] = {2, 4, 6, 8};
- Index using brackets, a[1] == 4
- What is the value of the following expressions?
  - **Arr**
    - 0x0004
  - ***arr**
    - 2
  - **Arr + 1**
    - 0x0008
    - Add arr + sizeof(int) ⇒ arr + 4 ⇒ 0x0008
  - ***(arr + 1)**
    - 4



Endianness

Big Endian: write out value towards higher address, the “big end”
Little Endian: write out value towards lower address, the “small end”



Strings

- Strings in C are pointers to 1 or more characters
- The end of a string is given by the null terminator '\0'
  - Be careful when you malloc
- sizeof (char) == 1
- String lengths can be found with strlen (this does NOT include '\0')
  - int strlen (char *str);
- String contents can be moved with strcpy (DOES copy '\0')
  - char * strcpy (char *dest, char *src);
- String contents can be compared with strcmp
  - int strcmp (char *s1, char *s2);

Strings Practice

- char *str = “Hello”;
  - Where is str[0] stored?
  - Stored in **static**

literal “Hello” stored in static,
str is just pointing to it’s address in static

- char b[6] = “Hello”;
  - Where is b[0] stored?
  - Initializing an array stores onto the **stack**

creating space on stack for this variable and then copying chars into variable

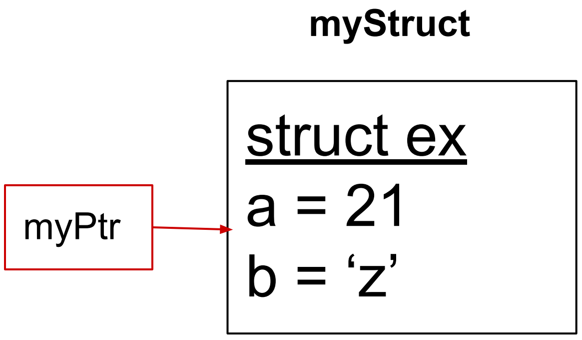
- char *c = malloc (sizeof (char) * 6);
- strcpy (c, “Hello”);
  - Where is c[0] stored?
  - Malloced strings go on the **heap**. Make sure to allocate space for the null terminator!

malloc/calloc/realloc
always => heap

Structs

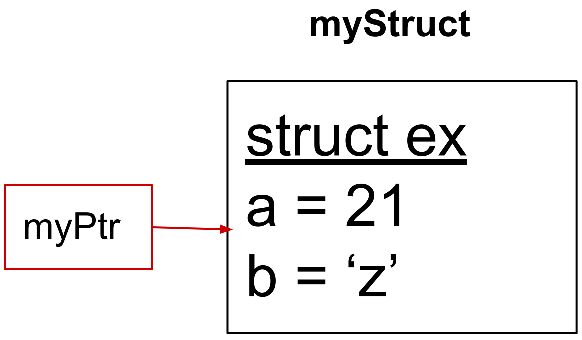
```
struct ex {
        int a;
        char b;
};
```

- Similar to objects in Java, classes in Python
- Struct ex myStruct;
- myStruct.a = 21;
- myStruct.b = ‘z’
- myStruct *myPtr = &myStruct



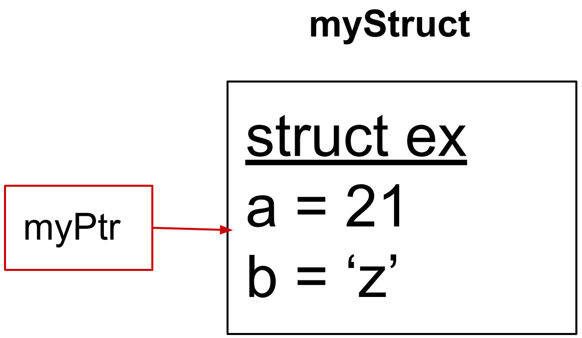
Careful with Functions!

- Assume we are given myStruct (to the right)
- Say we want a function, **funcA**, that should <u>modify struct.a to be 4</u>
- What is the issue with calling funcA(myStruct)?
    - Passing in the struct directly
    - **C is pass by value**



Careful with Functions!

```
void funcA(struct arg) {
    arg.a = 4;
    return;
}
```



We call funcA(myStruct)!

Careful with Functions!

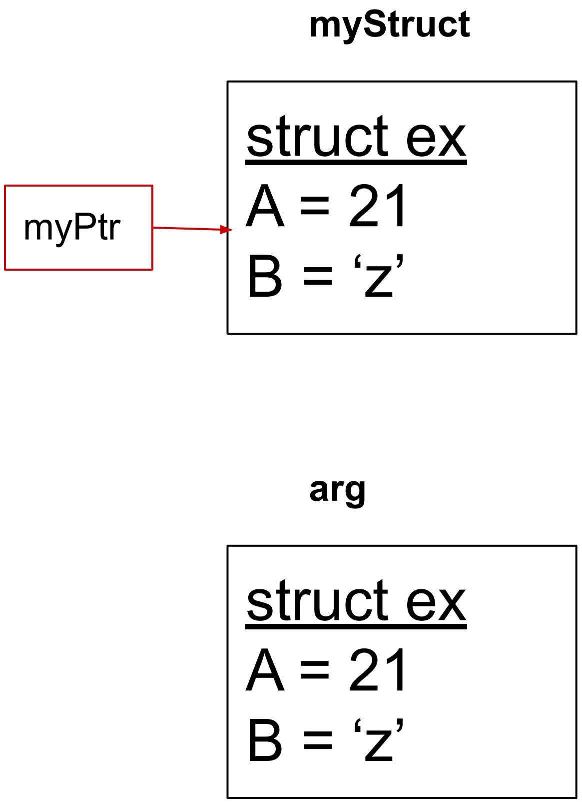
```
void funcA(struct arg) {

        arg.a = 4;

        return;

}
```

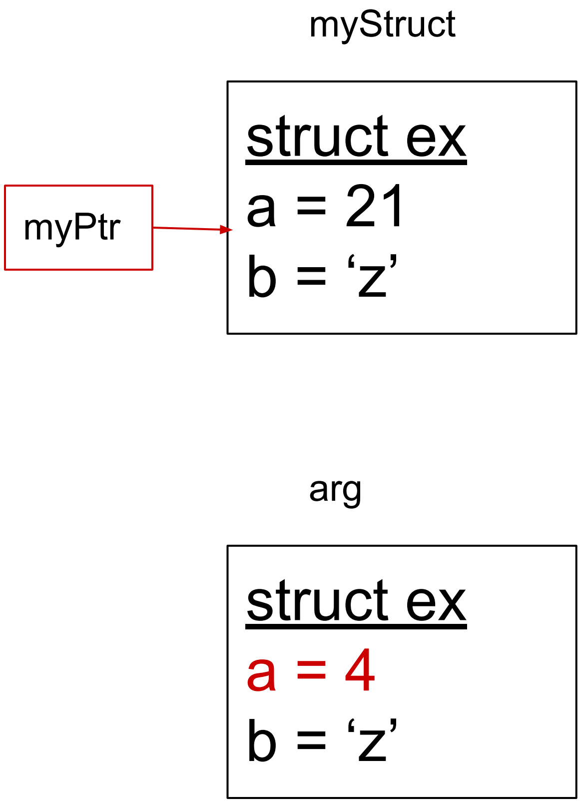
We call funcA(myStruct)!



Careful with Functions!

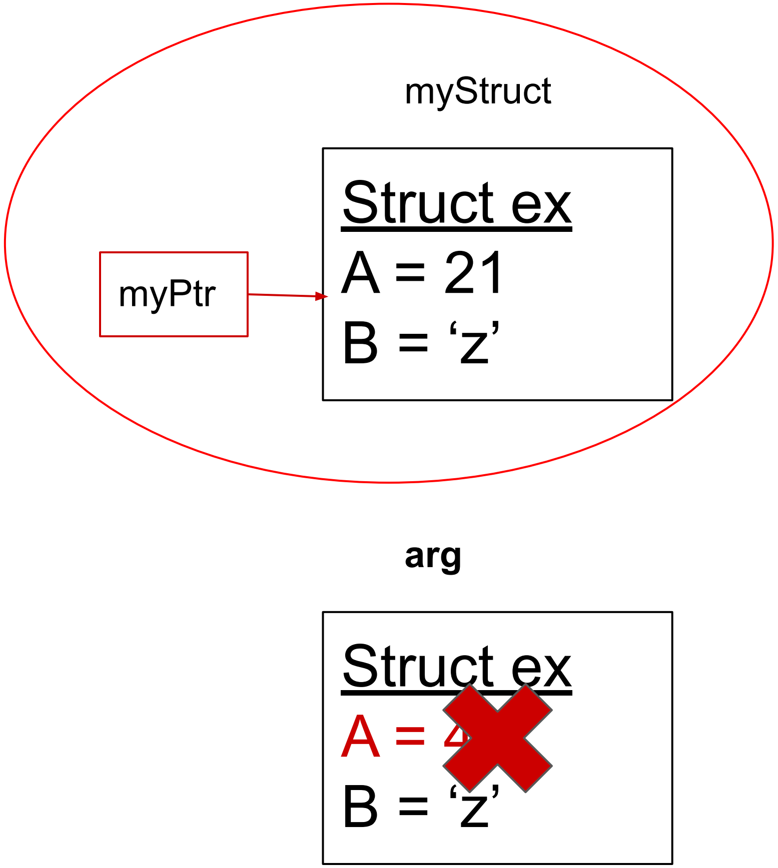
```
void funcA(struct arg) {
    arg.a = 4;
    return;
}
```

We call funcA(myStruct)!



Careful with Functions!

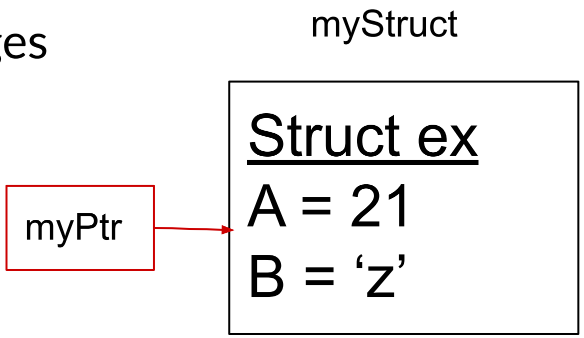
```
void funcA(struct arg) {
        arg.a = 4;
        return;
}
```



<u>The original struct is unchanged!</u>

Careful with Functions!

- Instead, pass a pointer if you want to retain changes




```
Void funcA(Struct *ptr) {
    (*ptr).A = 4      or      ptr->A = 4
    Return;
}
```

- Ptr points to myStruct
- (*ptr) is mStruct
- (*ptr).A accesses the field A
  - Dot operator accesses fields
- (*ptr).A == ptr->A

Memory Allocation

- `malloc(size_t size)` allocates a block of `size` bytes and returns the start of the block. The time it takes to search for a block is generally not dependent on `size`.
- `calloc(size_t count, size_t size)` allocates a block of `count * size` bytes, **sets every value in the block to zero**, then returns the start of the block.
- `realloc(void *ptr, size_t size)` "resizes" a previously-allocated block of memory to `size` bytes, returning the start of the resized block.
- `free(void *ptr)` deallocates a block of memory which starts at `ptr` that was previously allocated by the three previous functions.

C generics

int* pointers only point to **ints**, **char*** pointers only point to **chars/strings**

void * pointers - can point to any type of data

cannot dereference! **why ?**

int *a; int deref_val = *a -> how many bytes do we read? 4

void* b, *b -> how many bytes do we read? no clue

C generics

if we can’t dereference, how do we move data in void pointers around ?

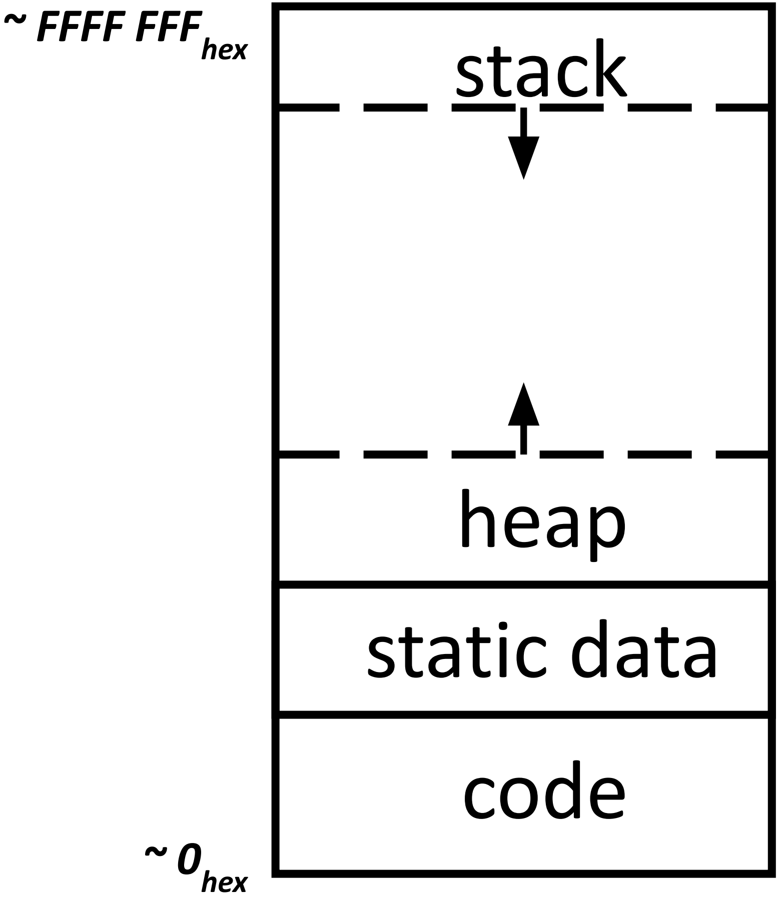
memcpy (void* dest, void* src, size_t num_bytes)

memmove (void* dest, void *src, size_t num_bytes)

move num_bytes bytes from the src pointer into the dest pointer

Memory Structure

- Memory goes into 1 of 4 segments
  - **Code:**
    - Actual program, RISC-V instructions
    - Does not change in size
  - **Static/Data:**
    - Memory for the whole program
    - Does not change in size
  - **Stack:**
    - Dynamic memory
    - Lives with a function call, local vars
    - Grows Down
  - **Heap:**
    - Dynamic memory
    - Expanded with malloc, shrunk with free
    - Grows Up



practice ! [fa23 midterm]

Q2 🍕🍉🍰🥧 *(C)* **(20 points)**

Recall in lab, we implemented some functions for a `vector_t` struct. In this question, we will add support for slicing a `vector_t`.

To do so, we've made some updates to the `vector_t` type from lab. You may assume that all necessary standard libraries are included.

```
typedef struct vector_t {
    // Number of elements in the vector; you may assume size > 0
    size_t size;

    // Pointer to the start of the vector
    int* data;

    // Number of child slices
    size_t num_slices;

    // Array of the vector's child slices, or NULL if num_slices == 0
    struct vector_t** slices;

    // true if the vector is a child slice of another vector, otherwise false
    bool is_slice;
} vector_t;
```

Implement the function `vector_slice`, which should return a slice of a `vector_t` at the given indices, with the following signature:

- `vector_t* v`: A pointer to the parent vector to create the slice from.
- `int start_index`: The beginning index of the new slice's data (inclusive)
- `int end_index`: The ending index of the new slice's data (exclusive). You may assume that `end_index > start_index`.
- Return value: A `vector_t*` representing data as described by `start_index` and `end_index`. A parent `vector_t` shares (portions of) its `data` array with all of its descendant slices.

For example:

```
// vec_a has the elements [0, 1, 2, 3, 4]
vector_t* vec_a = /* omitted */;

// vec_b should be of size 2 and have the elements [1, 2]
vector_t* vec_b = vector_slice(vec_a, 1, 3);

vec_b->data[1] = 10;
// At this point, vec_a should be [0, 1, 10, 3, 4]
//                vec_b should be [1, 10]
```

```
vector_t* vector_slice(vector_t* v, int start_index, int end_index) {

    vector_t* slice = ________________________________________;
                                     Q2.1

    if (slice == NULL) { allocation_failed(); }

    slice->__________________________________________________;
                               Q2.2

    slice->__________________________________________________;
                               Q2.3

    slice->__________________________________________________;
                               Q2.4

    v->slices = ______________________________________________;
                                Q2.5

    if (v->slices == NULL) { allocation_failed(); }

    v->slices[________________________] = ____________________;
                     Q2.6                         Q2.7

    v->_______________________________________________________;
                              Q2.8
    return slice;
}
```

```
vector_t* vector_slice(vector_t* v, int start_index, int end_index) {
    vector_t* slice = calloc(1, sizeof(vector_t));
                                 Q2.1
    if (slice == NULL) { allocation_failed(); }
    slice->size = end_index - start_index;
                      Q2.2
    slice->data = &v->data[start_index];
                     Q2.3
    slice->is_slice = true;
               Q2.4
    v->slices = realloc(v->slices, (v->num_slices+1)*sizeof(vector_t*));
                                      Q2.5
    if (v->slices == NULL) { allocation_failed(); }
    v->slices[v->num_slices] = slice;
                 Q2.6          Q2.7
    v->num_slices += 1;
            Q2.8
    return slice;
}
```

To accommodate these changes to the **vector_t** type, we need to update the **vector_delete** function to properly free our new data structure. When a **vector_t** is deleted, all descendant slices of the **vector_t** should also be freed. You may assume that **vector_delete** is only called on a **vector_t** that is not a slice.

```
void vector_delete(vector_t* v) {
    if (________________________________________________) {
                             Q2.9
        ________________________________________________;
                             Q2.10
    }
    for (int i = 0; i < v->num_slices; i++) {
        ________________________________________________;
                             Q2.11
    }
    if (v->num_slices > 0) {
        ________________________________________________;
                             Q2.12
    }
    ____________________________________________________;
                           Q2.13
}
```

```
void vector_delete(vector_t* v) {
    if (!v->is_slice) {
            Q2.9
        free(v->data);
             Q2.10
    }
    for (int i = 0; i < v->num_slices; i++) {
        vector_delete(v->slices[i]);
                    Q2.11
    }
    if (v->num_slices > 0) {
        free(v->slices);
              Q2.12
    }
    free(v);
     Q2.13
}
```

Practice

Spring 2019, Q6A

What lines are incorrect, and what would you change?

Problem 6 *C Reading* **(16 points)**

The function `parse_message` takes two inputs: an array of strings, and the length of the array. It copies the strings from the input array into a new buffer, ending the buffer with a NULL ptr rather than specifying a size. However if any of the strings are the string "STOP", then it terminates early and returns only strings before the stop message, again ending with a NULL terminator.

(a) The function below contains *at most 5 bugs* which cause the function to nondeterministically exhibit incorrect behavior. Bubble in the lines of code that may produce errors. **You may select more than one line.**

You may assume all calls to malloc succeed, `arr` and its contents are never NULL, `arr` always has at least `size` allocated, and we are using C99.

```
☐ 1.  char** parse_message (char** arr, size_t size) {
☐ 2.      int init_size = 8;
☐ 3.      char **output = malloc (sizeof (char *) * init_size);
☐ 4.      int i;
☐ 5.      for (i = 0; i < size; i++) {
☐ 6.          char *pointer = * arr + i;
☐ 7.          if (pointer == "STOP") {
☐ 8.              break;
☐ 9.          } else if (init_size == i - 1) {
☐ 10.             init_size *= 2;
☐ 11.             realloc (output, sizeof (char *) * init_size);
☐ 12.          }
☐ 13.          output[i] = malloc (sizeof (char) * strlen (pointer));
☐ 14.          strcpy (output[i], pointer);
☐ 15.     }
☐ 16.     output[i] = NULL;
☐ 17.     return output;
☐ 18. }
```

Practice

Spring 2019, Q6A

- **6:** missing parentheses around the arr + i
- **7:** incorrect method of comparison
- **9:** we resize too late
- **11:** realloc returns a new pointer and we don't do anything with it :(
- **13:** didn't malloc for null terminator

Problem 6 ***C Reading*** **(16 points)**

The function `parse_message` takes two inputs: an array of strings, and the length of the array. It copies the strings from the input array into a new buffer, ending the buffer with a NULL ptr rather than specifying a size. However if any of the strings are the string "STOP", then it terminates early and returns only strings before the stop message, again ending with a NULL terminator.

(a) The function below contains *at most 5 bugs* which cause the function to nondeterministically exhibit incorrect behavior. Bubble in the lines of code that may produce errors. **You may select more than one line.**

You may assume all calls to malloc succeed, `arr` and its contents are never NULL, `arr` always has at least `size` allocated, and we are using C99.

```
☐ 1.  char** parse_message (char** arr, size_t size) {
☐ 2.     int init_size = 8;
☐ 3.     char **output = malloc (sizeof (char *) * init_size);
☐ 4.     int i;
☐ 5.     for (i = 0; i < size; i++) {
☐ 6.         char *pointer = * arr + i;
☐ 7.         if (pointer == "STOP") {
☐ 8.             break;
☐ 9.         } else if (init_size == i - 1) {
☐ 10.             init_size *= 2;
☐ 11.             realloc (output, sizeof (char *) * init_size);
☐ 12.          }
☐ 13.          output[i] = malloc (sizeof (char) * strlen (pointer));
☐ 14.          strcpy (output[i], pointer);
☐ 15.     }
☐ 16.     output[i] = NULL;
☐ 17.     return output;
☐ 18. }
```

Q2 *#include "library.c"* **(24 points)**

```
#define MAX_BORROWS 25

typedef struct {
  char* book_name;
  bool borrowed;
} Book;

typedef struct {
  char* user_id;
  Book* borrowed_books[MAX_BORROWS];
} User;

typedef struct {
  User* users;
  int users_len;
  Book* books;
  int books_len;
} Library;
```

The city of Eddy B.C wants to build a new library! The `init_users` function receives the following input:

- `Library* lib`: A pointer to an uninitialized `Library` struct. You may assume that memory has already been correctly allocated on the heap for the `Library` struct.
- `char** user_ids`: An array of well-formatted strings of nonzero length except the last element. The last element is `NULL`. You may assume that all strings are allocated on the stack.

The function should make sure the following properties are held:

- `users_len` should be set to the number of strings in `user_ids`.
- Each `User` in `users` should be initialized as follows:
  - The `user_id` of the `i`th `User` in `users` should be set to the `i`th string in `user_ids`.
  - `borrowed_books` should be an array of `NULL`s to indicate that no `Book` has been borrowed.
- Every `User` and its contents must persist through function calls.

(Question 2 continued…)

Useful C function prototypes:

```
void* malloc(size_t size);
void free(void *ptr);
void* calloc(size_t num_elements, size_t size);
void* realloc(void *ptr, size_t size);

size_t strlen(char* s);
char* strcpy(char* dest, char* src);

// memset sets the first num bytes of the block of memory pointed to by ptr
// to the specified value (interpreted as an unsigned char).
void* memset(void* ptr, int value, size_t num);
```

(15 points) Fill in `init_users` so that it matches the described behavior. Assume that all necessary C libraries are included.

```
void init_users(Library* lib, char** user_ids) {
  int i = 0;

  while (______________________________________________) {
                          Q2.1

    lib_______________ = _________________________________;
             Q2.2                       Q2.3

    User* cur_user = __________________________________________;
                                   Q2.4

    cur_user_______________ = ______________________________;
                 Q2.5                      Q2.6

    strcpy(cur_user_______________, ________________________);
                        Q2.7                  Q2.8

    memset(cur_user_________________________, _______________,
                             Q2.9                  Q2.10

                  MAX_BORROWS * ______________________________);
                                             Q2.11
    i++;
  }

  lib____________________________________________ = i - 1;
                         Q2.12
}
```

Solution:

```c
void init_users(Library* lib, char** user_ids) {
  int i = 0;
  while (user_ids[i] != NULL) {
    lib->users = realloc(lib->users, sizeof(User) * (i + 1));
    User* cur_user = &lib->users[i];
    cur_user->user_id = malloc((strlen(user_ids[i]) + 1) * sizeof(char));
    strcpy(cur_user->user_id, user_ids[i]);
    memset(cur_user->borrowed_books, 0,
                   MAX_BORROWS * sizeof(Book *));
```



```c
    i++;
  }
  lib->users_len = i - 1;
}
```