```
#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();


    ///
    /// CONSTANTS
    ///

    // The game parameters.
    static const int k;  // how many to connect (4)
    static const int m;  // grid width (7)
    static const int n;  // grid height (6)


    ///
    /// API FUNCTIONS (PUBLIC MEMBER FUNCTIONS)
    ///

    /// Gets a view of the given column.
    ///
    /// **PRECONDITION:** `0 <= col_no < m()`
    const column_t* get_column(int col_no) const;

    /// Places the token for the given player in the given column, provided
    /// that column is not full.
    ///
    /// **PRECONDITIONS**:
    ///  - `0 <= col_no < m()`
    ///  - `get_column(col_no).size() < n()`
    void place_token(int col_no);

    /// Is the column number within bounds?
    bool is_good_col(int col_no) const;

    /// Is there room to play in the given column?
```

```
    bool is_playable_col(int col_no) const;

    /// Returns whose turn it is, or Player::neither for game over.
    Player get_turn() const { return turn_; };

    /// Returns the winner, or Player::neither for stalemates or when
    /// the game is not yet over.
    Player get_winner() const { return winner_; };

    ///
    /// INTERNAL FUNCTIONS (PRIVATE MEMBER FUNCTIONS)
    ///

    /// Gets a mutable view of the given column.
    ///
    /// **PRECONDITION:** `col_no < m()`
    column_t* get_column_(int col_no);

    /// **PRECONDITION** `col_no` is valid and where the last move
    /// was played.
    void update_winner_and_turn_(int col_no);

    /// Counts the number of instances of `goal`, moving in direction
    /// {change_col, change_row} starting at {start_col, start_row}, and
    /// not counting the Player at the starting position.
    int count_from_by_(Player goal,
                       int start_col, int start_row,
                       int change_col, int change_row) const;

    /// Returns whether there is a token at the given position.
    bool is_good_col_row_(int col_no, int row_no) const;

    ///
    /// FIELDS (PRIVATE DATA MEMBERS)
    ///

    // The current turn.
    Player turn_   = Player::first;

    // The winning player, if any.
    Player winner_ = Player::neither;

    // The grid itself.
    std::vector<column_t> columns_;
    // INVARIANT:
    //  - columns_.size() == m
    //  - for (column_t const& column : columns) column.size() <= n
    //  - none of the Player values is Player::neither
};
```

```
#include "model.h"
#include <catch.h>
#include <stdexcept>

TEST_CASE("can create default Connect Four")
{
    Connect4_model model;
}

TEST_CASE("game parameters")
{
    CHECK( Connect4_model::k == 4 );
    CHECK( Connect4_model::m == 7 );
    CHECK( Connect4_model::n == 6 );
}

TEST_CASE("can move")
{
    const Player red = Player::first,
                 blu = Player::second,
                 mt  = Player::neither;
    using Column = Connect4_model::column_t;

    Connect4_model c4;

    CHECK( c4.get_turn() == red );
    CHECK( c4.get_winner() == mt );
    c4.place_token(0);
    CHECK( *c4.get_column(0) == Column{red} );

    CHECK( c4.get_turn() == blu );
    c4.place_token(1);
    CHECK( *c4.get_column(1) == Column{blu} );

    CHECK( c4.get_turn() == red );
    c4.place_token(1);
    CHECK( *c4.get_column(1) == Column{blu, red} );

    CHECK( c4.get_turn() == blu );
    c4.place_token(2);
    CHECK( *c4.get_column(2) == Column{blu} );

    CHECK( c4.get_turn() == red );
    c4.place_token(3);
    CHECK( *c4.get_column(3) == Column{red} );

    CHECK( c4.get_turn() == blu );
    c4.place_token(2);
    CHECK( *c4.get_column(2) == Column{blu, blu} );

    CHECK( c4.get_turn() == red );
    c4.place_token(2);
    CHECK( *c4.get_column(2) == Column{blu, blu, red} );

    CHECK( c4.get_turn() == blu );
    c4.place_token(3);
    CHECK( *c4.get_column(3) == Column{red, blu} );

    CHECK( c4.get_turn() == red );
    c4.place_token(2);
    CHECK( *c4.get_column(2) == Column{blu, blu, red, red} );

    CHECK( c4.get_turn() == blu );
    c4.place_token(3);
    CHECK( *c4.get_column(3) == Column{red, blu, blu} );
```

```
    CHECK( c4.get_turn() == red );
    c4.place_token(3);
    CHECK( *c4.get_column(3) == Column{red, blu, blu, red} );

    CHECK( c4.get_turn() == mt );
    CHECK( c4.get_winner() == red );

    CHECK_THROWS_AS(c4.place_token(0), std::logic_error);
}

TEST_CASE("full column throws")
{
    const Player red = Player::first,
            blu = Player::second;
    using Column = Connect4_model::column_t;

    Connect4_model c4;

    for (int i = 0; i < c4.n; ++i)
        c4.place_token(2);

    CHECK_THROWS_AS(c4.place_token(2), std::invalid_argument);

    c4.place_token(0);

    CHECK( *c4.get_column(0) == Column{red} );
    CHECK( *c4.get_column(2) == Column{red, blu, red, blu, red, blu} );

    CHECK_THROWS_AS(c4.place_token(2), std::invalid_argument);
}
```

```cpp
#pragma once

#include "model.h"
#include <ge211.h>

// The radius of each rendered token.
int const token_radius = 50;

// Colors of rendered tokens.
ge211::Color const first_color       = ge211::Color::medium_red();
ge211::Color const second_color      = ge211::Color::medium_blue();
ge211::Color const first_move_color  = first_color.lighten(0.3);
ge211::Color const second_move_color = second_color.lighten(0.3);

// Code for how we interact with the model.
struct Connect4_ui : ge211::Abstract_game
{
    ///
    /// MEMBER FUNCTIONS
    ///
    /// Each of these member functions *overrides* a default implementation
    /// inherited from `ge211::Abstract_game`. For example, the default
    /// implementation of `on_mouse_move` does nothing, but we change it
    /// to keep track of the mouse position.
    ///

    // Displays the current state of the model by adding sprites to the given
    // `Sprite_set`.
    void draw(ge211::Sprite_set&) override;

    // Called by the game engine when the mouse moves.
    void on_mouse_move(ge211::Position) override;

    // Called by the game engine when the mouse button is clicked.
    void on_mouse_down(ge211::Mouse_button, ge211::Position) override;

    // Returns the dimensions that the window should have (based on the grid
    // dimensions).
    ge211::Dimensions initial_window_dimensions() const override;

    // Returns the title to put on the window.
    std::string initial_window_title() const override;

    ///
    /// FIELDS (PRIVATE DATA MEMBERS)
    ///

    // Holds the logical (presentation-independent) state of the game.
    Connect4_model model_;

    // Logical board column where the mouse was last seen.
    int mouse_column_ = -1;

    /// The sprites, for displaying player tokens.
    ge211::Circle_sprite const player1_token_{token_radius, first_color};
    ge211::Circle_sprite const player2_token_{token_radius, second_color};
    ge211::Circle_sprite const move1_token_{token_radius, first_move_color};
    ge211::Circle_sprite const move2_token_{token_radius, second_move_color};
};
```

```cpp
#include "ui.h"

using namespace ge211;

using column_t = Connect4_model::column_t;

static Position board_to_screen_(Position board_pos)
{
    int x = 2 * token_radius * board_pos.x;
    int y = 2 * token_radius * (Connect4_model::n - board_pos.y - 1);
    return {x, y};
}

static Position screen_to_board_(Position screen_pos)
{
    int col_no = screen_pos.x / (2 * token_radius);
    int row_no = Connect4_model::n - screen_pos.y / (2 * token_radius) - 1;
    return {col_no, row_no};
}

void Connect4_ui::draw(ge211::Sprite_set& sprites)
{
    // Here we add a sprite for each token in the game:
    for (int col_no = 0; col_no < Connect4_model::m; ++col_no) {
        const column_t* column = model_.get_column(col_no);
        for (int row_no = 0; row_no < column->size(); ++row_no) {
            Player player = (*column)[row_no];
            Position screen_pos = board_to_screen_({col_no, row_no});
            if (player == Player::first)
                sprites.add_sprite(player1_token_, screen_pos);
            else
                sprites.add_sprite(player2_token_, screen_pos);
        }
    }

    // Here we possibly add a sprite as a cursor that follows the mouse:
    if (model_.is_good_col(mouse_column_)) {
        int col_no              = mouse_column_;
        const column_t* column = model_.get_column(col_no);
        int row_no              = int(column->size());
        Player player           = model_.get_turn();
        if (row_no < Connect4_model::n && player != Player::neither) {
            Position screen_pos = board_to_screen_({col_no, row_no});
            if (player == Player::first)
                sprites.add_sprite(move1_token_, screen_pos);
            else
                sprites.add_sprite(move2_token_, screen_pos);
        }
    }
}

void Connect4_ui::on_mouse_move(Position screen_pos)
{
    mouse_column_ = screen_to_board_(screen_pos).x;
}

Dimensions Connect4_ui::initial_window_dimensions() const
{
    return {2 * token_radius * Connect4_model::m,
            2 * token_radius * Connect4_model::n};
}

std::string Connect4_ui::initial_window_title() const
{
    return "Connect Four";
```

```
}

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

    int col_no = screen_to_board_(screen_posn).x;

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

```cpp
#include "ui.h"

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

```cpp
#include "model.h"
#include <cassert>
#include <stdexcept>

Player other_player(Player p)
{
    switch (p) {
        case Player::first: return Player::second;
        case Player::second: return Player::first;
        default:
            throw std::invalid_argument("other_player: not a player");
    }
}

Connect4_model::Connect4_model()
        : columns_(m, column_t{})
{ }

const int Connect4_model::k = 4;  // how many to connect
const int Connect4_model::m = 7;  // grid width
const int Connect4_model::n = 6;  // grid height

void Connect4_model::place_token(int col_no)
{
    if (turn_ == Player::neither)
        throw std::logic_error("Model::place_token: game over");

    column_t* col = get_column_(col_no);
    if (col->size() == n)
        throw std::invalid_argument("Model::place_token: column full");

    col->push_back(turn_);
    update_winner_and_turn_(col_no);
}

void Connect4_model::update_winner_and_turn_(int const col_no)
{
    int const row_no = (int)columns_[col_no].size() - 1;

    auto const count = [&](int dcol, int drow) {
        return count_from_by_(turn_, col_no, row_no, dcol, drow);
    };

    int below       = count( 0, -1);
    int left        = count(-1,  0);
    int right       = count( 1,  0);
    int above_left  = count(-1,  1);
    int above_right = count( 1,  1);
    int below_left  = count(-1, -1);
    int below_right = count( 1, -1);

    if (below + 1 >= k ||
            left + 1 + right >= k ||
            above_left + 1 + below_right >= k ||
            above_right + 1 + below_left >= k) {
        winner_ = turn_;
    } else {
        for (column_t const& column : columns_) {
            if (column.size() < n) {
                turn_ = other_player(turn_);
                return;
            }
        }
    }
```

```
    turn_ = Player::neither;
}

const Connect4_model::column_t* Connect4_model::get_column(int col_no) const
{
    assert(is_good_col(col_no));
    return &columns_[col_no];
}

Connect4_model::column_t* Connect4_model::get_column_(int col_no)
{
    assert(is_good_col(col_no));
    return &columns_[col_no];
}

int Connect4_model::count_from_by_(Player goal,
                           int start_col, int start_row,
                           int change_col, int change_row) const
{
    int count = 0;

    for (;;) {
        start_col += change_col;
        start_row += change_row;

        if (!is_good_col_row_(start_col, start_row))
            return count;

        if (columns_[start_col][start_row] != goal)
            return count;

        ++count;
    }
}

bool Connect4_model::is_good_col(int col_no) const
{
    return 0 <= col_no && col_no < m;
}

bool Connect4_model::is_playable_col(int col_no) const
{
    return is_good_col(col_no) &&
           get_column(col_no)->size() < Connect4_model::n;
}

bool Connect4_model::is_good_col_row_(int col_no, int row_no) const
{
    return is_good_col(col_no) &&
           0 <= row_no && row_no < columns_[col_no].size();
}
```