Project 1: Object-Oriented Matrix Operations

Due Date: 2/10 by 11:59p

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

Aims of This Project

The aims of this project are as follows:

- To provide experience with writing a non-trivial C program.
- To provide familiarity with the tools used to build C programs under Linux.
- To get experience with advanced C features.

Background

This project will require you to implement a crude approximation to object-oriented programming in C. Specifically, you are required to provide alternate implementions of an interface involving some simple matrix operations, including matrix multiplication:

Abstract Matrix Implementation
: This implementation will implement those matrix operations which can be implemented in terms of other operations without knowledge of the concrete implementation. It will leave unimplemented those operations which depend on the concrete implementation. Hence this implementation will be abstract.

Dense Matrix Implementation
: This implementation will store all the matrix entries in contiguous memory locations in normal row-major order. It will inherit operations from the abstract implementation and implement only the operations not implemented in the abstract implementation.

Smart Multiplication Matrix Implementation
: The classical matrix multiplication algorithm accesses entries in the multiplier by column. This does not play well with caches as successive entries in a column of a large matrix may well be in different cache lines. These performance problems can be alleviated by transposing the multiplier before multiplying and changing the multiplication algorithm to use this transposed matrix suitably. This implementation will inherit operations from the dense matrix implementation and override only the matrix multiplication operation.

Note that since the classical matrix multiplication algorithm is `O(n^3)` and cache sizes of modern machines are large (the L3-cache may be several mega-bytes). Nevertheless, it is possible to observe speedups of around 20% on the `remote.cs` machines for matrices of around 1000x1000.

C11 allows anonymous nested `struct`'s and `union`'s. For example, C11 allows:

```
typedef struct {
  enum { CIRCLE, RECT } type;
  double x, y; //origin
  union { //anonymous union
    struct { //anonymous circle struct
      int radius;
    };
    struct { //anonymous rectangle struct
      int width, height;
    };
  };
} Shape;

Shape shapes[] = {
  { .type = CIRCLE, .x = 10, .y = 10, .radius = 10, },
  { .type = RECT, .x = 100, .y = 10, .width = 10, .height = 5, },
};
```

Pre-C11 did not allow anonymous fields and the above struct had to be declared as:

```
typedef struct {
  enum { CIRCLE, RECT } type;
  double x, y; //origin
  union {
    struct {
      int radius;
    } circle;
    struct {
      int width, height;
    } rect;
  } u;
} Shape;

Shape shapes[] = {
  { .type = CIRCLE, .x = 10, .y = 10, .u.circle.radius = 10, },
  { .type = RECT, .x = 100, .y = 10, .u.rect.width = 10, .u.rect.height = 5, },
};
```

This standard C11 feature allows anonymous fields whose types are defined inline within the struct but not for types which are defined elsewhere. The project uses a non-standard feature available in current versions of gcc and clang which allows anonymous fields even for types which are not defined inline. This feature is triggered using the `-fms-extensions` command-line option.

This extension is useful to extend types. Consider functions for shapes.

```
typedef struct {
  double (*area)(const Shape *s);
  double (*perimeter)(const Shape *s);
} ShapeFns;

typedef struct {
  ShapeFns;  //anonymous
  double (*radius)(const Shape *s);
```

```
} CircleFns;

typedef struct {
  ShapeFns; //anonymous
  double (*width)(const Shape *s);
  double (*height)(const Shape *s);
} RectFns;

RectFns rectFns = { ... };

Shape *rect = ...
double area = rectFns.area(rect); //direct access to ShapeFns field
```

This feature can be used for extending both interface types (like the above) as well as data types.

Project Specification

The ./files/matrix.h specifies an interface for matrix operations. You will need to provide 3 implementations for this interface:

`abstract_matrix.c`
: This file should provide operations which can be implemented in terms of other matrix operations without knowledge of the concrete matrix class. For example, `transpose()` can be implemented using `getNRows()`, `getNCols()`, `getElement()` and `setElement()`. It should provide access to a single static copy of the interface using the function `getAbstractMatrixFns()` (whose specification is contained in ./files/abstract_matrix.h).

`dense_matrix.c`
: This file should implement a concrete matrix with all entries stored in contiguous memory locations in row-major order using a flexi-array. It must implement the functions specified in ./files/dense_matrix.h. It must inherit the operations defined in `abstract_matrix.c` and define all the missing operations.

  All instances of dense-matrix must share the same `MatrixFns`.

`smart_mul_matrix.c`
: This file should implement a concrete matrix with all entries stored in contiguous memory locations in row-major order. It must implement the functions specified in ./files/smart_mul_matrix.h. It must inherit the operations defined in `dense_matrix.c` but override the `mul()` operation to use the transpose of the multiplier matrix in an effort to improve cache performance.

  All instances of smart-mul-matrix must share the same `MatrixFns`.

On completion, your project should link with the provided ./files/main.c to produce a `prj1` executable, which provides the following functionality:

1. Test the `mul()` and `transpose()` operations on predefined matrices for multiple combinations of dense and smart-multiplication matrix implementations.

```
./prj1 --predefined-tests
```

2. Test the `mul()` and `transpose()` operations on random matrices for multiple combinations of dense and smart-multiplication matrix implementations.

```
./prj1 --random-tests
```

3. Run performance tests measuring the runtime (in clock ticks) of the `mul()` operation on random matrices for multiple combinations of dense and smart-multiplication matrix implementations.

```
./prj1 --perf-matrix-size 1000
```

All combinations of the above options can be specified. (1) and (2) will terminate silently if all tests are successful. The tests can be printed out by specifying the `--output` or `-o` option. (3) will multiply a random square matrix of the specified dimension by itself and print out the user and system time in clock ticks.

Your executable should not produce any output other than that generated by the provided `main.c`. In particular, all debugging statements should be removed or deactivated before project submission.

Provided Files

The ./files directory contains the following:

Makefile
: This makefile provides the following targets:

  `prj1`
  : This will build the `prj1` executable.

  `clean`
  : This will clean out all generated files.

  `submit`
  : This will build the required `prj1.tar.gz` archive.

  Simply typing `make` will build the `prj1` program, typing `make clean` will remove all generated files and typing `make submit` will create a `prj1.tar.gz` compressed archive which can be submitted.

  You may edit this file if you choose to use a different organization for your project. When editing, watch out for tabs (the first character of any command-line **must be a tab character**).

README
: A template README; replace the `XXX` with your name, B-number and email. You may add any other information you believe is relevant to your project submission.

main.c
: A main program which allows you to run predefined and random functional tests for the `mul()` and `transpose()` operations, as well as run performance tests for the `mul()` operation. You should not change this file.

test.data
: This file contains test data for the predefined and random tests. It is `#include`'d by `main.c`. Pulling the test data out of `main.c` makes it easy for the TA to test your code by replacing your submitted `test.data` with one containing his tests before compiling your code.

./files/abstract_matrix.h, ./files/dense_matrix.h(./files/smart_mul_matrix.h
: Partial specifications for the matrix implementations. These `.h` header files are complete (but can be modified by you if necessary).

./files/abstract_matrix.c, ./files/dense_matrix.c, ./files/smart_mul_matrix.c
: These `.c` files are highly incomplete and need to be changed to implement your project. However, they are complete enough so that you can compile the project as-is and produce an executable complete enough to provide a usage message.

LOG
: A log of compiling and running this project.

The cs551 Library

The provided `main` driver uses some memory allocation and error reporting routines from the cs551 library. You should not need to use this library in your code since the matrix API reports all errors via error codes. However, you are welcome to use this library if necessary; the specs are in ../../include, the source code in ../../src/libcs551 and the precompiled library in ../../lib.

The provided Makefile is setup to link with this library.

Since the library is dynamic, the linking happens when (and after) you start your program. Hence when the program starts up, you need to let the system know where it can find this library. This can be done by setting the `LD_LIBRARY_PATH` environmental variable to contain the directory containing the library. This should already be setup in your environment.

Please use the command `echo $LD_LIBRARY_PATH` to check whether this environmental variable is setup properly for you. If not, please use the appropriate command below before running your program.

Under a `sh`-derived shell like `ksh` or `bash`, you would use

```
$ export LD_LIBRARY_PATH=$HOME/cs551-17s/lib:$LD_LIBRARY_PATH
```

whereas with a `csh`-based shell like `tcsh`, the syntax you would use is:

```
% setenv LD_LIBRARY_PATH $HOME/cs551-17s/lib:$LD_LIBRARY_PATH
```

Hints

You may choose to follow the following hints (they are not by any means required). They assume that you are using the project structure supported by the provided Makefile,.

For debugging you can use gdb or a gui frontend ddd. A cruder option is to simply add `fprintf(stderr, ...)` statements to your code; a slightly better option is to use the TRACE macro provided in the cs551 library.

1. Review the OO `shapes` program discussed in class. Understand the ./files/matrix.h specification for the matrix operations.

2. Decide which of the matrix operations can be implemented without knowing the concrete details of the matrix representations. Add these functions to ./files/abstract_matrix.c. Define a static `MatrixFns` structure and add to it the functions you implemented. Return this structure via `getAbstractMatrixFns()`.

3. Decide on a representation for your dense matrix (you will need to store the number of rows, number of columns and the elements; the specs require you to use a flexi-array for the latter).

   Implement the operations which were not implemented in `abstract_matrix.c`. Note that if a matrix has `nCols` columns, then the offset of the `[i][j]`'th entry relative to the start of the elements memory area is `i*nCols + j`.

   Define a static `DenseMatrixFns struct`; you can statically initialize this structure to contain the functions you defined in this file. Unfortunately, since C restricts static initializers to constant data (not expressions), it is not possible to initialize the operations which are to be inherited from the abstract matrix implementation. Instead, use a static boolean flag to ensure that this initialization is performed (using `getAbstractMatrixFns()`), the first time the `newDenseMatrix()` constructor is called.

   You can test your code at this point using the provided `main.c` driver by commenting out the `"smartMulMatrix"` line (around line 32) in main.c.

4. Your smart-mul-matrix can use the same representation as your dense-matrix. In fact, it can inherit everything from dense-matrix except for the `mul()` implementation which needs to be reimplemented to use the transpose of the multiplier.

   Set up your `SmartMulMatrixFns` the same way you set up `DenseMatrixFns` in (3), with the inheritance from `DenseMatrixFns` being done the first time the `newSmartMulMatrix()` constructor is called.

   You should now be able to link and test your program with both concrete matrix implementations.

5. Memory leaks can be a major problem with C programs. This can be avoided by getting into the habit of using memory debugging tools. One such tool is valgrind; you can run it by simply preceeding the normal command-line used to run your program with the command `valgrind`. If the command reports problems, then it suggests the options you can use to debug the problems further.

6. Test and review your code until it meets all requirements.

Submission

You will need to submit a compressed archive file `prj1.tar.gz` which contains all the files necessary to build your `word-count` executable. Additionally, this archive **must** contain a `README` file which should minimally contain your name, email, the status of your project and any other information you believe is relevant.

If you are using the suggested project structure, then the provided Makefile provides a `submit` target which will build the compressed archive for you; simply type `make submit`.

Note that it is your responsibility to ensure that your submission is complete so that simply typing `make` builds the `prj1` executable. To test whether your archive is complete, simply unpack it into a empty directory and see if it builds and runs correctly.

Submit your project using the submission link for this project, under **Projects** in Blackboard for this course.