Homework 6: Reversi

CS 211

Spring 2022

Code Due:	May 19, 2022, 11:59 PM, Central Time
Self-Eval Due:	May 22, 2022, 11:59 PM, Central Time
Partners:	Yes; mark your partner on Gradescope on each submission

Contents

Purpose

The goal is to get you writing more interesting algorithms and using more interesting data types.

Getting it

Download the project ZIP file to your computer, unzip it, and open the resulting directory in CLion. (Be careful that you open the `hw05` directory and not some sub- or superdirectory thereof. If you do, CLion will create a bogus `CMakeLists.txt` that won't be able to find SDL2.)

To complete this homework on your own computer, you need a C++14 toolchain and the SDL2 libraries. Follow these instructions to install the software you need.

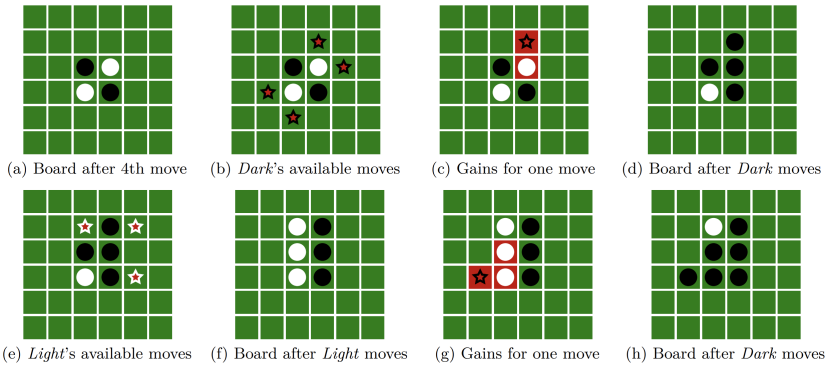
Specification

The game Reversi is played by two players, *Dark* and *Light*, laying dark- and light-colored tiles on an 8-by-8 board. The game proceeds in two phases.

In the opening phase, the players alternate turns, with *Dark* going first. In this phase, they may only play in the center four squares of the board ($(3,3)$, $(3,4)$, $(4,3)$, and $(4,4)$ if 0-based). The opening phase ends when those center four squares are occupied.

In the main phase, each move *must capture* at least one of the other player's tiles, as follows. The current player places a tile in an unoccupied square so that it forms at least one straight line—horizontal, vertical, or diagonal—with one or more of the other player's tiles in the middle and one of the current player's tiles on the other end. Then the other player's tiles in the line(s) are flipped to the current player. (See Figure 1 for some example moves.)

The players take turns unless one player cannot play, in which case the other player may play again. The game is over when neither player can play. The winner is the player with more tiles on the final board (or it may be a tie).



(a) Board after 4th move (b) *Dark*'s available moves (c) Gains for one move (d) Board after *Dark* moves

(e) *Light*'s available moves (f) Board after *Light* moves (g) Gains for one move (h) Board after *Dark* moves

Figure 1: Example of possible 5th through 7th moves on a 6-by-6 board

Generalization to smaller sizes

To facilitate testing, we generalize the rules of Reversi to allow boards with dimensions down to 2-by-2, including non-square boards (*e.g.,* 7-by-4). To generalize the opening phase of the game to a w-by-h board, we define the center four squares to be at positions $(c_x - 1, c_y - 1)$, $(c_x - 1, c_y)$, $(c_x, c_y - 1)$, and (c_x, c_y), where $c_x = \lfloor w/2 \rfloor$ and $c_y = \lfloor h/2 \rfloor$.

For your convenience, a rectangle containing these positions (which you can iterate over) is returned by `Board::center_positions() const`.

Your implementation

To start the game, the user runs the *reversi* executable with either no command-line arguments, or two arguments, the width and height of the board. If the width or height are out of range, or if some other number of arguments is given, then the program exits with an error message.

The user interface must make it possible to play the game using either the mouse or the keyboard (or both, if you wish). The game must allow the user to make all legal moves and no illegal moves. It must display the state of the model so that the user can see which squares are occupied by light tiles, which by dark tiles, and which by neither. When the game is not over, it should display whose turn it is and give some indication of which squares are valid to play in. For full credit, it should also give feedback on which tiles will flip as a result of each possible move. When the game ends, the UI must indicate which player, if any, has won.

It should not be possible for the user to cause the game to crash via any interaction with the UI.

Command line arguments can be passed in to the program by clicking "Edit configurations..." in the drop-down menu and then entering two numbers separated by a space in the "Program arguments" box. For example, you can enter "2 4" for a 4x2 board. Then apply the changes and run the program as normal.

Helper type reference

To help define the model, we have provided several types to build upon. We present these types in this section before discussing the model itself in Section "*The model*". The helper types are:

- A `Position_set` represents a set of game board positions. (See Section "*The `Position_set` class*".)
- Type `Move` represents a possible move as a pair of a single `ge211::Posn<int>` in which a player can place a tile, and the `Position_set` of all positions

gained by that move. (See Section "*Type aliases `Move` and `Move_map`*".)

- Type `Move_map` holds a collection of available `Move`s whose first components are distinct, and which supports looking up `Move`s by their first components. (This is the type of the `next_moves_` member variable of the `Model` class that your operations need to maintain.) (See Section "*Type aliases `Move` and `Move_map`*".)
- `Player` is an `enum class` with three enumerators: `dark`, `light`, and `neither`. (See Section "*The `Player` enumeraton*".)
- Class `Board` represents the state of the board. (See Section "*The `Board` class*".)

The `Position_set` class

The `Position_set` class is used to represent a set of positionsand offers a standard selection of set operations. `Position_set`s support equality and stream insertion (printing), which may be helpful for testing and debugging.

Positions are limited to those whose coordinates are both less than 8, which suffices for Reversi.

The full documentation may be found in the src/position_set.hxx header, but the highlights are described here.

```
using value_type = ge211::Posn<int>;
```

Type alias for the type of values stored in a `Position_set`.

```
Position_set::Position_set();
```

Constructs the empty set of positions.

```
Position_set::Position_set(
    std::initializer_list<value_type>);
```

Constructs the set of positions listed, like so:

```
Position_set pset{
    {2, 3}, {3, 2}, {4, 1}
};
```

```
bool Position_set::empty() const;
```

Returns whether this set is empty.

```
void Position_set::clear();
```

Removes all elements from this `Position_set`.

```
bool
Position_set::operator[](value_type)const;
```

Looks up the given position in the set, returning a `bool` indicating whether it is present.

```
Position_set::reference
Position_set::operator[](value_type);
```

Looks up the given position (within the square brackets) in the set, returning a reference-like object that can be assigned a `bool` to change whether the position is in the set. For example:

```
Position_set pset;

// add {2, 3} to `pset`:
pset[{2, 3}] = true;

// remove {2, 3} from `pset`:
pset[{2, 3}] = false;
```

```
Position_set&
Position_set::operator|=(
    Position_set);
```

Adds the positions in the passed in set to this set. Additionally, `Position_set` supports the full complement of set operations (the operations with the `=` store the result of the operation in the left operand):

- intersection: `a & b` and `a &= b`
- union: `a | b` and `a |= b`
- symmetric difference: `a ^ b` and `a ^= b`
- complement: `~a`

Note that set difference can be accomplished with intersection and complement: `a & ~b`.

```
Position_set::iterator
Position_set::begin() const;

Position_set::iterator
Position_set::end() const;
```

These functions return the iterators necessary to iterate over a `Position_set` using a range-based `for` loop, like so:

```
for (ge211::Posn<int> pos : pset) {
    ...
}
```

Type aliases `Move` and `Move_map`

```
struct Move
{
    ge211::Posn<int> const first;
    Position_set           second;
};
```

The `Move` type (see src/move.hxx) is an instantiation of the standard library's `std::pair` template struct. It has two member variables, `first` and `second`. The former contains the position of the move, and the latter is the set of all positions gained by the move, including both the move itself and any flips.

`Move`s support equality and stream insertion (printing), which may be helpful for testing and debugging.

```
using Move_map =
    std::unordered_map<
        ge211::Posn<int>,
        Position_set
    >;
```

Type `Move_map` (also in src/move.hxx) is an instantiation of the standard library's class template `std::unordered_map`with `ge211::Posn<int>` as the key type and `Position_set` as the value type. `Move_map`s support equality but not stream insertion.

Each key-value pair element in the `Move_map` is of type `Move`.

All the `std::unordered_map` operations are available, but you will mostly likely need only these three:

```
Position_set&
Move_map::operator[](ge211::Posn<int>);
```

Type `Move_map` overloads the indexing operator (square brackets) to take a `ge211::Posn<int>`. If the position is not already present then it inserts the given position paired with an empty `Position_set`. It then returns a reference to the `Position_set`, which allows the caller to modify or assign it. Thus, we can associate a position `pos` with a set of positions `pset` by indexing the move map with `pos` and assigning `pset` to the result:

```
mmap[pos] = pset;
```

```
bool Move_map::empty() const;
```

Returns whether this move map is empty, meaning no positions have been mapped to position sets.

```
void Move_map::clear();
```

Removes all moves from this move map.

The `Player` enumeraton

There are three `Player` values: `Player::dark` and `Player::light` represent the two players, and `Player::neither` represents absence of a player. `Player`s support equality and stream insertion (printing), which may be helpful for testing and debugging.

There is one operation you will need:

```
Player other_player(Player);
```

Returns the other player given the current player.

The `Board` class

The `Board` class stores the state of the Reversi board. It is, essentially, an updatable mapping from in-bounds `Board::Position`s (an alias for `ge211::Posn<int>`) to `Player`s. `Board`s support equality and stream insertion (printing), which may be helpful for testing and debugging.

The full documentation of the `Board` class is available in the src/board.hxx header file, but the highlights you are likely to want are described here.

```
using Dimensions = ge211::Dims<int>;
using Position   = ge211::Posn<int>;
using Rectangle  = ge211::Rect<int>;
```

Aliases for the geometry types used by the `Board` class (as well as the rest of the model.)

```
Board::Dimensions
Board::dimensions() const;
```

Returns the dimensions of the board. (Note that `Board::Dimensions` is a type alias for `ge211::Dims<int>`.)

```
Player
Board::operator[](Position)
const;
```

Returns the player at the given position.

```
Board::reference
Board::operator[](Position);
```

Returns a reference-like object that, when a `Player` is assigned to it, stores that `Player` in the board at the given position. For example, this statement stores `Player::dark` at board position $(2, 3)$:

```
board[{2, 3}] = Player::dark;
```

```
void
Board::set_all(Position_set, Player);
```

Stores the given `Player` in the board at all the positions in the given `Position_set`. For example, these statements store `Player::light` at two board positions:

```
Position_set pset{{2, 5}, {3, 4}};
board.set_all(pset, Player::light);
```

```
size_t
Board::count_player(Player) const;
```

Returns the number of times the given player appears on the board.

```
Board::Rectangle
Board::all_positions() const;
```

Returns a rectangle containing all of the board's positions. Since `Board::Rectangle`s are iterable, this can be used to iterate over the board's positions:

```
for (auto pos : board.all_positions()) {
    ...
}
```

```
Board::Rectangle
Board::center_positions() const;
```

Returns a rectangle containing just the four center positions that are playable in the opening phase. Since `ge211::Rect<int>`s are iterable, this can be used to iterate over the four center positions:

```
for (auto pos : board.center_positions()) {
    ...
}
```

```
static
std::vector<Board::Dimensions> const&
all_directions();
```

Returns a (borrowed) `std::vector` containing the eight direction vectors (as `ge211::Dims<int>`). The eight directions are all combinations of -1, 0, and 1 except for the zero vector $\langle 0, 0 \rangle$. This can be used to iterate over all possible line directions when evaluating a potential move:

```
for (auto dim : Board::all_directions()) {
    ...
}
```

Design orientation

In this section we describe the design of the three classes that you have to complete.

The model

The `Model` class (src/model.{hxx,cxx}) encapsulates the state of the game and its rules. In particular, it keeps track of:

- the current turn, if the game is ongoing (`Player turn_`),
- the winning player, if any (`Player winner_`),
- the state of the board (`Board board_`), and
- a cache of which moves are available to the current player (`Move_map next_moves_`).

While it is possible to generate the available moves on demand given the other three data members, this information is not cheap to compute, and the view and controller will most likely need it much more often than it changes. So it makes sense to compute the next possible moves when the game starts and then after each turn, rather than recomputing it whenever the UI wants to know which moves are valid.

In the `Model` class, we have defined a number of member functions that you may want to call from the view, the controller, or elsewhere in the model:

- `Model::board() const` returns a `ge211::Rectangle` that contains all positions in the board.
- `Model::is_game_over() const` returns a `bool` indicating whether the game is over.
- `Model::turn() const` returns the current player, if any.
- `Model::winner() const` returns the winning player, if any.
- `Model::operator[](Position) const` returns the `Player` at the given position on the board.
- `Model::find_move(Position) const` returns a pointer to the `Move` that would result from playing at the given position, if allowed, or `nullptr` if not allowed.

The last of these depends on the contents of `next_moves_` being correct. Ensuring that invariant is your job. In particular, there are two members of the `Model` class that are incomplete:

```
void
Model::play_move(Model::Position);
```

This function plays a move at the given position if allowed, or throws an exception if disallowed. We have already provided code to check the legality of the move for you and throw if necessary. Our starter code leaves a pointer to the valid `Move` in a local variable, `movep`. Your responsibility is to 1) actually execute the move by modifying the board, 2) advance the turn—to the other player if they can move, or back to the same player if the other player cannot move, or to game over if neither player can move, and 3) leave `next_moves_` in a correct state.

```
Model::Model(int width, int height);
```

This constructor initializes the model. We've provided you the member initializer for the board, but you need to write the code for filling `next_moves_` with the moves available to the first player on the first turn. (This should happen via a private helper function that `Model::play_move` calls as well.)

The view

The responsibility of the `View` class is to present the state of the model in such a way that users can play the game. We have not specified what the game should look like, other than that it must be playable as described in Section "*Your implementation*". You may emulate the style of the diagrams in Figure 1 if you wish, or design something else.

We have provided you with a minimal `View` class in src/view.{hxx,cxx}, which you will have to complete to make the game playable. This starter `View` class defines a single member variable, `Model const& model_`. It defines one constructor, which initializes `model_`; you may want to extend this constructor to initialize your sprites as well. Two member functions, for determining the window title and dimensions, are provided for you (though you may change them if you want to determine these things differently).

There is one function for you to write: `View::draw(ge211::Sprite_set&)`. This function is, of course, responsible for determining what appears on the screen. You will most likely want to add at least one parameter to it, so that the controller can communicate control state (such as the position of the mouse) to the view.

The controller

The responsibility of the `Controller` class is to receive input from the user and decide what to do with it. We have not specified how control should work, other than that the game must be playable as described in Section "*Your implementation*". You may provide mouse control, keyboard control, or whatever usable interface you desire.

We have provided you with a minimal `Controller` class in src/controller.{hxx,cxx}, which you will have to complete to make the game playable. This starter `Controller` class defines two member variables to hold the model and the view. It defines two constructors, each of which allows specifying the model dimensions, and initializes the model and the view. We have also overridden member functions `draw`, `initial_window_dimensions`, and `initial_window_title` in order to delegate those three responsibilities to the view.

You will need to add user-input handling to the controller by overriding additional member functions of `ge211::Abstract_game`, such as:

- `on_mouse_down` if you want to react to mouse clicks,
- `on_mouse_move` if you want to react to mouse motion, and
- `on_key` if you want to react to typing on the keyboard.

You will probably want to add at least one private member variable to the `Controller` class to keep track of the UI state. For example, if you want

the view to indicate the current player's available moves and their consequences based on where the mouse is pointing, then the controller needs to store the mouse position on each call to `on_mouse_move` so that it can then pass it to the view when it calls `View::draw` from `Controller::draw`.

Implementation hints

This section provides supplementary material to help you figure out how to implement the specification.

Model factoring

Your main responsibility with respect to the model implementation is to handle playing moves, and the most difficult part of that is computing the available moves to update `next_moves_`. In src/model.hxx we have declared six private helper functions that you should implement to break down this task. We strongly recommend that you implement these first, as they will make the other functions more straightforward. The private `Model` helpers are:

- `Position_set find_flips_(Position, Dimensions) const` takes the position of a prospective move by the current player and a direction to search in (as provided by `Board::all_directions()`). It searches for a straight line of opposing player tiles bounded by the given position at one end and an existing tile belonging to the current player on the other end. It returns the set of those opposing player positions (which will be empty if there is no such line).

  This is a helper for `evaluate_position_()`.

- `Position_set evaluate_position_(Position) const` takes the position of a prospective move and returns a `Position_set` containing all positions that would be gained by the current player playing in that position, if allowed (or the empty set if playing in the given position is disallowed).

  This is a helper for `compute_next_moves_()`.

- `void compute_next_moves_()` clears out `next_moves_` and then regenerates it with all moves currently available to the current player.

  This is a helper for the `Model(int, int)` constructor and for `advance_turn_()`.

- `bool advance_turn_()` switches the turn to the other player, regenerates `next_moves_`, and then returns whether any moves are actually available to the new current player.

  This is a helper for `really_play_move_()`.

- `void set_game_over_()` makes the game over by setting the current player to `Player::neither` and storing the winner, if any, in `winner_`.

  This is also a helper for `really_play_move_()`.

- `void really_play_move_(Move)` executes the given move by setting the appropriate positions on the board and then advancing the turn or setting game over. It needs to try advancing the turn twice—since if the other player cannot play then the current player gets to play again. Only if neither player has any moves available is the game over.

  You'll have to determine where calling this is useful.

Algorithm for computing moves

Computing `next_moves_` requires a somewhat involved algorithm, since it must evaluate every unoccupied board position, or just four in the opening phase (`compute_next_moves_`); and to evaluate each position (`evaluate_position_`), it must check for "flippable lines" of opposing player tiles in all eight directions (`find_flips_`).

Finding one line of flips

Given a starting, unoccupied position `start` and a direction `dir` to search in, we can find a line of flippable positions as follows. Start with an empty `Position_set` to hold the result, and begin checking positions moving away from `start`: `start + dir`, `start + 2 * dir`, and so on. At each position there are three possibilities:

- If we reach a position that would go off the board (check that first!) or is unoccupied (i.e., no player is at that position) then there is no flippable line to find, so the result is the empty set.
- If a position contains an opposing player tile then we add that position to our result `Position_set` and move on to the next.
- If we reach a position containing the current player's tile then we return the `Position_set` that we've accumulated.

Evaluating a position

We evaluate a position `pos` as the set of all positions that the current player would gain by playing there—or the empty `Position_set` if playing there is not allowed. First we check if it's unoccupied, since occupied positions are not playable and evaluate to the empty set. Otherwise, we need to search for flippable lines in all eight directions starting from `pos` (probably by iterating over the result of `Board::all_directions()`), and union together the eight resulting `Position_set`s. (You can do this by starting with an empty `Position_set` and then using the `|=` operator to union each result of `find_flips_` into it.) If the union of the sets is empty then position `pos` is not playable for the current player and the result of the evaluation is the empty set. Otherwise, we must add `pos` to the set of positions before returning it, since `pos` will be gained by the potential move as well.

Evaluating the whole board (as necessary)

Evaluating the whole board means first clearing `next_moves_`, then checking for available moves and adding them to `next_moves_`.

Before evaluating every board position, we need to check whether any of the four center positions (`board_.center_positions()`) are unoccupied, which would indicate that we are still in the opening phase of the game. Since playing in one of those positions would not flip any other tiles, each unoccupied center position gets mapped to the singleton set of itself:

```
next_moves_[pos] = {pos};
```

If, after adding any unoccupied center positions, `next_moves_` is non-empty, then we are still in the opening phase and should return `next_moves_` without checking the rest of the board.

Otherwise we are in the main phase, so we must evaluate each position in the board and record each *non-empty* evaluation in `next_moves_`. In particular, if some position `pos` is a legal move that evaluates to some `Position_set pset` then we store this fact in `next_moves_` like so:

```
next_moves_[pos] = pset;
```

Positions that evaluate to the empty set must not be added to `next_moves_`, as that would cause `play_move` to consider them to be available moves.

The UI

The UI description in Section "*Your implementation*" imposes a number of requirements on what the player can do. You are free to implement these requirements however you like, but here is a list of suggestions for how you could:

- Display the board as a grid of squares, with the *Dark* and *Light* players' tiles as slightly smaller black and white circles placed over them. (To place one sprite atop another, you need to provide different `z` values as a third argument to `Sprite_set::add_sprite`.)
- Allow the user to play a move by clicking in the desired square. (If the user clicks in a disallowed square or after the game is over, either don't react or display an error indication.)
- Once the game is over, indicate the winner by rendering all non-winning tiles in gray instead of black or white.
- Indicate the current turn (when the game isn't over) by having an image of the current player's tile (or something similar?) follow the mouse pointer.

  (This requires adding the mouse position as a private member variable in the `Controller` class.)
- When the mouse points to a square in which the current player is allowed to move, indicate the effect of moving in that position by changing the color of the squares in the positions that would be gained by the player. The view can easily discover this information by calling `Model::find_move` with the logical (board) position of the square that the mouse pointer currently points to.

  (This also requires adding the mouse position as a private member variable in the `Controller` class.)

  You should use the following helper functions defined in the `View` class.

```
View::Position
View::board_to_screen(
        Model::Position      logical)
    const;

Model::Position
View::screen_to_board(
        View::Position      physical)
    const;
```

You should also implement the following helper function in `View`.

```
void
View::add_player_sprite_(
        ge211::Sprite_set&  sprites,
        Player              which,
        ge211::Posn<int>    physical,
        int                 z_layer)
    const;
```

The first two convert positions from logical to physical and back. The third one adds the tile sprite for the given player at the given physical position and z layer, while ignoring `Player::neither` and turning non-winning players' tiles gray if the game is over.

Testing private members

Given that the model's move evaluation algorithm involves several steps and nested loops, how can you test some smaller portions of it? Thankfully, you are required to factor it into smaller, more testable pieces via the helper functions. But these are private, which means that your tests won't be able to access them, right?

Not exactly. We declared a *friend* `struct Test_access` in the `Model` class, which means that `Model` grants, to any members of a struct called `Test_access`, access to its own private members. This is there for the grading tests, but you can define a `Test_access` struct in order to provide your tests with privileged access to the model as well. You can also declare `struct Test_access` to be a friend of any other classes that you like, in order to facilitate testing them as well.

For example, if you wanted your tests to be able to access the board directly and to call the private `find_flips_` helper, you define `struct Test_access` as below. Then you could use it like the test case below named "simple flips case".

```
struct Test_access
{
    Model& model;

    Board& board()
    {
        return model.board_;
    }

    Position_set
    find_flips(Model::Position p,
               Model::Dimensions d)
    {
        return model.find_flips_(p, d);
    }
};
```

```
TEST_CASE("simple flips case")
{
    Model model;
    Test_access t{model};

    t.board()[{2, 2}] = Player::dark;
    t.board()[{2, 3}] = Player::light;

    Position_set f;

    f = t.find_flips({2, 4}, {0, 1});
    CHECK(f.empty());

    f = t.find_flips({2, 4}, {0, -1});
    CHECK(f == Position_set{{2, 3}});
}
```

Which files should I change? Which files may I change?

It may be difficult figuring out what is necessary to change, what is safe to change, and what will cause trouble with grading. This section divides all the provided starter code files into categories based on how you should change them.

- One file you definitely *must* change, but *carefully*:

  src/model.cxx – in particular:

  Do fill in the sections marked `TODO` in the `Model(int, int)` constructor and `play_move` member function,

  Do define any additional private helper functions you like, but

  Don't modify any of the existing, complete function implementations.

- Five files you definitely *must* change, and may change however you like:

  src/view.{hxx,cxx}

  src/controller.{hxx,cxx}

  test/model_test.cxx

- Eight files you *must not* change:

  src/board.{hxx,cxx}

  src/move.{hxx,cxx}

  src/player.{hxx,cxx}

  src/position_set.{hxx,cxx}

- Three files you *may* change at your discretion, but *carefully*:

  src/model.hxx – in particular:

  Do add any private members (most likely additional helper functions) you want, but

  Don't alter the declarations of any public members, and

Don't alter the definitions of private member variables `turn_`, `winner_`, `board_`, and `next_moves_`.

src/reversi.cxx – unlikely, but:

Don't change how command-line arguments are handled, but

Do change anything else, as you like.

CMakeLists.txt – unlikely, but:

Do add any new model .cxx files you create to the variable `MODEL_SRC`, but

Don't change anything else.

- Four files you *may* change, but probably don't have reason to:

  test/board_test.cxx

  test/move_test.cxx

  test/player_test.cxx

  test/position_set_test.cxx

Deliverables & evaluation

For this homework you must:

1. Complete the two partially-implemented `Model` members (the two-argument constructor and function `play_move`) and all the private helper functions in src/model.cxx.
2. Complete the design and implementation of the `View` and `Controller` classes in src/{view,controller}.{hxx,cxx} and the unimplemented helper function in `View` (`add_player_sprite_`), so that the game is playable.
3. Add more test cases to test/model_test.cxx in order to the test that the model functions properly.

As usual, self evaluation will spot-check your test coverage by asking for just a few particular test cases. You can't anticipate what cases we may ask about, so you should try to cover everything.

Your grade will be based on:

- the correctness of your `Model` implementation with respect to the specification,
- the playability of your UI,
- the presence of sufficient test cases to ensure your model code's correctness, and
- adherence to the CS 211 Style Manual. (Do note that there are new rules for C++ that didn't apply in C.)

Submission

Homework submission and grading will use Gradescope. You must include any files that you create or change. For this homework, that will definitely include:

- src/model.cxx
- src/view.hxx
- src/view.cxx
- src/controller.hxx
- src/controller.cxx
- test/model_test.cxx

You must also include any other files you modified. If you add any source files then it will includes those, as well as your updated CMakeLists.txt. See Section "*Which files should I change? Which files may I change?*" for a comprehensive list of which files you must and may change.

Per the syllabus, if you engaged in arms-length collaboration on this assignment, you must cite your sources. You may write citations either in comments on the relevant code, or in a file named README.txt that you submit along with your code. See the syllabus for definitions and other details.

Submit your files by uploading them directly to the Gradescope website. When you click on the assignment in Gradescope for the first time, you will get a window where you can upload files. You can drag-and-drop files or browse to select them. Make sure you include all the necessary files in the src/ and test/ directories! Submitting extra files is fine. To submit additional times, select the "Resubmit" button on the bottom right.

Partners

If you work with a partner, then you MUST register your partnership on Gradescope. Only one of you should submit and register the other. From the Gradescope course page, click on the assignment, then at the bottom click the button labeled "Group Members". A dropdown menu will allow you to select the name of your partner, then click save to send them an email. Group members can also be removed by the same process. For more details see the help page on Gradescope.

Please remember to mark your partner on each submission.