This exam is based on the final exam given in Fall 2017. The class was taught by Julie Zelenski and Chris Gregg. This was a 3-hour paper exam.

Problem 1: C-strings

1a) `"Tessi"`

1b)

```
void substring(char **p_input, size_t pos, size_t len)
{
    (*p_input) += pos;
    (*p_input)[len] = '\0';
}
```

1c)

```
int main(int argc, char *argv[])
{
    char name[16];
    strcpy(name, "Tessier-Lavigne");
    char *ptr = name;
    substring(&ptr, 3, 2);
    printf("%s\n", ptr);
}
```

Problem 2: Generics

2a)

```
void extract_min(void *addr, void *base, size_t *p_nelems, size_t width,
                 int (*cmp)(const void *, const void *))
{
    void *min = find_min(base, *p_nelems, width, cmp);
    memcpy(addr, min, width);
    size_t bytes_to_move = (*p_nelems * width) -
                           ((char *)min + width - (char *)base);
    memmove(min, (char *)min + width, bytes_to_move);
    (*p_nelems)--;
}
```

2b)

```
int cmp_len(const void *p, const void *q)
{
    return strlen(*(const char **)p) - strlen(*(const char **)q);
}

char *shortest(char *words[], size_t *p_nwords)
{
    char *min;
    extract_min(&min, words, p_nwords, sizeof(char *), cmp_len);
    return min;
}
```

Problem 3: Floating point

3a) -4.5

3b)

1.5
inf
nan
true

Problem 4: x86 Assembly

4a)

```
int pinky(char *param1, int *param2)
{
    char *str = NULL;
    int local = strtol(param1, &str, 16);
    if (local - 7 > 12)
    {
        local += param2[3];
    }
    return local/4;
}
```

4b) A strength reduction optimization is when GCC replaces expensive types of assembly instructions with cheaper ones. In this case, GCC substitutes an expensive **`div`** instruction in the return statement for a less-expensive shift. It can do this because the divisor is 2^N, and so division can be calculated as a bit shift right N positions (with fixup to round negative numbers

toward zero). Even though this trades four instructions for one, it still comes out ahead because `div` is so expensive.

4c) Line 3: An unsigned comparison is put in place of a signed comparison (`jle` -> `jbe`). Line 5: Unsigned divide does not require negative fixup, since it's now just a logical bit shift right N positions. Thus, the first four instructions at `.L1` are replaced with `shr $0x2.` (Note: there is an additional move instruction to get values in the right registers with the new shift instruction, but it's fine to not worry about this).

Problem 5: Runtime stack

5a) The `retq` instruction at the end of `concat` fails to execute. This is because during the execution of `concat`, the `strcpy/strcat` overflow the result array and overwrite the saved `%rbx` and return address. At the end of `concat`, the `retq` instruction pops the garbage address from the stack into `%rip` and attempts to resume execution at that location, which isn't a valid address, so it crashes.

5b) When a stack frame is deallocated, the memory remains accessible and contents remain as-is until a subsequent stack frame overwrites it. The stack frame for `printf` is overwriting what was the stack frame for `concat`, but since `concat` had a very large frame and `printf` has a much smaller one, it is overwriting the tail end of the stack array (which is unused), and never reaches down to the used part where the characters are stored. For this reason, it is still able to print out the correct result.

5c) Now that the `concat` frame is tightly-sized to exactly the string length, the `printf` frame is writing on the used contents of the stack array, garbling the string contents.

5d) The too-small version halts with error "stack smashing detected", because an overflow there will overwrite the canary value as it overwrites the saved registers. The too-large version does not exhibit overflow here because the input is not long enough, and the correct-sized version cannot exhibit overflow, so neither of those interfere with the canary value and thus execute unchanged.

Problem 6: Heap allocator

6a)

```
#define USED    (1UL << (HDRSIZE*8 - 1))
#define NWORDS  ~(USED)
```

The constant value 1 on the first line must be a `long` to avoid over-shifting. On the second line, we want to invert the binary representation, not the sign.

6b)

```
bool is_used(Header *hdr)
{
    return (*hdr) & USED;
}
```

6c)

```
Header *get_neighbor(Header *hdr)
{
    /* Note: we are adding multiples of 8-byte words here!
     * we also use void * here to avoid warnings when comparing
     * with segment_end.
     */
    void *neighbor = hdr + ((*hdr) & NWORDS) + 1;
    if (neighbor >= segment_end) return NULL;
    return neighbor;
}
```

6d) `&free_list`

6e) O(N). Removing the check means approximately 2 instructions fewer are executed per iteration of the loop (potentially a test/compare and a jump), which means the total number of instruction executions saved is proportional to N.

6f) `prev = *prev;`

6g) `*prev = *(void **)to_remove;`

6h) There is a mismatch in units – nbytes is being compared to a number of words. This will erroneously reject blocks that could be used because it believes they are too small.

6i)

```
*cur += *neighbor + 1; // can mask but don't need to, neighbor not in-use
remove_from_freelist(neighbor + 1); // note: arg is pointer to payload!
```

6j) There is no way to directly access the block header of a left neighbor, because we don't know how large the left neighbor is. Walking the block headers from segment_start (i.e. traverse implicit list) could find it, but in the worst-case requires a complete linear traversal of the entire heap! And even if this is done, expanding to the left for myrealloc still has to copy the payload data to the new start address, which is the expensive operation resize-in-place tries to avoid. Resize in place is only possible if the starting address stays the same, as in expanding right.