Array allocation in C

Acknowledgments: This lab is based on a lab proposed by Martin Quison

Objectives:

- Allocate/free memory with malloc/free;
- Manipulating pointers/arrays;
- Create new data types with the keyword `struct`;
- Use the gdb and valgrind tools in practice.

The goal of this lab is to manipulate the functions malloc and free, and to practice debugging a program stopping at SEGFAULT. Today's excuse will be the creation of a library for creating and manipulating vectors and matrices.

★ **TP Principle.** Please download the archive: TP-allocations.tgz

You will find several programs to complete in this archive. You can compile and test all the exercises from the session with the command `make test`

Each test corresponds to a program (the program name is displayed after FAILURE or SUCCESS) which you can run independently to determine your errors. The test <`toto`> is passed if your program `toto` displays exactly the same thing as what is found in the file `toto.result`. Feel free to consult the content of the files provided to understand how the system works. If your program seems to work, but the tests indicate the opposite; compare the content of the file `toto.output` produced by your program to the file `toto.result`, which is the expected output from the test. For this, use the command `diff -u toto.result toto.output` which displays the differences line by line. It is of course possible to modify `toto.result` so that the tests no longer detect the problem, but that's not the goal ;)

To understand your segfaults, use valgrind by running: `valgrind --leak-check=full ./toto`

★ **Exercise 1: Vectors.** We wish to implement the functions necessary for managing a type vector.

Question: Complete ***vector.c*** which contains all the vector handling functions that we want to implement. The file ***vector.h*** contains all the required information: function description and type declaration vector. Once all functions are complete, type `make t11-vector` to compile, `./t11-vector` to execute the test program and make test to test if the result of the the test program is correct (if it displays the same thing as what is located in ***t11-vector.result***).

Question: Add a test to the access function, and return NULL if the requested index is outside the object's boundaries concerned. Test your modification with the program `./t12-vector-bound`

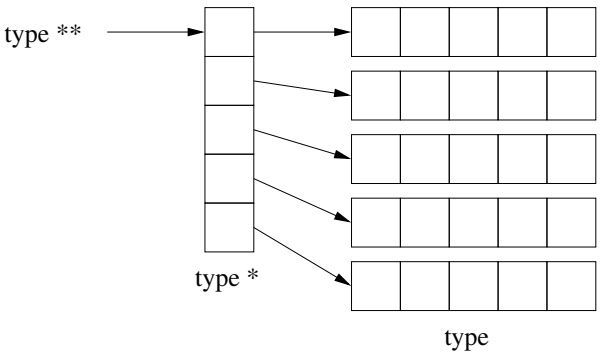
Question: In the previous question, it was assumed that accessing a non-existent entry is an error. Sometimes, it's preferable to make sure that the vector automatically grows to create on demand the non-existent entries. As it is difficult to predict the behavior desired by the user, we will configure the behavior with the `dynamic` field of the vector structure. If it is FALSE (`dynamic == 0`), we will use the behavior from the previous question: if access to nonexistent, it returns NULL. If the field is TRUE (`dynamic != 0`), the table needs to be enlarged (with `realloc()`) when the index exceeds the threshold.

Test your modification with the program `./t13-vector-realloc`.

★ **Exercise 2: Matrices.** We want to implement a type `matrix_t`, representing matrices as follows:

To allocate such a structure, we first allocate an array of pointers, whose size corresponds to the number of lines desired. Then each of the cells in this first table is a pointer to another array allocated separately to represent one of the lines. The corresponding code is given on the following page.

We then access the cells of the table as we would for a statically allocated array: with the operator `[ ]` for each of the dimensions:



Structure d'un tableau bidimensionnel dynamique

Initialization

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 10; j++)
        array[i][j] = 0;
```

Allocation of a 10*10 array of doubles

```
1 double** array;
2
3 array = malloc(sizeof(double*) * 10);
4 for (int i = 0; i < 10; i++) {
```

```
    array[i] = malloc(sizeof(double) * 10);
}
```

Same with error handling

```
double** array;

array = malloc(sizeof(double *) * 10);
if (array == NULL) { // malloc signals an error
    abort(); // kills the program by reporting an internal bug
}
for (int i = 0; i < 10; i++) {
    array[i] = malloc(sizeof(double) * 10);
    if (array[i] == NULL) {
        abort();
    }
}
```

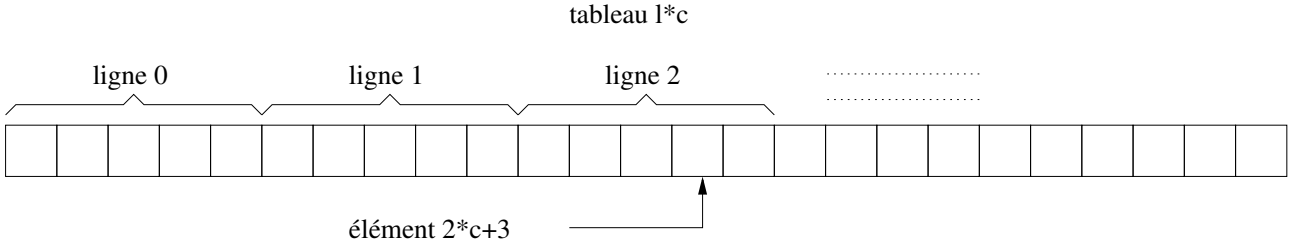
Question: Complete the file ***matrix.c*** which contains all the Matrix management functions that we want to implement. The file ***matrix.h*** contains all the necessary information: function description and declarations of type `matrix_t`. Once the functions are completed, type `make t21-matrix` to compile. `./make t21-matrix` to run the test program and `make test` to test if the result of the test program is correct.

Question: Modify your access function to return NULL if you access an entry that doesn't exist, and test your work. with `./t22-matrix-bounds`.

Question: Now modify your access function so that it increases the size of the matrix when accessing a cell that does not yet exist, provided that the matrix is dynamic. Test your work with `./t23-matrix-realloc`

★ **Exercise 3: Linear Matrices (optional)**

In practice, matrices are very rarely represented in memory as we did in the previous exercise. Instead, we uses a single large one-dimensional array. The rows of the matrices are then stored one after the other, in a contiguous way. The access function translates a pair of matrix indices into a single index in the table. This saves on the costs of indirections, as well as the cost of memory management structures associated with each `malloc` by the system. Static multidimensional arrays are automatically stored in this form in memory.



Question: Implement the type 'matlin_t' (for *mat*rix *lin*ear) in the file ***matlin.c***. As usual, the documentation is located in ***matlin.h*** and you can test your work with `./t31-matlin`.

Question: Extend your access function to return NULL when accessing a non-existent cell, and test your work with `./t32-matlin-bounds`.

Question: Extend your access function again to enlarge the matrix when accessing a non-existent cell of a matrix dynamic, then test your work with `./t33-matlin-realloc`.

★ **Exercise 4: Memory Operations (optional)** Complete the file ***memory.c*** to implement the following memory manipulation functions:

- `my_memcpy`: copying a memory area in the same way as `memcpy` (cf. man);
- `my_memmove`: copy of a memory area with Possible overlap, meaning that the area to be written on can partially overlap the area to be read (cf. `man memmove`). It is sometimes necessary to start by copying the bytes from the end;
- `is_little_endian`: returns true if the architecture target uses the convention *little endian*;
- `reverse_endianess`: returns the value passed in argument with its bytes reversed.

Check your work with `./t41-memory-cpy` `./t42-memory-move` and `./t43-memory-endianess`.