CS107 Midterm Exam

Question Booklet

CS107 Spring 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**

Note: you may need to scroll to fully view blocks of code.

Note 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.

Answer the following short answer questions below.

A) What is the decimal number -107 in 8-bit signed (two's complement)?

B) A coworker is trying to write a `truncate` function that takes in a string and a desired length, and creates a new truncated copy of the provided string that is at most the specified length. For example, you should be able to use the function as follows:

```
...
char *str = "This is a very long string";
char *truncated = truncate(str, 7);
printf("%s", truncated);

// should print out "This is"
...
```

They initially propose to implement the function as follows. There is one core memory issue with this implementation - identify the issue in three sentences or less.

```
char *truncate(char *str, size_t length) {
    // Find the length of the new string
    size_t str_length = strlen(str);
    size_t smaller_length = length < str_length ? length : str_length;

    // Make the new string and copy characters over
    char new_str[smaller_length + 1];
    strncpy(new_str, str, length);  // copy at most n characters
    new_str[smaller_length] = '\0';

    return new_str;
}
```

C) *This part refers to the same `truncate` function as described in part B*. After becoming aware of their mistake, your coworker attempts to re-implement the function by instead taking a third parameter, which is where the new truncated string should be stored. Their new implementation is as follows - however, they notice that when passing in a string destination, it is not changed to store the new string. Identify the cause of this issue in `truncate` in three sentences or less.

```
void truncate(char *str, size_t length, char *dest) {
    // Find the length of the new string
    size_t str_length = strlen(str);
    size_t smaller_length = length < str_length ? length : str_length;

    // Make the new string and copy characters over
    dest = malloc(smaller_length + 1);
    strncpy(dest, str, length);
    dest[smaller_length] = '\0';
}

int main() {
    ...
    char *str = "This is a very long string";
    char *truncated;
    truncate(str, 7, truncated);
    printf("%s", truncated);

    // prints out garbage! :(
    ...
}
```

2) Error-Correcting Codes

25 Points/110 Total

Note: you may need to scroll to fully view blocks of code.

For this question, example bit representations are written with the most-significant bit on the left, and the least-significant bit on the right.

An *error-correcting code* is a way of sending data with redundancies such that it is resilient to small errors that may occur during transmission, for instance wirelessly or over a network where parts of data can be lost or corrupted.

For this problem, let's take one example of such a redundant format when transmitting bits, which is **duplicating each bit an additional two times** and sending the result. In other words, if someone wanted to send us the bit message `010`, they would actually send the redundant message `000111000`. Then, we would parse it in 3-bit chunks to decode it and get the original message `010`.

The utility of this is even if one bit per chunk is flipped during transmission, we can still read the encoded message. Here's how: for each of these chunks, the receiver assumes that the *majority* bit in that chunk is the bit that it encodes. In other words, if the chunk contains a majority (two) of 0s, then it assumes the block encodes a 0. If the chunk contains a majority (two) of 1s, then it assumes the block encodes a 1. For completeness, here are all of the possible 3-bit patterns the receiver could be given, and what bit they are assumed to encode:

Encoding `1`	**Encoding `0`**
`111`	`000`
`110`	`001`
`101`	`010`
`011`	`100`

For example, let's say the receiver receives the following bit message (spaced out for clarity):

```
111 010 001 110 101
```

To decode this message, we read three bits at a time from least-significant to most-significant, and each time we look at the majority bit in that block to know what bit is encoded. In this case,

the above message would be decoded to the following bit pattern (spaced out for clarity):

```
1 0 0 1 1
```

Your task for this problem is to help complete the implementation of the `decode` function that takes in a bit pattern as described above and returns its decoded bit pattern. Each part of the problem has you fill in one blank below with the correct expression. Each part has certain restrictions - violating solutions may get partial credit.

```
long decode(unsigned long message) {
    unsigned long decoded = 0;

    /* loop over each 3-bit chunk from least-significant
     * to most-significant (ignore remainder) of `message`.
     */
    for (int i = 0; i < sizeof(message) * 8 / 3; i++) {

        /* `block` should store in its least-significant
         * 3 bits the ith least-significant block of 3
         * bits from `message`.  All other bits should
         * be zero.
         */
        unsigned long block = ______(A)_______ & 0x7;

        /* If `block` encodes 1, make the i-th
         * least-significant bit of `decoded` a 1.
         */
        if (______(B)_______ != 0) {
            decoded = decoded _____(C)_____;
        }
    }

    return decoded;
}
```

A) For the following line of code from the function above, fill in the blank with an expression so that this line stores the i-th least-significant block of 3 bits from `message` into the 3 least-significant bits of `block`. All other bits in `block` should be 0.

```
unsigned long block = ______(A)_______ & 0x7;
```

For example, if `message` is the bit pattern `0...0 111 110 010` (spaced out for clarity) and `i` is `1`, then after this line is executed, `block` should be the bit pattern `0...0 110`.

B) Assuming the definition of `block` as given in the provided code above, for the following line of code from the function above fill in the blank with an expression so that this loop condition evaluates to `true` if `block` encodes a `1`, or `false` if `block` encodes a `0`. You are not allowed to enumerate or hardcode the explicit possible blocks in your expression, or explicitly check each bit to see if it is a 1 or 0 – you must write one expression that exploits certain patterns.

```
if (______(B)_______ != 0) { ...
```

For example, if `block` is the bit pattern `0...0110`, then the loop condition should evaluate to `true` because `110` is majority `1`s.

Hint: excluding `000`, what types of numbers are represented by the other 3-bit patterns that encode 0?

C) For the following line of code from the function above, fill in the blank with an expression so that this line sets the i-th least-significant bit of `decoded` to be a `1`.

```
decoded = decoded _____(C)_____;
```

For example, if `i` is `2` and `decoded` is a bit pattern of all zeros, then after this line is executed, `decoded` should be the bit pattern `0...0100`.

3) Find and Replace **30 Points/110 Total**

Note: you may need to scroll to fully view blocks of code.

A common operation in modern text editors is *find and replace* – given a pattern to search for, replace each occurrence of that pattern in the entered text with a certain replacement. If we were to implement this operation in C, it might have something like the following signature:

```
char *find_and_replace(const char *str, const char *find,
                       const char *replace);
```

The function would create a new heap-allocated string of the appropriate size containing the contents of the string in `str`, but with each occurrence of the pattern in `find` replaced with the pattern in `replace`. It is the caller's responsibility to free the returned string when they are done with it. Here is an example usage of the function:

```
...
char *original = "CS107 uses Unix a lot; I like Unix!";
char *replaced = find_and_replace(original, "Unix", "C");
printf("%s", replaced);

// prints "CS107 uses C a lot; I like C!"
...
```

Implement the `find_and_replace` function. The technique you must use is to allocate temporary space on the stack of size `MAX_STR_LEN` (a provided constant) to store the string, only use heap allocation once you know the full size of the string, and then return a pointer to the heap allocated string. You may assume that `find` will be a non-empty string and that all parameters are not `NULL`. The resulting implementation should compile cleanly and not have any memory errors or leaks. You should use built-in string library functions whenever possible. **You should use only one loop in your solution.** Solutions that violate the listed restrictions may get partial credit.

4) `get_resized` **40 Points/110 Total**

Note: you may need to scroll to fully view blocks of code.

The generic `get_resized` function, with the signature and parameters described below, takes as parameters information about an array of elements of any type and returns a new heap-allocated array that is the "resized" equivalent of the original. Specifically, it returns an array where each element appears in the same order as in the original array, but the number of times each element appears is specified by the provided *times* function.

As an example, let's say we pass information about the following `int` array to the `get_resized` function:

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

If we provide a times function that says an integer should appear a number of times equivalent to its value, the resulting array should look like this:

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

Note that the elements appear in the same order as in the original array, but 0 appears 0 times, 1 appears 1 time, 2 appears twice (consecutively), and so on.

The function signature and parameters are specified as follows:

```
void *get_resized(void *base, size_t nelems,
                  size_t elem_size_bytes, size_t *p_nelems,
                  size_t (*times_fn)(void *));
```

- `base` : a pointer to the first element of an array
- `nelems` : the number of elements in the provided array
- `elem_size_bytes` : the size of a single provided array element, in bytes
- `p_nelems` : a pointer to a `size_t` that should be updated to store the number of elements in the returned array
- `times_fn` : a function pointer that accepts a single parameter, a pointer to an element in the array, and returns the number of times the element should appear consecutively in the returned array.

It is the caller's responsibility to free the array when no longer needed. Since `get_resized` does not know initially how big the resulting array will be, it should use a resize-as-you-go approach. The array should start out **empty**. Every time an element is examined in the original array, the new array should be enlarged *just enough to make space* for the number of times that element should appear. Your implementations below should compile cleanly and not have any memory errors or leaks. Solutions that violate the specified restrictions may get partial credit.

Hint: You should initialize the array being returned as `NULL`. If you realloc a null pointer, it is equivalent to calling `malloc` with the realloc'ed size.

A) Implement the generic `get_resized` function.

B) Implement the `times_int` function that could be used as a parameter to `get_resized` with an array of `ints` to make each integer appear a consecutive number of times equivalent to its value. In other words, an integer with value 3 should appear 3 consecutive times in the resulting array, and an integer with value 0 should not appear.

C) Implement the `times_string_length` function that could be used as a parameter to `get_resized` with an array of strings to make each string appear a number of consecutive times equivalent to its length. In other words, a string with length 4 should appear 4 consecutive times in the resulting array, and a string of length 0 should not appear.