This document contains the questions to the midterm given in Fall 2017. The class was taught by Julie Zelenski & Chris Gregg. This was an 80-minute exam.

Midterm questions

Problem 1: Bits, bytes, and numbers

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 true? Yes / No

If Yes, explain:

If No, the following initialization of `x` will make `result` false:

Problem 2: C-strings

2a) The function `strip_leading(char *input, const char *discard)` removes the leading characters from the string `input` that occur in the string `discard`. Here are some examples:

```
char *word = strdup("details");

strip_leading(word, "s")          no change to word
strip_leading(word, "due")        changes word to "tails"
strip_leading(word, word)         changes word to ""
```

Requirements:

- Your function should not allocate, deallocate, or resize any memory. Instead it should destructively modify the input string. If the input string loses some characters, its memory will be over-allocated at the end; your implementation should leave the memory as-is.
- Re-implementing functionality that is available in the standard library will result in loss of credit. *Hint*: your code should not have any explicit loops, instead call the library functions!
- The string-copying routines (e.g. **`strcpy/strcat`**) cannot be used when the source and destination overlap. *Hint*: remember what alternatives you have in the standard library!

```
void strip_leading(char *input, const char *discard) {
```

2b) Your colleague wants to add this final line to **`strip_leading`** to fix the over-allocation issue:

```
input = realloc(input, strlen(input) + 1);
```

Not only is this call not likely to shrink the memory, it causes two distinct memory errors. What are they?

Problem 3: Pointers and generics

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;
}
```

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) {

}

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

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++) {
```