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

The function **`substring (char *input, size_t pos, size_t len)`** is intended to trim **`input`** to the sequence of characters starting at **`pos`** and continuing for **`len`** characters. If **`pos`** or **`len`** is out of range for the input string, the behavior is undefined. The implementation below is buggy:

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

You write a program to test the function:

```
int main(int argc, char *argv[])
{
    char name[16];
    strcpy(name, "Tessier-Lavigne");
    buggy_substring(name, 3, 2);
    printf("%s\n", name);
    return 0;
}
```

1a) The above test program is intended to print **`"si"`**. What is printed instead?

1b) You change the prototype of **`substring`** to take the input string by reference. Implement the body of the function so that it works correctly and is compatible with the new prototype.

```
void substring(char **p_input, size_t pos, size_t len)
{
```

1c) Complete the program started below to test the **`substring`** function. Compute the substring of **`name`** starting at position 3 of length 2 and print it.

```
int main(int argc, char *argv[])
{
    char name[16];
    strcpy(name, "Tessier-Lavigne");
```

Problem 2: Generics

2a) The function `extract_min` extracts the smallest element from a generic array according to a comparison function. The client supplies an address as the first argument; the smallest element is written to this address and that element is removed from the array. As an example, `extract_min` on the array `{8, 5, 19, 11}` with ordinary integer comparison writes `5` to the client's address, changes the array contents to `{8, 19, 11}`, and decrements the number of elements from 4 to 3. The order of the remaining array elements is preserved and the array's storage is not resized. Assume the array has at least one element.

The provided `find_min` function returns a pointer to the smallest element in a generic array. The use of `find_min` below is correct and you can assume `find_min` is correctly implemented. Complete the implementation of `extract_min` below.

```
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);
```

2b) Complete the `shortest` function to remove the shortest string from an array of words and return it. If two strings tie for shortest, either choice is fine. You must fill in the blank line in `shortest` with a single call to `extract_min` and can assume the function works correctly. You will also need to write the comparison function.

```
int cmp_len(const void *p, const void *q)
{
    ____________________________________________________________________;
}

char *shortest(char *words[], size_t *p_nwords)
{
    char *min;
    ____________________________________________________________________;
    return min;
}
```

Problem 3: Floating point

A normalized IEEE 32-bit float is stored as the following bit pattern:

```
N EEEEEEEE SSSSSSSSSSSSSSSSSSSSSSS
```

where `N` is the sign bit (1 if negative), `E` is the 8-bit exponent (with a bias of 127), and `S` is the 23-bit significand, with an implicit leading "1."

3a) Given the following 32-bit float binary, what is the corresponding (simplified) decimal number that it represents?

```
1 10000001 00100000000000000000000
```

3b) What is the output of the following code?

```
float val = 1.5;
printf("%g\n", val);

val = val * FLT_MAX;
printf("%g\n", val);

val = 0.0;
printf("%g\n", val / val);

float a = 1.234e20;
float b = a + 100;
printf("a == b? %s\n", (a == b) ? "true" : "false");
```

Problem 4: x86 Assembly

Below is the assembly generated for the **`pinky`** function. It was compiled **`-Og`** (mild level of optimization) and the compiler applied a strength reduction optimization when generating assembly for the final line of the function.

```
pinky:
    push   %rbx
    sub    $0x10,%rsp
    mov    %rsi,%rbx
    movq   $0x0,0x8(%rsp)
    mov    $0x10,%edx
    lea    0x8(%rsp),%rsi
    callq  <strtol>
    mov    %eax,%edx
    lea    -0x7(%rax),%ecx
    cmp    $0xc,%ecx
    jle    .L1
    add    0xc(%rbx),%edx
.L1:
    lea    0x3(%rdx),%eax
    test   %edx,%edx
    cmovns %edx,%eax
    sar    $0x2,%eax
    add    $0x10,%rsp
    pop    %rbx
    retq
```

4a) Fill in the blanks in the C code below to match the assembly above. There should be **no typecasts**. Note this is nonsense code, not intended to do anything at all meaningful.

```
int pinky(char *param1, int *param2)
{
    char *str = NULL;

    int local = ____________________________________________________;

    if ( ____________________________________________________ )

    {

        ____________________________________________________;

    }

    ____________________________________________________;
}
```

4b) What is a strength reduction optimization and how is it applied here?

4c) For many operations, the same assembly instruction, e.g. `add`, can be used for either signed or unsigned values, but this is not true for all operations. The declaration of `local` is changed to `unsigned int` and `pinky` is recompiled. Consider the five lines of C in the `pinky` function. Three of the lines generate the exact same sequence of assembly instructions as before; two do not. Identify which two lines generate different assembly and describe how the assembly changed.

Problem 5: Runtime stack

The function **concat** is a poorly-coded attempt at concatenation. Not only does it declare the result array with an arbitrary small size, it returns the address of that stack-allocated array. Review the C source below on the left and the generated assembly on the right.

```
char *concat(const char *s, const char *t)
{
    char buffer[10];
    char *result = buffer;

    strcpy(result, s);
    strcat(result, t);
    return result;
}


int main(int argc, char *argv[])
{
    char *title = concat(argv[1], argv[2]);
    printf("%s\n", title);
    return 0;
}
```

```
concat:
    push   %rbx
    sub    $0x10,%rsp
    mov    %rsi,%rbx
    mov    %rdi,%rsi
    mov    %rsp,%rdi
    callq  <strcpy>
    mov    %rbx,%rsi
    mov    %rsp,%rdi
    callq  <strcat>
    mov    %rsp,%rax
    add    $0x10,%rsp
    pop    %rbx
    retq

main:
    sub    $0x8,%rsp
    mov    %rsi,%rax
    mov    0x10(%rsi),%rsi
    mov    0x8(%rax),%rdi
    callq  <concat>
    mov    %rax,%rsi
    mov    $0x401850,%edi
    mov    $0x0,%eax
    callq  <printf>
    mov    $0x0,%eax
    add    $0x8,%rsp
    retq
```

5a) Running **./program LelandStanfordJunior University** crashes during execution. Circle the assembly instruction at which the crash occurs and explain why this instruction fails to execute.

5b) Seeing the array is too small, the implementer enlarges the size of the **buffer** declared within **concat** to:

```
char buffer[4096];
```

After this change, running **./program LelandStanfordJunior University** now prints **LelandStanfordJuniorUniversity**. Explain why the program appears to work correctly despite the fact that **concat** returns the address of a stack-allocated array.

5c) Not wanting to squander memory, they then change the declaration of `buffer` to the proper size:

```
char buffer[strlen(s) + strlen(t) + 1];
```

After this change, `./program LelandStanfordJunior University` now prints garbage. Explain how the change in sizing the array leads to this change in behavior.

5d) They decide to add the compiler flag `-fstack-protector` to see if that will change anything. This flag tells GCC to add instructions at the start of each function to write something called a *canary value* on the stack. This is an arbitrary constant value written directly below the saved registers and above any local variables. GCC also adds instructions at the end of each function to check the integrity of this canary value, and if it has changed, terminate the program with the error message "stack smashing detected", as it knows that some previous instructions improperly wrote to that memory area.

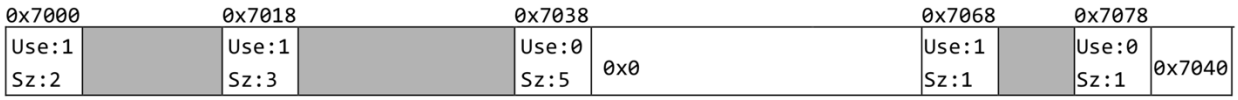
Thus, they recompile the three program versions (too-small array, too-large array, right-sized array) with the `-fstack-protector` flag and execute each as `./program LelandStanfordJunior University.` Only one of the versions has any noticeable change when executing. Explain which version behaves differently and why.

Problem 6: Heap allocator

You are writing code for an allocator that uses a block header and maintains an explicit free list. Implementation details of this allocator include:

- All requests are rounded up to a multiple of 8-bytes and all returned pointers are aligned to 8-byte boundaries. The minimum payload size is 8 bytes.
- The header is an unsigned long (8 bytes) that bit-mashes together the block information:
  - **most** significant bit is 1 if block is in-use, 0 if free
  - **lower** 63 bits store the payload size, expressed as **count of 8-byte words**
- The allocator maintains an explicit free list as a **singly**-linked list stored in the payload. A global variable points to the **payload** of a free block (or NULL if no free blocks). The first 8 payload bytes of each free block store a pointer to the **payload** of another free block. The last free block on the list stores NULL in its payload.

  Here is an example heap after a few requests have been serviced:



This segment starts at address 0x7000 and ends at 0x7088. Three blocks are in-use, two are free. The in-use payloads are shown shaded in gray. The first header has the most significant bit on (block is in-use) and the lower bits store 2 (size of payload is 2 words = 16 bytes). The `free_list` points to the payload at 0x7080, the payload at 0x7080 stores a pointer to the payload at 0x7040, and the payload at 0x7040 stores NULL.

Below are the allocator's global variables, constants, and type definitions.

```
// define Header type that is actually unsigned long
typedef unsigned long Header;
#define HDRSIZE sizeof(Header)

// mask to extract used bit from header
#define USED   0x8000000000000000

// remaining header bits store payload size as count of 8-byte words
#define NWORDS 0x7fffffffffffffff

static void *segment_start;    // base address of heap segment
static void *segment_end;      // end address of heap segment
static void *free_list;        // pointer to payload of first free block
                               // (NULL if no free blocks)
```

6a) The `USED` and `NWORDS` bitmasks isolate the most significant bit from the lower bits. A code reviewer is displeased with defining these as fixed constants and instead recommends building the masks by construction. Your first attempt to follow their recommendation is shown below.

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

Your approach is correct but the details are a little off. You examine the mask values in GDB and see that both are zero -- oops! Rewrite the two mask definitions below, correcting where they have gone astray. Only one character needs to be edited in each definition.

6b) The `is_used` function is given a pointer to a block's header and returns true if the block is in- use, false otherwise. Complete the function by filling the blank with the correct expression.

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

6c) Implement `get_neighbor`. Given a pointer to a block's header, it returns a pointer to the header of the neighbor to the right, i.e., at the next higher address in the heap. If `hdr` has no right neighbor (i.e., it is the rightmost block in the heap segment), the function returns `NULL`.

```
Header *get_neighbor(Header *hdr)
{
```

The function `remove_from_freelist` is given a pointer to a **free payload**. It searches the free list for that entry and removes it from the list. An implementation is started below. The two blanks in the code will be completed when answering parts d-g.

```
void remove_from_freelist(void *to_remove)
{
    void **prev = NULL;

    for (prev = ___________; prev != NULL; memcpy(&prev, prev, sizeof(prev))) {

        if (*prev == to_remove) break;
    }

    ____________________________________________;

}
```

6d) The loop initialization is missing the expression to be assigned to **`prev`**. Circle the correct expression from the choices below:

`to_remove`	`&to_remove`	`&prev`
`free_list`	`&free_list`	`*free_list`

6e) The function should only be called with a pointer that is on the free list. Given this constraint, the loop test **`prev != NULL`** can be removed entirely. Approximately how many fewer instructions are executed during a call to **`remove_from_freelist`** as a result of this change? Express your answer in Big-O notation in terms of the length of the freelist, N (e.g. O(logN) would mean the reduction in instructions executed is proportional to logN).

6f) The **`memcpy`** call in the loop increment expression is correct, but slow. Rewrite the increment expression below to achieve an equivalent effect, but efficiently, with no use of **`memcpy`**.

6g) The function is missing the splice operation. Complete the function by filling in the blank line at the end of the function with a statement to correctly take **`to_remove`** out of the free list.

To add resize-in-place to **`myrealloc`**, you are writing the **`expand_right`** helper. The function is given a pointer to a header and the number of additional bytes needed. If the right neighbor is unavailable or too small, the function makes no changes and returns false. Otherwise, it absorbs the right neighbor into the current block, updates all heap data structures, and returns true. The function does not split any excess, nor does it attempt to expand beyond the first neighbor. The implementation of **`expand_right`** is started below and will be completed in parts h-i.

```
bool expand_right(Header *cur, size_t nbytes)
{
    Header *neighbor = get_neighbor(cur);
    if (neighbor == NULL || is_used(neighbor) || (*neighbor + 1) < nbytes) {
        return false;
    } else {
        
        return true;
    }
}
```

6h) The expression (**`*neighbor + 1) < nbytes`** is supposed to reject the neighbor if it is too small for the requested increase, but the test condition does not work as intended. Identify the problem and describe the observed consequence when testing the flawed code.

6i) Assume the issue above is resolved. Now complete the implementation of **`expand_right`** by writing the code to add into the boxed area so that the function works correctly. You should call **`remove_from_freelist`** and can assume it works correctly.

6j) You consider adding **`expand_left`** to absorb the left neighbor but decide against it. Explain why expanding left cannot be efficiently implemented in the current design and furthermore why expanding left will not provide the same benefit to **`myrealloc`** as expand right.