# CISC 340 – HW 0.5
# An exercise in data management
**(2 points)**

## Instructions:
*This is an individual homework. All code must be developed on rusty. Submit your solution files via github classroom.*

*You should start off with the supplied template git repository. You should not modify the (file/directory) structure of the repository. Pushing your work back to the master branch on github should trigger an autograded report of your submission.*

*Developing solutions on platforms other than rusty can introduce very subtle errors (like different newline characters) that can cause your submissions to not pass the test suite.*

***I guarantee all of the programming skills you will learn from this homework will prove useful.***

1. Create the C source code file spass.c and the Makefile to compile it.

In spass.c you should:

A) Use GNU getopt so your program can use -r and -c command line flags. These flags allow the user to supply a number of rows and a number of columns, respectively. You can assume the supplied values will be reasonably small.

B) Create a struct (see Section A1) that stores three variables. The number of rows, the number of columns, and a 2-dimensional array of those dimensions.
HINT: You will need to use malloc() to make this array (see Section A2).
Use malloc() to create your struct variable on the heap.
See the section below on structs (A1).

C) Create a function that fills the two dimensional array in the struct with values. The value of an element is the product of its indices. As input, this function takes a pointer to the struct.
This function should be called in main().

D) Print the contents of the 2D array in the struct to the screen as a grid (elements separated by spaces). For example:

```
5 4 3
3 2 4
6 2 5
```

E) Free() all of your memory allocations.

HINT: If you need to debug memory accesses (segmentation faults, typically) see the section on gdb (Section A3).

## A1- Helpful section on structs:

In C, you can use structs to group variables. You can think of these as a class with nothing but public data members. For example:

```
typedef struct three_ints{
        int a;
        int b;
        int c;
} three_ints_t;  // typedefs are a way to name types whatever you want to
```

**Declaring a struct and accessing data:**

The "three_ints" struct groups three ints into a single type that I named "three_ints_t". It can be used as:

```
three_ints_t  my_three_ints;
my_three_ints.a = 5;
my_three_ints.b = 4;
my_three_ints.c  =my_three_ints.a – my_three_ints.b;  //results in 1
```

**Malloc()ing a struct and accessing data:**

If I malloc a struct I need to dereference the pointer to access the data members.

```
three_ints_t*  my_three_ints = (three_ints_t*)malloc(sizeof(three_ints_t));
(*my_three_ints).a = 5;
(*my_three_ints).b = 4;
(*my_three_ints).c  =(*my_three_ints).a – (*my_three_ints).b;  //results in 1
```

**Malloc()ing a struct and accessing data – more elegently:**

The dereference to access a data member pattern is so common the designers of C created some syntactic sugar for us programmers. This is arrow notation. Using an arrow (->, dash greater than) is the same as a pointer dereference and . to access a data member. The following is identical to the previous example, except for the use of arrow notation:

```
three_ints_t*  my_three_ints = (three_ints_t*)malloc(sizeof(three_ints_t));
my_three_ints->a = 5;
my_three_ints->b = 4;
my_three_ints->c  =my_three_ints->a – my_three_ints->b;  //results in 1
```

## A2: Dynamically allocated 2D arrays

A 2D array is really just an array of arrays. If we want to use malloc to allocate at runtime, we need to allocate a 1D array that is the first dimension (that will store 1D arrays an elements). Each of the elements (1D arrays) need to then be allocated in each element of the first array. Here is an example:

```
int x = 3;
int y = 4;
int **arr = (int **)malloc(y * sizeof(int *));
for (i=0; i<y; i++){
     arr[i] = (int *)malloc(x * sizeof(int));
}
```

```
//After allocation, the array (arr) can be accessed using [] notation
//like normal.
```

Googling “dynamically allocated 2D arrays in C” is also helpful.

## A3: A helpful section on troubleshooting memory

The GNU debugger (gdb) is extremely helpful when troublshooting memory issues (segfaults, stack smashing, etc.). The following steps should help reveal the source of problems.

1) Recompile your project with the -ggdb option. (Add -ggdb to the compilation lines in your Makefile. NOTE: You should NOT submit a Makefile with -ggdb enabled.)

2) Open your program in the debugger: $gdb ./myprogram

3) Run your program with appropriate inputs: run -f inputfile

4) GDB should then identify an exact file and line number in your program where the error is occurring. The GDB command “bt”, which stands for backtrace, can reveal the entire call sequence that led to the error. Further googling will reveal commands for more sophisticated usage, but our our class this should be sufficient.

5) At this point you should be able to intelligently trouble shoot the memory issue.