Objective

The objective of this programming project is to practice using pointers and dynamic memory allocation in C.

Background

For this project, we will be using real dynamic memory allocation, via calls to malloc(). However, it is often the case that even when applications use malloc() and free(), they sometimes implement a form of intermediate caching to speed up memory management. We do this because calls to malloc() and free() can be expensive. It is cheaper to reduce calls to malloc() by pre-allocating several objects at a time, distributing parts/fractions of them out one at a time to the caller in subsequent calls, and reduce calls to free() by just adding deleted objects to a list of free objects. This is particularly true when the units of allocation are uniform, which is the case with the structures that you will work with, representing fractions of memory space. This struct should be defined in a file called frac_heap.h. The struct must have fields called sign, numerator and denominator. These should be, respectively, signed char, unsigned int and unsigned int. Typedef and struct will be used to define a fraction type.

(From this point on in this document, when we use the term "block", we will be referring to "space large enough to hold one fraction", i.e., a struct fraction-sized block of space. This is because the design you implement will work with managing any kind of collection of uniform-sized blocks.)

For this project, you will implement a system that manages a heap of fraction-sized blocks. You will use malloc() to get the space for any and all necessary blocks. However, you will not be simply calling malloc() every time the user wants a new fraction. You will implement a hybrid memory management scheme to make freeing and subsequently reallocating blocks much more efficient. It will do so by allocating space with malloc() in large chunks, big enough for several free blocks at once, which it will then manage and hand out one block at a time. When it has dispensed all the available blocks, if yet another block is requested by the caller, it will call malloc() again, once again for a chunk big enough for several blocks. So, depending on how many blocks the user ends up allocating, you might have called malloc() multiple times, getting a small array of blocks each time.

Conversely, when the user frees up fractions, your package will not actually return them to the system via calls to free(), but will instead just link each fraction-sized block into a list of free blocks to keep around so that they can be very efficiently reallocated to the user when new allocation requests come in.

At this point, it might seem like the allocation function (new_frac()) is getting complicated: should it dispense blocks that it got from a call to malloc(), or should it dispense a block from the free list that the deallocation function (del_frac()) is building up? The solution is simple: if the free list is empty, then call malloc(), and take all the blocks in the new array and add them to the free list. Then, just dispense the first item from the free list.

This scheme allows us to create as many new fractions as the user desires, up to the limits of the program's memory, which is a Good Thing. However, it also adds a complication: our collection of blocks--some allocated to the caller, the rest in our free list--will now be potentially in any of several arrays, since each call to malloc() returns a distinct array, a fraction block. We want to treat the fractions as just blocks of space, anyway, which means, we would link free blocks together using actual pointers. We will achieve this by using a C language feature called a union.

C unions
You have already used C structs, which allow you to compose a set of members into a single data structure. There is another C structure called a union. You can tell C that a structure is actually serving multiple, mutually exclusive purposes at the same time, by defining a union. The syntax for a union looks like this:

```
union foo {
    int    i;
    double d;
    char   c;
};

union foo my_foo;
```

This looks very similar to a struct definition, but with the word "struct" replaced by "union". The above example first defines a new type of union called "foo", then declares a variable of that union type called "my_foo". The syntax for accessing the members is also the same: you would access the integer member of my_foo by using "my_foo.i". However, instead of the members("i", "d", and "c" in the above example) being laid out one after the other in sequential memory locations, all of the members occupy the same space (or at least, start at the same place), and the union takes up only as much space as its largest member. This means only one of the members can be in use at any time, and you have to remember which, but it will usually be clear from context. More later on using unions in this project.

Assignment

Note: This project description deliberately avoids using C syntax. This is so you figure out how to look up the syntax of various elements of the C programming language.

For this assignment, you will work with structs that represent fractions. This struct is defined in a file called frac_heap.h, provided to you as a part of the skeleton code. The struct has fields called sign, numerator and denominator. These should be, respectively, signed char, unsigned int and unsigned int. Typedef and struct is used to define a fraction type.

Your memory allocator will be using `malloc()` to dynamically allocate chunks of 10 free blocks at a time. In other words, each call to `malloc()` will request space for an array of 10 blocks. You will only call malloc() when a request for a new fraction (via a call to new_frac()) finds there are no free blocks available. Since you're malloc()'ing a bunch of blocks each time, you will set one aside to return to the user, and place the rest into a linked list of free blocks to be used for future calls to `new_frac()`. This must be defined in the file frac_heap.c, along with another

global variable for the beginning of the free block list. Both the global variables should be declared static to give them file scope and ensure that code in other files cannot access these variables directly.

Again, in the file frac_heap.c, you must supply four functions: init_heap(), new_frac(), del_frac() and dump_heap().

init_heap() must be called once by the program using your functions before calls to any other functions are made. This allows you to set up any housekeeping needed for your memory allocator. It will just initialize an empty free list.

new_frac() must return a pointer to fraction. If the free list has any available free blocks, it removes the first one, and returns a pointer to it.
If the free list is empty, it should allocate an array of 10 new free blocks using a single call to malloc(). (Note: the free list will be empty the very first time this function is called.) It should return one of these to the caller, saving the rest on the free list.

To help the graders, each time malloc() is called, you must print out a debugging message saying: "Called malloc(%d): Returned 0x%Lx\n\n", where the %d and 0x%Lx print out the size passed to, and pointer returned from, malloc().

del_frac() takes a pointer to fraction and adds that item to the free block list. The programmer using your functions promises to never use that item again, unless the item is given to her/him by a subsequent call to new_frac(). (Note that you never give space back to the system using free().)

dump_heap() is for debugging/diagnostic purposes. It only prints out the addresses of each block on the free list. If the free list is empty, it should print out "Free list is empty". This allows you to see how your memory allocator is working.

You are provided with a header file called frac_heap.h, mentioned earlier. Any program that includes frac_heap.h should be able to use your memory allocator without any additional declarations or definitions. Remember that this file should just declare the external interface: the struct definitions and prototypes for the functions that need to be accessed by users of your functions.

You should make your declarations and definitions compatible with the sample main program: **proj6.c**, provided to you. You need to copy this file into your own directory. The main program should compile with your code without any modification. You should infer the correct function prototypes of the four functions listed above from how they are used in the main program.
In addition to this sample main program, you are provided with **six** main programs that test various features of your memory allocator (test1.c, test2.c … test6.c). You are required to compile each test separately and

Implementation Notes

You must use ONLY C! The graders will build your assignment using gcc, not g++, and your program must have the extension ".c", not ".cpp" or ".cxx". This means you must use C library functions like "printf" instead of C++ library functions like "std::cout".

Please name the file that has the memory allocator functions frac_heap.c. The provided test programs (test1.c, test2., test3.c …) fully exercise your memory allocation functions. The program proj6.c does not individually address all issues.

Make sure your code builds against proj6.c without any modifications to proj6.c Your output does not need to exactly match that of proj6.txt, depending on your allocation/deallocation strategy. However, any custom formatting messages printed to the output via standard error or standard output must match **exactly** that of proj6.txt. Guidelines for formatting to exact specification is available below.
Following is an exhaustive list of custom messages with right formatting to be included in appropriate locations within your code while printing them to standard output via printf.

When your malloc returns a NULL (Out of space)

"\nError: No more memory space left for allocation!\n"

Debugging message each time malloc is called

"\nCalled malloc(%d): Returned 0x%Lx\n\n"

When a NULL pointer is passed to del_frac

"\nError: del_frac() issued on NULL pointer.\n"

dump_heap message format

Dump of heap should begin with the following string, (One newline at start and exactly two newline characters at the end)

"\n**** BEGIN HEAP DUMP ****\n\n"

followed by the fraction block pointers that are dumped out, followed by a single newline character after each address (Four space characters in the beginning)

" 0x%Lx\n"

Followed by the footer string (One newline at start and exactly two newline characters at the end)

"\n**** END HEAP DUMP ****\n\n"

You should not have any extra fields in fraction. You should just have sign, numerator and denominator.

Your memory allocator will need the head of the free list (a pointer) to be a global variable, again declared static to give it file scope to ensure that code in other files does not access it directly. It should be declared to be a pointer to the correct type (see discussion below on unions).

The internal implementation of free list must use pointers to link the free blocks together. This is necessary, because you might have allocated more than one array over the life of the program (you only allocate arrays of 10 blocks at a time). Once you have a definition for a fraction (via typedef, provided in frac_heap.h), you would do something like the following:

```
union fraction_block_u {
    union fraction_block_u *next;
    fraction frac;
};

typedef union fraction_block_u fraction_block;
```

Here, the frac_block would very likely be exactly the same size as a fraction, since the "frac" member, would be the larger of the two members of the union. When you access the "frac" member, the union will act exactly like a fraction.
In new_frac(), you would actually treat the chain of free blocks as a chain of fraction_block unions (so obviously your free list head would be a "fraction_block *"). Assuming you removed the first element from the free list and were pointing to it with "fbp", you would not even need a cast: you could just say:

```
  return (&fbp->frac);
```

This would work because the address of any of the members of a union would be the same as the address of the union itself, but with a different associated type (in this case, "pointer to fraction" instead of "pointer to fraction_block").
Later, when the user frees up the fraction, you would cast the "fraction *" parameter to a "fraction_block *", using something like:

```
del_frac(fraction *fp) {
    fraction_block *fbp;

    fbp = (fraction_block *) fp;
    /* Now, you can access fbp->next */
```

and... Voila!... you have your fraction_block back!

Each time you need additional space in new_frac(), you must allocate a chunk of space large enough for exactly 10 new fractions.

If malloc() returns a NULL, indicating you've truly run out of memory, you should print an error message and exit.

A sample output is provided, proj6.txt. Your output does not have to look identical to this, but it gives you an idea of what should happen when you run the program.

Your del_frac() function receives a pointer parameter which is to be used for linking into the free blocks list. The del_frac() function should do some error checking and make sure that the pointer passed to it is actually pointing to an item in your global free list of fraction blocks. If it detects an error, it should display an error message and exit the program.

What to Submit

Use the UNIX submit command on the GL system to turn in your project.

You must submit the header file frac_heap.h, the implementation frac_heap.c and the test programs test1.c, test2.c, ...
In addition, submit a typescript file showing that the test programs compiled and ran.

You should also submit a README file explaining anything the graders might need to know about compiling and/or running your programs.

The UNIX command to do this should look something like:

```
submit CMSC313_Deepak proj6 frac_heap.h frac_heap.c
submit CMSC313_Deepak proj6 test1.c test2.c test3.c test4.c test5.c test6.c
submit CMSC313_Deepak proj6 typescript README
```