COMP26020 Part 1 Lab Assignment: Matrix Processing Library in C

The goal of this assignment is to implement a matrix processing library in C, offering various matrix operations such as transposition, matrix-matrix multiplication, scalar product, etc. To that aim you are given a header file `matrix.h` containing the definitions (prototypes) of the different functions offered by the library API, and **your task is to implement each of these functions in a single C source file**, `matrix.c`. To partially validate your implementation, you are given a test suite in the form of a program that uses the library and performs various sanity checks.

You can download an archive with the library header file, an empty `matrix.c` source file, as well as the test suite sources and supporting files here:
https://olivierpierre.github.io/comp26020/lab/comp26020-lab1.zip.

Matrix Processing Library

The `matrix.h` header defines a data structure representing a matrix and a set of functions to create and destroy matrix objects, to perform operations on matrices, and to save/load matrices on/from disk as files.

Matrix Data Structure. The data structure `matrix_t` represents a matrix. It has 3 fields: two integers recording the number of rows and columns of the matrix, and a pointer of pointer of integers (i.e. a two-dimensional array) holding the matrix's content. Creating a matrix object is a two step process: the two dimensional array must first be dynamically allocated, and then filled with the content of the matrix.

Exported Functions. The library's header exports an API made of 13 functions to be implemented in the source file. They can be classified into the following categories:

- *Matrix allocation, initialisation, and destruction*: `matrix_allocate` is used to allocate a matrix object given its dimensions, and `matrix_free` deallocates an object. `matrix_init_rand`, `matrix_init_n`, `matrix_init_zeros`, and `matrix_init_identity` respectively initialise the content of an already allocated matrix object with random integers, a fixed integer *n*, zeros, and content corresponding to an identity matrix[1].
- *Matrix operations*[2]: `matrix_equal` checks if two matrices are equal. `matrix_sum` performs the sum of two matrices. `matrix_scalar_product` multiplies a matrix by a scalar. `matrix_transposition` transposes a matrix, and `matrix_product` performs the product of two matrices.
- *Saving/Loading matrices as files*: `matrix_dump_file` saves a matrix object into a file, and `matrix_allocate_and_init_file` allocates a matrix object and loads its content from a file. The format of such files is discussed below.

You should check the comments in `matrix.h` for more details about the expected behaviour of each function, in particular what values functions should return, and whether they operate on matrix objects that need to have been previously allocated or not.

You are free to develop additional functions within `matrix.c`. For example, a `print_matrix` function displaying a matrix on the standard output could be useful for debugging purposes.

On-Disk Matrix File Format. The matrix file format is text-based: each row of the matrix is represented as a line in the file, with the elements constituting the row being separated by spaces on that line. You can find examples of matrix files in the `matrix-samples/` directory of the archive.

Ideally, the matrix file loading function should be able to process files adhering to a non-strict definition of that format, e.g. with variable amounts of spaces between the numbers on a line, possible empty lines to be ignored, etc.

---

1 https://en.wikipedia.org/wiki/Identity_matrix
2 https://en.wikipedia.org/wiki/Matrix_(mathematics)#Basic_operations

Test Suites

Basic Test Suite. To test your code you are given a basic test suite implemented in a source file, `basic-test-suite.c`. It is a C program (it contains a `main` function) that includes `matrix.h`. It creates matrix objects and tests functions of the library's API by performing sanity checks. This program uses the *Unity*[3] test framework and requires 3 additional source files to be compiled: `unity.c`, `unity.h`, and `unity_internals.h`. The test program can be compiled as presented in the lecture concerning modular compilation.

The test suite should drive your implementation: you are encouraged to investigate the content of `basic-test-suite.c` to understand how the library's functions to be implemented should behave. The test suite is made of a series of test cases, each invoked from `main` with the `RUN_TEST()` macro and defined in a self-contained function. A test case will fail when one of the assertions it contain fails. For example the assertion `TEST_ASSERT_TRUE( <condition> )` fails when `condition` evaluates to false (`0` in C). See Unity's website[4] for a reference on the other assertions used in the test suite. To complete the assignment, you do not need to understand the content of `unity.c`, `unity.h`, and `unity_internals.h`. These files implement the Unity engine and are rather complex.

Advanced Test Suite. The basic test suite is not fully comprehensive and, although passing all the tests it contains means that a good chunk of the assignment has been accomplished and that the implementation seems functional, it does not mean that everything is perfect. When marking, in addition to the basic test suite, an extended one will be used, for which you do not have access to the tests' details. This suite will include additional functional tests, and will also check the robustness of your implementation against mistakes made in the usage of the library API. To prepare for this advanced test suite, try to reason about:

- Possible functional tests missing in the basic test suite;
- How the library may be used in a C program and what programming mistakes could be made in the usage of its API (e.g. trying to allocate a matrix with negative dimensions), as well as what runtime-level issues could break assumptions you made when implementing certain functions (e.g. trying to load a matrix from a file of wrong format.)

Here, being robust means that your implementation should make efforts to detect API/runtime misuses and take proper actions when it happens.

Deliverables, Submission & Deadline

There is a single deliverable: the completed `matrix.c` file. The submission is made through the CS Department's Gitlab. You should have a fork of the repository named "*26020-lab1-s-matrix_<your username>*". Submit your deliverable by pushing the file on the default (main) branch and creating a tag named `lab1-submission` to indicate that the submission is ready to be marked. **Make sure you push to that precise repository and not another one, and to tag your submission properly**, failure to do so is likely to result in a mark of 0 for this exercise.

You do not need to submit the library header file or any of the test suite's files. This means that, when working on the assignment, although you can edit these files for debugging purposes, upon submission your code should eventually work with unmodified versions of these files.

The deadline for this assignment is Friday 14/11/2024 (14th of November) 2pm London time.

Marking Scheme

The exercise will be marked out of 10, using the following marking scheme:

- The program is functional and passes the basic test suite /4
- The program passes the advanced test suite /5
- The program follows the good C programming practices covered in the course regarding dynamic memory allocation and usage of the C standard library functions /1

3 https://github.com/ThrowTheSwitch/Unity

4 https://github.com/ThrowTheSwitch/Unity/blob/master/docs/UnityAssertionsReference.md