Final Exam Solutions

1. Floats (25)

A: 0 10011 1000000000 in decimal

```
24     // 1.5 * 2^(19-15)
```

B: 0 00000 1100000000 in scientific notation

```
0.75 * 2^(-14)
```

C:

The size of gaps between sequential floats increases as floats get larger, because the increasingly large exponent multiplies the mantissa by a larger and larger amount each step. For this reason, as the number gets bigger, eventually incrementing by one will equal the same number because the step size to the next representable float is more than 1.

D: (explanations were not required)

true // reasonable float arithmetic

true // extremely large number with extremely small number loses precision!

false // this float division is not the same when done in different orders because of precision loss

2. Generics (45)

A:

Sample Solution

```
void remove_at(void *base, size_t nelems, size_t elem_size_bytes,
               size_t remove_index, void *dest) {
    void *elem_loc = (char *)base + (remove_index * elem_size_bytes);

    // copy the element into destination
    memcpy(dest, elem_loc, elem_size_bytes);

    // shift further-right elements to the left by 1
    memmove(elem_loc, (char*)elem_loc + elem_size_bytes,
            (nelems - remove_index - 1) * elem_size_bytes);
}
```

B:

Sample Solution

```
void set_at(void **base, size_t *p_nelems, size_t elem_size_bytes, void *elem,
            size_t index) {
    // check if we need to resize
    if (index >= *p_nelems) {
        // resize to index + 1 elements
        *base = realloc(*base, elem_size_bytes * (index + 1));
        *p_nelems = index + 1;
    }

    // regardless of whether we resize, copy the element into index
    memcpy((char *)*base + index * elem_size_bytes, elem, elem_size_bytes);
}
```

C:

Sample Solution

```
void order_pairs(void *base, size_t nelems, size_t elem_size_bytes,
                 int (*cmp_fn)(void *, void *)) {
    // loop over each pair (discarding any remainder if # elems is odd)
    for (int i = 0; i < nelems / 2; i++) {
        void *first = (char *)base + 2 * i * elem_size_bytes;
        void *second = (char *)first + elem_size_bytes;

        // Check if we need to swap elements
        if (cmp_fn(first, second) > 0) {
            char buf[elem_size_bytes];
            memcpy(buf, first, elem_size_bytes);
            memcpy(first, second, elem_size_bytes);
            memcpy(second, buf, elem_size_bytes);
        }
    }
}
```

D:

Sample Solution

```
int compare_ints(void *a, void *b) {
    return *(int *)b - *(int *)a;
}
```

E:

```
set_at(&arr, &length, sizeof(int), &elem_to_append, length);
```

F:

```
order_pairs(arr, length, sizeof(int), compare_ints);
```

3. Assembly (40)

A: Reverse Engineering

Sample Solution

```
int func2(int x, int y) {
    int z = (x * 6) / 4;
    return z << y;
}

void func1(int *arr, int count) {
    int z = 4;
    for (int i = 0; i < count - 1; i += 2) {
        z += func2(arr[i], arr[i + 1]);
    }

    printf("%d\n", z);
}
```

Note: *because of the confusion surrounding the printf call in the assembly, we were lenient in the answers we accepted for the last blank if they were reasonable given the problem information.*

B: Caller-owned registers are being pushed and popped at the start and end of this function because they are guaranteed to the caller to be preserved across function calls so we must preserve them if we want to use them.

We were looking for the following points:

- Identify caller-owned registers as the ones being pushed/popped
- Mention that functions must have the caller value preserved and restored

C: instructions 40057d (mov (%r12, %rdx, 4)) and 400581 (mov (%r12,%rax, 4)) both do pointer arithmetic and multiply the index by 4, indicating it is doing pointer arithmetic with a 4 byte pointee type.

We were looking for the following points:

- Identify instruction
- Pointer arithmetic scale factor of 4 identifies that it is a 4 byte pointee type

D: %rip stores the address of the next instruction to execute. The callq instruction pushes the current %rip value onto the stack before calling the function to preserve where we should resume after the callee finishes, and when the callee is done the **retq** instruction will restore the saved %rip value so that execution resumes in the right place in the caller.

Note: you did not have to specifically mention callq, retq or %rip to receive full credit.

4. ATM Part 2 (25)

A: We can successfully log in if we enter a 12-character passcode. The vulnerability is that the password is copied into an 8 character buffer regardless of how long the password is. Therefore, if the password is longer than 7 characters (+ null terminator), it will overflow. The GDB output indicates that buffer is 12 bytes below real_password, so if we enter a 12-character passcode, its null terminator will be copied into the first character of real_password, making the real_password appear empty, and logging us in.

Note: it's not possible to overwrite real_password so that it matches buffer, because they are compared using strcmp – for this reason, buffer will be e.g. a 20-character string, and real_password will be e.g. an 8 character string, so they will not be equal.

B: There are various fixes that can prevent the overflow, such as:

- Using strncpy instead of strcpy and copying in 7 characters, and then manually copying in a null terminator (this is more than a 1 line change, but was accepted)
- Sizing buffer to be strlen(user_password) + 1

All of these fix the issue by preventing overflow – either making the buffer big enough to store the entire password, or truncating the user password to fit in the 8 byte buffer.

5. Heap Allocators (45)

A:

```
bool is_used(size_t *header) {
      return *header & ALLOCATED;
}
```

B:

```
size_t get_size(size_t *header) {
      return *header & ~ALLOCATED;
}
```

C:

```
void set_size(size_t *header, size_t new_size) {
      bool used = is_used(header);
      *header = new_size | (used ? 1 : 0);
}
```

D:

```
size_t *get_next_header(size_t *header) {
    char *next = (char *)header + sizeof(size_t) + get_size(header);
    if (next >= (char *)heap_start + heap_size) {
        return NULL;
    }
    return (size_t *)next;
}
```

E:

```
void coalesce_starting_at(size_t *header) {
    size_t *next_header = get_next_header(header);
    while (next_header && !is_used(next_header)) {
        set_size(header, get_size(header) + get_size(next_header) +
                 sizeof(*header));
        next_header = get_next_header(header);
    }
}
```

F:

```
void coalesce_all() {
    size_t *ptr = heap_start;
    while (ptr != NULL) {
        if (!is_used(ptr)) {
            coalesce_starting_at(ptr);
        }
        ptr = get_next_header(ptr);
    }
}
```

G:

```
void *next_payload = *(void **)((char *)ptr - sizeof(void *));
```

H:

```
memcpy(&next_payload, (char *)ptr - sizeof(void *), sizeof(void *))
```