CS 107
Midterm Review Session
Saturday, May 4, 2024

Based on slides created by Eduardo Higuera and Christine Cheng

Review Topics

Integers, bits, and bytes

Bitwise operators

Strings

Pointers

Stack and heap

Generics

Bits

Signed vs Unsigned Numbers

Unsigned numbers: are 0 or positive (no negatives)

Signed numbers: can be negative

Most significant bit will tell you if it’s positive or negative (signed)

If you compare a signed with unsigned both numbers are read as unsigned

Switching between the values a and -a (two’s complement)
1) Flip every bit
2) Add 1
Works in both direction

Binary vs Hex

Switching between binary and hex

0b10110011 ------------> 0xB3

0xF7 ------------> 0b11110111

Bitwise Operators

Bit operations: &, |, ^, ~, >>, <<

- AND (&): a **&** b = 1 if both a and b are 1
- OR (|): a **|** b = 1 if either are 1
- XOR (^): a **^** b = 1 if only one of them is 1 (XOR with 1 flips the bit, XOR with 0 keeps it the same)
- NOT (~): **~**a flips every bit in a
- LEFT SHIFT (**<<**): a **<<** n moves every bit to the left by n spaces, fills bottom n bits with 0
- RIGHT SHIFT (**>>**): a **>>** n moves every bit to the right by n spaces
  - If a is **signed**, fills top n bits with the original most significant bit (aka signed bit)
  - If a is **unsigned**, fills top n bits with 0

Common Use Case: Masking

- AND

  If AND with 0, always outputs 0 (turns off)
  If AND with 1, always keeps the other bit

- OR

  If OR with 1, always outputs 1 (turns on)
  If OR with 0, always keeps the other bit

Practice Question 1

Consider the mystery function. The marked line (Line 5) does most of the work of the function.

```
int mystery(unsigned int v) {
        int c;
        for (c = 0; v; c++) {
            v &= v - 1; // Line 5
        }
        return c;
}
```

1a) Identify the change in bit pattern between a non-zero unsigned value number and its numeric predecessor (number - 1).

1b) How does the bit pattern of v change after executing line 5?

1c) In terms of the bit pattern for v, what value is returned by the call mystery(v)? 1d) The following statements appear in a C program running on our myth computers.

```
int x = /* initialization here */
bool result = (x > 0) || (x - 1 < 0);
```

Either argue that result is true for all values of x or give a value of x for which result is false. Is result always Yes / No If Yes, explain: If No, the following initialization of x will make result false:

Practice Question 1 - Solution

A: The least significant 1 bit is now a 0 and any bits further to right are all 1s.

B: The least significant 1 bit is changed to a 0.

C: The count of 1 bits in v

D: No. If x = INT_MIN, result is false.

Practice Question 2

Write a function that takes an unsigned int and returns true if its binary representation contains at least one instance of at least two consecutive zeros.

Examples:

- Input 00110111101111101111111111011111 Return: true
- Input: 11111011110111111110000111111111 Return: true
- Input: 01010101010101010101010101010101 Return: false
- Input: 11111111111111111111111111111111 Return: false

Write a function that uses a loop to each pair of bits to detect a pair of zeros

Practice Question 2 - Practice Midterm 2, Q5

```
bool zeros_detector_loop(unsigned int n) {







}
```

Practice Question 2 - Solution

```
bool zeros_detector_loop(unsigned int n) {
    unsigned int mask = 0x3; // 0b000....00011
    for (int i = 0; i < 31; i++) {
        if (!(n & mask)) return true;
        mask <<= 1;
    }
    return false;
}
```

Questions?

Strings

Strings in C

- Array of chars
- Must have a **null terminator (\0)** at the end
- Each character is an element in the array, including the null terminator
- Multiple ways to declare strings
- Strange things happen if there is no null terminator

```
char str[] = “Hello”

char *str = “Hello”

char str[6];
strcpy(str, “Hello”);
```

index	*0*	*1*	*2*	*3*	*4*	*5*
char	'H'	'e'	'l'	'l'	'o'	'\0'

String Operations (#include <string.h>)

- strcat(dest, src): appends the string src to destination (dest = dest + src)
- strcpy(dest, src): copies the string src to destination (dest = src)
- strlen(str): returns the length of the string str, not counting the null terminator
- strspn(const char *str, const char *accept)
    - Returns the count of initial characters in **str that are in accept**
        - strspn("abcbad", "abc") // returns 5
        - strspn("AaAa", "A") // returns 1
- strcspn(const char* str, const char* reject)
    - Opposite of strspn, but same idea
- strcmp(const char* str1, const char* str2)
    - Compares two strings, returns 0 if they are the same.
        - strcmp("hi", "hi") //returns 0
        - strcmp("hi", "bye") //doesn't return 0
    - Returns < 0 if str1 should come before str2, and > 0 if str2 should come before str1

Pointers

In C pass by value

When passing parameters, C always passes a **copy** of whatever parameter is passed

To change a value and have its changes persist outside of a function, we must pass a **pointer to the value** (To change an int, pass an int*)

Pointers

A pointer is a variable storing an 8 byte memory address and is denoted with a *

- A **char *** is a variable storing an 8 byte memory address that points to a character, which can be part of a string or a standalone char
- You can add or subtract numbers from a pointer to change its location with pointer arithmetic

Every variable in C has a **memory address, name,** and **value**

```
int x = 120;
```

i. x is the **name** of the variable
ii. 120 is the **value**
iii. The **memory address** is an 8 byte location in memory that you can get via **&x**

Why Pointers?

- Pointing to a location in memory
- Pointing to a *series* of locations in memory (*arrays*)
- Representing data types (*char*'s representing strings)*
- Passing in modifiable references to an object (*pointers to variables and double-pointers*)
- Dynamically-allocated memory (*pointers returned from malloc / free*)

Arrays and Pointers

When passing arrays as parameters, the array is automatically converted to a pointer to the first element

Arrays of pointers are arrays of 8 byte values; whatever they point to is not stored in the array itself. e.g. an array of strings stores pointers to their first characters, not the characters themselves

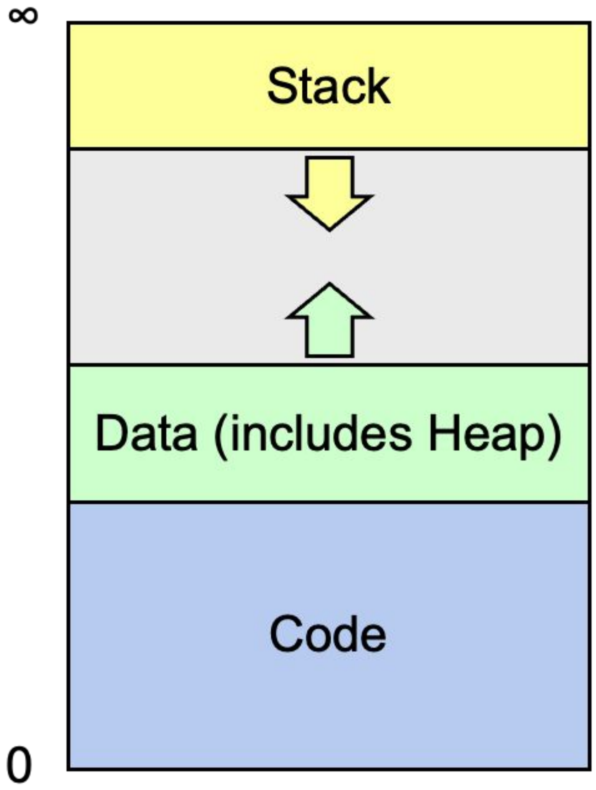
Pointer Arithmetic

Pointer arithmetic works in units of the size of the **pointee type**.

Ex: +1 for an int * means advance one int, so 4 bytes

```
int arr[] = {1, 2, 3, 4, 5};
int * y = arr;
y = y + 2          // *y is now 3
```

Memory



The Stack

Stores local variables and parameters

Each function call has its own stack frame, which is created when the call starts, and goes away when the function ends

The stack grows downwards and shrinks upwards

The Heap

We manually allocate this memory

Beneficial if we want to store data that persists beyond a single function call

Must clean up the memory when we are done

- C no longer knows when we are done with it, as it does with stack memory.
- Check up the function called free(...)

Malloc/calloc/etc. to request memory on the heap

- Request number of bytes
- Get void* to newly-allocated memory

Memory is resizable with realloc

Stack vs Heap

Stack

- Easy cleanup -- we don't have to worry about it
- Fast allocation -- no malloc, etc.
- Type safety -- compiler doesn't know about data allocated dynamically on the heap so it can't help you :(

Heap

- Large size
- Ability to resize allocations -- array size fixed in stack
- Persistence beyond function's stack frame

Memory Allocation

void *malloc(size_t size);

- Allocates size bytes of uninitialized storage.

void* calloc(size_t num, size_t size);

- Allocates memory for an array of num objects of size and initializes all bytes in the allocated storage to zero.

void *realloc(void *ptr, size_t new_size);

- Reallocates the given area of memory. It must be previously allocated by malloc(), calloc() or realloc() and not yet freed with a call to free or realloc. Otherwise, the results are undefined.
- The return value can be the same as ptr but it's not guaranteed

void free(void* ptr);

- Deallocates the space previously allocated by malloc(), calloc() or realloc().

Practice Question 2a - Extra Practice Problems, Q3

Consider the following code, compiled using the compiler and settings we have been using for this class.

```
char *str = "Stanford University";
char a = str[1];
char b = *(char*)((int*)str + 3);
char c = str[sizeof(void*)];
```

What are the char values of variables a, b, and c? (as an example, a = ‘t’) Write “ERROR” if the line of code declaring the variable won’t compile.

Practice Question 2a - Solution

Consider the following code, compiled using the compiler and settings we have been using for this class.

```
char *str = "Stanford University";
char a = str[1];
char b = *(char*)((int*)str + 3);
char c = str[sizeof(void*)];
```

What are the char values of variables a, b, and c? (as an example, a = ‘t’) Write “ERROR” if the line of code declaring the variable won’t compile.

b is 'v'

c = ' ' (space)

Practice Question 2b - Extra Practice Problems, Q3

The code to the right has three buggy lines of code in it. The three buggy parts of the code are noted in bold. Next to each buggy line, write a new line of code that fixes the bug. You may have an idea for restructuring the program that would also fix the bugs, but you must only write code to replace the lines shown in bold—one line of replacement code per one line of buggy code.

The purpose of this function is to take an array of strings (always size 3) and return a heap-allocated array of size 2, where the first entry is the concatenation of the first two strings in the input array, and the second entry is a copy of the third string in the input array. The two strings in the returned array are <u>both newly allocated on the heap</u>. The input is not modified in any way. You may assume that the input is always valid: the array size is always 3, none of the array entries is NULL, and all strings are valid strings.

```
char **pair_strings(char **three_strings) {
        char *return_array[2];
        ____________________;
        size_t str0len = strlen(three_strings[0]);
        return_array[0] = malloc(str0len +
strlen(three_strings[1]));
        ______________________;
        strcpy(return_array[0], three_strings[0]);
        for (size_t i = 0; i < strlen(three_strings[1]); i++) {
        for (___________________________________________) {
        return_array[0][str0len + i] = three_strings[1][i];
    }
    return_array[1] = strdup(three_strings[2]);
    return return_array;
}
```

Practice Question 2b - Extra Practice Problems, Q3

The code to the right has three buggy lines of code in it. The three buggy parts of the code are noted in bold. Next to each buggy line, write a new line of code that fixes the bug. You may have an idea for restructuring the program that would also fix the bugs, but you must only write code to replace the lines shown in bold—one line of replacement code per one line of buggy code.

The purpose of this function is to take an array of strings (always size 3) and return a heap-allocated array of size 2, where the first entry is the concatenation of the first two strings in the input array, and the second entry is a copy of the third string in the input array. The two strings in the returned array are both newly allocated on the heap. The input is not modified in any way. You may assume that the input is always valid: the array size is always 3, none of the array entries is NULL, and all strings are valid strings.

```
char **pair_strings(char **three_strings) {
        char *return_array[2];
        char **return_array = malloc(2 * sizeof(char*));
        size_t str0len = strlen(three_strings[0]);
        return_array[0] = malloc(str0len + 
strlen(three_strings[1]));
        return_array[0] = malloc(str0len + 
strlen(three_strings[1]) + 1);
        strcpy(return_array[0], three_strings[0]);
        for (size_t i = 0; i < strlen(three_strings[1]); i++) {
        // or i < strlen(three_strings[1]) + 1
        for (size_t i = 0; i <= strlen(three_strings[1]); i++) {
        return_array[0][str0len + i] = three_strings[1][i];
    }
    return_array[1] = strdup(three_strings[2]);
    return return_array;
}
```

Practice Question 3 - Extra Midterm Practice, Q2

Problem 3: Memory Diagram (13pts)

For this problem, you will draw a memory diagram of the state of memory (like those shown in lecture) as it would exist at the end of the execution of this code:

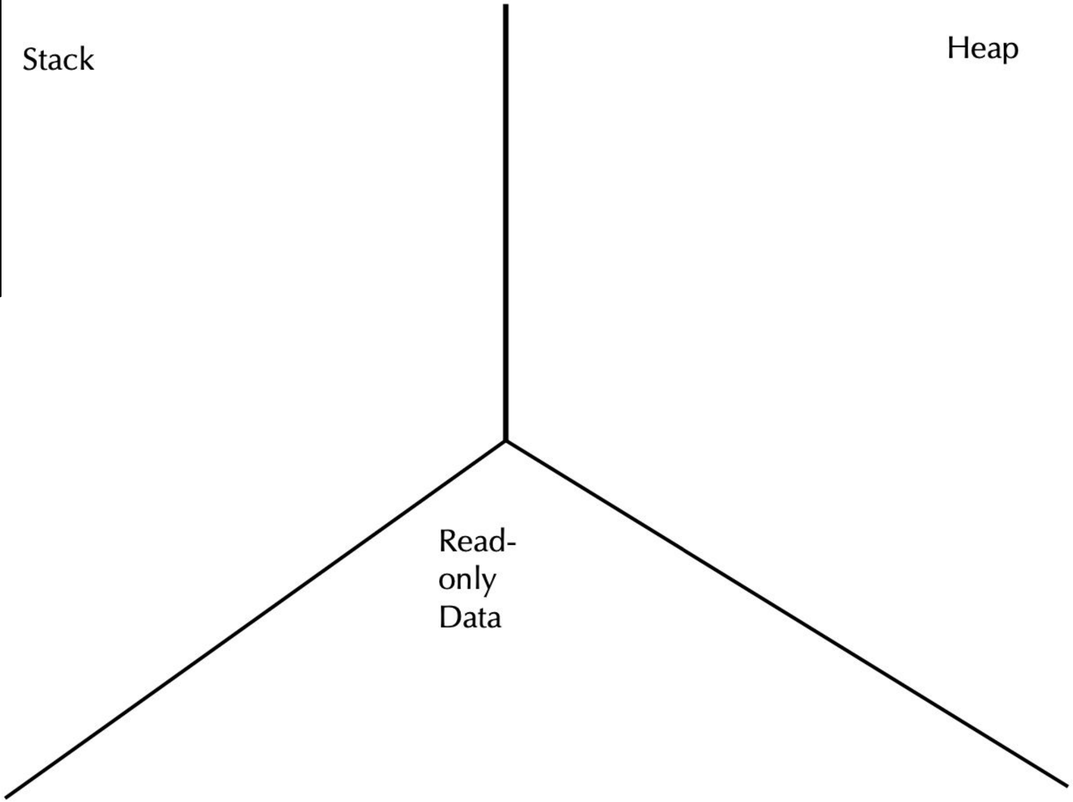
```
char *aaron = "burr, sir";
int *the_other = malloc(12);
the_other[0] = 51;
char *eliza[2];
*eliza = strdup("satisfied");
*(int *)((char *)the_other + 4) = 85;
aaron++;
eliza[1] = aaron + 2;
```

Instructions:

- Place each item in the appropriate segment of memory (stack, heap, read-only data).
- Please write array index labels (0, 1, 2, ...) next to each box of an array, in addition to any applicable variable name label. (With the array index labels, it doesn't matter if you draw your array with increasing index going up or down--or sideways for that matter.)
- Draw strings as arrays (series of boxes), with individual box values filled in appropriately and array index labels as described above.
- Take care to have pointers clearly pointing to the correct part of an array.
- Leave boxes of uninitialized memory blank.
- NULL pointer is drawn as a slash through the box, and null character is drawn as '\0'.

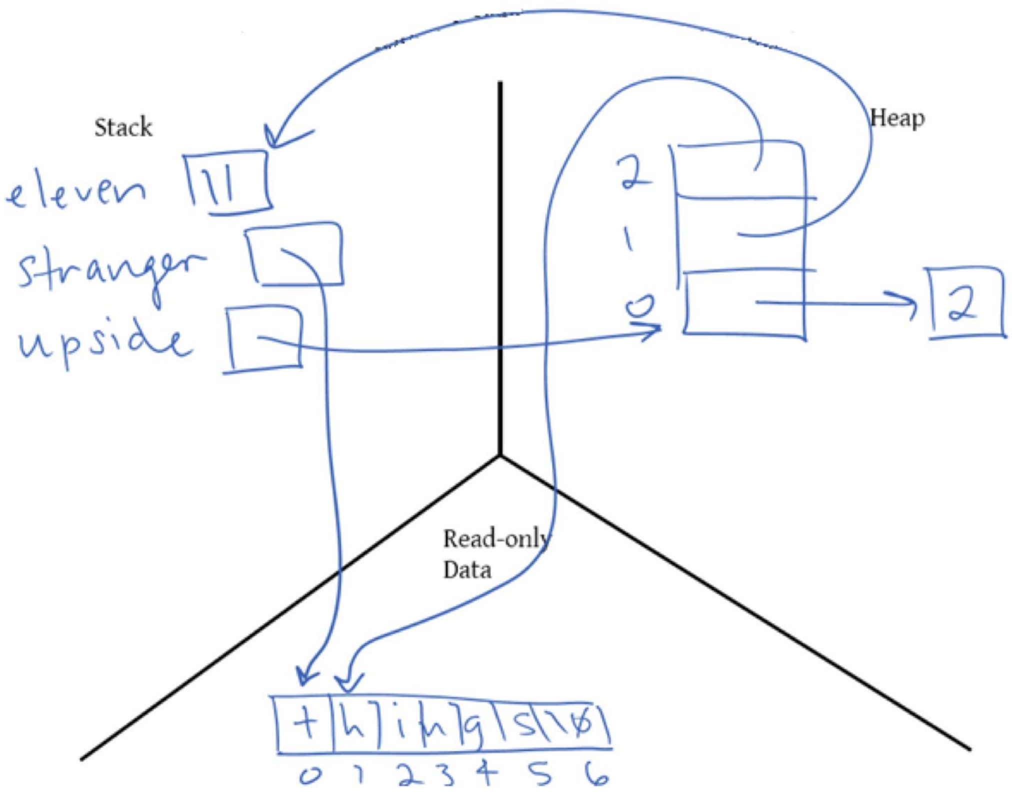
Practice Question 3 - Extra Midterm Practice, Q2

```c
int eleven = 11;
char *stranger = "things";
int **upside = malloc(3 * sizeof(int*));
upside[0] = malloc(4);
*upside[0] = 2;
upside[1] = &eleven;
upside[2] = (int*) ((char*)stranger + 1);
```



Practice Question 3 - Solution

```
int eleven = 11;
char *stranger = "things";
int **upside = malloc(3 * sizeof(int*));
upside[0] = malloc(4);
*upside[0] = 2;
upside[1] = &eleven;
upside[2] = (int*) ((char*)stranger + 1);
```



Questions?

Generics

We can use void * to represent a **generic** pointer to "something"

void * loses information about data types - there is less error checking and it is more prone to mistakes

memcpy/memmove are functions that let us copy around arbitrary bytes from one location to another

- memcpy does **not** support overlapping src/destination memory regions
- memmove does

We can use pointer arithmetic plus casting to **char *** to do pointer arithmetic with a void * to advance it by a **specific number of bytes**

Comparison Functions

Comparison functions have the following prototype:

```
int my_compare(const void *a, const void *b);
```

Comparison functions work using <u>pointers to what you're comparing!</u>

Generic Comparison Function Formula

Generic Comparison Function Formula :

1. **Cast the void*** argument and set a pointer of **known pointee type** equal to it.
2. **Dereference** the typed pointer to access the value. (Steps 1 and 2 are often combined to cast and dereference in one expression.)
3. **Compare values** to determine the result to return.

Reminder!

Comparison functions return:

< 0 if a comes before b

$= 0$ if a and b are equal

> 0 if a comes after b

Example from Lab

What would a callback comparison function that can be passed to qsort look like if we wanted to arrange an array of ints in order of increasing absolute value?

(hint - the builtin C abs() function can calculate absolute value)

```
int abs_fn(const void *a, const void *b) {
    return abs(*(int *)a) - abs(*(int *)b);
}
```

Function Pointers

We can pass functions as parameters to other functions by specifying one or more function pointer parameters

A function pointer lets us pass logic that the caller has access to to the callee

- For instance, we pass a function to bubble_sort that knows how to compare two elements of the type we are sorting.

Functions with generic operations **must always deal with pointers to the data they care about**

- For instance, comparison functions must cast and dereference their received elements before comparing them.

Function Pointers: HowTo

return_type (*function_name) (arg1, arg2)

Ex: int (*my_compare) (void* a, void* b)

Comparison Function Pointer

```
int process_cmp(int (*compare_ints)(const void *a, const void *b)) {
        int a = 1;
        int b = 2;
 return compare_ints(&a, &b);
}
```

Remember, we must always **pass a pointer** to whatever we want to compare!

Practice Question 4 - Practice Midterm 3, Q3

3a) The generic find_min searches an array for its smallest element according to a client-supplied callback function. The function arguments are the array base address, the count of elements, the size of each element in bytes and a comparison function. The function returns a pointer to the minimum array element. As an example, **find_min** on the array **{3.7, 9.4, 1.1, -6.2}** with ordinary float comparison returns a pointer to the last element in the array. Fill in each of the three blank lines with the necessary expression so that the function works correctly.

```
void *find_min(void *base, size_t nelems, size_t width,
     int (*cmp)(const void *, const void *)) {

     assert(nelems > 0);   // error if called on empty array
     void *min = ______________________________;   // Line 1
     for (size_t i = 1; i < nelems; i++) {
          void *ith = ____________________________; // Line 2
          if (_________________________________) { // Line 3
               min = ith;
          }
     }
     return min;
}
```

Practice Question 4 - Solution

3a) The generic find_min searches an array for its smallest element according to a client-supplied callback function. The function arguments are the array base address, the count of elements, the size of each element in bytes and a comparison function. The function returns a pointer to the minimum array element. As an example, **find_min** on the array **{3.7, 9.4, 1.1, -6.2}** with ordinary float comparison returns a pointer to the last element in the array. Fill in each of the three blank lines with the necessary expression so that the function works correctly.

```
void *find_min(void *base, size_t nelems, size_t width,
     int (*cmp)(const void *, const void *)) {

     assert(nelems > 0);   // error if called on empty array
     void *min = base;   // Line 1
     for (size_t i = 1; i < nelems; i++) {
          void *ith = __________________________________________; // Line 2
          if (_________________________________________________) { // Line 3
               min = ith;
          }
     }
     return min;
}
```

Practice Question 4 - Solution

3a) The generic find_min searches an array for its smallest element according to a client-supplied callback function. The function arguments are the array base address, the count of elements, the size of each element in bytes and a comparison function. The function returns a pointer to the minimum array element. As an example, **find_min** on the array **{3.7, 9.4, 1.1, -6.2}** with ordinary float comparison returns a pointer to the last element in the array. Fill in each of the three blank lines with the necessary expression so that the function works correctly.

```
void *find_min(void *base, size_t nelems, size_t width,
     int (*cmp)(const void *, const void *)) {

     assert(nelems > 0);    // error if called on empty array
     void *min = base;    // Line 1
     for (size_t i = 1; i < nelems; i++) {
          void *ith = (char *)base + i * width; // Line 2
          if (_________________________________) { // Line 3
               min = ith;
          }
     }
     return min;
}
```

Practice Question 4 - Solution

3a) The generic find_min searches an array for its smallest element according to a client-supplied callback function. The function arguments are the array base address, the count of elements, the size of each element in bytes and a comparison function. The function returns a pointer to the minimum array element. As an example, **find_min** on the array **{3.7, 9.4, 1.1, -6.2}** with ordinary float comparison returns a pointer to the last element in the array. Fill in each of the three blank lines with the necessary expression so that the function works correctly.

```
void *find_min(void *base, size_t nelems, size_t width,
     int (*cmp)(const void *, const void *)) {

     assert(nelems > 0);   // error if called on empty array
     void *min = base;   // Line 1
     for (size_t i = 1; i < nelems; i++) {
          void *ith = (char *)base + i * width; // Line 2
          if (cmp(ith, min) < 0) { // Line 3
               min = ith;
          }
     }
     return min;
}
```

Practice Question 4 - Practice Midterm 3, Q3

3b) Complete the program started below to use find_min to find the command-line argument with the minimum first character ("minimum" means smallest ASCII value) and print that character. For example, if invoked as ./program red green blue, the program prints 'b'. You must fill in the blank line in main with a call to find_min and can assume that this function works correctly.   You will also need to implement the comparison callback function. Hint: remember that the command-line arguments start at index 1 in the argv array.

```
int cmp_first(const void *p, const void *q) {
    // TODO
}

int main(int argc, char *argv[]) {
    char ch = ________________________________________________________;
    printf("Min first char of my arguments is %c\n", ch);
    return 0;
}
```

Practice Question 4 - Solution

3b) Complete the program started below to use find_min to find the command-line argument with the minimum first character ("minimum" means smallest ASCII value) and print that character. For example, if invoked as ./program red green blue, the program prints 'b'. You must fill in the blank line in main with a call to find_min and can assume that this function works correctly. You will also need to implement the comparison callback function. Hint: remember that the command-line arguments start at index 1 in the argv array.

```
int cmp_first(const void *p, const void *q) {
    // TODO
}

int main(int argc, char *argv[]) {
    char ch = **(char **)find_min(argv + 1, argc - 1, sizeof(*argv), cmp_first);
    printf("Min first char of my arguments is %c\n", ch);
    return 0;
}
```

Practice Question 4 - Solution

3b) Complete the program started below to use find_min to find the command-line argument with the minimum first character ("minimum" means smallest ASCII value) and print that character. For example, if invoked as ./program red green blue, the program prints 'b'. You must fill in the blank line in main with a call to find_min and can assume that this function works correctly. You will also need to implement the comparison callback function. Hint: remember that the command-line arguments start at index 1 in the argv array.

```
int cmp_first(const void *p, const void *q) {
    return **(const char **)p - **(const char **)q;
}

int main(int argc, char *argv[]) {
    char ch = **(char **)find_min(argv + 1, argc - 1, sizeof(*argv), cmp_first);
    printf("Min first char of my arguments is %c\n", ch);
    return 0;
}
```

Practice Question 4 - Practice Midterm 3, Q3

3b) Complete the program started below to use find_min to find the command-line argument with the minimum first character ("minimum" means smallest ASCII value) and print that character. For example, if invoked as ./program red green blue, the program prints 'b'. You must call to find_min and can assume that this function works correctly.   You will also need to implement the comparison callback function. Hint: remember that the command-line arguments start at index 1 in the argv array.

```
int cmp_first(const void *p, const void *q) {
    // TODO
}

int main(int argc, char *argv[]) {
    // TODO; get char and store in ch

    printf("Min first char of my arguments is %c\n", ch);
    return 0;
}
```

Practice Question 4 - Solution

3b) Complete the program started below to use find_min to find the command-line argument with the minimum first character ("minimum" means smallest ASCII value) and print that character. For example, if invoked as ./program red green blue, the program prints 'b'. You must call to find_min and can assume that this function works correctly.   You will also need to implement the comparison callback function. Hint: remember that the command-line arguments start at index 1 in the argv array.

```
int cmp_first(const void *p, const void *q) {
    // TODO
}

int main(int argc, char *argv[]) {
    char *result = *(char **)find_min(argv + 1, argc - 1, sizeof(*argv), cmp_first);
    ch = result[0];
    printf("Min first char of my arguments is %c\n", ch);
    return 0;
}
```

Practice Question 4 - Solution

3b) Complete the program started below to use find_min to find the command-line argument with the minimum first character ("minimum" means smallest ASCII value) and print that character. For example, if invoked as ./program red green blue, the program prints 'b'. You must call to find_min and can assume that this function works correctly.   You will also need to implement the comparison callback function. Hint: remember that the command-line arguments start at index 1 in the argv array.

```
int cmp_first(const void *p, const void *q) {
     return **(const char **)p - **(const char **)q;
}

int main(int argc, char *argv[]) {
     char *result = *(char **)find_min(argv + 1, argc - 1, sizeof(*argv), cmp_first);
     ch = result[0];
     printf("Min first char of my arguments is %c\n", ch);
     return 0;
}
```

Practice Question 4 - Practice Midterm 3, Q3

3c) The selection sort algorithm works by repeatedly selecting a minimum element and swapping it into position. On the first iteration, it finds the minimum array element and swaps it with the first element. The second iteration finds the minimum element of the subarray starting at the second position and swaps it into the second position. This process repeats on shorter and shorter subarrays until the entire array is sorted. Implement the selection_sort function below to perform the selection sort algorithm on a generic array. You will need to call find_min and can assume that the function works correctly.

```
void selection_sort(void *base, size_t nelems, size_t width,
   int (*cmp)(const void *, const void *)) {

     for (size_t i = 0; i < nelems - 1; i++) {
```

Practice Question 4 - Solution

```
void selection_sort(void *base, size_t nelems, size_t width,
 int (*cmp)(const void *, const void *)) {

    for (size_t i = 0; i < nelems - 1; i++) {
        void *ith = (char *)base + i * width;
        void *min = find_min(ith, nelems - i, width, cmp);
        char tmp[width];
        memcpy(tmp, ith, width);
        memcpy(ith, min, width);
        memcpy(min, tmp, width);
    }
}
```

Questions?

Good Luck Tuesday :)