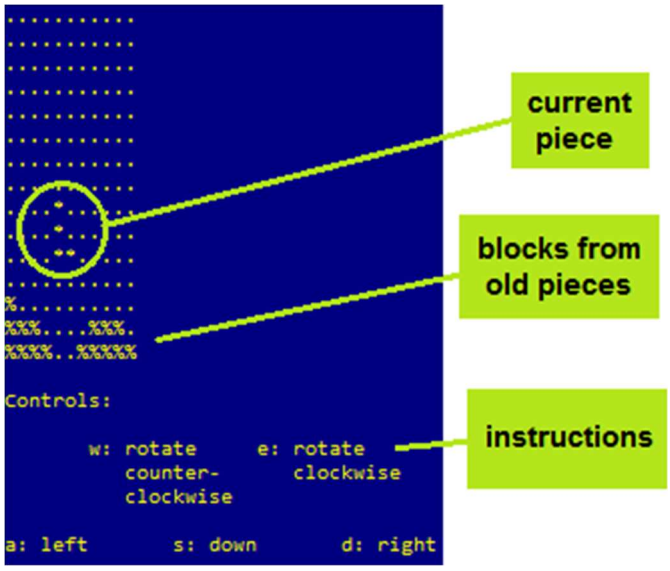
ECE220: Computer Systems & Programming **Fall 2020 ZJUI**
Machine Problem 6 **due: Saturday 7 November, 11:59:59 p.m.**

Blocky: A Game of Falling Blocks



Your task this week is to implement a game of falling blocks. The game is similar to some you may have seen and played in the past, but uses dazzling Text Graphics™!

In particular, you must write C subroutines that perform a variety of functions for the game, such as clearing the board, inserting pieces, moving pieces, rotating pieces, and removing full rows.

The objective for this week is for you to gain experience with using multi-dimensional arrays to represent information in C.

Background

The screenshot above is taken from the animated version of the game, which you can play once you have completed and debugged your code. Until that work is done, you can use the full text mode for debugging, in which each command must be followed by the **`<Enter>`** key, avoiding any time pressure but making for a somewhat less enjoyable game.

In the game, pieces of various shapes appear at the top of the game board. Each piece consists of four blocks, each represented by one ASCII character in the display. The pieces can be moved left and right, and can be rotated clockwise and counter-clockwise. The pieces can also be moved downward.

Pieces can only move within the board, and only into empty areas. If the player attempts to move or rotate a piece so as to make it leave the board, or so as to move the blocks in the piece over blocks left from previous pieces (see the bottom of the board in the screenshot), the move simply fails.

If a downward move fails, the current piece is fixed in place, which is shown by the piece's blocks changing from asterisks (`'*'`) to percent signs (`'%'`). Any rows filled with old blocks are then removed from the board, and any rows above them move downward in the board (one space per row removed, so any gaps remain).

A new piece is then added at the top of the board, and play continues.

If the board does not have space for a new piece, the game is over.

Pieces

Your program will consist of a total of three files:

`mp6.h` This header file provides type definitions, function declarations, and brief descriptions of the subroutines that you must write for this assignment. **You should read through the file before you begin coding.**

`mp6.c` The main source file for your code. A version has been provided to you with full headers for each of your subroutines. You need merely fill in the body of each subroutine.

A third file is also provided to you:

`main.c` A source file that interprets commands, calls your subroutines, implements some game logic, and provides you with a few helper functions (described later). You need not read this file, although you are welcome to do so. You may want to read the headers of the helper functions before using them.

The Task

You must write a total of 11 C subroutines in this assignment. Don't panic: the total amount of code needed in my version was only 110 extra lines! I recommend implementing the functions in the order described in this document, which allows you to perform some amount of debugging as you implement rather than trying to debug everything at once.

Step 1: Empty the board.

Implement the C function

```
int32_t empty_board (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH]);
```

in **`mp6.c`**. The function must fill the board **`b`** with **`SPACE_EMPTY`**. The height and width of the board are given by the preprocessor constants **`BOARD_HEIGHT`** and **`BOARD_WIDTH`**. The function should return 1 on success, or 0 on failure (which shouldn't happen once you implement the function, but the caller does check). After implementing the function, you can compile the program and execute it. Instead of being told that emptying the board failed, the code should now tell you 'good game' and terminate (not crash—a crash means that you have a bug!).

Step 2: Print the board.

Implement the C function

```
void print_board (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH]);
```

in **`mp6.c`**. The function must print the board **`b`** to the monitor. Each index of the first array dimension (the height) should map to a separate row on the screen, and each index of the second array dimension (the width) should map to a separate character. How each type of space should appear on the monitor is described in the function header in the code, and matches the illustration on the first page of this specification. Terminate each line with a newline character ("\n"). After implementing this function, you can compile the program and execute it. Now the program should print an empty board before terminating.

Step 3: Test and place pieces.

This step requires that you understand arrays with more than two dimensions. The shape of each piece, in terms of its four blocks, is described by the four-dimensional array **`piece_def`** in **`mp6.c`**. **Read the description of the array in the code before you try to implement this step.**

Implement the following two C functions (in order; the second will be fairly easy once you understand how to do the first)

```
fit_result_t test_piece_fit
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y);

void mark_piece
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y, space_type_t v);
```

in **`mp6.c`**.

The function **`test_piece_fit`** checks whether a piece fits into board **`b`**. Three results (return values) are possible:

- First, if ANY block in the piece lies outside of the board, the function must return **`FIT_OUT_OF_BOARD`**.
- Next, if the board space under ANY block in the piece has a value other than **`SPACE_EMPTY`**, the function must return **`FIT_NO_ROOM_THERE`**.
- Otherwise, the function must return **`FIT_SUCCESS`**.

Note that the function must not make any changes to the board, only examine the locations that the piece described by the parameters **`p`**, **`orient`**, **`x`**, and **`y`** might occupy.

The function **`mark_piece`** marks a board with value **`v`** for each block in the piece described by the parameters **`p`**, **`orient`**, **`x`**, and **`y`**. The function does not need to make any tests as to whether the piece fits within the board, and can simply overwrite the current content of the board at the appropriate indices.

After implementing these two functions, you can compile the program and execute it. Now the program should add a single piece (a horizontal line) to the top of the board, then terminate (since you have yet to write the function to move a piece down).

Step 4: Moving pieces around

Next are five C functions corresponding to the five ways to move a piece. We suggest that you implement and debug the first of these, which will allow pieces to fall downward in the board, before implementing the remaining four. The structure is almost identical, so you are likely to copy any remaining bugs (and pay for that mistake!) if you fail to debug the first function before continuing with the others.

The first C function for moves is

```
int32_t try_to_move_down
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y);
```

in **`mp6.c`**. This function attempts to move the current piece (as described by the parameters **`p`**, **`orient`**, **`x`**, and **`y`**) downward in the board **`b`**, and returns 1 on success, and 0 on failure. On success, the piece has also been moved (the board **`b`** has changed). On failure, the board **`b`** must not be changed.

Note that the function must assume that the piece is currently marked in the board, so the function must remove the piece, then test that it fits in the new location, and finally add the piece back, either to the new location or back in the old location. Two helper functions have been provided to you in **`main.c`** to simplify your code:

```
void add_piece
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y);

void remove_piece
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y);
```

The **`add_piece`** routine adds a piece into the board **`b`**, and the **`remove_piece`** routine removes it. If you read the implementations, you'll notice that both simply call your **`mark_piece`** function, so any bugs are still your responsibility. **You must use these helper functions for the C functions in this step!**

Once you have implemented and debugged **`try_to_move_down`**, implement the following four C functions to try the other types of moves. The parameters, return values, and implementation strategies are the same as for **`try_to_move_down`**.

```
int32_t try_to_move_left
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y);

int32_t try_to_move_right
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y);

int32_t try_to_rotate_clockwise
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y);

int32_t try_to_rotate_cc
    (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], piece_type_t p,
     int32_t orient, int32_t x, int32_t y);
```

Note that the orientation ranges from 0 to 3, with clockwise rotations corresponding to the positive direction (mod 4) and counter-clockwise rotations corresponding to the negative direction. But you knew that already from having read about the **`piece_def`** array in the code.

With these functions complete, the game should be almost playable. Full rows won't go away, but eventually you're going to lose anyway, so why postpone the inevitable?

Step 5: Removing rows

Only one logical task remains: removing full rows from the board. However, you must implement this step using two C functions, which helps you to separate the logic of checking for full rows from the actions needed to remove a single row.

The two C functions that you must implement are the following:

```
int32_t is_row_full (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH], int row);

void remove_full_rows (space_type_t b[BOARD_HEIGHT][BOARD_WIDTH]);
```

The **`is_row_full`** function checks whether a single row in board **`b`** is filled with **`SPACE_FULL`** or not. The index of the row to check is given by parameter **`row`**. The function should return 1 if the row is full, and 0 otherwise.

The **remove_full_rows** function uses **is_row_full** to check for full rows in board **b** and removes those rows. To remove a row, rows above the specified row are copied downward (towards higher index values), and row 0 is filled with **SPACE_EMPTY**. Remember that removing a row changes the board, so you may need to re-check the same row number after removing a row.

Specifics

Be sure that you have read the type definitions, function headers, and other information in the code before you begin coding.

- Your code must be written in C and must be contained in the **mp6.c** file in the **mp/mp6** subdirectory of your repository. We will NOT grade any other files. **Changes made to any other files WILL BE IGNORED during grading.** If your code does not work properly without such changes, you are likely to receive 0 credit.
- You must implement the **empty_board**, **print_board**, **test_piece_fit**, **mark_piece**, **try_to_move_down**, **try_to_move_left**, **try_to_move_right**, **try_to_rotate_clockwise**, **try_to_rotate_cc**, **is_row_full**, and **remove_full_rows** functions correctly.
- You may NOT make any assumptions about the values of preprocessor constants or type definitions. You MUST use their symbolic names for full credit. We may choose to test your code with modified versions of **mp6.h**.
- You may assume that all parameter values are valid when your functions are called, provided that you also pass only valid parameters to functions that you call (such as the helper functions **add_piece** and **remove_piece**, as well as **remove_row**). In particular,
    - board arguments will point to board arrays,
    - piece types will be between 0 and NUM_PIECE_TYPES – 1 (inclusive), and
    - orientations will be between 0 and 3 (note that you may pollute these values if you call **add_piece** with a bad orientation, for example).
- Your routine's return values and outputs must be correct.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook.

Compiling and Executing Your Program

When you are ready to compile, type:

```
gcc -g -Wall main.c mp6.c -o blocky
```

As mentioned in MP1, the "**-g**" argument tells the compiler to include debugging information so that you can use **gdb** to find your bugs (you will have some).

The "**-Wall**" argument tells the compiler to give you warning messages for any code that it thinks likely to be a bug. Track down and fix all such issues, as they are usually bugs. Also note that if your code generates any warnings, you will lose points.

The "**-o blocky**" argument tells the compiler to name the resulting program "**blocky**". If compilation succeeds, you can then execute the program by typing, "**./blocky**" (no quotes).

Debugging Tip

You may wish to reduce the board size by changing the preprocessor constants in **`mp6.h`**. A smaller board can make it easier to fully test your code without needless keystrokes. However, be sure to return to the original board size and test again before turning in your code. **Remember that testing is your responsibility.**

Time for Fun!

Once your code is fully working and debugged—we strongly advise against trying to debug in graphics mode!—you may want to try a more challenging game by turning on the graphics mode. In **`mp6.h`**, change the preprocessor constant **`USE_NCURSES`** from 0 to 1. Then re-compile with "**`-l ncurses`**" added to the end of the compile line (that's a lower case L for "library"). Time how long you can last and compete with your friends! For a more serious challenge, find the call to **`halfdelay (5)`** in **`main.c`** and change the 5 to a 2, or a 1 if you dare. Note that if you want to use the VM provided by the TAs for this step, you must first install the ncurses library by typing:

```
sudo apt -y install libncurses
```

in a terminal window.

Grading Rubric

Functionality (60%)

- 5% - **`empty_board`** function works correctly
- 5% - **`print_board`** function works correctly
- 10% - **`test_piece_fit`** function works correctly
- 5% - **`mark_piece`** function works correctly
- 5% - **`try_to_move_down`** function works correctly
- 5% - **`try_to_move_left`** function works correctly
- 5% - **`try_to_move_right`** function works correctly
- 5% - **`try_to_rotate_clockwise`** function works correctly
- 5% - **`try_to_rotate_cc`** function works correctly
- 5% - **`remove_row`** function works correctly
- 5% - **`remove_full_rows`** function works correctly

Style (25%)

- 10% - **`test_piece_fit`** and **`mark_piece`** functions use loops to walk over blocks in a piece
- 5% - **`try_to_move_*`** and **`try_to_rotate_*`** functions use the helper functions **`add_piece`** and **`remove_piece`**
- 10% - all code makes use of enumerated constants (symbolic names, such as SPACE_*, PIECE_TYPE_*, and FIT_*) rather than hardcoded numerical values

Comments, Clarity, and Write-up (15%)

- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 10% - code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no functionality points. As always, your functions must be able to be called many times and produce the correct results, so we suggest that you avoid using any static storage (or you may lose most/all of your functionality points).