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):

```c
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:

```c
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:

```c
int remove_duplicates(void *arr, size_t nelems,
                      int width, int (*cmp)(void *, void *))
{
```
1b) Write a comparison function on longs that will work for remove_duplicates:

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

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(______, ___,__, __);
    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:

\[
\text{N EEEEEEEE SSSSSSSSSSSSSSSSSSSSSSSSSS}
\]

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 \(-17\)?

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

\[
0 0111110 100000000000000000000000
\]

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

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

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

```assembly
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 (______) {                  // line 1
        ___                           // line 2
        printf("%lu\n",______);        // 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    0x400554 <mystery+69>
0x40056c <+61>:  mov    %rbx,%r12
0x40056d <+62>:  add    $0x1,%rbp
0x400570 <+65>:  cmp    %r14,%rbp
0x400571 <+66>:  jb     0x400554 <mystery+69>
0x400572 <+67>:  mov    %r12,%rax
0x400573 <+68>:  add    $0x18,%rsp
0x400575 <+70>:  pop    %rbx
0x400576 <+71>:  pop    %rbp
0x400577 <+72>:  pop    %r12
0x400578 <+73>:  pop    %r14
0x400579 <+74>:  retq
```

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

    for (__) { // line 2

        void *y = _____ // line 3

        if (__) { // line 4
            ___________________ // line 5
        }
    }

    return _______; //
} 
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).

```c
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 userpwcory[16];
14    get_realpw(rearpw);
15    strncpy(userpwcory,userpw,16);
16    if (strcmp(userpwcory,realpw) == 0) {
17        printf("Welcome to CIAnet!\n");
18    } else {
19        printf("Your password, %s, is incorrect.\n",userpwcory);
20    }
21 }
22
23 int main(int argc, char **argv)
24 {
25    char userpw[1024];
26    printf("Password?\n");
27    fgets(userpw,1024,stdin);
28    // remove trailing newline
29    userpw[strlen(userpw)-1] = 0;
30    authenticate(userpw);
31    return 0;
32 }
```
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 = 0x7fffffff5e80 "abcdefg"
(gdb) p userpwcopy
$3 = "abcdefg\000\000\000\000\000\000\000\000"
(gdb) p &realpw
$4 = (char (*)[16]) 0x7fffffff560
(gdb) p &userpw
Address requested for identifier "userpw" which is in register $rbx
(gdb) p/x $rbx
$7 = 0x7fffffff580
(gdb) p &userpwcopy
$5 = (char (*)[16]) 0x7fffffff550
(gdb) x/64bx &userpwcopy
0x7fffffff550: 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x00
0x7fffffff558: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffff560: 0x64 0x75 0x6d 0x66 0x79 0x20 0x73 0x73
0x7fffffff568: 0x77 0x77 0x77 0x6f 0x72 0x64 0x00 0x00
0x7fffffff570: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffff578: 0x7f 0x07 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffff580: 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x00
0x7fffffff588: 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:
4b) Write a note to your boss explaining how and why your exploit works.

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

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

```c
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).

```cpp
int get_size(void *curr)
{
}
```

(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.

```cpp
bool is_allocated(void *curr)
{
}
```

(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.

```cpp
bool is_reallocated(void *curr)
{
}
```
(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.

def right_block(headerT *curr):

(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.

```c
void myfree(void *ptr)
{
}
```
(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.

```c
void *myrealloc(void *ptr, size_t size)
{
}
```
Problem 6 (gcc / make) (9 points) (suggested time: 10 minutes)

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

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

6c) You have the following Makefile:

```plaintext
# 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)