CS107 Midterm Exam

Question Booklet

CS107 Fall 2019 – Instructor: Nick Troccoli

You may not use any internet devices. You will be graded on functionality – but good style saves time and helps graders understand what you were attempting. You have 110 minutes. We hope this exam is an exciting journey.

Note: DO NOT WRITE in this booklet. Only work in the answer booklet will be graded.

1) Short Answer **15 Points/110 Total**

IMPORTANT NOTES for all questions on the exam: For coding questions, the majority of the points are typically focused on the correctness of the code. However, there may be deductions for code that is roundabout/awkward/inefficient when more appropriate alternatives exist. For any coding questions, your answers should compile cleanly and not have any memory leaks or errors. You may need to scroll vertically or horizontally to fully view blocks of code. Solutions that violate any specified restrictions may get partial credit. For these problems, you do not need to worry about calling `assert` to check for heap errors.

Answer the following short answer questions below.

A) What is the decimal number -84 in 8-bit signed (two's complement)?

B) Give an example of an issue that would cause a memory error and give an example of an issue that would cause a memory leak. You are not required to include code in your answer. Please limit your description of each example to no more than 2 sentences.

C) One of your coworkers is working on a program using heap memory. Over the course of the program, they allocate an array of integers on the heap, resize that array to a larger size, and free the memory at the end when they are done with it. They have provided you an excerpt of the code with some comments - the ellipses (...) indicate code that was omitted. There are two core memory issues in the provided code; briefly identify each issue and how to correct it, using no more than 3 sentences per issue.

```
// Allocate the initial heap array
int num_elems = 5;
int *arr = malloc(sizeof(int) * num_elems);
assert(arr);

...

// Resize it to add space for 2 more elements
int *new_arr = realloc(arr, num_elems + 2);
assert(new_arr);

...

// Free memory to avoid leaks
free(arr);
free(new_arr);
```

2) C Strings **30 Points/110 Total**

Implement the function `remove_delimiters` that takes in a string `str` and a collection of delimiter characters `delimiters` and returns a new heap-allocated string that contains the same characters as `str` but with all characters removed that are contained in `delimiters`. It is the caller's responsibility to free this returned string.

```
char *remove_delimiters(const char *str,
                        const char *delimiters);
```

Here is an example usage of the function:

```
// prints "The  quick  brwn"
char *str = "The - quick - brown.";
char *no_delims = remove_delimiters(str, "o-.");
printf("%s\n", no_delims);
free(no_delims);
```

The core approach your function should take is to call the `scan_token` function from assign2 (assume it is implemented for you) repeatedly in a loop on the provided string and concatenate the tokens onto a growing heap-allocated string. In this way, you will build up a copy of the string that omits any delimiters. As a reminder, `scan_token` has the following signature:

```
bool scan_token(const char **p_input, const char *delimiters,
                char buf[], size_t buflen);
```

`scan_token` scans the input string pointed to by `p_input` to determine the extent of the next token, using the delimiters as separators, and then writes the token characters to `buf`, which is assumed to be `buflen` bytes large, terminated with a null char. The function returns true if a token was written to `buf`, and false otherwise. `scan_token` updates the string pointed to by `p_input` to point to the next character in the input that follows what was just scanned. If a token does not fit in `buf`, the function writes `buflen - 1` characters into `buf`, writes a null terminator in the last slot, and the pointer held by `p_input` is updated to point to the next character following the `buflen - 1` characters in the token.

Your implementation must follow these steps:

1. create a heap-allocated string of size `INITIAL_SIZE` (a provided constant).
2. Create a loop that continually tokenizes `str` via the provided delimiters string `delimiters`. Each time, you should read the next token into a stack array of size `INITIAL_SIZE`. This loop should be the only loop in your implementation.
3. After reading in a token, you should append this token to the heap-allocated string you created previously. If there is not enough space in the heap-allocated string to append this new token, you should first double the size of the heap-allocated string to make more space before appending the token.
4. Once you have read in all tokens in `str`, you should return the heap-allocated string you have built up.

You may assume that the parameters will be non-NULL, though the strings may be empty. You should use built-in string library functions whenever possible.

3) Extract Min **40 Points/110 Total**

Implement the generic function `extract_min` which copies the minimum element in an array, calculated using the provided comparison function, to the specified destination, and removes it from the array.

```
void extract_min(void *base, void *dest, size_t nelems,
                 size_t elem_size_bytes,
                 int (*cmp_fn)(const void *, const void *));
```

As an example, let's say we call `extract_min` with the following int array with 4 elements:

```
[3, 1, 0, 2]
```

If we provide a comparison function that orders the numbers in ascending order, 0 will be copied to the destination we specify, and then it will be removed by shifting the 2 left one place, resulting in the following 3-element array:

```
[3, 1, 2]
```

The function signature and parameters are specified as follows:

```
void extract_min(void *base, void *dest, size_t nelems,
                 size_t elem_size_bytes,
                 int (*cmp_fn)(const void *, const void *));
```

- `base` : a pointer to the first element of an array
- `dest` : a pointer to the location where the minimum element should be copied to. Assumed to point to `elem_size_bytes` bytes of valid memory.
- `nelems` : the number of elements in the provided array. Assumed to be greater than 0.
- `elem_size_bytes` : the size of a single provided array element, in bytes. Assumed to be greater than 0.
- `cmp_fn` : a function pointer that accepts two parameters, both pointers to elements in the array, and returns a negative number if the first parameter is considered less than the second, 0 if the first parameter and the second parameter are considered equal, or a positive number if the first parameter is considered larger than the second.

This function should find the minimum element in the array according to the provided comparison function, copy it to the specified destination, and remove it from the array by shifting over the remaining elements if needed. If there are multiple equivalent minimum elements, you may extract any one. Note that this function does not handle any logic regarding the array length - it is assumed that the caller knows that each call to `extract_min` will decrease the array size by 1.

A) Write the `extract_min` function.

B) Implement the `cmp_strings_asc` function that could be used as a parameter to `extract_min` with an array of strings to extract the *shortest string*. In other words, with an array containing pointers to the strings "Hello", "Hi" and "Howdy", a single call to `extract_min` would copy the pointer to the string "Hi" to the specified destination and remove that pointer from the array.

C) Fill in the 5 blanks to finish the implementation of the `main` function below to print out the contents of the hardcoded array of strings in order of ascending length. When implemented correctly, this code should print out the strings in the following order (1 per line): "C", "Unix", "CS107", "Assembly".

```
int main(int argc, char *argv[]) {
    char *strs[] = {"CS107", "Assembly", "Unix", "C"};
    size_t nelems = sizeof(strs) / sizeof(strs[0]);
    for (int i = 0; i < sizeof(strs) / sizeof(strs[0]); i++) {
        char *min;
        /* use extract_min along with cmp_strings_asc
         * function to extract the minimum element into min.
         */
        extract_min(__1__, __2__, __3__, __4__, __5__);
        printf("%s\n", min);
        nelems--;
    }

    return 0;
}
```

4) Bits and Bytes **25 Points/110 Total**

The `get_bit_range` function takes in an `unsigned long` and indices of two bits and returns a new `unsigned long` containing the bits between those two indices (inclusive) shifted as far right as possible, with all other bits being 0. The function has the following signature:

```
unsigned long get_bit_range(unsigned long bits,
                            int leftmost_index,
                            int rightmost_index);
```

The specified indices are defined as having index 0 be the least significant bit, and index 63 being the most significant bit. Thus, given a 64-bit unsigned long, `get_bit_range` should return a new `unsigned long` with its least-significant bit equal to the bit at index `rightmost_index` in `bits`, its second-least-significant bit equal to the bit at `rightmost_index + 1` in `bits`, etc. for all the bits between `rightmost_index` and `leftmost_index`, inclusive. Here are some example executions of this function:

```
unsigned long x = 0b0....01011101;
unsigned long result_1 = get_bit_range(x, 2, 0);
// result_1 = 0b0....0101;

unsigned long result_2 = get_bit_range(x, 5, 2);
// result_2 = 0b0....0111

unsigned long result_3 = get_bit_range(x, 4, 4);
// result_3 = 0b0...01
```

In the above examples, notice how the bits are shifted as far right as possible in the result. For example, for `result_3`, the index-4 bit in the input becomes the index-0 bit in the output, and all other bits are 0. Similarly, for `result_2`, the index 2-5 bits in the input become the index 0-3 bits in the output.

The following parts have you implement pieces of this `get_bit_range` function using bit operations. You may assume that `leftmost_index` is at least as large as `rightmost_index`, and that both indices are between 0 and 63. The strategy we will use is to first shift all the bits over to put the in-range bits in the correct positions. Then, we will create a mask to zero out the bits outside the specified indices.

```
unsigned long get_bit_range(unsigned long bits,
                            int leftmost_index,
                            int rightmost_index) {
    // shift `bits` right to make the rightmost_index
    // bit index 0, rightmost_index + 1 bit index 1, etc.
    unsigned long shifted = __1__;

    // Create a mask with 1s in the indices of the bits
    // in `shifted` to keep, and 0s in the indices of bits
    // in `shifted` to zero out.
    unsigned long mask = __2__;

    // return result with this mask applied to `shifted`
    // to zero out bits outside of range
    return __3__;
}
```

For example, if `bits` is `0b0....01011101`, `leftmost_index` is 5 and `rightmost_index` is 2, then `shifted` should be `0b0....010111`, `mask` should be `0b0....01111`, and the returned value should be `0b0....0111`.

A) Fill in the 3 blanks to finish the implementation of the `get_bit_range` function above.

B) An alternative approach is to first mask off the bits outside of the specified range in `bits`, and then shift the result to the right into the correct position. In other words, using the same example where `bits` is `0b0....01011101`, `leftmost_index` is 5 and `rightmost_index` is 2, first we would create a mask where bits 2 - 5 are 1, and all other bits are 0; in other words, `0b0....0111100`. We use this to get just the bit values in `bits` we care about, and then we shift them to the right as needed.

It turns out that this mask can be expressed using subtraction with two expressions. Fill in the blanks to create this mask in terms of `leftmost_index` and `rightmost_index`.

```
unsigned long mask = __1__ - __2__;
```