This document contains the questions and solutions to the CS107 Final given in Winter 2018 by instructor Chris Gregg. This was a 3-hour exam.

Problem 1 (Void * generics) (24 points) (suggested time: 30 minutes)

Given a *sorted* `void *` array of elements, the `remove_duplicates` function will remove all duplicates in the array, returning the new size of the array. Your function should modify the array in-place (e.g., to remove a duplicate, move the rest of the array backwards over it):

```
int remove_duplicates(void *arr, size_t nelems,
                      int width, int (*cmp)(void *, void *));
```

For example, given the following code, and assuming a correct `int` comparison function:

```
int iarr[] = {1, 1, 1, 2, 2, 3, 3, 3, 4, 7, 8, 8};

int newsz = remove_duplicates(iarr,12,4,cmp_int);
```

`newsz` holds the value `6` and the first six elements of `iarr` are:

```
{1, 2, 3, 4, 7, 8}
```

1a) Write the generic `remove_duplicates` function:

```
int remove_duplicates(void *arr, size_t nelems,
                      int width, int (*cmp)(void *, void *))
{
    int i = 0;
    while (i < nelems - 1) {
        void *ith = (char *)arr + i * width;
        void *ithplus1 = (char *)arr + (i+1) * width;
        if (cmp(ith,ithplus1) == 0) {
            // remove
            memmove(ithplus1,(char *)ithplus1+width,(nelems-i-2)*width);
            nelems--;
        } else {
            i++;
        }
    }
    return nelems;
}
```

1b) Write a comparison function on `longs` that will work for `remove_duplicates`:

```
int cmp_long(void *p, void *q)
{

    if (*(long *)p > *(long *)q) return 1;
    else if (*(long *)p < *(long *)q) return -1;
    return 0;


}
```

1c) Fill in the four blanks in `main` that will produce the following output when run from the command line:

```
$ ./remove_dup 1 1 1 2 2 3 3 3 4 7 8 8
1,1,1,2,2,3,3,3,4,7,8,8
1,2,3,4,7,8

// function declaration for function written in 1b.
int cmp_long(void *p, void *q);

int main(int argc, char **argv)
{
    int nelems = argc-1;
    long arr[nelems];

    for (int i=0; i < nelems; i++) {
        arr[i] = atol(argv[i+1]);
        printf("%ld",arr[i]);
        printf("%s",i == nelems-1 ? "\n" : ",");
    }

    int newsz = remove_duplicates(__arr____, _nelems__,

                                  _sizeof(long)_, _cmp_long_);

    for (int i=0; i < newsz; i++) {
        printf("%ld",arr[i]);
        printf("%s",i == newsz-1 ? "\n" : ",");
    }
    return 0;
}
```

Problem 2 (Floats) (17 points) (suggested time: 20 minutes)

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". Note: binary `10000000` equals `128` in decimal.

2a) What is the bit representation for **`-17`**?

1 10000011 00010000000000000000000

2b) Given the following 32-bit float binary, what is the corresponding decimal number that it represents?

```
0 01111110 10000000000000000000000
```

0.75

2c) How can you tell if a 32-bit float is normalized, denormalized, or exceptional by looking at the binary representation?

Normalized: exponent is not all 1s or all 0s

Denormalized: exponent is all 0s

Exceptional: exponent is all 1s

2d) What does the following code print?

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

int main(int argc, char **argv)
{
    float a = 0.5;
    float b = 1.0;
    float c = 2.0;
    float d = 1 << 31;
    float inf = +INFINITY; // floating point +inf

    print_equality(a + b, 1.5);
    print_equality(b - c + c, c + b - c);
    print_equality(d * d, inf);
    print_equality(d - c + c, c + d - c);
    print_equality(d + c - d, d - d + c);

    return 0;
}
```

Answer:

true
true
false
true
false

Problem 3 (Assembly) (24 points) (suggested time: 35 minutes)

3a) Given the following assembly code, re-construct the C code that produced it.

```
0x400566 <+0>:   test   %rsi,%rsi
0x400569 <+3>:   je     0x400599 <mystery+51>
0x40056b <+5>:   push   %rbp
0x40056c <+6>:   push   %rbx
0x40056d <+7>:   sub    $0x8,%rsp
0x400571 <+11>:  mov    %rsi,%rbx
0x400574 <+14>:  mov    %rdi,%rbp
0x400577 <+17>:  shr    %rsi
0x40057a <+20>:  callq  0x400566 <mystery>
0x40057f <+25>:  mov    -0x8(%rbp,%rbx,8),%rsi
0x400584 <+30>:  mov    $0x400694,%edi
0x400589 <+35>:  mov    $0x0,%eax
0x40058e <+40>:  callq  0x400430 <printf@plt>
0x400593 <+45>:  add    $0x8,%rsp
0x400597 <+49>:  pop    %rbx
0x400598 <+50>:  pop    %rbp
0x400599 <+51>:  repz retq
```

```
void mystery(long *arr, size_t count)
{
    if (__count > 0____) {            // line 1

        __mystery(arr, count / 2);_      // line 2

        printf("%lu\n",__arr[count-1]____); // line 3
    }
}
```

3b) Given the following assembly code, re-construct the C code that produced it. *Tip*: the assembly looks long, but all of the pushing and popping are necessary due to the nature of the function; make sure you note where all of the arguments are moved to at the beginning of the assembly function.

```
0x40052f <+0>:   push   %r15
0x400531 <+2>:   push   %r14
0x400533 <+4>:   push   %r13
0x400535 <+6>:   push   %r12
0x400537 <+8>:   push   %rbp
0x400538 <+9>:   push   %rbx
0x400539 <+10>:  sub    $0x18,%rsp
0x40053d <+14>:  mov    %rdi,%r13
0x400540 <+17>:  mov    %rsi,%r14
0x400543 <+20>:  mov    %edx,0xc(%rsp)
0x400547 <+24>:  mov    %rcx,%r15
0x40054a <+27>:  mov    %rdi,%r12
0x40054d <+30>:  mov    $0x1,%ebp
0x400552 <+35>:  jmp    0x400574 <mystery+69>
0x400554 <+37>:  movslq 0xc(%rsp),%rbx
0x400559 <+42>:  imul   %rbp,%rbx
0x40055d <+46>:  add    %r13,%rbx
0x400560 <+49>:  mov    %rbx,%rsi
0x400563 <+52>:  mov    %r12,%rdi
0x400566 <+55>:  callq  *%r15
0x400569 <+58>:  test   %eax,%eax
0x40056b <+60>:  jns    0x400570 <mystery+65>
0x40056d <+62>:  mov    %rbx,%r12
0x400570 <+65>:  add    $0x1,%rbp
0x400574 <+69>:  cmp    %r14,%rbp
0x400577 <+72>:  jb     0x400554 <mystery+37>
0x400579 <+74>:  mov    %r12,%rax
0x40057c <+77>:  add    $0x18,%rsp
0x400580 <+81>:  pop    %rbx
0x400581 <+82>:  pop    %rbp
0x400582 <+83>:  pop    %r12
0x400584 <+85>:  pop    %r13
0x400586 <+87>:  pop    %r14
0x400588 <+89>:  pop    %r15
0x40058a <+91>:  retq
```

```
void *mystery(void *arr, size_t nelems,
int width, int(*cmp)(void *, void *))
{
    void *x = ___arr_________ // line 1

    for (_size_t i = 1; i < nelems; i++_) { // line 2

        void *y = __(char *)arr + i * width___ // line 3

        if (_cmp(x,y) < 0_) {   // line 4

            _____x = y_____________  // line 5
        }
    }

    return ____x____; // line 6
}
```

Problem 4 (Runtime Stack) (20 points) (suggested time: 30 minutes)

Now that you have made it through CS 107, you have been offered $100,000,000 to figure out how to break into the CIA computer system. You promptly dispense with your ethics and your patriotism, and you accept the job.

You are given the following C code. You can run the code on your local computer, but you don't have access to the `get_realpw` function, so you make that function up yourself for your testing. However, you do know that the password only changes once a day. Tomorrow you need to tell a spy inside the CIA how to run the program and break in. The spy can run the program multiple times, but not in gdb.

After looking at the code, you decide to look up the four string library functions with man, because you don't recall exactly what they do (see attached).

```
 1 // file: cia_login.c
 2 #include<stdio.h>
 3 #include<stdlib.h>
 4 #include<string.h>
 5
 6 void get_realpw(char *pw) { // function made up by you
 7     strcpy(pw,"dummy password");
 8 }
 9
10 void authenticate(char *userpw)
11 {
12     char realpw[16];
13     char userpwcopy[16];
14     get_realpw(realpw);
15     strncpy(userpwcopy,userpw,16);
16
17     if (strcmp(userpwcopy,realpw) == 0) {
18         printf("Welcome to CIAnet!\n");
19     } else {
20         printf("Your password, %s, is incorrect.\n",userpwcopy);
21     }
22 }
23
24 int main(int argc, char **argv)
25 {
26     char userpw[1024];
27     printf("Password?\n");
28     fgets(userpw,1024,stdin);
29     // remove trailing newline
30     userpw[strlen(userpw)-1] = 0;
31     authenticate(userpw);
32     return 0;
33 }
```

You run gdb on the binary file after you compile it. Here is the text of the gdb session:

```
$ gdb cia_login
The target architecture is assumed to be i386:x86-64
Reading symbols from cia_login...done.
(gdb) break 22
Breakpoint 1 at 0x40073a: file cia_login.c, line 22.
(gdb) run
Starting program: cia_login
Password?
abcdefg
Your password, abcdefg, is incorrect.

Breakpoint 1, authenticate (userpw=userpw@entry=0x7fffffffe580
"abcdefg") at cia_login.c:22
22    }
(gdb) p realpw
$1 = "dummy password\000"
(gdb) p userpw
$2 = 0x7fffffffe580 "abcdefg"
(gdb) p userpwcopy
$3 = "abcdefg\000\000\000\000\000\000\000\000"
(gdb) p &realpw
$4 = (char (*)[16]) 0x7fffffffe560
(gdb) p &userpw
Address requested for identifier "userpw" which is in register $rbx
(gdb) p/x $rbx
$7 = 0x7fffffffe580
(gdb) p &userpwcopy
$5 = (char (*)[16]) 0x7fffffffe550
(gdb) x/64bx &userpwcopy
0x7fffffffe550:  0x61  0x62  0x63  0x64  0x65  0x66  0x67  0x00
0x7fffffffe558:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x7fffffffe560:  0x64  0x75  0x6d  0x6d  0x79  0x20  0x70  0x61
0x7fffffffe568:  0x73  0x73  0x77  0x6f  0x72  0x64  0x00  0x00
0x7fffffffe570:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x7fffffffe578:  0x7f  0x07  0x40  0x00  0x00  0x00  0x00  0x00
0x7fffffffe580:  0x61  0x62  0x63  0x64  0x65  0x66  0x67  0x00
0x7fffffffe588:  0x00  0x0d  0xa1  0xf7  0xff  0x7f  0x00  0x00
```

Once you reach this point, you know how to break into the CIA computer!

4a) Write your note to the spy about how to break into the CIA computer:

Run the program and type a password that is at least 16 characters long. You will see a message that says, “your password, xxxxxxxxxxxxxxxxxxx, is incorrect”, however it will list your password AND the real password. Run the program again and type the real password, and you will break in!

4b) Write a note to your boss explaining how and why your exploit works.

The

```
strncpy(userpwcopy,userpw,16);
```

line will not null-terminate the copy if it is 16 or more characters long. Therefore, the `userpwcopy` variable will not be null-terminated, and the

```
  printf("Your password, %s, is incorrect.\n",userpwcopy);
```

line will continue printing the `realpw` variable, in effect concatenating the two variables. This will print out the real password to the screen, which can be used to run the program again and gain access.

4c) Give a 1-line fix (or modification) that will make the CIA program more secure.

There are a number of different answers. One would be:

After the `strncpy(userpwcopy,userpw,16);` line, add:

```
userpwcopy[15] = 0;
```

There are other alternatives — one is to not bother using a copy of the user's password and simply to print out the original:

```
printf("Your password, %s, is incorrect.\n",userpw);
```

Problem 5 (Heap Allocator) (36 points) (suggested time: 50 minutes)

Consider a heap allocator implementation designed as follows:

- All blocks must be aligned on 4-byte boundaries, and the max request size is `INT_MAX`.
- The minimum payload size is 4 bytes, and payloads are always given multiples of 4 bytes (*e.g.*, a malloc of 10 bytes would be given a payload of 12 bytes).
- Each block has a **20-byte header**: 4 bytes (`int`) payload size, a "next" pointer (pointer to the header of the next block in the free list or NULL for last element of the list), and a "previous" pointer (pointer to the header of the previous block in the free list or NULL for first element in the list), where the pointers are for use in a doubly-linked free list (these pointers are always present in the header—not payload—but are simply not used if the block is allocated).
- Each block has a **4-byte footer**: 4 bytes (`int`) payload size.
- Because block sizes must be multiples of 4 bytes, the **rightmost (least significant) two bits of the payload size** in the header and footer are not really needed (they would always be zero). So we use the least significant bit to indicate allocated or free (1=allocated, 0=free). The next bit is only used for allocated blocks (must be 0 for free blocks), and it indicates whether this block has been the subject of a call to realloc after its malloc (1=has been reallocated, otherwise 0). Therefore, **to find the actual payload size of an allocated block, these two bits must be zeroed out.** Note: both the header and footer have allocated and realloc bits.
- There is a single explicit free list implemented as an **unsorted linked list** of free blocks.
- To maintain alignment, the header of **the first block in the heap segment starts 4 bytes after the start of the heap segment**. The entire heap after that is always part of one or more blocks (initialization of heap segment starts with one giant free block that is split over time).
- **Example 1:** An **allocated** block that was first malloced as 25 bytes, then subsequently realloced as 50 bytes would have an actual payload size of 52 bytes (closest multiple of 4 greater than or equal to 50), and the header and footer would record the payload size as 55 (two rightmost bits set to 1). The header would include unused next/prev pointers.
- **Example 2:** For a **free** block with an actual payload size of 16 bytes, the header and footer would record the payload size as 16 (two rightmost bits set to 0). The header would include valid next/prev pointers.

Assume the following global typedefs, constants, and variables have already been set up:

```
typedef struct headerT {
   int payloadsz; // FOR THIS PROBLEM, you may assume that the memory
   struct headerT *next; // layout for this struct is in the order given here,
   struct headerT *prev; // with no padding.
} headerT;

#define MIN_SIZE 4
int roundup(int size, int mult); // rounds size up to multiple of mult

static void *heapStart; /* base address of entire heap segment */
static size_t heapSize; /* number of bytes in heap segment */
headerT *free_list;     /* front of the free list */
```

(a) (4pts) Write a helper function that, given a pointer to a header or footer, reads and returns the actual payload size of that block, in bytes (i.e., returns the size with the two least significant bits zeroed out).

```
int get_size(void *curr)
{

    int mask = -1 << 2;
    return (*((int*)curr)) & mask;




}
```

(b) (4pts) Write a helper function to identify when a block is allocated. Given a pointer to a header or footer, return true if the block is allocated, otherwise return false.

```
bool is_allocated(void *curr)
{

    int mask = 0x1;
    return (*((int*)curr)) & mask;




}
```

(c) (4pts) Write a helper function to identify when a block has been reallocated since the time of its malloc. Given a pointer to a header or footer, return true if the block has been reallocated, otherwise return false.

```
bool is_reallocated(void *curr)
{

    int mask = 0x2;
    return (*((int*)curr)) & mask;









}
```

(d) (4pts) Write a helper function that, given a pointer to the header of a block, returns the address of the header of the block to its right in memory. If there is none (i.e., at the boundary of the heap), return NULL.

```
headerT *right_block(headerT *curr) {

    headerT *next = (headerT *)(((char*)curr + get_size(curr))
            + sizeof(headerT) + sizeof(int));
    if ((char *)next >= (char*)heapStart + heapSize) return NULL;
    return next;



}
```

(e) (10pts) Write a simple implementation of `myfree` given this design. Your `myfree` should do the following:

- If `ptr` is outside the range of addresses of the heap, return without doing anything. Otherwise assume ptr points to a valid payload.
- Check that the block is currently allocated (if not, return without doing anything).
- Mark the header and footer values appropriately for a free block (don't forget to set the realloc bit!)
- Insert the block at the front of the free list.
- *This simple version does not attempt to do coalesce or other advanced features.*

```
void myfree(void *ptr)
{

    ptr = (char*)ptr - sizeof(headerT); // rewind to block header
    if ((char *)ptr >= ((char*)heapStart + heapSize)) return; // past end

    // before beginning—include   offset padding

    if ((char *)ptr < ((char*)heapStart + 4)) return;

    if (!is_allocated(ptr)) return;
    int mask = -1 << 2;
    *(int*)ptr &= mask; // clear out alloc/realloc bits of header

    // clear footer
    *(headerT **)((char*)ptr + get_size(ptr) + sizeof(headerT)) = ptr;

    (((headerT*)ptr)->next) = free_list; // current head of free list is next

    // no prev since this is new head of free list
    ((headerT*)ptr)->prev = NULL;

    if (free_list != NULL) free_list->next = ptr; // new head of free list


}
```

(f) (10pts) Write a simple implementation of myrealloc given this design. Your myrealloc should do the following:

- Assume ptr points to a valid, allocated block's payload. If the realloc size is less than or equal to the current payload size, do nothing and return ptr. (you do not need to change the value of the reallocated bit in this case)
- If the block has previously been reallocated, proceed as if the request is for double the actual request size, in anticipation that there may be future myrealloc calls that will now be very fast so long as they fit within this preemptive doubling.
- Perform the realloc by calling myfree and mymalloc (do not manually do any of the work for this). Be sure to copy the caller's data for them. Even though you are the heap allocator, order these operations in a way that respects the usual convention that one does not access freed memory.
- Be sure that the allocated and reallocated flag bits are set appropriately to show this was reallocated.
- *This simple version does not attempt to look for free blocks to the right to expand into, nor does coalesce or other advanced features.*

```
void *myrealloc(void *ptr, size_t size)
{

    // rewind to block
    headerT *header = (headerT *)((char*)ptr - sizeof(headerT));

    int cursz = get_size(header);
    if (size <= cursz) return ptr;
    if (is_reallocated(header)) size *= 2;

    void *block = mymalloc(size);
    memcpy(block, ptr, cursz);
    myfree(ptr);
    int mask = 0x2;

    // rewind to header, mark realloc
    *(int*)((char*)block - sizeof(headerT)) |= mask;

    // mark footer realloc
    header = (headerT *) ((char *) block - sizeof(headerT)); //new header
    *(int *)((char *)header+get_size(header)+sizeof(headerT)) |= mask;

    return block;



}
```

Problem 6 (gcc / make) (9 points) (suggested time: 10 minutes)

6a) What does the `text` segment in an ELF binary hold?

The text segment holds the binary of the program code.

6b) What does the C Preprocessor do with a `#include` statement when it appears in a C program?

The C Preprocessor replaces the #include line with the actual text from the file being included.

6c) You have the following Makefile:

```
# Generic Makefile
# CS 107 - Winter 2018

########################## SETTINGS ############################
# (1) Compiler to use
CC=gcc

# (2) Compiler flags
CFLAGS=-g3 -std=c99 -pedantic -Wall

# (3) Name of executable
PROG_NAME=utf8

########################### RULES ##############################
# If just "make" is called, then make the program
all: $(PROG_NAME)

# Build the executable from object files
$(PROG_NAME): $(PROG_NAME).o
	$(CC) $(CFLAGS) -o $@ $^

# Build the object file from source files
$(PROG_NAME).o: $(PROG_NAME).c $(PROG_NAME).h
	$(CC) $(CFLAGS) -c $(PROG_NAME).c

# Clean up
clean:
	$(RM) $(PROG_NAME) *.o
```

You have created the `utf8.c` and `utf8.h` files, and then you type `make`. What does the command line print out? **(note: order matters! you must create the .o file first)**

```
gcc -g3 -std=c99 -pedantic -Wall -c utf8.c
gcc -g3 -std=c99 -pedantic -Wall -o utf8 utf8.o
```