10GE211

Contents

1 CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.3)
2  project(connect4 CXX)
3  include(.cs211/cmake/CMakeLists.txt)
4
5  # Builds program connect4 from the listed source files.
6  add_program(connect4
7          src/connect4.cxx
8          src/ui.cxx
9          src/model.cxx)
10 target_link_libraries(connect4 ge211)
11
12 # Builds test program connect4_test from the two listed source files.
13 # Neither source file contains a main(), because that is provided by
14 # the `add_test_program` command.
15 add_test_program(connect4_test
16         test/model_test.cxx
17         src/model.cxx)
```

2 src/connect4.cxx

```cpp
#include "ui.hxx"

int main()
{
    Connect4_ui().run();
}
```

3 src/model.hxx

```cpp
#pragma once

#include <vector>

// How we represent players or the absence thereof.
enum class Player
{
    first, second, neither
};

// Returns the other player, if given Player::first or Player::second;
// throws std::invalid_argument if given Player::neither.
Player other_player(Player);

// Models a Connect Four game.
struct Connect4_model
{
    ///
    /// A TYPE ABBREVIATION:
    ///

    // A type member used below - we represent a column as a vector of
    // `Player`s:
    using column_t = std::vector<Player>;


    ///
    /// CONSTRUCTOR
    ///

    // Constructs an empty Connect Four game model.
    Connect4_model();
```

```
33
34
35      ///
36      /// CONSTANTS
37      ///
38
39      // Game size parameters.
40      // (Defined in model.cxx.)
41      static const int k;  // how many to connect (4)
42      static const int m;  // grid width (7)
43      static const int n;  // grid height (6)
44
45
46      ///
47      /// API FUNCTIONS (PUBLIC MEMBER FUNCTIONS)
48      ///
49
50      // Places the token for the current player in the given column.
51      //
52      // **PRECONDITION**: `is_playable(col_no)` (throws)
53      void place_token(int col_no);
54
55
56      // Returns whose turn it is, or Player::neither for game over.
57      Player turn() const { return turn_; };
58
59      // Returns the winner if there is one; returns Player::neither for
60      // stalemates or when the game isn't over yet.
61      Player winner() const { return winner_; };
62
63      // Gets a read-only view of the given column.
64      //
65      // **PRECONDITION:** `is_good_col(col_no)` (throws)
66      const column_t& column(int col_no) const;
67
68
69      // Is the column number within bounds?
70      bool is_good_column(int col_no) const;
71
72      // Can we play in the given column? Returns true if the game is
73      // not over and the column is not full.
74      bool is_playable(int col_no) const;
75
76      // Is the game over?
77      bool is_game_over() const { return turn_ == Player::neither; }
```

```
78
79      ///
80      /// INTERNAL HELPER FUNCTIONS ("PRIVATE" MEMBER FUNCTIONS)
81      ///
82
83      // Checks for a winner, or advances the turn if there isn't one.
84      //
85      // **PRECONDITION** `col_no` where the most recent move was played
86      // (whch also means it's in bounds).
87      void update_winner_and_turn_(int col_no);
88
89      // A cheap position type (to avoid model depending on GE211).
90      struct Posn { int col, row; };
91
92      // Counts the number of instances of `turn_`, moving in direction
93      // {dcol, drow} starting at `start`, and *not* counting the Player
94      // at the starting position.
95      int count_from_by_(Posn start, int dcol, int drow) const;
96
97      // Returns whether there is a token at the given position.
98      bool is_occupied_(int col_no, int row_no) const;
99
100     // Checks that `col_no` is in bounds, throwing an exception if not.
101     void check_column_(int col_no) const;
102
103     // Checks that we can play in `col_no`, throwing an exception
104     // if we can't.
105     void check_playable_(int col_no) const;
106
107     ///
108     /// FIELDS (PRIVATE DATA MEMBERS)
109     ///
110
111     // The grid of tokens, by column. Each column vector is indexed from
112     // the bottom and only as long as the number of player tokens in it.
113     // (It ISN'T padded to the full board height with `Player::neither`s,
114     // so `board_` can be thought of as a "ragged" 2-D array.)
115     std::vector<column_t> board_;
116
117     // The current turn.
118     Player turn_   = Player::first;
119
120     // The winning player, if any.
121     Player winner_ = Player::neither;
122
```

```
123     // INVARIANT (game invalid if false):
124     //
125     //  - board_.size() == Model::m
126     //
127     //  - for (column_t c : board_) c.size() <= Model::n
128     //
129     //  - for (column_t c : board_) for (Player p : c) p != Player::neither
130     //
131     //  - turn_ == Player::neither || winner_ == Player::neither
132     //
133     //  - If `turn_ != Player::neither` then there is no line of length
134     //    `Model::k` on the board.
135     //
136     //  - For either `Player p`, `winner_ == p` if and only if there is a
137     //    length-`Model::k` line of `p` tokens on the board.
138 };
```

4 src/model.cxx

```
1  #include "model.hxx"
2
3  // For `std::logic_error` and `std::invalid_argument`:
4  #include <stdexcept>
5
6  Player other_player(Player p)
7  {
8      switch (p) {
9      case Player::first:
10         return Player::second;
11     case Player::second:
12         return Player::first;
13     default:
14         throw std::invalid_argument("other_player: not a player");
15     }
16 }
17
18 // Constructor for Connect4_model.
19 //
20 // The second line (`: board_(m)`) constructs the `board_` member
21 // variable. Since `board_` is declared like
22 //
23 //     std::vector<column_t> board_;
24 //
```

```
25  // the member initializer is as if we were constructing it like
26  //
27  //      std::vector<column_t> board_(m);
28  //
29  // which initializes `board_` to have `m` elements, each of which
30  // is a default-constructed (empty) `column_t`.
31  Connect4_model::Connect4_model()
32          : board_(m)
33  { }
34
35
36  const int Connect4_model::k = 4;  // how many to connect
37  const int Connect4_model::m = 7;  // grid width
38  const int Connect4_model::n = 6;  // grid height
39
40
41  void Connect4_model::place_token(int col_no)
42  {
43      check_playable_(col_no);
44      board_[col_no].push_back(turn_);
45      update_winner_and_turn_(col_no);
46  }
47
48  const Connect4_model::column_t& Connect4_model::column(int col_no) const
49  {
50      check_column_(col_no);
51      return board_[col_no];
52  }
53
54  bool Connect4_model::is_good_column(int col_no) const
55  {
56      return 0 <= col_no && col_no < m;
57  }
58
59  bool Connect4_model::is_playable(int col_no) const
60  {
61      return turn() != Player::neither &&
62             is_good_column(col_no) &&
63             column(col_no).size() < Connect4_model::n;
64  }
65
66
67  void Connect4_model::update_winner_and_turn_(int const col_no)
68  {
69      Posn move{col_no, (int)board_[col_no].size() - 1};
```

```
70
71      int below       = count_from_by_(move, 0, -1);
72      int left        = count_from_by_(move, -1, 0);
73      int right       = count_from_by_(move, 1, 0);
74      int above_left  = count_from_by_(move, -1, 1);
75      int above_right = count_from_by_(move, 1, 1);
76      int below_left  = count_from_by_(move, -1, -1);
77      int below_right = count_from_by_(move, 1, -1);
78
79      if (below + 1 >= k ||
80          left + 1 + right >= k ||
81          above_left + 1 + below_right >= k ||
82          above_right + 1 + below_left >= k) {
83          winner_ = turn_;
84      } else {
85          for (column_t const& column : board_) {
86              if (column.size() < n) {
87                  turn_ = other_player(turn_);
88                  return;
89              }
90          }
91      }
92
93      turn_ = Player::neither;
94  }
95
96  void Connect4_model::check_column_(int col_no) const
97  {
98      if (!is_good_column(col_no))
99          throw std::invalid_argument("Model::place_token: column out of bounds");
100 }
101
102 void Connect4_model::check_playable_(int col_no) const
103 {
104     check_column_(col_no);
105
106     if (is_game_over())
107         throw std::logic_error("Model::place_token: game over");
108
109     if (!is_playable(col_no))
110         throw std::invalid_argument("Model::place_token: column full");
111 }
112
113 int Connect4_model::count_from_by_(Posn start, int dcol, int drow) const
114 {
```

```
115     int count = 0;
116
117     for (;;) {
118         start.row += drow;
119         start.col += dcol;
120
121         if (!is_occupied_(start.col, start.row))
122             return count;
123
124         if (board_[start.col][start.row] != turn_)
125             return count;
126
127         ++count;
128     }
129 }
130
131 bool Connect4_model::is_occupied_(int col_no, int row_no) const
132 {
133     return is_good_column(col_no) &&
134            0 <= row_no && row_no < board_[col_no].size();
135 }
```

5 src/ui.hxx

```
1  #pragma once
2
3  #include "model.hxx"
4  #include <ge211.hxx>
5
6  // Sizes of grid and tokens:
7  int const grid_size                = 100,
8            token_radius             = grid_size / 2;
9
10 // Colors of rendered tokens.
11 ge211::Color const playing_bg   = ge211::Color::medium_yellow().lighten(0.7),
12                    stalemate_bg = ge211::Color::white(),
13                    first_color  = ge211::Color::medium_red(),
14                    second_color = ge211::Color::medium_blue(),
15                    first_tint   = first_color.lighten(0.5),
16                    second_tint  = second_color.lighten(0.5);
17
18 // Code for how we interact with the model.
19 struct Connect4_ui : ge211::Abstract_game
```

```
20  {
21      ///
22      /// MEMBER FUNCTIONS
23      ///
24      /// Each of these member functions *overrides* a default implementation
25      /// inherited from `ge211::Abstract_game`. For example, the default
26      /// implementation of `on_mouse_move` does nothing, but we change it
27      /// to keep track of the mouse position.
28      ///
29
30      // Displays the current state of the model by adding sprites to the given
31      // `Sprite_set`.
32      void draw(ge211::Sprite_set&) override;
33
34      // Called by the game engine when the mouse moves.
35      void on_mouse_move(ge211::Position) override;
36
37      // Called by the game engine when the mouse button is clicked.
38      void on_mouse_down(ge211::Mouse_button, ge211::Position) override;
39
40      // Returns the dimensions that the window should have (based on the grid
41      // dimensions).
42      ge211::Dimensions initial_window_dimensions() const override;
43
44      // Helper function for computing the physical dimensions of the board.
45      // `static` means it doesn't require a `Connect4_ui` object to call it,
46      // which means we can call it when constructing a `Connect4_ui` (and
47      // we will).
48      static ge211::Dimensions board_pixels();
49
50      // Returns the title to put on the window.
51      std::string initial_window_title() const override;
52
53      ///
54      /// FIELDS (PRIVATE DATA MEMBERS)
55      ///
56
57      // Holds the logical (presentation-independent) state of the game.
58      Connect4_model model_;
59
60      // Logical board column where the mouse was last seen.
61      int mouse_column_ = -1;
62
63      // The sprites, for displaying player tokens.
64      ge211::Circle_sprite const player1_token_{token_radius, first_color};
```

```
65      ge211::Circle_sprite const player2_token_{token_radius, second_color};
66      ge211::Circle_sprite const player1_shadow_{token_radius, first_tint};
67      ge211::Circle_sprite const player2_shadow_{token_radius, second_tint};
68  };
```

6 src/ui.cxx

```
1   #include "ui.hxx"
2
3   using namespace ge211;
4
5   // Converts a logical board position to the physical screen position
6   // of the upper-left corner of the corresponding grid square.
7   static Position board_to_screen_(Position board_pos)
8   {
9       int x = 2 * token_radius * board_pos.x;
10      int y = 2 * token_radius * (Connect4_model::n - board_pos.y - 1);
11      return {x, y};
12  }
13
14  // Converts a physical screen position to the logical board position
15  // that corresponds to it.
16  static Position screen_to_board_(Position screen_pos)
17  {
18      int col_no = screen_pos.x / (2 * token_radius);
19      int row_no = Connect4_model::n - screen_pos.y / (2 * token_radius) - 1;
20      return {col_no, row_no};
21  }
22
23  void Connect4_ui::draw(ge211::Sprite_set& sprites)
24  {
25      // Here we add a sprite for each token in the game:
26      for (int col_no = 0; col_no < Connect4_model::m; ++col_no) {
27          const Connect4_model::column_t& column = model_.column(col_no);
28          for (int row_no = 0; row_no < column.size(); ++row_no) {
29              Player player = column[row_no];
30              Position screen_pos = board_to_screen_({col_no, row_no});
31
32              if (player == Player::first)
33                  sprites.add_sprite(player1_token_, screen_pos);
34              else
35                  sprites.add_sprite(player2_token_, screen_pos);
36          }
```

```
37      }
38
39      // Here we possibly add a sprite as a cursor that follows the mouse:
40      if (model_.is_playable(mouse_column_)) {
41          int      col_no     = mouse_column_,
42                   row_no     = (int) model_.column(col_no).size();
43          Position screen_pos = board_to_screen_({col_no, row_no});
44
45          if (model_.turn() == Player::first)
46              sprites.add_sprite(player1_shadow_, screen_pos);
47          else
48              sprites.add_sprite(player2_shadow_, screen_pos);
49      }
50
51      // Select the background color for the window based on the
52      // winner or lack thereof:
53      if (model_.winner() == Player::first)
54          background_color = first_tint;
55      else if (model_.winner() == Player::second)
56          background_color = second_tint;
57      else if (model_.is_game_over())
58          background_color = stalemate_bg;
59      else
60          background_color = playing_bg;
61  }
62
63  void Connect4_ui::on_mouse_move(Position screen_pos)
64  {
65      mouse_column_ = screen_to_board_(screen_pos).x;
66  }
67
68  Dimensions Connect4_ui::initial_window_dimensions() const
69  {
70      return board_pixels();
71  }
72
73  ge211::Dimensions Connect4_ui::board_pixels()
74  {
75      return {2 * token_radius * Connect4_model::m,
76              2 * token_radius * Connect4_model::n};
77  }
78
79  std::string Connect4_ui::initial_window_title() const
80  {
81      return "Connect Four";
```

```
}

void Connect4_ui::on_mouse_down(Mouse_button btn, Position screen_posn)
{
    if (model_.turn() == Player::neither || btn != Mouse_button::left)
        return;

    int col_no = screen_to_board_(screen_posn).x;

    if (model_.is_playable(col_no))
        model_.place_token(col_no);
}
```

7 test/model_test.cxx

```
#include "model.hxx"
#include <catch.hxx>
#include <iostream>
#include <stdexcept>


/////
/////
///// TESTING HELPERS
/////
/////

static const Player RED = Player::first,
                    BLU = Player::second,
                    MT  = Player::neither;

using Model  = Connect4_model;
using Column = Model::column_t;

// Helper struct for checking (writing down, comparing, printing) the
// board state.
struct Board
{
    // Holds the whole board; padded to height `Model::n` with
    // `Player::neither`.
    std::vector<Column> columns;

    // Constructs an empty board.
```

```
29      Board()
30              : columns(Model::m, Column(Model::n, Player::neither))
31      { }
32
33      // Constructs a board by copying it from a model.
34      Board(const Model& model)
35              : Board()
36      {
37          for (int col_no = 0; col_no < Model::m; ++col_no)
38              col(col_no, model.column(col_no));
39      }
40
41      // Replaces the column at the given index with the given column.
42      Board& col(int col_no, const Column& col)
43      {
44          columns[col_no] = col;
45          columns[col_no].resize(Model::n, Player::neither);
46          return *this;
47      }
48  };
49
50  bool operator==(const Board& a, const Board& b)
51  {
52      return a.columns == b.columns;
53  }
54
55  // How we'll print players when printing a board.
56  static char char_of(Player player)
57  {
58      switch (player) {
59      case Player::first: return '1';
60      case Player::second: return '2';
61      case Player::neither: return '_';
62      }
63  }
64
65  // Prints a board.
66  std::ostream& operator<<(std::ostream& os, const Board& board)
67  {
68      os << "board:\n";
69      for (int row = Model::n - 1; row >= 0; --row) {
70          for (const Column& col : board.columns)
71              os << char_of(col[row]);
72          os << '\n';
73      }
```

```
74      return os;
75  }
76
77
78  /////
79  /////
80  ///// TEST CASES
81  /////
82  /////
83
84  TEST_CASE("Can construct model")
85  {
86      Model model;
87  }
88
89  TEST_CASE("Game parameters match classic Connect Four")
90  {
91      CHECK( Model::k == 4 );
92      CHECK( Model::m == 7 );
93      CHECK( Model::n == 6 );
94  }
95
96  TEST_CASE("New model in expected state")
97  {
98      Model c4;
99
100     CHECK(c4 == Board());
101     CHECK(c4.turn() == RED);
102     CHECK(c4.winner() == MT);
103     CHECK_FALSE(c4.is_game_over());
104 }
105
106 TEST_CASE("Player 1 can move")
107 {
108     Model c4;
109
110     c4.place_token(1);
111
112     CHECK(c4 == Board().col(1, {RED}));
113     CHECK(c4.turn() == BLU);
114     CHECK(c4.winner() == MT);
115 }
116
117 TEST_CASE("Player 2 can play next to player 1")
118 {
```

```
119     Model c4;
120 
121     c4.place_token(1);
122     c4.place_token(2);
123 
124     CHECK(c4 == Board().col(1, {RED})
125                        .col(2, {BLU}));
126 }
127 
128 
129 TEST_CASE("Player 2 can play atop to player 1")
130 {
131     Model c4;
132 
133     c4.place_token(1);
134     c4.place_token(1);
135 
136     CHECK(c4 == Board().col(1, {RED, BLU}));
137 }
138 
139 TEST_CASE("Can play a whole game.")
140 {
141     Model c4;
142 
143     CHECK(c4.turn() == RED );
144     CHECK(c4.winner() == MT );
145     c4.place_token(0);
146     CHECK(c4 == Board().col(0,{RED}));
147 
148     CHECK(c4.turn() == BLU );
149     c4.place_token(1);
150     CHECK(c4 == Board().col(0, {RED})
151                        .col(1, {BLU}));
152 
153     CHECK(c4.turn() == RED );
154     c4.place_token(1);
155     CHECK(c4 == Board().col(0, {RED})
156                        .col(1, {BLU, RED}));
157 
158     c4.place_token(2);
159     CHECK(c4 == Board().col(0, {RED})
160                        .col(1, {BLU, RED})
161                        .col(2, {BLU}));
162 
163     c4.place_token(3);
```

```
164     CHECK(c4 == Board().col(0, {RED})
165                        .col(1, {BLU, RED})
166                        .col(2, {BLU})
167                        .col(3, {RED}));
168 
169     c4.place_token(2);
170     CHECK(c4 == Board().col(0, {RED})
171                        .col(1, {BLU, RED})
172                        .col(2, {BLU, BLU})
173                        .col(3, {RED}));
174 
175     c4.place_token(2);
176     CHECK(c4 == Board().col(0, {RED})
177                        .col(1, {BLU, RED})
178                        .col(2, {BLU, BLU, RED})
179                        .col(3, {RED}));
180 
181     c4.place_token(3);
182     CHECK(c4 == Board().col(0, {RED})
183                        .col(1, {BLU, RED})
184                        .col(2, {BLU, BLU, RED})
185                        .col(3, {RED, BLU}));
186 
187     c4.place_token(2);
188     CHECK(c4 == Board().col(0, {RED})
189                        .col(1, {BLU, RED})
190                        .col(2, {BLU, BLU, RED, RED})
191                        .col(3, {RED, BLU}));
192 
193     c4.place_token(3);
194     CHECK(c4 == Board().col(0, {RED})
195                        .col(1, {BLU, RED})
196                        .col(2, {BLU, BLU, RED, RED})
197                        .col(3, {RED, BLU, BLU}));
198 
199     c4.place_token(3);
200     CHECK(c4 == Board().col(0, {RED})
201                        .col(1, {BLU, RED})
202                        .col(2, {BLU, BLU, RED, RED})
203                        .col(3, {RED, BLU, BLU, RED}));
204 
205 
206     CHECK(c4.winner() == RED );
207     CHECK(c4.turn() == MT );
208 
```

```
209     CHECK_THROWS_AS(c4.place_token(0), std::logic_error);
210 }
211
212 TEST_CASE("full column throws")
213 {
214     Model c4;
215
216     for (int i = 0; i < Model::n; ++i)
217         c4.place_token(2);
218
219     CHECK_THROWS_AS(c4.place_token(2), std::invalid_argument);
220
221     c4.place_token(0);
222
223     CHECK(c4 == Board().col(0, {RED})
224                       .col(2, {RED, BLU, RED, BLU, RED, BLU}));
225
226     CHECK_THROWS_AS(c4.place_token(2), std::invalid_argument);
227 }
```