# CS107 Final 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 180 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.

**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.

# 1) Floats **25 Points/180 Total**

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

While generally useful for high-precision computation, in areas such as machine learning where precision is less important than runtime, 32-bit floating point numbers are unnecessarily costly due to their size. For this reason, there is active research into smaller floating point representations that still provide sufficient precision.

One such example in a 2018 paper published by IBM is an 8 bit float called FP8 that is structured the same as an IEEE float, but with 1 sign bit, 5 exponent bits, and 2 significand bits, and an exponent bias of 15. Use this format to answer the following questions below.

**A)** What is the corresponding (simplified) decimal number for the FP8 float `0 10101 00` ?

**B)** The smallest amount by which you can increment a float such as with FP8 is by incrementing the significand by one bit. For example, the next largest representable float to the one from part A would be `0 10101 01`. What is the corresponding (simplified) decimal number for this new number?

**C)** Describe one advantage the 32-bit signed integer representation has over the 32-bit float representation (IEEE) and one disadvantage.

**D)** Regardless of bitwidth, floats exhibit strange behaviors when performing arithmetic. What is the output of the following program, which uses regular IEEE 32-bit floats?

```
void print_equality(float x, float y) {
    printf("%s\n", x == y ? "true" : "false");
}

int main(int argc, char **argv) {
    float a = 1.125;
    float b = 1.0;
    float c = 1 << 31;
    float d = 0.0 / 0.0;
    float inf = INFINITY; // floating point +infinity
    print_equality(a + b, 2.125);
    print_equality(c - b, c + b);
    print_equality(2 * d, INFINITY);
    print_equality(inf * inf, INFINITY);
    return 0;
}
```

# 2) Strings and Memory **35 Points/180 Total**

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

So far, we have seen an array of strings as a way to model a string list. One alternative approach to modeling a list of strings is as a large block of memory storing all the strings' characters, one string's characters immediately following the other. In other words, a string list storing the string "Hi" followed by the string "Hello" would look in memory like "Hi\0Hello\0" (where in this example each \0 is the null terminator character, not the individual characters "\0").

Your task is to implement the `add_string` function. The function should attempt to add the specified new string to the specified string list, which is laid out in memory as described above. If that string is already present in the string list, it does nothing and returns false. If the string is not already present in the string list, you should resize the string list just enough to fit this new string, and then insert the new string at the *start of this string list*. You should assume that the string list characters live on the heap. The function would therefore have the following signature:

```
bool add_string(char **string_list_ptr, size_t total_bytes, char *new_string);
```

It takes in the following parameters:

- `string_list_ptr` : a pointer **to the address of** the start of the string list. The string list characters are assumed to live in heap memory. If the location of the string list in memory changes, you should update the value at the address stored in `string_list_ptr` to be the new location of the string list.
- `total_bytes` : The total number of bytes in the passed-in string list, including null terminators.
- `new_string` : the string to insert in the provided string list. You may assume it is a valid C string.

You do not need to worry about asserting for this problem. Here are some examples of using the function. Note that in the examples below, each \0 is the null terminator character, not the individual characters "\0":

- with a string list `Hi\0Hello\0` , total bytes 9 and new string "Hello", the function should not modify the string list and should return false.
- with a string list `Hi\0Hello\0` , total bytes 9 and new string "Howdy", the function should increase the memory size of the string list to be 15, should update the string list to be `Howdy\0Hi\0Hello\0` and should return true.

Fill in the blanks below to complete the `add_string` function.

```
bool add_string(char **string_list_ptr, size_t total_bytes, char *new_string) {

    // Pointer to use to loop over the string list
    // Should point to the first character in string list
    char *string_list_curr = ____1____;
    while (____2____) {

        // If this string equals the string to add, return false
        if (____3____) {
            return false;
        }

        // Update string_list_curr to point to next string
        ____4____ (one line only)
    }

    // We didn't find the string; resize and add it to the front of the string list
    ____5____ (3 lines)

    return true;
}
```

# 3) Assembly **40 Points/180 Total**

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

Consider the following code, generated by gcc with the usual -0g and other settings for this class, and use it to answer the following questions:

```
0000000000400546 <bar>:
  400546:   89 f8                   mov    %edi,%eax
  400548:   03 06                   add    (%rsi),%eax
  40054a:   c3                      retq

000000000040054b <foo>:
  40054b:   41 55                   push   %r13
  40054d:   41 54                   push   %r12
  40054f:   55                      push   %rbp
  400550:   53                      push   %rbx
  400551:   48 83 ec 18             sub    $0x18,%rsp
  400555:   49 89 fc                mov    %rdi,%r12
  400558:   41 89 f5                mov    %esi,%r13d
  40056b:   c7 44 24 04 6b 00 00    movl   $0x6b,0x4(%rsp)
  400573:   bb 01 00 00 00          mov    $0x1,%ebx
  400578:   bd 00 00 00 00          mov    $0x0,%ebp
  40057d:   eb 23                   jmp    4005a2 <foo+0x57>
  40057f:   48 85 db                test   %rbx,%rbx
  400582:   79 11                   jns    400595 <foo+0x4a>
  400584:   48 8d 74 24 04          lea    0x4(%rsp),%rsi
  400589:   48 89 df                mov    %rbx,%rdi
  40058c:   e8 b5 ff ff ff          callq  400546 <bar>
  400591:   01 c5                   add    %eax,%ebp
  400593:   eb 03                   jmp    400598 <foo+0x4d>
  400595:   44 01 ed                add    %r13d,%ebp
  400598:   48 89 da                mov    %rbx,%rdx
  40059b:   48 f7 da                neg    %rdx
  40059e:   48 8d 1c 12             lea    (%rdx,%rdx,1),%rbx
  4005a2:   4c 39 e3                cmp    %r12,%rbx
  4005a5:   7c d8                   jl     40057f <foo+0x34>
  4005a7:   8d 45 6b                lea    0x6b(%rbp),%eax
  4005bf:   48 83 c4 18             add    $0x18,%rsp
  4005c3:   5b                      pop    %rbx
  4005c4:   5d                      pop    %rbp
  4005c5:   41 5c                   pop    %r12
  4005c7:   41 5d                   pop    %r13
  4005c9:   c3                      retq
```

**A)** Fill in the C code below (also included in the answer area) so that it is consistent with the above x86-64 assembly. Your C code should fit the blanks as shown, so do not try to squeeze in additional lines or otherwise circumvent this (this may mean slightly adjusting the syntax or style

of your initial decoding guess to an equivalent version that fits). All literals of type signed/unsigned int/long must be written in decimal. Your C code should not include any casting.

```
int bar(long i, int *x) {
    return ________;
}

int foo(long max, int counter) {
    int sum = 0;
    int x = ________;
    for (long i = ________; i < ________; ________) {
        if (________) {
            sum += ________;
        } else {
            sum += ________;
        }
    }
    return ________;
}
```

**B)** Where does the variable `x` live? Why must it live there?

**C)** One of the lines in the assembly is of the form `test ARG, ARG`. What is the purpose of a `test` instruction with two of the same arguments? Explain how this impacts the following `jns` instruction.

**D)** At the start of `foo`, there are four push instructions. What type of registers are these pushed registers? What is the purpose of these instructions, and why must they be done?

# 4) Gaming The System **30 Points/180 Total**

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

After hearing about your success identifying vulnerabilities in ATM code, an online gaming company recruits you to help debug an issue they're seeing with player scores. Specifically, their anti-cheating measures don't seem to be working in certain cases, and they're not sure why.

They've narrowed down the issue to a single function, `sum_and_check`, for which they have provided you with the source code. The function is supposed to take in an array of scores from games a single player has played, along with the number of scores, and return their total score (which is the sum of all the individual scores). Game scores can only be 0, 1, 2 or 3; so, as anti-cheating measures, if they see a score that is more than 3, they change the score to 0 before they are summed, since this person must have cheated to obtain that score. While this seems to prevent impossibly high scores from being included in certain cases, it fails in others. How is this possible, they say?!

```
unsigned int sum_and_check(unsigned int scores[], int length) {
    unsigned int scores_copy[length];
    memcpy(scores_copy, scores, length * sizeof(*scores));

    for (int i = length; i >= 0; i--) {
        if (scores_copy[i] > 3) {
            scores_copy[i] = 0;
        }
    }

    // assume the sum() function works correctly
    return sum(scores_copy, length);
}
```

The engineers have identified two cases, one of which works, and one of which does not. Specifically, if scores is the array `[4, 4]` then the function correctly returns the total as 0, since these are cheating scores. However, if scores is the array `[4, 4, 4, 4]` then the function returns the total as 16, when it should be 0! They've asked you to specifically investigate this second case and how this player is still able to cheat.

You start investigating with GDB, and write a program that takes the scores as command line arguments, converts them to an array of unsigned ints, and passes this information to `sum_and_check`. Assume all code outside `sum_and_check` (not shown here) works correctly.

Running this, you uncover the following information about the program state:

```
$ gdb game
Reading symbols from game...done.
(gdb) b 24
Breakpoint 1 at 0x4006c1: file game.c, line 24.
(gdb) r 4 4 4 4
Starting program: /root/code/game 4 4 4 4

...
Breakpoint 1, sum_and_check (scores=0x7fffffffe950, length=4) at game
.c:24
24          if (scores_copy[i] > 3) {
(gdb) p &scores_copy[0]
$1 = (unsigned int *) 0x7fffffffe920
(gdb) p &i
$2 = (int *) 0x7fffffffe930
(gdb)

...
```

Answer the following questions below about this session of the program with this input.

**A)** After printing out the values above in GDB, you are curious about some additional values, so you also print out the value of `i` and `&scores_copy[i]` during the first iteration of the loop. What would GDB output as the values of these two expressions for this execution of the program?

**B)** You decide to add the following code to help with your investigative efforts: before the loop, you declare a new counter integer variable initialized to zero. Inside the loop, you increment this counter by 1 each iteration. After the loop, you print out the value of this counter. What value will be printed out as the value of this counter after the loop for the same input of `[4, 4, 4, 4]`? Assume that the memory layout information for the program reported in GDB above is unchanged with this new counter.

**C)** In 3 sentences or less, explain to the engineers exactly how, for this execution of the program with this input, the returned score is 16? Be as specific as possible, where necessary referring to specific code and memory information.

**D)** In 3 sentences or less, explain which line of code in `sum_and_check` should be changed to resolve the issue, what the change should be, and why this change resolves the issue.

# 5) Heap Allocators **50 Points/180 Total**

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

After seeing the promising results of your heap allocator implementations on assignment 7, you decide to experiment with other allocator designs to make memory management easier for the client. In particular, you decide to try implementing a **region-based heap allocator**. A region-based allocator does not allow the client to free memory allocation-by-allocation. Instead, it only allows freeing of entire *regions* of the heap. What this means is that you could, for instance, allocate nodes for a linked list all within the same region, and then instead of having to iterate through the linked list later to free each node, you can simply tell the allocator to free the entire region containing these nodes, and it will do so. Nice!

Here is the interface for this new type of allocator, followed by more information about what each function does:

```
zone_ptr *zone_alloc(size_t size);
void *malloc_within_zone(zone_ptr *zone, size_t size);
void zone_free(zone_ptr *zone);
```

`zone_alloc` creates a new zone on the heap of at least the specified size and returns a unique pointer that represents this zone. This pointer can be used by the client as a parameter for `malloc_within_zone` and `zone_free`, but should not be used by the client for any other purpose, e.g. it should not be dereferenced, the data it points to should not be changed, etc. If size is 0, then `zone_alloc()` returns NULL.

`malloc_within_zone` allocates at least `size` bytes within the specified zone and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, if the specified zone is NULL, or if there is not enough remaining space in this zone to satisfy this request, then this function returns NULL.

`zone_free` frees the specified zone, including all allocations it contains. The specified zone must have been returned by a previous call to `zone_alloc()`. If zone is NULL, no operation is performed.

Here's a sample of how a client might use such an allocator:

```
int *arr[5];
zone_ptr *zone = zone_alloc(256);
for (int i = 0; i < 5; i++)  {
    arr[i] = malloc_within_zone(zone, sizeof(int));
}

// We don't have to free each individual allocation;
// since they are all in the same zone, we free the whole zone!
zone_free(zone);
```

While this allocator design may seem very far off from the implicit and explicit allocators discussed in class, it turns out they are more similar than you might think. In particular, we can imagine it being the same as an explicit allocator, except instead of *blocks* we have *zones*. Each zone has a header and a payload. The difference is a zone's payload acts as a miniature bump allocator; whenever a request comes in to allocate memory within a zone, we look at the next-largest available address within that zone's payload and return it. (As a reminder, a bump allocator keeps a pointer to the next available address, and for each allocation increments this pointer as more memory is used up - space in a bump allocator cannot be reused). Allocations within a zone do not have headers; the only headers are for each zone. As a result, individual allocations within a zone cannot be freed or resized; only the entire zone can be freed. A zone cannot be resized.

Here's more information about the specifics of this design:

- All request sizes are rounded up to a multiple of 8 bytes and all returned pointers are aligned to 8-byte boundaries.
- Each zone has a 16-byte header storing the zone payload size, whether the zone is allocated or free, and the next available address within its payload to use for an allocation. Because zone payload sizes must be multiples of 8 bytes, the **SECOND** bit from the right (second-least significant) of the size is unused and can thus be used to store whether a zone is free (0) or allocated (1).
- The allocator maintains an explicit free list of zones as an unsorted doubly-linked list. A global variable points to the header of the first free zone in the linked list (or NULL if there are no free zones).
- Allocated zones use the pointer in their header to store the next available address within its payload. Freed zones use the pointer in their header to store the location of the previous free zone in the free list, and use the first 8 bytes of the zone payload to store the location of the next free zone in the free list.

Below are the allocator's global variables, constants, type definitions and the provided `round_up` function:

```
#define FREE 0
#define ALLOCATED 1

typedef struct zone_header {
    size_t payload_size_and_status;
    void *ptr;
} zone_header;

// zone_ptr is really zone_header, but client doesn't know
typedef zone_header zone_ptr;

// pointer to the start of the heap
static zone_header *heap_start_header;

// pointer to first header in free list
static zone_header *free_list_head;

/* Function: round_up
 * -----------------
 * This function rounds up the given number to the given multiple, which
 * must be a power of 2, and returns the result.
 */
size_t round_up(size_t sz, size_t mult) {
    return (sz + mult-1) & ~(mult-1);
}
```

Each of the parts below has you implement functionality for this allocator design. For full credit, you can (and should) call other functions in this problem as needed rather than re-implementing logic.

**A)** Implement `is_used` . Given a pointer to a zone's header, it should return true if the zone is allocated, or false if the zone is free.

```
bool is_used(zone_header *header);
```

**B)** Implement `get_size`. Given a pointer to a zone's header, it should return the size of that zone's payload, in bytes.

```
size_t get_size(zone_header *header);
```

**C)** Implement `set_status` . Given a pointer to a zone's header, and a new status, either `FREE` or `ALLOCATED` (the numbers 0 or 1), it should update the header to store the updated status. Its payload size should not change.

```
void set_status(zone_header *header, int new_status);
```

**D)** Implement `get_next_header`. Given a pointer to a zone's header, it should return a pointer to the header of the zone to the immediate right. If the specified header is the last header on the heap, the function should return the address right after the heap end.

```
zone_header *get_right_header(zone_header *header);
```

**E)** Implement `malloc_within_zone` as described in the problem summary.

```
void *malloc_within_zone(zone_ptr *zone, size_t size);
```

**F)** Implement `set_next` . Given a pointer to a zone's header and a pointer to another free zone, it should update the pointer stored in the first zone's payload to store the location of this next free zone.

```
void set_next(zone_header *free_header, zone_header *next);
```

**G)** To the dismay of your style instincts, your coworker insists that instead you implement the `set_next` function using a single `memcpy` call. Fill in the blanks below.

```
void set_next(zone_header *free_header, zone_header *next) {
    memcpy(________, ________, ________);
}
```

**H)** You decide to experiment with an alternative header design for each zone where instead of storing the zone payload size explicitly, you instead directly store a pointer to the immediate right neighbor zone header. This way, instead of having to perform pointer arithmetic to get the right neighbor, you can access it immediately. The downside is it takes a bit more calculation if you do need to access the payload size.

Implement `get_size`, assuming the following new definition of `zone_header`. Given a pointer to a zone's header, it should return the size of that zone's payload, in bytes.

```
typedef struct zone_header {
    struct zone_header *right_neighbor_header;
    bool is_free;
    void *ptr;  // same as previously
} zone_header;

size_t get_size(zone_header *header);
```