# Final Exam Solutions

## 1. Floats (25)

**A:**

```
64     // 1.0 * 2^(21-15)
```

**B:**

```
80     // 1.25 * 2^(21-15)
```

**C:** Some possible answers:

- Advantage: integers have only one representation of zero because of their non-dedicated sign bit
- Advantage: integers can represent every integer number in their range
- Advantage: arithmetic is simpler with integers than floats because of two's complement
- Disadvantage: integers cannot represent large scientific notation numbers as well as floats
- Disadvantage: integers overflow, while floats have infinity for representing too-large numbers (easier to work with)
- Disadvantage: because of their non-dedicated sign bit, integers don't have the same number of negative numbers as positive numbers, while floats do

**D:** (explanations were not required)

true // reasonable float arithmetic

true // extremely large number with extremely small number loses precision in both cases, so equal

false // d is NaN, so 2 * d is NaN and is not equal to INFINITY

true // infinity * infinity is still infinity

## 2. Strings and Memory (35)

*Sample Solution*

```
// blank 1
*string_list_ptr

// blank 2
string_list_curr < *string_list_ptr + total_bytes

// blank 3
strcmp(string_list_curr, new_string) == 0

// blank 4
string_list_curr += strlen(string_list_curr) + 1

// blank 5-1
*string_list_ptr = realloc(*string_list_ptr, total_bytes + strlen(new_string) + 1);
memmove(*string_list_ptr + strlen(new_string) + 1, *string_list_ptr, total_bytes);
strcpy(*string_list_ptr, new_string);
```

## 3. Assembly (40)

**A: Reverse Engineering**

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

int foo(long max, int counter) {
    int sum = 0;
    int x = 107;
    for (long i = 1; i < max; i *= -2) {
        if (i < 0) {
            sum += bar(i, &x);
        } else {
            sum += counter;
        }
    }

    // or sum + x
    return sum + 107;
}
```

**B:** The variable x lives on the stack, as shown on the assembly line **`movl $06b, 0x4(%rsp)`**. This shows the value 107 being moved onto the stack, and later we use the address of **`x`** when calling **`bar`**, as in the instruction **`lea 0x4(%rsp), %rsi`**. **`x`** must live on the stack because we get its address using the & operator, and only in-main-memory values have addresses.

*Note:* we also accepted answers that interpreted "why must it live there?" as "justify your answer" instead of "why is it required to live there".

**C:** **`test`** with two of the same argument is useful to test the sign of that specified value. This is because this will & the argument with itself, which equals itself, and update the condition codes to reflect this value. This impacts the following **`jns`** instruction because it sets the condition codes that **`jns`** checks when deciding whether or not to jump (specifically, **`jns`** checks the sign flag).

*We gave full credit* for any answer for the first part of C that described the general correct behavior for test with two of the same argument, including answers like "it checks the sign of the value" or "it sets flags based on the value of the argument", but did not give full credit for answers that omitted significant information such as "it sets the sign flag based on the value of the argument" (since it also sets the zero flag).

**D:** Caller-owned registers are being pushed at the start of this function. They must be pushed because they are guaranteed to the caller to be preserved across the function call so we must preserve them if we want to use them and pushing them pushes their values onto the stack to save for later.

*We were looking for the following points:*

- Identify caller-owned registers as the ones being pushed
- Mention that functions must have the caller value preserved and restored, and push preserves the value on the stack for later.

## 4. Gaming The System (30)

**A:** At the start of the loop, i is 4 and &scores_copy[i] is 0xfffffffe930. (We also accepted answers that interpreted the question as asking what the values are at the end of the first iteration, which would be i = 0 and &scores_copy[i] = 0xfffffffe920).

**B:** The counter would be printed out as 1 (while an explanation is not required, this is because the loop terminates after the first iteration because i is overwritten to zero, and then decremented by the loop, meaning the loop condition the second iteration will be false and the loop will stop).

**C:** During the first iteration of the loop, i is overwritten to 0, because we access the index-4 element in an array which only goes through index 3. The element one beyond the array bounds is i, because i lives in memory immediately above (0x7fffffffe930) the location of the score array (0x7fffffffe920 to 0x7fffffffe920 + 4 * sizeof(int)). Thus, i is overwritten to 0 immediately, then decremented by one by the loop, meaning the loop breaks before it ever examines any of the scores and the original cheating scores are still summed up.

**D:** The line that should be changed is the for loop line **`for (int i = length; i >= 0; i--)`**. It should be changed to have i initialized to length – 1, because this is the maximum element index in a zero-indexed array with **length** elements. This fixes the issue because all indices between 0 and length – 1 are valid indices in the scores_copy array, meaning the code will no longer accidentally access or overwrite i.

## 5. Heap Allocators (50)

**A:**

```
bool is_used(zone_header *header) {
    return ((header->payload_size_and_status) & 2) != FREE;
}
```

**B:**

```
size_t get_size(zone_header *header) {
    return (header->payload_size_and_status) & ~2;
}
```

**C:**

```
void set_status(zone_header *header, int new_status) {
    header->payload_size_and_status = get_size(header) | (new_status << 1);
}
```

**D:**

```
zone_header *get_right_header(zone_header *header) {
    return (zone_header *)((char *)header + sizeof(*header)
        + get_size(header));
}
```

**E:**

```
void *malloc_within_zone(zone_ptr *zone, size_t size) {
    if (size == 0 || zone == NULL) {
        return NULL;
    }

    size = round_up(size, 8);

    // Check if enough space left in this zone
    if ((char *)(zone->ptr) + size > (char *)get_right_header(zone)) {
        return NULL;
    }

    void *next_available_address = zone->ptr;
    zone->ptr = (char *)(zone->ptr) + size;
    return next_available_address;
}
```

**F:**

```
void set_next(zone_header *free_header, zone_header *next) {
    *((void **)(free_header + 1)) = next;
}
```

**G:**

```
memcpy(free_header + 1, &next, sizeof(next));
```

**H:**

```
size_t get_size(zone_header *header) {
    return (char *)(header->right_neighbor_header) - (char *)(header + 1);
}
```