Homework #5 (120 pts)

ECS30 / UC Davis / Rob Gysel / Winter 2016

LOGISTICS

1. Turn your programs in on SmartSite by **Friday, 3/11/16 11:55pm**.
2. Make sure your program compiles and runs as intended on the CSIF machines.
3. Be careful with your submission. We will no longer re-grade submissions errors.
    a. Double-check your filenames, they must match the filenames listed in this document.
    b. Do not retrieve your code from the CSIF by cutting and pasting. This can easily lead to source that, for one reason or another, will not compile.
    c. **Double-check your submitted source code**. Download your SmartSite submission, upload it to the CSIF (using sftp, scp, winftp, or a similar file transfer program), and compile it.

GRADING

- Grading is done by matching test cases.
    - The code you submit on SmartSite must compile and have the correct file names in order for us to run the test cases. Failure to do so will result in a 0 score.
    - Your program's output must match the output of the supplied executables to pass a test case. Use the `diff` command to compare my output from your output, like this:

```
robs_executable < test.txt > robs_test_result.txt
your_executable < test.txt > your_test_result.txt
diff robs_test_result.txt your_test_result.txt
```

      Skeleton files (source, header, and driver files), test files, and sample executables are on the CSIF at:

```
/home/rsgysel/ecs30/hw5
```

- Comments: 10 pts
    - Each program needs to start with the following comments:

```
/*
```

```
 * filename.c
 * your name and student ID
 * A brief summary of what your program does (2-10 lines; no more
no less)
 */
```

- Each function (in both header and source files) must be commented to describe its inputs (if any - omit this section if there are no inputs), outputs (if any - omit this section if the return type is void), along with a summary of what it does. Some examples:

```
/*
 * Input:
 *    int x, describe what x is
 *    char c, describe what c is
 * Summary:
 *    This is what my foo does.
 */
void foo(int x, char c);

/*
 * Input:
 *    int x, an integer to find the square root of
 * Output:
 *    Returns square root of x as a double
 * Summary:
 *    Calculates the square root of x using Newton's method
 */
double square_root(int x);
```

- Style: 10 pts
  - Your code must be legible. Indenting is a must. Try to use meaningful variable names. Indentation will be checked using the python script located here. Every 5 errors will result in -1, up to -10. Usage:

```
python c_indentation_checker my_source_file.c
```

- Programs: 100 pts total, see below.

TEST CASES & EXECUTABLES

Test cases and executables for this homework are at:

```
/home/rsgysel/ecs30/hw5
```

See Homework 1 for a reminder of how to test your program against test cases. If no test cases are given, use the executable to check your work as follows:

1. Come up with your own test case and save it to a file.
2. Redirect your test case as input to your program, then redirect the output a file. Do something similar for my program.
3. `diff` the files created in step 2. If you are having trouble seeing the difference (maybe it's whitespace?), use `cmp` instead.

LIBRARIES

You may only use the libraries `stdio.h`, `stdlib.h`, `ctype.h`, `string.h`, and `math.h`. You are not required to use all of these libraries.

SUBMISSION CHECKLIST

- ❏ All files commented and indented
  - ❏ Student Name & ID at top of each file
- ❏ Testing:
  - ❏ `coordinate_driver`
  - ❏ `binary_matrix_driver`
  - ❏ `list_driver`
  - ❏ `conways_game_of_life_driver`
  - ❏ indentation script (forthcoming)
  - ❏ comment script (forthcoming)
- ❏ Files to turn in on SmartSite (Upload them, then double-check your submission by downloading and compiling it):
  - ❏ `coordinate.c coordinate.h`
  - ❏ `binary_matrix.c binary_matrix.h`
  - ❏ `list.c list.h`
  - ❏ `conways_game_of_life.c conways_game_of_life.h`
  - ❏ `Makefile`

PART #1: coordinate.h, coordinate.c (15 pts total)

Implement the functions in the source file `coordinate.c`. The header file `coordinate.h` has already

been written for you. You only need to modify `coordinate.c`. Each function in `coordinate.c` looks like:

```
/* Delete this comment and implement this function
Coordinate ConstructCoordinate(int x, int y) {
}
*/
```

As stated in the code, delete the comments and implement the function.

To test your code, use `coordinate_driver.c` to test it. First compile `coordinate.c` and `coordinate_driver.c` to produce object files by using `gcc` with the `-c` flag. Then compile the object files and name your executable `coordinate_driver`. Read the comments of `coordinate_driver.c` to see what it does, and use coordinate_driver to test your implementation. These tests are not exhaustive, and are only meant to *start* your testing and to test integration with the rest of the code.

Points:

- (5 pts) `ConstructCoordinate` implementation
- (5 pts) `IsNeighbor` implementation
- (5 pts) `SwapCoordinates` implementation

PART#2: binary_matrix.h, binary_matrix.c (40 pts total)

Implement the functions in the source file `binary_matrix.c` and complete the header file `binary_matrix.h`. You must modify both `binary_matrix.c` and `binary_matrix.h`.

First, implement the type `BinaryMatrix` as a struct in `binary_matrix.h` with the following elements:

```
1. int num_rows;
2. int num_cols;
3. int** data;
```

The entry `data[i][j]` is the i^{th} row and j^{th} column of data. Each entry is a 0 or a 1.

As stated in the code, delete the comments and implement the functions in `binary_matrix.c`, and add the prototypes to `binary_matrix.h`. The functions you will need to implement are:

- (10 pts) `BinaryMatrix* ConstructBinaryMatrix(int num_rows, int num_cols);`
    - Dynamically allocates a new `BinaryMatrix` with `num_rows` rows and `num_cols` cols. Each entry of the matrix should be 0.
    - Check that `num_rows` and `num_cols` are greater than 0, and if not, print the error message

`Error in CreateMatrix: number of rows and columns must be positive`

ending with a newline and `exit` the program.

- (10 pts) `void DeleteBinaryMatrix(BinaryMatrix* M);`
  - Deallocates the memory used for `*M`.
- (5 pts) `void UpdateEntry(BinaryMatrix* M, int row, int col, int content);`
  - Updates the (`row`, `col`)th entry of `*M` with `content`.
  - Check that `row` and `col` are valid indices for `*M`, and if not, print the error message `Error in UpdateEntry: index out of bounds` ending with a newline and `exit` the program.
  - Check that `content` is a 0 or a 1, and if not, print the error message `Error in UpdateEntry: content must be 0 or 1` ending with a newline and `exit` the program.
- (5 pts) `int IsMatrixIndex(BinaryMatrix* M, int row, int col);`
  - Returns true if `row` and `col` are valid indices for `*M` and false otherwise.
  - Check that `M` is not `NULL`, and if it is, print the error message `IsMatrixIndex Error: NULL parameter passed` ending with a newline and exit the program.
- (10 pts) `void PrintMatrix(BinaryMatrix* M);`
  - Prints the entries of `*M`, one row at a time.

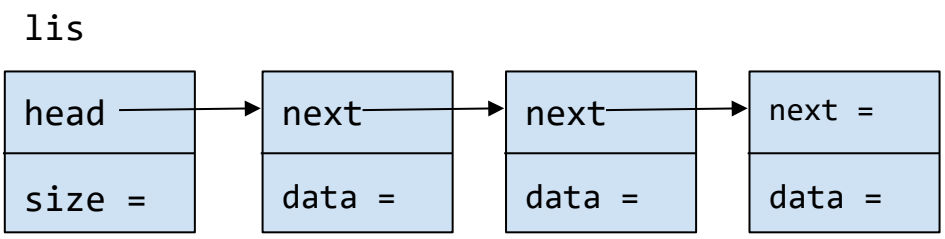
You may use `binary_matrix_driver.c` to test your code. Compile as you did with the driver in part 1. Read the comments of `binary_matrix_driver.c` to see what it does, and use `binary_matrix_driver` to test your implementation. These tests are not exhaustive, and are only meant to *start* your testing and to test integration with the rest of the code.

PART#3: list.h, list.c (10 pts total)

Implement the function `PrintList` in the source file `list.c`. You only need to modify `list.c`. Its prototype is:

`void PrintList(List* list);`

`PrintList` prints every `Coordinate` in `list`, one line at a time, starting with the `Coordinate` at the `head` node. For example, if `list` has data:



Then the output should be:

```
Coordinate: (3, 1)
Coordinate: (0, 0)
Coordinate: (2, 5)
```

This is the only function you need to modify, and is worth 10 points. You can test your code with list_driver.c.

PART#4: conways_game_of_life.h, conways_game_of_life.c (25 pts total)

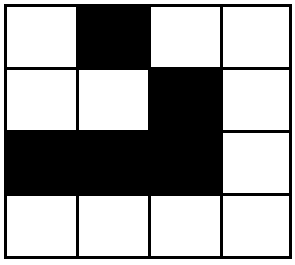
Implement the functions in the source file `conways_game_of_life.c` and write the header file `conways_game_of_life.h` (you must write all of `conways_game_of_life.h` from scratch). You must modify the file `conways_game_of_life.c` and create the file `conways_game_of_life.h`. Before discussing which functions to implement, we introduce Conway's game of life.

Conway's game of life is not a "game" as in a video game, it is a type of machine called a *cellular automata* that works as follows. The game takes place on a 2-dimensional infinite grid of squares called *cells* (we will use a finite grid, and for simplicity we will not worry about what occurs on the boundary). Each cell is *alive* or *dead*. The game moves from one generation to the next (one "game board configuration" to the next) according to the rules listed below. It starts in an initial configuration called a *seed*. Check out a web-based simulation here (the yellow figure is a *glider* - see `glider_seed.txt`).

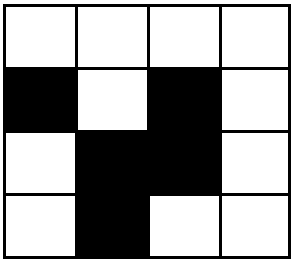
Cell birth, death, and survival rules:

- A cell is *born* in the next generation if it was dead and had three neighbors in the previous generation.
- A cell *survives* in the next generation if it was alive and had two or three neighbors in the previous generation.
- A cell *dies due to overcrowding* in the next generation if it was alive and had four or more neighbors in the previous generation.
- A cell *dies due to starvation* in the next generation if it was alive and had zero or one neighbors in the previous generation.

For example, if the first generation (defined by the seed) is:



The second generation would be:



The functions you will need to implement are:

- (5 pts) `BinaryMatrix* FirstGeneration(int num_rows, int num_cols, List* seed_cells);`
    - Creates a BinaryMatrix representing the first generation on a `num_rows` by `num_cols` grid. The first generation is described by the coordinates in `seed_cells`.
- (10 pts) `BinaryMatrix* NextGeneration(BinaryMatrix* generation);`
    - Creates a BinaryMatrix representing the next generation after `*generation`.
- (5 pts) `int Live(BinaryMatrix* generation, Coordinate coord);`
    - Returns true if `coord` describes a live cell of `*generation`.
- (5 pts) `int LivingNeighbors(BinaryMatrix* M, List* neighbors)`
    - Returns the number of cells described by a `coordinate` in neighbors.

Test your program by running `conways_game_of_life_driver` to see if it functions as you expect. You can also check your results with the simulator above.

PART#5: Makefile (10 pts total)

Create a makefile to compile your project. Your makefile must have the following targets:

- `all`
    - Creates the executables for every driver
- `clean`
    - Deletes all `*.o` object files and `*_driver` executable files
- `binary_matrix_driver`
    - Builds the `binary_matrix_driver` executable
- `coordinate_driver`
    - Builds the `coordinate_driver` executable
- `conways_game_of_life_driver`
    - Builds the `conways_game_of_life_driver` executable
- `list_driver`
    - Builds the `list_driver` executable
- `binary_matrix.o`
    - Builds object file from `binary_matrix.c`

- `coordinate.o`
    - Builds object file from `coordinate.c`
- `conways_game_of_life.o`
    - Builds object file from `conways_game_of_life.c`
- `list.o`
    - Builds object file from `list.c`