Project 1: snek

Deadline: Thursday, February 9, 11:59:59 PM PT

Welcome to the first project of 61C! In this project, you'll get some practice with C coding by creating a playable snake game. If you're not familiar with snake, you can try out a demo at this link .

Content in scope for this project: Lectures 3-5, Discussions 1-2, Labs 1-2, and Homework 2. Also, make sure you've finished the setup in Lab 0.

Setup

This assignment can be done alone or with a partner.

> **Warning**: Once you create a Github repo, you will *not* be able to change (add, remove, or swap) partners for this project, so please be sure of your partner before starting the project. You must add your partner on both `galloc` and to every Gradescope submission.

If there are extenuating circumstances that require a partner switch (e.g. your partner drops the class, your partner is unresponsive), please reach out to us privately.

1. **Visit** Galloc . **Log in** and **start** the Project 1 assignment. This will create a GitHub repository for your work. This will create a GitHub repository for your work. If you have a partner, one partner should create a repo and invite the other partner to that repo. The other partner should accept the invite without creating their own repo.
2. **Clone** the repository on a **hive machine**.

```
$ git clone git@github.com:61c-student/sp23-proj1-USERNAME.git 61c-proj1
```

(**replace** `username` with your GitHub username)

3. **Navigate** to your repository:

```
$ cd 61c-proj1
```

4. **Add** the starter repository as a remote:

```
$ git remote add starter https://github.com/61c-teach/sp23-proj1-starter.git
```

Conceptual Overview

Snakes

A snake game can be represented by a grid of characters. The grid contains walls, fruits, and one or more snakes. An example of a game is shown below:

```
##############
#            #
#    dv      #
#     v   #  #
#     v   #  #
#   s >>D #  #
#   v     #  #
# *A<  *  #  #
#            #
##############
```

The grid has the following special characters:

- `#` denotes a wall.
- (space character) denotes an empty space.
- `*` denotes a fruit.
- `wasd` denotes the tail of a snake.
- `^<v>` denotes the body of a snake.
- `WASD` denotes the head of a snake.
- `x` denotes the head of a snake that has died.

Each character of the snake tells you what direction the snake is currently heading in:

- `w`, `W`, or `^` denotes up
- `a`, `A`, or `<` denotes left
- `s`, `S`, or `v` denotes down
- `d`, `D`, or `>` denotes right

At each time step, each snakes moves according to the following rules:

- Each snake moves one step in the direction of its head.
- If the head crashes into the body of a snake or a wall, the snake dies and stops moving. When a snake dies, the head is replaced with an `x`.
- If the head moves into a fruit, the snake eats the fruit and grows by 1 unit in length. Each time fruit is consumed, a new fruit is generated on the board.

In the example above, after one time step, the board will look like this:

```
##############
#         *  #
#     s      #
#     v   #  #
#     v   #  #
#   s >>>D#  #
#   v     #  #
# A<<  *  #  #
#            #
##############
```

After one more time step, the board will look like this:

```
##############
#          *  #
#      s      #
#      v   #  #
#      v   #  #
#      >>>x#  #
#    s     #  #
#A<<<  *   #  #
#           #
##############
```

Snakes are guaranteed to be at least three units long.

Numbering snakes

Each snake on the board is numbered depending on the position of its tail, in the order that the tails appear in the file (going from top-to-bottom, then left-to-right). For example, consider the following board with four snakes:

```
##############
#  s  d>>D  #
#  v   A<a  #
#  S    W   #
#       ^   #
#       w   #
##############
```

Snake 0 is the snake with tail `s`, snake 1 has tail `d`, snake 2 has tail `a`, and snake 3 has tail `w`.

Once the snakes are numbered from their initial positions, the numbering of the snakes does not change throughout the game.

Game board

A game board is a grid of characters, not necessarily rectangular. Here's an example of a non-rectangular board:

```
##############
#            #######
#####             ##
#   #             ##
#####             ######
#                 ##   #
#                 ######
#                 ##
#                  #
#      #####       #
########   #########
```

Note that each row can have a different number of characters, but will start and end with a wall (`#`). You can also assume that the board is an enclosed space, so snakes can't travel infinitely far in any direction.

The `game_state_t` struct

A snake game is stored in memory in a `game_state_t` struct. The struct contains the following fields:

- `unsigned int num_rows`: The number of rows in the game board.
- `char** board`: The game board in memory. Each element of the `board` array is a `char*` pointer to a character array containing a row of the map.
- `unsigned int num_snakes`: The number of snakes on the board.
- `snake_t* snakes`: An array of `snake_t` structs.

The `snake` struct

Each `snake` struct contains the following fields:

- `unsigned int tail_row`: The row of the snake's tail.
- `unsigned int tail_col`: The column of the snake's tail.
- `unsigned int head_row`: The row of the snake's head.
- `unsigned int head_col`: The column of the snake's head.
- `bool live`: `true` if the snake is alive, and `false` if the snake is dead.

Please don't modify the provided struct definitions. You should only need to modify `state.c` `snake.c`, and `custom_tests.c` in this project.

Task 1: `create_default_state`

Implement the `create_default_state` function in `state.c`. This function should create a default snake game in memory with the following starting state (which you can hardcode), and return a pointer to the newly created `game_state_t` struct.

```
####################
#                  #
# d>D    *         #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
#                  #
####################
```

```
create_default_state
```

Arguments	None	
Return values	`game_state_t *`	A pointer to the newly created `game_state_t` struct.

Hints

- The board has 18 rows, and each row has 20 columns. The fruit is at row 2, column 9 (zero-indexed). The tail is at row 2, column 2, and the head is at row 2, column 4.
- Which part of memory (code, static, stack, heap) should you store the new game in?
- `strcpy` may be helpful.

Testing and debugging

You can run `make run-unit-tests` to check your implementation for each task. Please note that the unit tests are *not* comprehensive, and passing them does not guarantee that your implementation is fully correct. However, they should be helpful to get you started with debugging.

If your implementation isn't working, it's time to start debugging. You can add `printf` statements in your code to print out variables during code execution, and then run `make run-unit-tests` again to see the output of your print statements.

Also, you can use `make debug-unit-tests` to start CGDB. In CGDB, you can set a breakpoint in your own code (hint: see the GDB reference card for how to set a breakpoint in a different file). Then type `run` or `r` to start the program, and it'll pause at your breakpoint.

Tip: If you see "Segmentation fault (core dumped)", this means that your program crashed. One way to start debugging is by starting CGDB, `run`ning the program with no breakpoints, and then typing `backtrace` or `bt` to see what line of code the program crashed at.

Task 2: `free_state`

Implement the `free_state` function in `state.c`. This function should free all memory allocated for the given state, including all `snake` structs and all `map->board` contents.

`free_state`		
Arguments	`game_state_t* state`	A pointer to the `game_state_t` struct to be freed

Return values	None

Testing and debugging

To test if we correctly freed memory for the game state, run `make valgrind-test-free-state` to check for memory leaks. If nothing is leaked, then you've passed the unit test for this task.

Task 3: `print_board`

Implement the `print_board` function in `state.c`. This function should print out the given game board to the given file pointer.

`print_board`		
Arguments	`game_state_t* state`	A pointer to the `game_state_t` struct to be printed
	`FILE* fp`	A pointer to the file object where the board should be printed to
Return values	None	

Hints

- The `fprintf` function will help you print out characters and/or strings to a given file pointer.

Testing and debugging

Run `make run-unit-tests` and `make debug-unit-tests` to test and debug, just like before.

If your function executes successfully (doesn't segfault or crash) but doesn't print the correct output, the board you printed will be in `unit-test-out.snk`. A correctly-printed board should match the default board from Task 1.

Task 4: `update_state`

Implement the `update_state` function in `state.c`. This function should move the snakes one timestep according to the rules of the game.

You are free to implement this function however you want, but if you'd like, you can work through this task by implementing the helper functions we've provided. Helper functions are not graded; for this task, we'll only be checking that `update_state` is correct.

Task 4.1: Helpers

We have provided the following helper function definitions that you can implement. These functions are entirely independent of any game board or snake; they only take in a single character and output some information about that character.

- `bool is_tail(char c)`: Returns true if `c` is part of the snake's tail. The snake's tail consists of these characters: `wasd`. Returns false otherwise.
- `bool is_head(char c)`: Returns true if `c` is part of the snake's head. The snake's head consists of these characters: `WASDx`. Returns false otherwise.
- `bool is_snake(char c)`: Returns true if `c` is part of the snake. The snake consists of these characters: `wasd^<v>WASDx`. Returns false otherwise.
- `char body_to_tail(char c)`: Converts a character in the snake's body (`^<v>`) to the matching character representing the snake's tail (`wasd`). The output may be undefined for characters that are not a snake's body.
- `char head_to_body(char c)`: Converts a character in the snake's head (`WASD`) to the matching character representing the snake's body (`^<v>`). The output may be undefined for characters that are not a snake's head.
- `unsigned int get_next_row(unsigned int cur_row, char c)`: Returns `cur_row + 1` if `c` is `v` or `s` or `S`. Returns `cur_row - 1` if `c` is `^` or `w` or `W`. Returns `cur_row` otherwise.
- `unsigned int get_next_col(unsigned int cur_col, char c)`: Returns `cur_col + 1` if `c` is `>` or `d` or `D`. Returns `cur_col - 1` if `c` is `<` or `a` or `A`. Returns `cur_col` otherwise.

Unit tests are not provided for these helper functions, so you'll have to write your own tests in `custom_tests.c` to make sure that these are working as expected. Make sure that these tests comprehensively test your helper functions--our autograder will run your tests on buggy implementations to make sure that your tests can catch bugs!

When writing a unit test, the test function should return `false` if the test fails, and `true` if the test passes. You can use `printf` to print out debugging statements. Some of the assert helper functions in `asserts.h` might be useful.

Once you've written your own unit tests, you can run them with `make run-custom-tests` and `make debug-custom-tests`.

Task 4.2: `next_square`

Implement the `next_square` helper function in `state.c`. This function returns the character in the cell the given snake is moving into. This function should not modify anything in the game stored in memory.

`next_square`		
Arguments	`game_state_t* state`	A pointer to the `game_state_t` struct to be analyzed
	`int snum`	The index of the snake to be analyzed
Return values	`char`	The character in the cell the given snake is moving into

As an example, consider the following board:

```
##############
#            #
#            #
#            #
#   d>D*     #
#            #
#        s   #
#        v   #
#        S   #
##############
```

Assuming that `state` is a pointer to this game state, then `next_square(state, 0)` should return `*`, because the head of snake 0 is moving into a cell with `*` in it. Similarly, `next_square(state, 1)` should return `#` for snake 1.

The helper functions you wrote earlier might be helpful for this function (and the rest of this task too). Also, check out `get_board_at` and `set_board_at`, which are helper functions we wrote for you.

Use `make run-unit-tests` and `make debug-unit-tests` to run the provided unit tests. You can also use `p print_board(state, stdout)` to print out your entire board while debugging in `cgdb`.

Task 4.3: `update_head`

Implement the `update_head` function in `state.c`. This function will update the head of the snake.

Remember that you will need to update the head both on the game board and in the `snake_t` struct. On the game board, add a character where the snake is moving. In the `snake_t` struct, update the row and column of the head.

`update_head`		
Arguments	`game_state_t* state`	A pointer to the `game_state_t` struct to be updated

	`int snum`	The index of the snake to be updated
Return values	None	

As an example, consider the following board:

```
##############
#   d>D      #
#        *   #
#        W   #
#        ^   #
#        ^   #
#        w   #
#            #
#            #
##############
```

Assuming that `state` is a pointer to this game state, then `update_head(state, 0)` will move the head of snake 0, leaving all other snakes unchanged. In the `snake_t` struct corresponding to snake 0, the `head_col` value should be updated from 6 to 7, and the `head_row` value should stay unchanged at 1. The new board will look like this:

```
##############
#   d>>D     #
#        *   #
#        W   #
#        ^   #
#        ^   #
#        w   #
#            #
#            #
##############
```

Note that this function ignores food, walls, and snake bodies when moving the head.

Use `make run-unit-tests` and `make debug-unit-tests` to run the provided unit tests. You can also use `p print_board(state, stdout)` to print out your entire board while debugging in `cgdb`.

Task 4.4: `update_tail`

Implement the `update_tail` function in `state.c`. This function will update the tail of the snake.

Remember that you will need to update the tail both on the game board and in the `snake_t` struct. On the game board, blank out the current tail, and change the new tail from a body character (`^<v>`) into a tail character (`wasd`). In the `snake_t` struct, update the row and column of the tail.

```
update_tail
```

Arguments	`game_state_t* state`	A pointer to the `game_state_t` struct to be updated
	`int snum`	The index of the snake to be updated
Return values	None	

As an example, consider the following board:

```
##############
#   d>D      #
#        *   #
#        W   #
#        ^   #
#        ^   #
#        w   #
#            #
#            #
##############
```

Assuming that `state` is a pointer to this game state, then `update_tail(state, 1)` will move the tail of snake 1, leaving all other snakes unchanged. In the `snake_t` struct corresponding to snake 1, the `tail_row` value should be updated from 6 to 5, and the `tail_col` value should stay unchanged at 9. The new board will look like this:

```
##############
#   d>D      #
#        *   #
#        W   #
#        ^   #
#        w   #
#            #
#            #
#            #
##############
```

Use `make run-unit-tests` and `make debug-unit-tests` to run the provided unit tests. You can also use `p print_board(state, stdout)` to print out your entire board while debugging in `cgdb`.

Task 4.5: `update_state`

Using the helpers you created, implement `update_state` in `state.c`.

As a reminder, the rules for moving a snake are as follows:

- Each snake moves one step in the direction of its head.
- If the head crashes into the body of a snake or a wall, the snake dies and stops moving. When a snake dies, the head is replaced with an `x`.
- If the head moves into a fruit, the snake eats the fruit and grows by 1 unit in length. (You can implement growing by 1 unit by updating the head without updating the tail.) Each time fruit is consumed, a new fruit is generated on the board.

The `int (*add_food)(game_state_t* state)` argument is a function pointer, which means that `add_food` is a pointer to the code section of memory. The code that `add_food` is pointing at is a function that takes in `game_state_t* state` as an argument and returns an `int`. You can call this function with `add_food(x)`, replacing `x` with your argument, to add a fruit to the board.

`update_state`		
Arguments	`game_state_t* state`	A pointer to the `game_state_t` struct to be updated
	`int (*add_food)(game_state_t* state)`	A pointer to a function that will add fruit to the board
Return values	None	

Use `make run-unit-tests` and `make debug-unit-tests` to run the provided unit tests. You can also use `p print_board(state, stdout)` to print out your entire board while debugging in `cgdb`.

Task 5: `load_board`

Implement the `load_board` function in `state.c`. This function will read a game board from a file into memory.

Remember that each row of the game board might have a different number of columns. Your implementation should be memory-efficient and should not allocate more memory than necessary to store the board. For example, if a row is 3 characters long, you shouldn't be allocating 100 bytes of space for that row. We highly recommend against using `getline` since it is not memory efficient and will most likely fail autograder tests.

Tasks 5 and 6 combined will create a `game_state_t` struct in memory with all its fields set up. In this task, please set `num_snakes` to 0 and set the `snakes` array to `NULL`, since these will be initialized in task 6.

`load_board`		
Arguments	`char* filename`	The name of the file where the board is stored
Return values	`game_state_t *`	A pointer to the newly created `game_state_t` struct. `NULL` if the file does not exist.

Use `make run-unit-tests` and `make debug-unit-tests` to run the provided unit tests. You can also use `p print_board(state, stdout)` to print out your entire board while debugging in `cgdb`.

Task 6: `initialize_snake`

Implement the `initialize_snake` function in `state.c`. This function takes in a game board and creates the array of `snake_t` structs.

You are free to implement this function however you want, but if you'd like, you can work through this task by implementing the helper function we've provided.

Task 6.1: `find_head`

Implement the `find_head` function in `state.c`. Given a `snake_t` struct with the tail row and column filled in, this function traces through the board to find the head row and column, and fills in the head row and column in the struct.

`find_head`		
Arguments	`game_state_t* state`	A pointer to the `game_state_t` struct to be analyzed
	`int snum`	The index of the snake to be analyzed
Return values	None	

As an example, consider the following board:

```
##############
#            #
#        *   #
#            #
#   d>v      #
#     v      #
#  W  v      #
#  ^<<<      #
#            #
##############
```

Assuming that `state` is a pointer to this game state, then `find_head(state, 0)` will fill in the `head_row` and `head_col` fields of the snake 0 struct with 6 and 3, respectively.

Use `make run-unit-tests` and `make debug-unit-tests` to run the provided unit tests. You can also use `p print_board(state, stdout)` to print out your entire board while debugging in `cgdb`.

Task 6.2: `initialize_snake`

Using `find_head`, implement the `initialize_snake` function in `state.c`. You can assume that the state passed into this function is the result of calling `load_board`, but you may not assume that the `snakes` array is defined. This means the board-related fields are already filled in, and you only need to fill in `num_snakes` and create the `snakes` array.

You may assume that all snakes on the board start out alive.

`initialize_snakes`		
Arguments	`game_state_t* state`	A pointer to the `game_state_t` struct to be filled in
Return values	`game_state_t* state`	A pointer to the `game_state_t` struct with fields filled in. This can be the same as the struct passed in (you can modify the struct in-place).

Use `make run-unit-tests` and `make debug-unit-tests` to run the provided unit tests. You can also use `p print_board(state, stdout)` to print out your entire board while debugging in `cgdb`.

Task 7: `main`

Using the functions you implemented in all the previous tasks, fill in the blanks in `snake.c`. Each time the `snake.c` program is run, the board will be updated by one time step.

To test your full implementation, run `make run-integration-tests`.

To debug your implementation, run `cgdb --args ./snake -i tests/TESTNAME-in.snk -o tests/TESTNAME-out.snk`. To check for memory leaks or out-of-bounds reads/writes, you can run `valgrind ./snake -i tests/TESTNAME-in.snk -o tests/TESTNAME-out.snk`. Replace `TESTNAME` with one of the test names in the `tests` folder:

- `01-simple`
- `02-direction`
- `03-tail`
- `04-food`
- `05-wall`
- `06-small`
- `07-medium`
- `08-multisnake`
- `09-everything`
- `10-filled`

- `11-manyclose`
- `12-corner`
- `13-sus`
- `14-orochi`
- `15-hydra`
- `16-huge`
- `17-wide`
- `18-tall`
- `19-101-127`
- `20-long-line`
- `21-bigL`

You can also run `make run-nonexistent-input-file-test` to make sure that your program correctly exits with error code -1 if the input file doesn't exist.

Task 8: Partner/Feedback Form

Congratulations on finishing the project! This is a relatively new project, so we'd love to hear your feedback on what can be improved for future semesters.

Please fill out this short form , where you can offer your thoughts on the project and (if applicable) your partnership. Any feedback you provide won't affect your grade, so feel free to be honest and constructive.

Submission and Grading

Submit your code to the Project 1 Gradescope assignment. Make sure that you have only modified `snake.c`, `state.c`, and `custom_tests.c`. You can submit to Gradescope as many times as you want, and the score you see on Gradescope will be your final score for this project.

Just for fun: play snake

Now you can play a game with the code you've written by `make interactive-snake` followed by `./interactive-snake`. Use the `wasd` keys to control your snake!

To speed up or slow down the game, you can run `./interactive-snake -d 0.5` (replacing 0.5 with the number of seconds between time steps). During the game, you can also press `]` to move faster and `[` to move slower.
