Assignment 7 — Pointers and Arrays

Due: Friday, May 27th

Large matrix operations are at the heart of many scientific and graphical computations. As a result, they are a prime target for optimization in performant, close-to-the-hardware languages such as C. This improves running time by removing bloat from the machine code, and it improves memory usage by giving programmers more control over data organization[1].

Deliverables:

GitHub Classroom: `https://classroom.github.com/a/axuuDh7P`
Required Files: `matrix.c`
Optional Files: `*.c`, `*.h`

Part 1: Ground Rules

Throughout this class, any C code you write must compile using the command:

```
>$ gcc -Wall -Werror -ansi -pedantic ...
```

Furthermore, your C programs are expected to compile and run on Cal Poly's Unix servers.

Part 2: Multidimensional Arrays as Arguments

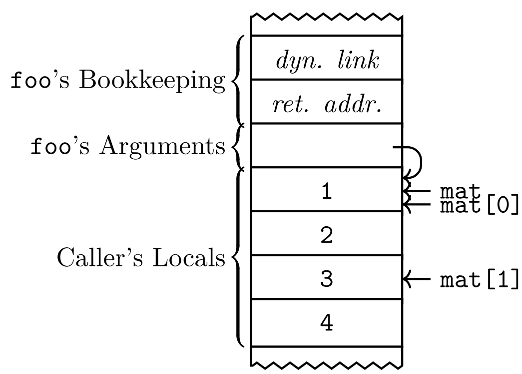
In C, multidimensional arrays are stored in *row-major order*: they are allocated as contiguous blocks of memory, one row after another. As a result, a reference to a multidimensional array remains the address of its first element, just as with a one-dimensional array, and must be passed as an ordinary single pointer.

Thus, the following code:

```
void foo(int *);

int mat[2][2] = {{1, 2}, {3, 4}};
foo(&mat[0][0]);
```

...produces the following pointers and arrays:



In this case, within the callee, the compiler is no longer aware that a multidimensional array exists, and cannot handle code such as "`mat[i][j]`" – after all, it makes no sense to index a single pointer twice.

However, as programmers who understand how the array will be stored in memory, we can perform the appropriate indexing math by hand[2]: "`mat[i * n + j]`", where "`n`" is the length of each row.

[1]It is also relatively easy to parallelize operations and transfer them to the GPU from within a C program.
[2]Though common practice, this is technically undefined behavior for local arrays. Future programs will be more portable.

Part 3: Arrays of Pointers as Arguments

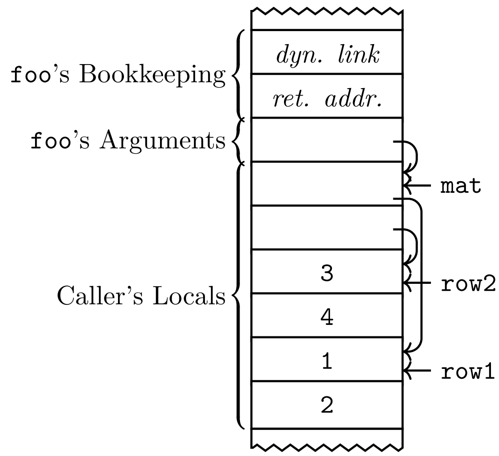
In contrast, in C, arrays of pointers are stored as separate arrays: one "outer" array, each of whose elements are pointers to the first elements of "inner" arrays. As a result, a reference to an array of pointers is the address of a pointer, making it a proper double pointer.

Thus, the following code:

```
void foo(int **);

int row1[2] = {1, 2}, row2[2] = {3, 4}, *mat[2];
mat[0] = row1;
mat[1] = row2;
foo(mat);
```

...produces the following pointers and arrays:



In this case, within the callee, the compiler *is* aware that a double pointer exists. Since array indexing is equivalent to pointer offsetting and dereferencing, the compiler can now handle code such as "`mat[i][j]`".

Part 4: Matrix Operations

Consider four common matrix operations[3]:

- *Scalar multiplication*, which multiplies each element by a constant:

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

- *Transposition*, which mirrors the elements about the diagonal:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

- *Addition*, which adds each element of one matrix to the corresponding element of another:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+4 & 2+3 \\ 3+2 & 4+1 \end{bmatrix} = \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$$

- *Multiplication*, which computes the dot product of each row in one matrix and each column in another:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 4 + 2 \cdot 2 & 1 \cdot 3 + 2 \cdot 1 \\ 3 \cdot 4 + 4 \cdot 2 & 3 \cdot 3 + 4 \cdot 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

[3]See also: `https://en.wikipedia.org/wiki/Matrix_(mathematics)#Basic_operations`

Implement these operations by completing the C functions in `matrix.c`:

- For each operation, you must implement two variations: one (e.g., "`matmul`") that accepts a two-dimensional array, and one (e.g., "`matpmul`") that accepts an array of pointers to arrays.
- You may add helper functions to `matrix.c`, and you may add additional C source or header files, if desired. However, you may *not* alter the contents of `matrix.h`.

You may assume that each matrix is square, containing exclusively integers, and that the arguments passed to each function will be valid.

Part 5: Testing

C does not have a built-in unit testing framework. The C standard library does, however, provide a single lightweight function[4], `assert`, which checks to see if a boolean expression evaluates to "true".

A minimal set of tests has been provided in `mattests.c`. If an `assert` succeeds, execution continues on; if one fails, an error message is printed and the program is terminated.

For example:

```
>$ gcc -Wall -Werror -ansi -pedantic matrix.c mattests.c
>$ ./a.out
```

These tests are by no means exhaustive, and you are encouraged — though not required — to write additional unit tests. The quality of your tests will not be assessed as part of grading.

Part 6: Submission

The following files are required and must be pushed to your GitHub Classroom repository by the deadline:

- `matrix.c` — A working C implementation of matrix operations, as specified.

The following files are optional:

- `*.c` — Any additional C source code specific to your implementation of `matrix.c`.
- `*.h` — Any additional C header files specific to your implementation of `matrix.c`.

Any files other than these will be ignored.

[4] Actually, "`assert`" is a macro — macros that take arguments are beyond the scope of this class.