CSE 220: Systems Fundamentals I

Stony Brook University

Homework Assignment #4

Fall 2018

Assignment Due: November 30, 2018 by 11:59 pm

Important Information about CSE 220 Homework Assignments

- Read the entire homework documents twice before starting. Questions posted on Piazza whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. It is much easier for your TAs and the professor to help you if we can quickly figure out what your code does.
- You personally must implement homework assignments in MIPS Assembly language by yourself. You may not write or use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS Assembly code you submit as part of the assignments.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- You must use the Stony Brook version of MARS posted on Piazza. Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you will need to complete the homework assignments.
- Do not submit a file with the functions/labels `main` or `_start` defined. You are also not permitted to start your label names with two underscores (`__`). You will obtain a zero for an assignment if you do this.
- Submit your final `.asm` file to Blackboard by the due date and time. Late work will not be accepted or graded. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

With minor exceptions, all aspects of your homework submissions will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this homework assignments you will be writing *functions* in assembly language. The functions will be tested independently of each other. This is very important to note, as you must take care that no function you write ever has side-effects or requires that other functions be called before the function in question is called. Both of these are generally considered bad practice in programming.

Some other items you should be aware of:

- All test cases must execute in 100,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be

necessary, or a large data structure must be traversed. To find the instruction count of your code in Mars, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

- Any excess output from your program (debugging notes, etc.) might impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Getting Started

Visit Piazza and download the file `hw4.zip`. Decompress the file and then open `hw4.zip`. Fill in the following information at the top of `hw4.asm`:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)

Having this information at the top of the file helps us locate your work. If you forget to include this information but don't remember until after the deadline has passed, don't worry about it – we will track down your submission.

Inside `hw4.asm` you will find several function stubs that consist simply of `jr $ra` instructions. Your job in this assignment is implement all the functions as specified below. Do not change the function names, as the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, but they must be saved in `hw4.asm`. Helper functions will not be graded.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS assembly code.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

Finally, do not define a `.data` section in your `hw4.asm` file. A submission that contains a `.data` section will probably receive a score of zero.

Register Conventions

You must follow the register conventions taught in lecture and reviewed in recitation. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any `$s` registers it overwrites by saving copies of those registers on the stack and restoring them before returning.

- If a function calls a secondary function, the caller must save `$ra` before calling the callee. In addition, if the caller wants a particular `$a`, `$t` or `$v` register's value to be preserved across the secondary function call, the best practice would be to place a copy of that register in an `$s` register before making the function call.
- A function which allocates stack space by adjusting `$sp` must restore `$sp` to its original value before returning.
- Registers `$fp` and `$gp` are treated as preserved registers for the purposes of this course. If a function modifies one or both, the function must restore them before returning to the caller. There is no reason for your code to touch the `$gp` register, so leave it alone.

The following practices will result in loss of credit:

- "Brute-force" saving of all `$s` registers in a function or otherwise saving `$s` registers that are not overwritten by a function.
- Callee-saving of `$a`, `$t` or `$v` registers as a means of "helping" the caller.
- "Hiding" values in the `$k`, `$f` and `$at` registers or storing values in main memory by way of offsets to `$gp`. This is basically cheating or at best a form of laziness, so don't do it. We will comment out any such code we find.

Welcome to the Dungeon!

In this homework you will continue to work with 2D arrays, while also learning the basics of working with files in MIPS and implementing algorithms that use the stack for computations.

You will be implementing a game called MipsHack, inspired by the 1987 ASCII game NetHack. Our intrepid hero, Sir CodesALot, has entered a deep, dark dungeon in search of riches. He will need to find his way out, while defeating enemies and filling his sack with loot! His objective is to collect at least 3 coins of treasure and escape to the surface without getting killed.

Let the adventure begin!

Data Structures

The game relies on two data structures, primarily:

- a struct for representing the game world, and
- a struct for representing the player

The `Map` struct has the following definition:

```
struct Map {
    unsigned byte num_rows;    // byte #0
    unsigned byte num_cols;    // byte #1
    unsigned byte cells[][];   // bytes #2 through #num_rows*num_cols+1
}
```

As implied in the definition, a `Map` specifies a rectangular region, where `cells[0][0]` is the upper-left corner

of the map and `cells[num_rows-1][num_cols-1]` is the lower-right corner. Each element (byte) in the `cells[][]` array stores a 7-bit ASCII character in the lowest 7 bits. The most significant bit of a cell indicates whether the cell is hidden (`1`) or not (`0`) from the player. Valid ASCII characters include the following:

- **`@`** Sir CodesALot, our hero
- **`.`** empty floor
- **`#`** a wall
- **`/`** a door
- **`>`** a dungeon exit
- **`$`** a single coin
- **`*`** a gem, worth 5 coins
- **`m`** a minion monster
- **`B`** a boss monster
- **`?`** inaccessible area (should always be hidden)

The `Player` struct has the following definition:

```
struct Player {
    unsigned byte row;    // byte #0 (unsigned)
    unsigned byte col;    // byte #1 (unsigned)
             byte health; // byte #2 (signed)
    unsigned byte coins;  // byte #3 (unsigned)
}
```

The player's position in the game map is `row,col`. Provided that Sir CodesALot's health is greater than 0, he is still alive and can continue to adventure in the dungeon.

File Format

The game map and other initial game state are stored in a plaintext file with a fixed format:

```
NUM_ROWS
NUM_COLS
MAP_DATA
STARTING_HEALTH
```

Every line ends with a `'\n'` character, which is a UNIX-style line ending. When you create plaintext files in Windows, the OS will end each line with the two-character combination `'\r\n'`. *This WILL cause problems for students who make custom maps in Windows. A solution is explained in the specification for the `init_game` function below.*

As an example of how this file format is used, here is the contents of the 7-row, 25-column game world in `map3.txt` file provided with this PDF:

```
07
```

```
25
#######>########??#######?
#.....m.m......#???#..#??
#.....m*m......#####..###
#.@...mmm....../........#
#...........$..#####....#
#..B...........#???#....#
##################??#######
10
```

Note that any numbers in the file are given as two ASCII digit characters. Your code will need to perform the required conversions to 4-byte integers.

File I/O in MIPS Assembly

To assist with reading and writing files, MARS has several system calls:

Service	Code in `$v0`	Arguments	Results
open file	13	`$a0` = address of null-terminated file-name string `$a1` = flags `$a2` = mode	`$v0` contains file descriptor (negative if error)
read from file	14	`$a0` = file descriptor `$a1` = address of input buffer `$a2` = maximum # of characters to read	`$v0` contains # of characters read (0 if end-of-file, negative if error)
close file	16	`$a0` = file descriptor	

Service 13: MARS implements three *flag* values: 0 for read-only, 1 for write-only with create, and 9 for write-only with create and append. It ignores *mode*. The returned file descriptor will be negative if the operation failed. MARS maintains file descriptors internally and allocates them starting with 3. File descriptors 0, 1 and 2 are always open for reading from standard input, writing to standard output, and writing to standard error, respectively. An example of how to use these syscalls can be found on the MARS syscall web page.

Main File

We have provided you part of a main file and game loop in `hw4_main.asm`. We encourage you to add to it and write a few helper functions like `print_map` and `print_player_info`. The provided main has fixed sizes for the `map` and `visited` structs. You will need to manually set the sizes of these arrays depending on which map file you load. Guidance on how large to set these arrays is given in the main file.

Here is the general flow of what your main file should look like:

```
print "welcome" message
zero-out the map and player structs
filename = "map3.txt"  # or other file
init_game(filename, map_ptr, player_ptr)  # assuming successful execution
reveal_area(map_ptr, player_ptr.row, player_ptr.col)
move = 0
```

```
while player_ptr.health > 0 and move == 0:  # 0 means keep playing
    print_map()          # these functions take no arguments because the
    print_player_info()  # map and player structs are available globally
    char = read 1 char from the keyboard
    move = 0
    if char == 'w' then
        move = player_turn(map_ptr, player_ptr, 'U')
    elif char == 'a' then
        move = player_turn(map_ptr, player_ptr, 'L')
    elif char == 's' then
        move = player_turn(map_ptr, player_ptr, 'D')
    elif char == 'd' then
        move = player_turn(map_ptr, player_ptr, 'R')
    elif char == 'r' then
        flood_fill_reveal(map_ptr, player_ptr.row, player_ptr.col, visited)
    if move == 0 then
        reveal_area(map_ptr, player_ptr.row, player_ptr.col)

print_map()

if player_ptr.coins >= 3 and player_ptr.health > 0:
    print "congratulations" message
else:
    if player_ptr.health <= 0:
        print "you died" message
    else:
        print "you failed" message
print_player_info()
```

Hint: when you are writing the `print_map` function, as you load characters out of the map, if the character has `1` for the hidden flag, print a space instead of the character you loaded. Otherwise, assuming the character is visible, simply print it using system call 11.

A Reminder on How Your Work Will be Graded

It is **imperative** (crucial, essential, necessary, critically important) that you implement the functions below exactly as specified. Do not deviate from the specifications, even if you think you are implementing the game in a better way. Modify the contents of memory only as described in the function specifications!

Remember how your work will be graded: we will test each function individually. We will not be playing your game and watching things unfold – you will not even be submitting your main files, remember.

Part I: Initialize Game Data Structures

```
int init_game(string map_filename, Map *map_ptr, Player *player_ptr)
```

This function opens (syscall 13), reads (syscall 14), processes and then closes (syscall 16) the file named `map_filename`, which contains the starting game information. You may assume that the input file is always formatted properly. The function reads the number of rows and columns from the game map and stores them at bytes 0 and 1, respectively, of the `Map` struct that `map_ptr` points to. It then proceeds to read the contents of the game world, character-by-character. As it reads each character, the function sets the "hidden" flag at bit number 7 (the most significant bit) of the character before writing the character to memory. Note that these characters are written to the `Map` struct starting at byte 2. Every character in the map is hidden initially. The bytes of the map are stored in row-major order.

At some point while reading the map contents, the function will encounter the `@` character. The function writes the row and column numbers of the `@` character as bytes 0 and 1, respectively, of the `Player` struct that `player_ptr` points to. At the end of the file the function will find the starting health of the player. It writes that value at byte 2 of the `Player` struct, and the number 0 at byte 3 of the `Player` struct to initialize the number of coins held by the player to zero. Finally, the function closes the file.

Note that all numbers in the file are stored as two-digit ASCII characters, with leading zeroes as needed. For example, in `map3.txt`, the number of columns in the map is twenty-five. This value is represented by the ASCII characters `'2'` and `'5'` on line 2 of the file. You will need to perform the appropriate conversion to turn this two-character string into an integer.

The function assumes that every line ends only with a `'\n'` character, *not* the two-character combination `'\r\n'` employed in Microsoft Windows. If you create your own maps for testing purposes, use MARS to edit the file. If you are developing on a Windows computer, do not use a regular text editor like Notepad. Such an editor will insert both characters. In contrast, MARS will insert only a `'\n'` at the end of each line, *so only use MARS to create custom maps*.

The function takes the following arguments, in this order:

- `map_filename`: A string the provides the filename of the file containing the map information and the player's starting helth.
- `map_ptr`: A pointer to a `Map` struct that is large enough to store the dimensions of the map, as well as the map contents.
- `player_ptr`: A pointer to a `Player` struct that is large enough to store the four bytes that define the player's attributes.

Returns in `$v0`:

- `0` if the file was successfully opened and its contents were processed, or `-1` for error.

Returns `-1` in `$v0` for error if a file with the name `map_filename` could not be read off the disk.

Additional requirements:

- The function must not write any changes to main memory except as specified.

Example:

Suppose `init_game` has been called with `map3.txt` as the filename. The `Player` and `Map` structs would be

initialized as follows, where the bytes of the game world are given in hexadecimal and all other values are given in decimal. All values shown below are integers. Note that each byte in the `cells` array is ≥ 128 because all cells are initially hidden:

```
player_ptr.row = 3
player_ptr.col = 2
player_ptr.health = 10
player_ptr.coins = 0
map_ptr.rows = 7
map_ptr.cols = 25
map_ptr.cells = [
 A3 A3 A3 A3 A3 A3 A3 BE A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
 A3 AE AE AE AE AE ED AE ED AE AE AE AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
 A3 AE AE AE AE AE ED AA ED AE AE AE AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
 A3 AE C0 AE AE AE ED ED ED AE AE AE AE AE AE AF AE AE AE AE AE AE AE AE A3
 A3 AE AE AE ED AE AE AE AE AE AE AE A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
 A3 AE AE C2 AE AE AE AE AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
 ]
```

Part II: Check for a Valid Cell Position

`int is_valid_cell(Map *map_ptr, int row, int col)`

This function determines whether `(row,col)` represents a valid pair of indices into the game world. That is, whether a potential access to `map_ptr.cells[row][col]` would be valid.

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `row`: The 0-based row index of the desired byte.
- `col`: The 0-based column index of the desired byte.

Returns in `$v0`:

- `0` if `(row,col)` is a valid index pair, or `-1` if not

Returns `-1` in `$v0` in any of the following cases:

- `row` < 0
- `row` $\geq$ `map_ptr.num_rows`
- `col` < 0
- `col` $\geq$ `map_ptr.num_cols`

Additional requirements:

- The function must not write any changes to main memory.

Examples:

In these examples, suppose that the game world contains 25 rows and 7 columns.

Function Call	Return Value
`is_valid_cell(map_ptr, 5, 3)`	0
`is_valid_cell(map_ptr, 0, 0)`	0
`is_valid_cell(map_ptr, 24, 6)`	0
`is_valid_cell(map_ptr, 25, 3)`	-1
`is_valid_cell(map_ptr, -3, 4)`	-1

Part III: Get the Contents of a Cell

```
int get_cell(Map *map_ptr, int row, int col)
```

This function returns the byte stored at map position `(row,col)`. It does not modify the hidden flag or perform any processing before returning the value in the map.

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `row`: The 0-based row index of the desired byte.
- `col`: The 0-based column index of the desired byte.

Returns in `$v0`:

- The byte at `map_ptr.cells[row][col]`.

Returns `-1` in `$v0` for error in any of the following cases:

- `row` < 0
- `row` $\geq$ `map_ptr.num_rows`
- `col` < 0
- `col` $\geq$ `map_ptr.num_cols`

Additional requirements:

- `get_cell` must call `is_valid_cell`.
- The function must not write any changes to main memory.

Example:

Suppose that `map3.txt` was loaded and the state of the game world (visually) is as follows:

```
###
#.....m
#.....m
#.....mm
#.....@.
#..*....
####
```

In memory, in hexadecimal, the game world will look like this:

```
23 23 23 A3 A3 A3 A3 BE A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
23 2E 2E 2E 2E 2E 6D AE ED AE AE AE AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
23 2E 2E 2E 2E 2E 6D AA ED AE AE AE AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
23 2E 2E 2E 2E 2E 6D 6D ED AE AE AE AE AE AE AF AE AE AE AE AE AE AE AE A3
23 2E 2E 2E 2E 2E 40 2E AE AE AE AE A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
23 2E 2E 2A 2E 2E 2E 2E AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
23 23 23 23 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

Function Call	Return Value (decimal)
`get_cell(map_ptr, 3, 8)`	237
`get_cell(map_ptr, 4, 6)`	64
`get_cell(map_ptr, 25, 3)`	-1
`get_cell(map_ptr, -3, 4)`	-1

Part IV: Set the Contents of a Cell

```
int set_cell(Map *map_ptr, int row, int col, char ch)
```

This function changes the byte stored at map position `(row,col)` to `ch`. It does not modify the hidden flag of `ch` or perform any processing of `ch` before writing the value to the map.

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `row`: The 0-based row index of the byte to be modified.
- `col`: The 0-based column index of the byte to be modified.
- `ch`: The byte to be written to index `(row,col)`.

Returns in `$v0`:

- `0` if the change to the map was successful, or `-1` on error

Returns `-1` in `$v0` for error in any of the following cases:

- `row` < 0

- `row` $\geq$ `map_ptr.num_rows`
- `col` < 0
- `col` $\geq$ `map_ptr.num_cols`

Additional requirements:

- `set_cell` must call `is_valid_cell`.
- The function must not write any changes to main memory except for the intended byte.

Examples:

Suppose that `map3.txt` was loaded and the state of the game world (visually) is as follows:

```
###
#.....m
#.....m
#.....mm
#.....@.
#..*....
####
```

In memory, in hexadecimal, the game world will look like this:

```
23 23 23 A3 A3 A3 A3 BE A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
23 2E 2E 2E 2E 2E 6D AE ED AE AE AE AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
23 2E 2E 2E 2E 2E 6D AA ED AE AE AE AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
23 2E 2E 2E 2E 2E 6D 6D ED AE AE AE AE AE AE AF AE AE AE AE AE AE AE AE A3
23 2E 2E 2E 2E 2E 40 2E AE AE AE AE A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
23 2E 2E 2A 2E 2E 2E 2E AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
23 23 23 23 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

Function Call	Return Value (decimal)
`set_cell(map_ptr, 3, 6, '$')`	0
`set_cell(map_ptr, 4, 8, '.')`	0
`set_cell(map_ptr, 25, 3, '.')`	-1
`set_cell(map_ptr, -3, 4, '@')`	-1

For test cases 1 and 2, the `cells` array would be updated at the given indices. For test cases 3 and 4, no changes would be made to the `cells` array. As an example, for test case 1 the `cells` array would be updated to the following:

```
23 23 23 A3 A3 A3 A3 BE A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
23 2E 2E 2E 2E 2E 6D AE ED AE AE AE AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
23 2E 2E 2E 2E 2E 6D AA ED AE AE AE AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
23 2E 2E 2E 2E 2E 24 6D ED AE AE AE AE AE AE AF AE AE AE AE AE AE AE AE A3
23 2E 2E 2E 2E 2E 40 2E AE AE AE AE A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
23 2E 2E 2A 2E 2E 2E 2E AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
```

```
23 23 23 23 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

Part V: Reveal the Cells in a 3×3 Area

`void reveal_area(Map *map_ptr, int row, int col)`

This function reveals all 9 characters in the 3×3 area centered at `(row,col)` in a game world. If `(row,col)` itself is not a valid position, but some cells in a 3×3 area centered at `(row,col)` in the game world are valid, then the valid cells are revealed. For example, suppose `(row,col) = (-1,0)`, which is an invalid position. The 3×3 area centered at this position consists of these (theoretical) indices:

```
-2,-1  -2, 0  -2, 1
-1,-1  -1, 0  -1, 1
 0,-1   0, 0   0, 1
```

For this example, the cells at indices `0,0` and `0,1` would be revealed.

A cell is revealed by setting bit 7 of the cell to 0. For example, the `#` character has hexadecimal ASCII code `0x23`, which is `0b00100011` in binary. Therefore, a hidden wall would be stored as `0b10100011` = `0xA3` in memory. This function would change that byte to `0b00100011`. A cell that is already revealed is not modified by this function.

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `row`: The 0-based row index of the byte in the center of the 3×3 area to be revealed.
- `col`: The 0-based column index of the byte in the center of the 3×3 area to be revealed.

Additional requirements:

- `reveal_area` must call `get_cell` and `set_cell`.
- The function must not write any changes to main memory except for the intended bytes.

Examples:

Suppose that `map3.txt` was loaded and the state of the game world (visually) is as follows:

```
 #####
.....m.
.....@*
.....mm
...
```

In memory, in hexadecimal, the game world will look like this:

```
A3 A3 23 23 23 23 23 BE A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
A3 2E 2E 2E 2E 2E 6D 2E ED AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
A3 2E 2E 2E 2E 2E 40 2A ED AE AE AE AE AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
A3 2E 2E 2E 2E 2E 6D 6D ED AE AE AE AE AE AE AF AE AE AE AE AE AE AE AE AE A3
A3 2E 2E 2E AE AE AE AE AE AE AE AE A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
A3 AE AE C2 AE AE AE AE AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

Now suppose we call `reveal_area(map_ptr, 6, 4)`. Although it is true that the player is not in the neighborhood of this cell, the function specification does not require that the player be in the center of the revealed cell. In addition, the cell at index `(6,4)` is a hidden wall character at the edge of the map. Again, the specification does not say it is illegal/invalid to call the function on a cell that is wall or that is on the edge of the map. For this particular function call the game map would theoretically be updated as follows (visually):

```
  #####
.....m.
.....@*
.....mm
...
  B..
  ###
```

and in memory, in hexadecimal, the `cells` array would look like this:

```
A3 A3 23 23 23 23 23 BE A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
A3 2E 2E 2E 2E 2E 6D 2E ED AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
A3 2E 2E 2E 2E 2E 40 2A ED AE AE AE AE AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
A3 2E 2E 2E 2E 2E 6D 6D ED AE AE AE AE AE AE AF AE AE AE AE AE AE AE AE AE A3
A3 2E 2E 2E AE AE AE AE AE AE AE AE A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
A3 AE AE 42 2E 2E AE AE AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
A3 A3 A3 23 23 23 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

Part VI: Acquiring a Target to Attack

```
int get_attack_target(Map *map_ptr, Player *player_ptr, char direction)
```

This function inspects a cell adjacent to a player, as indicated by `direction`, and returns the character at that "target" cell, provided that the cell is an attackable target, namely, a minion monster (`m`), a boss monster (`B`) or a door (`/`). If the targeted cell is not one of these targets or of the target cell is not valid (i.e., invalid indices), the function returns `-1`. Otherwise, the function simply returns the character at the targeted cell. The function assumes that the targeted cell is visible (i.e., not hidden).

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index

`[0][0]` of the game world.

- `player_ptr`: The starting address of a `Player` struct.
- `direction`: The character `'U'`, `'D'`, `'L'` or `'R'`.
  - `'U'` indicates that the player is attempting to attack a target in the same column, but previous row of the game world (i.e., index `(player_ptr.row-1,player_ptr.col)`).
  - `'D'` indicates that the player is attempting to attack a target in the same column, but next row of the game world (i.e., index `(player_ptr.row+1,player_ptr.col)`).
  - `'L'` indicates that the player is attempting to attack a target in the same row, but previous column of the game world (i.e., index `(player_ptr.row,player_ptr.col-1)`).
  - `'R'` indicates that the player is attempting to attack a target in the same row, but next column of the game world (i.e., index `(player_ptr.row,player_ptr.col+1)`).

Returns in `$v0`:

- The targeted cell, which can be only one of `'m'`, `'B'` or `'/'`.

Returns `-1` in `$v0` for error in any of the following cases:

- `direction` is not one of `'U'`, `'D'`, `'L'` or `'R'`.
- The targeted cell is not at a valid index. Although during normal gameplay this shouldn't be possible, the function must accommodate this possibility.
- The targeted cell is not one of `'m'`, `'B'` or `'/'`.

Additional requirements:

- `get_attack_target` must call `get_cell`.
- The function must not write any changes to main memory.

Example:

Suppose that `map3.txt` was loaded and the state of the game world (visually) is as follows:

```
        ####
        m@.....
.....m m......#
.....mmm.......
...........$..#
   .......
```

In memory, in hexadecimal, the game world will look like this:

```
A3 A3 A3 A3 A3 A3 A3 BE 23 23 23 23 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
A3 AE AE AE AE AE ED AE 6D 40 2E 2E 2E 2E 2E A3 BF BF BF A3 AE AE A3 BF BF
```

```
A3 2E 2E 2E 2E 2E 6D AA 6D 2E 2E 2E 2E 2E 2E 23 A3 A3 A3 A3 AE AE A3 A3 A3
A3 2E 2E 2E 2E 2E 6D 6D 6D 2E 2E 2E 2E 2E 2E 2E AE AE AE AE AE AE AE AE A3
A3 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 24 2E 2E 23 A3 A3 A3 A3 AE AE AE AE A3
A3 AE AE C2 2E 2E 2E 2E 2E 2E 2E AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

Function Call	Return Value	Explanation
`get_attack_target(map_ptr, player_ptr, 'U')`	`-1`	cannot attack a wall
`get_attack_target(map_ptr, player_ptr, 'D')`	`-1`	cannot attack floor
`get_attack_target(map_ptr, player_ptr, 'L')`	`109`	can attack a minion
`get_attack_target(map_ptr, player_ptr, 'R')`	`-1`	cannot attack floor
`get_attack_target(map_ptr, player_ptr, 'Z')`	`-1`	invalid argument

Part VII: Completing a Player's Attack

```
void complete_attack(Map *map_ptr, Player *player_ptr, int target_row,
                     int target_col)
```

This function is essentially a helper function for `player_turn`. It is called once it has been determined external to this function that the player can validly attack a targeted cell located at index `(target_row,target_col)` of the game world. The possible outcomes of calling this function are as follows:

- The targeted cell is `'m'`. The player completes a successful kill of the minion, which counterattacks during the fight and causes 1 point of damage to the player. The value of `player_ptr.health` is updated accordingly. The `'m'` in the targeted cell is replaced with `'$'`. The player's position does not change.
- The targeted cell is `'B'`. The player completes a successful kill of the boss monster, which counterattacks during the fight and causes 2 points of damage to the player. The value of `player_ptr.health` is updated accordingly. The `'B'` in the targeted cell is replaced with `'*'`. The player's position does not change.
- The targeted cell is `'/'`. The player destroys the targeted door. The `'/'` in the targeted cell is replaced with `'.'`. The player's position does not change.

While battling a monster, the player's health might drop to 0 or -1, meaning that the player has died. In this case, the `'@'` for the player is replaced with `'X'` in the game world. Note that the monster is still killed and loot (`'$'` or `'*'`) is still dropped.

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `player_ptr`: The starting address of a `Player` struct.
- `target_row`: The row of a targeted cell.
- `target_col`: The column of a targeted cell.

Additional requirements:

- `complete_attack` must call `get_cell` and `set_cell`.
- The function must not write any changes to main memory except for the targeted cell and possibly changing the `'@'` to `'X'`, if the player has died.

Examples:

Suppose that `map3.txt` was loaded and the state of the game world (visually) is as follows:

```
        ####
        m@.....
  .....m m......#
  .....mmm.......
  ............$..#
     .......
```

In memory, in hexadecimal, the game world will look like this:

```
A3 A3 A3 A3 A3 A3 A3 BE 23 23 23 23 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
A3 AE AE AE AE AE ED AE 6D 40 2E 2E 2E 2E 2E A3 BF BF BF A3 AE AE A3 BF BF
A3 2E 2E 2E 2E 2E 6D AA 6D 2E 2E 2E 2E 2E 2E 23 A3 A3 A3 A3 AE AE A3 A3 A3
A3 2E 2E 2E 2E 2E 6D 6D 6D 2E 2E 2E 2E 2E 2E 2E AE AE AE AE AE AE AE AE A3
A3 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 24 2E 2E 23 A3 A3 A3 A3 AE AE AE AE A3
A3 AE AE C2 2E 2E 2E 2E 2E 2E 2E AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

The player is located at `map_ptr.cells[1][9]`. Suppose we make the function call `complete_attack(map_ptr, player_ptr, 1, 8)`. At `map_ptr.cells[1][8]` we can see that a minion is present. Therefore, the player suffers 1 point of damage (i.e., `player_ptr.health` decreases by 1) and the contents of `map_ptr.cells[1][8]` is changes to `'$'`.

The state of the game world (visually) will change to this:

```
        ####
        $@.....
  .....m m......#
  .....mmm.......
  ............$..#
     .......
```

In memory, in hexadecimal, the game world will now look like this:

```
A3 A3 A3 A3 A3 A3 A3 BE 23 23 23 23 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
A3 AE AE AE AE AE ED AE 24 40 2E 2E 2E 2E 2E A3 BF BF BF A3 AE AE A3 BF BF
```

```
A3 2E 2E 2E 2E 2E 6D AA 6D 2E 2E 2E 2E 2E 2E 23 A3 A3 A3 A3 AE AE A3 A3 A3
A3 2E 2E 2E 2E 2E 6D 6D 6D 2E 2E 2E 2E 2E 2E 2E AE AE AE AE AE AE AE AE A3
A3 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 24 2E 2E 23 A3 A3 A3 A3 AE AE AE AE A3
A3 AE AE C2 2E 2E 2E 2E 2E 2E 2E AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

Part VIII: When Monsters Attack

`int monster_attacks(Map *map_ptr, Player *player_ptr)`

This function is essentially a helper function for `player_turn`. It returns the number of points of damage potentially inflicted by monsters against the player. Suppose `(R,C)` is the position of the player. The four grid positions `(R-1,C)`, `(R+1,C)`, `(R,C-1)` and `(R,C+1)` are inspected to check if a monster is located there. Every such minion (`m`) potentially inflicts one point of damage, and every boss (`B`) potentially inflicts two points of damage.

Note that the function itself does not modify the `Player` struct's `health` field. Rather, the function simply returns the number of points of damage that the monsters at the given location *could* inflict on the target.

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `player_ptr`: The starting address of a `Player` struct.

Returns in `$v0`:

- The number of points of damage that monsters immediately up, down, left and right of `map_ptr.cells[player_ptr.row][player_ptr.col]` could potentially cause to the player.

Additional requirements:

- `monster_attacks` must call `get_cell`.
- The function must not write any changes to main memory.

Example #1:

Suppose that `map3.txt` was loaded and the state of the game world (visually) is as follows:

```
      #>####
      m.....
.....m@m...
.....mmm...
..........
  B........
```

In memory, in hexadecimal, the game world will look like this:

```
A3 A3 A3 A3 A3 A3 23 3E 23 23 23 23 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
A3 AE AE AE AE AE 6D 2E 2E 2E 2E 2E AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
A3 2E 2E 2E 2E 2E 6D 40 6D 2E 2E 2E AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
A3 2E 2E 2E 2E 2E 6D 6D 6D 2E 2E 2E AE AE AE AF AE AE AE AE AE AE AE AE A3
A3 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
A3 AE AE 42 2E 2E 2E 2E 2E 2E 2E 2E AE AE AE A3 BF BF BF A3 AE AE AE AE A3
A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

The function call `monster_attacks(map_ptr, player_ptr)` will return the value `3` because there are 3 minions immediately above, below, left and/or right of the player. The player's health remains unchanged. The state of the `cells` array of the `Map` struct remains unchanged.

Example #2:

Suppose that `map3.txt` was loaded and the state of the game world (visually) is as follows:

```
.....
.....m
....@.
  B...
```

In memory, in hexadecimal, the game world will look like this:

```
A3 A3 A3 A3 A3 A3 A3 BE A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
A3 AE AE AE AE AE ED AE ED AE AE AE AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
A3 2E 2E 2E 2E 2E ED AA ED AE AE AE AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
A3 2E 2E 2E 2E 2E 6D ED ED AE AE AE AE AE AE AF AE AE AE AE AE AE AE AE A3
A3 2E 2E 2E 2E 40 2E AE AE AE AE AE A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
A3 AE AE 42 2E 2E 2E AE AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

The function call `monster_attacks(map_ptr, player_ptr)` will return the value `0` because there are no monsters (neither minions nor bosses) immediately above, below, left and/or right of the player. The player's health remains unchanged. The state of the `cells` array of the `Map` struct remains unchanged.

Example #3:

Suppose that `map3.txt` was loaded and the state of the game world (visually) is as follows:

```
.....m
.....m
...m..
  B@..
  ####
```

In memory, in hexadecimal, the game world will look like this:

```
A3 A3 A3 A3 A3 A3 A3 BE A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 BF BF
A3 AE AE AE AE AE ED AE ED AE AE AE AE AE AE A3 BF BF BF A3 AE AE A3 BF BF
A3 2E 2E 2E 2E 2E 6D AA ED AE AE AE AE AE AE A3 A3 A3 A3 A3 AE AE A3 A3 A3
A3 2E 2E 2E 2E 2E 6D ED ED AE AE AE AE AE AE AF AE AE AE AE AE AE AE AE A3
A3 2E 2E 2E 6D 2E 2E AE AE AE AE AE A4 AE AE A3 A3 A3 A3 A3 AE AE AE AE A3
A3 AE AE 42 40 2E 2E AE AE AE AE AE AE AE AE A3 BF BF BF A3 AE AE AE AE A3
A3 A3 A3 23 23 23 23 A3 A3 A3 A3 A3 A3 A3 A3 A3 BF BF BF A3 A3 A3 A3 A3 A3
```

The function call `monster_attacks(map_ptr, player_ptr)` will return the value `3` because the monsters immediately above, below, left and/or right of the player (one minion, one boss) *could* collectively cause 3 points of damage. The player's health remains unchanged. The state of the `cells` array of the `Map` struct remains unchanged.

Part IX: Moving the Player

```
int player_move(Map *map_ptr, Player *player_ptr, int target_row,
                int target_col)
```

This function is essentially a helper function for `player_turn`. It is called once it has been determined external to this function that the player can validly *attempt* to move to the targeted cell located at index `(target_row,target_col)` of the game world.

Before attempting to move the player, the function calls `monster_attacks` to check whether any nearby monsters land an attack on the player. The return value of `monster_attacks` is subtracted from the player's health.

The possible outcomes of calling this function are as follows:

- Nearby monsters killed the player (i.e., the player's health is ≤ 0). The `'@'` at the player's position in the game world is replaced with `'X'`. The function returns `0`.
- The targeted cell is `'.'`. The `'@'` at the player's position in the game world is replaced with `'.'`, and the `'.'` at the targeted cell in the game world is replaced with `'@'`. The `Player` struct's position is updated accordingly. The function returns `0`.
- The targeted cell is `'$'`. The `'@'` at the player's position in the game world is replaced with `'.'`, and the `'$'` at the targeted cell in the game world is replaced with `'@'`. The `Player` struct's position is updated accordingly. The `Player` struct's `coins` field is incremented by 1. The function returns `0`.

- The targeted cell is `'*'`. The `'@'` at the player's position in the game world is replaced with `'.'`, and the `'*'` at the targeted cell in the game world is replaced with `'@'`. The `Player` struct's position is updated accordingly. The `Player` struct's `coins` field is incremented by 5. The function returns `0`.
- The targeted cell is `'>'`. The `'@'` at the player's position in the game world is replaced with `'.'`, and the `'>'` at the targeted cell in the game world is replaced with `'@'`. The `Player` struct's position is updated accordingly. The function returns `-1`. (Updated 11/20/2018.)

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `player_ptr`: The starting address of a `Player` struct.
- `target_row`: The row of the cell the player will move to.
- `target_col`: The column of the cell the player will move to.

Return values for `$v0` are provided in the description above.

Additional requirements:

- `player_move` must call `get_cell`, `set_cell` and `monster_attacks`.
- The function must not write any changes to main memory except as specified.

Examples:

In the examples below, the player is playing a game using `map3.txt`.

Example #1: The player dies while trying to move

Suppose the state of the game visually is:

```
.....m
.....m
...m..
  B@..
  ####
```

Suppose that the player has 1 point of health remaining, is at index `(5,4)` and the function call `player_move(map_ptr, player_ptr, 5, 5)` is made. `monster_attacks` is called, causing 3 points of damage to the player. The player dies from the monsters' attacks. The state of the map becomes:

```
.....m
.....m
...m..
  BX..
  ####
```

and the function returns `0`. Note that the cell at the player's position becomes `'X'` and the player does not move.

Example #2: The player moves to adjacent floor spot

Suppose the state of the game visually is:

```
.....m
.....$
....@.
 .B...
```

Suppose that the player is at index `(4,5)` and the function call `player_move(map_ptr, player_ptr, 5, 5)` is made. The state of the map becomes:

```
.....m
.....$
......
 .B.@.
   ###
```

and the function returns `0`.

Example #3: The player moves and picks up a coin

Suppose the state of the game visually is:

```
.....m
.....m
...$@.
  B...
  ####
```

Suppose that the player is at index `(4,5)` and the function call `player_move(map_ptr, player_ptr, 4, 4)` is made. The state of the map becomes:

```
.....m
.....m
...@..
  B...
  ####
```

and the function returns `0`. Note that the player's `coins` field is incremented by 1 during execution of the function.

Example #4: The player moves and picks up a gem

Suppose the state of the game visually is:

```
.....m*m
.....m@m
........
  B.....
  ####
```

Suppose that the player is at index `(3,7)` and the function call `player_move(map_ptr, player_ptr, 2, 7)` is made. The state of the map becomes:

```
      m.m
.....m@m
.....m.m
........
  B.....
  ####
```

and the function returns `0`. Note that the player's `coins` field is incremented by 5 during execution of the function.

Example #5: The player escapes the dungeon

Suppose the state of the game visually is:

```
 #####>#
......@m
.....m*m
....
...
```

Suppose that the player is at index `(1,7)`, has 5 points of health remaining, and the function call `player_move(map_ptr, player_ptr, 0, 7)` is made. The state of the map becomes:

```
 #####@#
.......m
.....m*m
....
...
```

and the function returns `-1`.

Example #6: The player dies while trying to escape the dungeon

Suppose the state of the game visually is:

```
      #>###
      m@...
.....m*m..
..........
...m......
```

Suppose that the player has 1 point of health remaining, is at index `(1,7)`, and the function call `player_move(map_ptr, player_ptr, 0, 7)` is made. The state of the map becomes:

```
      #>###
      mX...
.....m*m..
..........
...m......
```

and the function returns `0` (not `-1`) because the player died.

Part X: Taking a Turn

`int player_turn(Map *map_ptr, Player *player_ptr, char direction)`

This function is a top-level function and relies directly or indirectly on almost every other function on the assignment. `player_turn` is intended to be called inside the *game loop* to process a player's turn in the game. The algorithm it must implement proceeds as follows:

1. Check if `direction` is one of `'U'`, `'D'`, `'L'` or `'R'`. If not, then return `-1`. Otherwise, continue to step 2.
2. Check if the targeted cell is at a valid index. If not, then return `0`. Otherwise, continue to step 3. Although during normal gameplay shouldn't be possible to target an invalid index, the function must accommodate for this possibility.
3. Call `get_cell` to check where the player is attempting to move to or attack. If the target cell is `'#'`, return `0`.
4. Assuming the target is a valid index to move to or attack, call `get_attack_target` to see if the target cell is attackable.
   - If the target cell is attackable, call `complete_attack` and then return `0`.
   - Otherwise, call `player_move` and return that function's return value as the return value of `player_turn`.

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `player_ptr`: The starting address of a `Player` struct.
- `direction`: The character `'U'`, `'D'`, `'L'` or `'R'`.
  - `'U'` indicates that the player is attempting to attack a target in the same column, but previous row of the game world (i.e., index `(player_ptr.row-1,player_ptr.col)`).
  - `'D'` indicates that the player is attempting to attack a target in the same column, but next row of the game world (i.e., index `(player_ptr.row+1,player_ptr.col)`).
  - `'L'` indicates that the player is attempting to attack a target in the same row, but previous column of the game world (i.e., index `(player_ptr.row,player_ptr.col-1)`).
  - `'R'` indicates that the player is attempting to attack a target in the same row, but next column of the game world (i.e., index `(player_ptr.row,player_ptr.col+1)`).

Return values for `$v0` are provided in the description above.

Additional requirements:

- `player_turn` must call `get_cell`, `get_attack_target`, `complete_attack` and `player_move`.
- The function must not write any changes to main memory except as specified.

Examples:

In the examples below, the player is playing a game using `map3.txt`.

Example #1: The player attempts to move into a wall

Suppose the state of the game visually is:

```
.....m
.....m
...m....
   ...@.
   #####
```

Suppose that the player is at index `(5,7)`, and the function call `player_turn(map_ptr, player_ptr, 'D')` is made. Because there is wall immediately below the player, the player cannot move. Therefore, the function simply returns `0`.

Example #2: The player wants to make an attack

Suppose the state of the game visually is:

```
.....m
.....mmm.
...m...@.
   ......
   ######
```

Suppose that the player is at index `(4,8)`, and the function call `player_turn(map_ptr, player_ptr, 'U')` is made. Since the target cell is not `'#'`, `player_turn` then calls `get_attack_target`. `get_attack_target` in this case will not return `-1`, meaning that the player can attack the target cell. Therefore, `player_turn` then calls `complete_attack` and, after `complete_attack` returns, `player_turn` returns `0`.

Example #3: The player wants to move to an adjacent cell

Suppose the state of the game visually is:

```
.....m
.....mmm..
...m....@.
   .......
   #######
```

Suppose that the player is at index `(4,9)`, and the function call `player_turn(map_ptr, player_ptr, 'R')` is made. Since the target cell is not `'#'`, `player_turn` then calls `get_attack_target`. `get_attack_target` in this case will return `-1`, meaning that the player cannot attack the target cell. Therefore, `player_turn` then calls `player_move`. `player_turn` will return `player_move`'s return value as its own return value.

Part XI: Revealing a Large Area of the Game World

```
int flood_fill_reveal(Map *map_ptr, int row, int col, bit[][] visited)
```

This function performs a "flood fill" operation to reveal empty floor spaces (`'.'`) in the neighborhood of the cell at index `(row,col)` of the world map. The function uses the stack to track all adjacent cells which could be revealed. See the pseudocode below for the algorithm you must implement. `$fp` is the **frame pointer**, a register we can use in concert with `$sp` to manage the stack. The frame pointer is a preserved register, and therefore must be preserved just like an `$s` register. In this algorithm the frame pointer is used to help us keep track of what cells of the game world we still need to process and possibly reveal during the search.

```
if (row, col) represents an invalid index then
    return -1

$fp = $sp
$sp.push(row)
$sp.push(col)
offsets = [(-1, 0), (1, 0), (0, -1), (0, 1)]   # a list of pairs
while $sp != $fp:
    col = $sp.pop()
    row = $sp.pop()
    make the cell at index (row,col) visible in the world map
    foreach pair (i,j) of values in offsets[]:
        if the cell at index (row+i, col+j) represents empty floor AND     (*)
           the cell has not been visited yet, then
               (1) set that cell as having been visited
               (2) $sp.push(row+i)
               (3) $sp.push(col+j)
return 0
```

In the if-statement marked with the `(*)` above, the floor cell might be be hidden or revealed. Regardless of whether the floor at that index is hidden or revealed, it must be pushed on the stack if it has not been visited yet.

The function takes the following arguments, in this order:

- `map_ptr`: The starting address of a `Map` struct. Note: this is NOT the address of the character at index `[0][0]` of the game world.
- `row`: The row index where the flood fill begins.
- `col`: The column index where the flood fill begins.
- `visited`: A 2D array of bits that record whether a particular cell has been visited by the algorithm. The

dimensions of the bit vector match the dimensions of the `cells` array. The function may assume that this bit-vector has been initialized with all zero bits.

Returns in `$v0`:

- `0` if a flood fill was executed, or `-1` on error.

Returns `-1` in `$v0` for error in any of the following cases:

- `row` < 0
- `row` $\geq$ `map_ptr.num_rows`
- `col` < 0
- `col` $\geq$ `map_ptr.num_cols`

Additional requirements:

- `flood_fill_reveal` must call `get_cell` and `set_cell`.
- The function must not write any changes to main memory except as specified.

Examples:

In the examples below, the player is playing a game using `map3.txt`.

Example #1:

The player is at index `(3,2)`.

Suppose that before the function call `flood_fill_reveal(map_ptr, 3, 2, visited)` the state of the map is:

```
...
.@.
...
```

After the function call it will become:

```
.....   ......
.....   ......
.@...   ......
... ....... ..
.. ...........
```

Example #2:

The player is at index `(3,14)`.

Suppose that before the function call `flood_fill_reveal(map_ptr, 3, 14, visited)` the state of the map is:

```
.....m       ..#
.....mmm.....@.
...m..........#
   ............#
            ###
```

After the function call it will become:

```
.....    ......      ..
.....m   ......#     ..
.....mmm.....@.........
...m..........#     ....
..  ............#    ....
            ###
```

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work to Blackboard you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the College of Engineering and Applied Sciences.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating in a homework may be discovered and penalized after the grades for that homework have been recorded.

8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.

How to Submit Your Work for Grading

To submit your `hw4.asm` file for grading:

1. Login to Blackboard and locate the course account for CSE 220.
2. Click on "Assignments" in the left-hand menu and click the link for this assignment.
3. Click the "Browse My Computer" button and locate the `hw4.asm` file. Submit only that one `.asm` file.
4. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the steps again. We will grade only your last submission.