Reflections

1. Explain why structs are useful.

Structs are useful because you can have “objects” basically that contain many variables and you can refer to those objects in storage. Having structs allows for this condensation of many variables into one kind of variable. You can store multiple variables basically into the struct, meaning you can refer to that struct and get whichever variable you need from that struct. Saving multiple structs allows you to keep things organized and separated out from everything.

2. When would you use binary versus text I/O?

You would use binary in a situation where you would not want someone to see the output of a program. Binary can mask the text and variables outputted from a program. There isn’t necessarily an advantage or disadvantage to using either for any situation, it is more using it at the appropriate time or as an appropriate tool. So using binary to hide prying eyes from seeing an outputted text file would be a good use for binary. Using text i/o would be useful in the scenario you want people to be able to read the outputted information on a text file. Binary can be somewhat useful when writing and reading structs with specific allocated memory sizes but same for regular text.

3. Give an example of the dangers of pointers.

Pointers can cause crashes if used wrongly. Pointers point to a value in memory and you aren’t always aware of where it is pointing to, sometimes you can have memory leaks where say a struct is leaking memory because it wasn’t allocated the correct amount of memory so it keeps leaking memory. Pointers can also be dangerous if you leave them without a specific position to point to because it can have some random memory address, and that address may have a number that can screw up your math if you use that pointer in the math. Pointers must be really taken into consideration and handled correctly to not cause program issues or memory leaking issues. Also with character arrays, not using the full array can cause major issues when writing and then trying to read back from a file as it will contain garbage to fill the rest of the array/pointer.

4. Explain the process of memory allocation and freeing.

The process of memory allocation is when you allocate a certain size of memory storage for either a variable, struct, or string etc. When you define a character array of size 64 you are specifically allocating space for 64 characters. You can malloc() to allocate memory for the size of something like a struct, that way you can write in binary

to a text file and read back from it by knowing that malloc() size. Freeing memory is necessary to stop memory leaks from occurring. When you free memory, you are essentially taking allocated memory and unallocating it allowing for those memory addresses to be used again and freed up not doing this can crash a program.

5. Explain what a constructor and destructor do in light of memory allocation and freeing, and why they're useful.

A constructor with memory allocation is very useful because if you are writing to a bin file and then reading back from that bin file, having a memory allocation size allows for you to recall that struct from a read function. A destructor allows for you to free up the memory used or allocated by a constructor that had been allocated. Destructor's and Constructors are useful in an object oriented view because you are keeping structs/objects separate and allocating specific memory sizes for those structs, then the destructors free up and un-allocate that memory taken up by the allocated struct. Constructors and destructors are basically an intro to object oriented programming and how to keep things separated and allocate the right amount of memory and free up that memory when you are done using it.

Details

6. Explain the difference between **s->a** and **a**.

s->a would be assigning a variable of a within a struct, while a alone would just be assigning a variable a. The format -> is used when referencing and making changes to a struct variable. In this case struct s and you are looking at variable a in struct s.

7. Explain what adding a value to a pointer does, and how it's different than adding a value to an integer.

When you add a value to a pointer you do not add a literal number to the pointer, you are changing the position the pointer is pointing to by the integer value. When you add a value to an integer, you are literally adding that number to the variable, such as; i+2; when i=5, this will result in i=7; When you add say 6 to a pointer, you are saying move the pointer 6 positions.

8. Explain **sizeof()** and why we often use it with **malloc()**.

Sizeof is a function that we use to find the allocated memory size of anything such as an array, a struct, an int, float, double, etc. Sizeof is used with malloc because we want to specifically allocate memory (malloc) that is the exact sizeof() whatever we are trying to allocate memory to. That is why when we create a struct, we allocate the memory size with sizeof and malloc. EX: `coord *c = malloc(sizeof(coord));`

Problems: 36 points total

*Each of these very short programs has **two things** wrong with it. Explain them, **and explain how to fix them**. Your answers should each be a paragraph or less.*

Program 1 (12 points)

```
#include <math.h>
#include <stdio.h>
#include <string.h>

struct coord_struct {
   double x;
   double y;
};

int main(void) {
   char *s;
   coord_struct c;

   printf("Enter x: ");
   fgets(s, 79, stdin);
   c.x = atof(s);

   printf("Enter y: ");
   fgets(s, 79, stdin);
   c.y = atof(s);
```

```
   printf("Coords are (%lf, %lf).\n", c.x, c.y);
   return 0;
}
```

This program has 2 simple errors, for one, when defining the struct coord_struct c; it is missing the struct before the coord_struct c; You must declare it is of type struct. The second error is also very simple, atof requires the stdlib.h library. Adding the stdlib.h library and adding struct before coord_struct c; will allow this program to run correctly.

Program 2 (12 points)

```
#include <math.h>
#include <stdio.h>
#include <string.h>

struct coord_struct {
   double x;
   double y;
};

typedef struct coord_struct coord;

int main(void) {
   char s[80];
   coord *c;

     printf("Enter x: ");
   fgets(s, 79, stdin);
   c.x = atof(s);

   printf("Enter y: ");
```

```
    fgets(s, 79, stdin);
    c.y = atof(s);

    printf("Coords are (%lf, %lf).\n", c.x, c.y);
    return 0;
}
```

Once again, this program is missing the stdlib.h library for atof. This function requires the stdlib.h library to run correctly. As well, c should not be a pointer, removing the * in front of c will fix this program. A struct declaration should not necessarily be a pointer.

Program 3 (12 points)

```
#include <math.h>
#include <stdio.h>
#include <string.h>

struct coord_struct {
    double x;
    double y;
};

typedef struct coord_struct coord;

int main(void) {
    char *s = malloc(sizeof(char) * 80);
    coord *c = malloc(sizeof(coord));

      printf("Enter x: ");
    fgets(s, 79, stdin);
    c->x = atof(s);
```

```
    printf("Enter y: ");
    fgets(s, 79, stdin);
    c->y = atof(s);


      printf("Coords are (%f, %f).\n", c->x, c->y);
    return 0;
}
```

This program has 2 very simple issues. Once again, malloc() requires the stdlib.h library. Once this library is included, the program works fine. HOWEVER, you should free the memory that you allocated after you are done with it, this will cause a memory leak… bad practice :) So add free(s); and free(c); before return 0;