CMSC 313 — Spring 2017
Project 5 — Project 5: DIY Memory Allocation

Assigned	Thursday, April 6th
Program Due	Wednesday, April 12th by 11:59pm
Updates:	None yet.

Objectives

The objective of this programming project is to help familiarize with the syntax of various elements of the C programming language, including: arrays, structs and pointers.

Background

There are times in programming when you might want to do your own memory allocation. This might be the case if you only need to allocate fixed-size blocks of memory and for some reason you do not want to rely on the operating system's memory allocator (e.g., you want your program to run on several platforms that use different memory allocation methods).

A simple way to achieve this do-it-yourself memory allocation is to declare a "large" preallocated global array of blocks. Each time your program needs a block of memory, you use an item of the array instead of asking the operating system for a chunk of memory. You keep track of a list of free blocks in your array by linking the unused items together in a linked list.

You do not need to create a new data structure to link the free blocks together: simply reuse one of the fields of the array item as the "next" pointer. You also do not need to use real pointers to link the free blocks together, you can just use the array index of the next item in the list of free blocks. Thus, you can simply reuse any integer field as the "next pointer". When your program deallocates a block of memory, that block is added to the linked list of free blocks. The beginning of the free block list will be stored in a global variable. Initially, the list of free blocks is the entire array.

If your memory allocations needs are modest, doing your own memory allocation can be faster than making calls to library functions.

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 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`. You must use `typedef` and `struct` to define a `fraction` type.

Your memory allocator should use a "large" global array of `fractions`. 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. For example, this is when you can initialize your linked list of free blocks.
- `new_frac()` must return a pointer to `fraction`. This must be the address of an item in your global array of `fractions`. It should be an item taken from the list of free blocks. (Don't forget to remove it from the list of free blocks!)
- `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()`.
- `dump_heap()` is for debugging/diagnostic purposes. It should print out the entire contents of the global array and the head of the free block list. This allows you to see how your memory allocator is working.

You must also supply 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 this sample main program: `main5.c`. (The file is also available at `/afs/umbc.edu/users/p/a/park/pub/cmsc313/spring17/proj5/main5.c`) 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 should write *several* main programs that test various features of your memory allocator.

Implementation Notes

- You **must** allocate memory from your global array. You are not allowed to use `malloc()`, `calloc()`, `free()` or any variation thereof.

  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` and name your test files: `test1.c`, `test2.c`, `test3.c`, ...

- Make sure that your test programs fully exercise your memory allocation functions. The program `main5.c` does not.
- Make sure your code builds against main5.c without any modifications to main5.c Your output does not need to exactly match that of main5.txt, depending on your allocation/deallocation strategy.
- The size of your global array should be exactly 20 elements in length.
- For readability, zero out the `sign` and `numerator` fields when `del_frac()` is called. That way, when you dump out the array, it is more obvious which items of the array are free. This is assuming that you are using the `denominator` field for linking the free block list.
- You should not have any extra fields in `fraction`. You should just have `sign`, `numerator` and `denominator`.
- When you link the free blocks together, use the index of the item in the array for linking, not its address. For example, if item #18 follows item #13 in the free blocks list, then you should store the number 18 in the `denominator` field of item #13. Since `denominator` is an unsigned integer, the compiler will not complain. If you tried to store a pointer in the `denominator` field, it will be a type mismatch error. (Yes, you can force the compiler to do this by type casting, but let's not start your first C program by telling the compiler to ignore types.)
- You cannot use 0 to indicate the end of the free blocks list, since 0 is a perfectly fine index for the first item in the global array. You also cannot use -1: the field is unsigned. What can you use instead? (The sample output file provided below has that "special number" replaced with "##" to hide what we used.)
- Your `del_frac()` function receives a pointer parameter, but you really want to know the index of the array item that corresponds to that pointer. (So, you can use it for linking in the free blocks list.) Pointer arithmetic saves the day. If we have the declarations:

  ```
  int A[20], *p1, *p2, i ;
  ```

  then after the statements:

  ```
  p1 = &A[0] ;
  p2 = &A[15] ;
  i = p2 - p1 ;
  ```

  the variable `i` will have the value 15. You can think of this as a case where "pointer algebra" works, since the statements

  ```
  p1 = &A[0] ;
  i = 15 ;
  p2 = p1 + i;
  ```

will make `p2` point to `A[15]`. If we "solved" for `i`, we would get

```
   i = p2 - p1 ;
```

- Your `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 array and not some random place in memory. If it detects an error, it should display an error message and exit the program.
- If you run out of free fraction structs to allocate, then `new_frac()` should return NULL.
- Here is a sample output from the sample main program: main5.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. (The file is also available at `/afs/umbc.edu/users/p/a/park/pub/cmsc313/spring17/proj5/main5.txt`)

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 your test programs `test1.c`, `test2.c`, ...

In addition, submit a `typescript` file showing that your test programs compiled and ran.

You may optionally 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 cs313_park proj5 frac_heap.h frac_heap.c
submit cs313_park proj5 test1.c test2.c test3.c test4.c test5.c test6.c
submit cs313_park proj5 typescript README
```