ThinkChess++

Build a chess app with C++ and learn to play along the way.

Oliver Krischer

March 18, 2024

If you have ever wondered how to program a simple chess app for yourself, this tutorial is the right starting point. We are going to explore the machanics of the game, how to maintain and display the game state, and how to find a good next move.

To that goal we we'll also dive into data structures (*trees, graphs*), and explore some basic search algorithms based on these data structures.

After working through this tutorial, you will not only have a running chess app, but also will be proficient in playing chess at an amateur level.

The souce code of the program and the documentation is available at github.

Contents

1 Display the Game

I've chosen the *Simple and Fast Multimedia Library* SFML for creating the graphical user interface of the app. It lives up to its name and is available on all major platforms and programming languages.

In this chapter we'll learn how to display the board and the chess pieces on it. In order to show valid moves for each piece, we will also learn the basic rules of the game.

I'm using CMake for building the C^{++} code, and the SFML CMake project template, which will build the SFML libraries. So you will need to have the following components installed on your machine:

- a decent C^{++} compiler (any of the major compilers will do)

- the *git* tool

- the *cmake* tool

- the required system packages for SFML

On a linux system, all those components can be installed with your systems package manager (e.g. with `apt-get` on Ubuntu). For Mac OS, just use the included `clang++` compiler and install missing components with homebrew: `<brew install git cmake sfml>`.

When everything is in place, just clone my repository with
`<git clone https://github.com/okrischer/ThinkChess.git>`
and execute `<cmake -B build>` from the root folder of your local copy.
If everything went well, change to the `build` folder and execute `<cmake --build .>`.
Voila, you have a simple chess app, which you can start with `<./ThinkChess>`.

I strongly encourage you to code all the following steps with an editor of your choice for yourself and see if you can get the code running. Nothing is gained if you just skim over the provided source code.

If you have LaTeX installed on your machine, you can also build the documentation with `<pdflatex -shell-escape ThinkChess++.tex>` from the `doc` folder, which will produce this document.

1.1 Displaying the board and the pieces

Let's start with the basic framework for displaying something with SFML:

```cpp
#include <SFML/Graphics.hpp>

int main() {
  sf::ContextSettings settings;
  settings.antialiasingLevel = 8;
  auto window = sf::RenderWindow{ {640u, 640u}, "ThinkChess++",
                sf::Style::Default, settings };
  window.setFramerateLimit(10);

  while (window.isOpen()) {
    for (auto event = sf::Event{}; window.pollEvent(event);) {
      if (event.type == sf::Event::Closed) {
        window.close();
      }
    }
    window.draw(bs);
    window.display();
  }
}
```

This is the main file for our app (`app/main.cpp`). Thus, it defines the `int main()` function (3) as the starting point of the app. The numbers in paranthesis `(x)` always refer to the last code snippet.

In order to access SFML functionality, we have to `#include` the SFML Graphics library (1), which was built by *cmake*.

First, we define the context (4) and set the antialiasing level to 8 within the context (5). Next, we define the main window for our app (6), setting its size, title, style, and the context settings.

Then we set the framerate to 10 (8), i.e. 10 frames per second; we don't need more for such a static app, and the app will keep responsive with that.

Next comes the central part: entering the main game loop (10-18). The game loop usually contains three steps:

1. an event loop, processing all user inputs for the current frame (11-15)

2. several drawing instructions (16) for all items to appear in the frame

3. a call to `window.display()`, which causes all drawn elements to be actually displayed.

1.1.1 The board

We don't have anything to draw yet; let's change this by adding a chessboard within the main function, just before entering the game loop:

```
1    sf::Texture bi;
2    bi.loadFromFile("../img/chessboard.jpg");
3    sf::Sprite bs;
4    bs.setTexture(bi);
```

Here, we create a SFML texture (1) and load an image from the file system into that texture (2). Then we create a SFML sprite (3) and set its texture to that image (4).

If you run the app now, you would see an empty chess board:
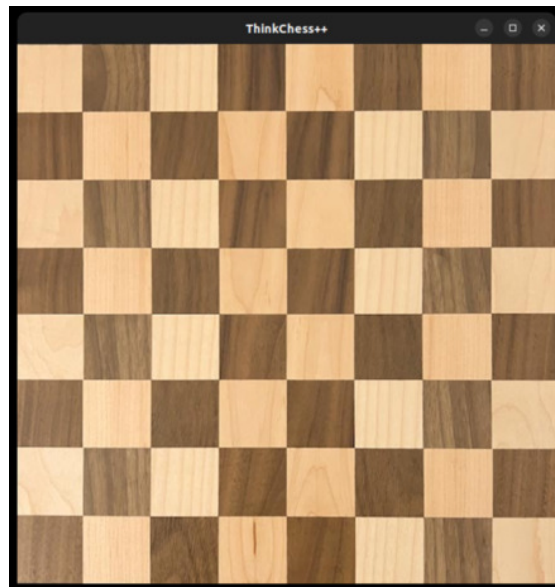
1.1.2 The pieces

Not that interesting so far, so we're going to add the chess pieces at their initial position. For that, let's introduce the pieces at first: they come in two colors, *white* and *black*, each for one player, and they are called as follows (from left to right in the image below):

Piece	King	Queen	Bishop	Knight	Rook	Pawn
quantity	1	1	2	2	2	8
value	-	8	3	3	4	1

Every piece, except the king, has a value assigned to it; this is just by convention and not necessary for the original game play. The value indicates the strength of the piece and serves for a basic evaluation of each players position during the game.

The *Queen* is the most powerful piece in the game, followed by the *Rook*; those two are also called *major pieces*. Then we have the *Bishop* and the *Knight*, both of equal value, building the group of *minor pieces*. The least worthy piece is the *Pawn*.

But why has the *King* no value assigned?
The goal of the game is to put your opponents *King* in a position, where it is attacked by any of your own pieces (an attack is the threat to capture a piece with the next move). We call this position *check*.
If a player cannot respond to a *check* in the very next move (i.e. defend his *King*), the game is over, and that player has lost the game. We call this position *checkmate*.
Thus, the *King* is never actually captured, it stays on the board until the end of the game. With that, it doesn't make any sense to give it a value for evaluating a players position.

We'll cover the movement of the pieces in great detail in the next section 1.2.2. But, for now, let's concentrate on how to display the pieces:

```cpp
1    sf::Texture figures;
2    figures.loadFromFile("../img/figures.png");
3
4    sf::Sprite wk;
5    wk.setTexture(figures);
6    wk.setTextureRect(sf::IntRect(0,0,60,60));
7
8    sf::Sprite bk;
9    bk.setTexture(figures);
10   bk.setTextureRect(sf::IntRect(0,60,60,60));
11   // --- snip ---
```

We have loaded a texture, containing the figures of all pieces (2). Then we define distinct sprites for every piece, calling them *wk* for the white king (4), *bk* for the black king (8), and so forth.
Finally, we cut out the matching parts of the `figures` texture and assign them to every distinct sprite (6, 10).

Notice, that I've taken care that all the sprites have the same size of 60×60 pixels. As the board has a size of 640×640 pixels, containing 8×8 light and dark squares, every square on the board has a size of 80×80 pixels. That allows us to place the sprites evenly on the board with an offeset of 10 pixels to the boundary of each square.

1.1.3 Keeping track of the pieces

In order to actually place the pieces on the board, we need a data structure, being able to keep track of the 64 squares. As it turns out, an 8×8 vector matrix is a good fit for that:

```
vector<vector<Piece*>> board(8, vector<Piece*>(8));
```

You could have used any type of vector matrix for keeping track of the pieces, but I've chosen an *object-oriented* approach: every piece is represented by a raw pointer (indicated with the *) to an instance of the type `Piece`. That allows for greater flexibility implementing the game logic: instead of putting the logic for the pieces in the main file, we are going to implement it in a new file `app/pieces.cpp`.

So, let's have a look on how `Piece` is actually implemented:

```cpp
#pragma once
#include <vector>

// public interface for pieces
class Piece {
public:
  virtual ~Piece() {}
  virtual char getType() = 0;
  virtual char getValue() = 0;
  virtual bool isWhite() = 0;
  virtual bool isCaptured() = 0;
  virtual int getRow() = 0;
  virtual int getCol() = 0;
  virtual void capture() = 0;
  virtual bool isValid(std::vector<std::vector<Piece*>>& bd,
                       int r, int c) = 0;
};
```

8

As you may have guessed, this code snippet is not from the `pieces.cpp` file, but from its *header* file, located at `include/pieces.hpp`. I've decided to separate the compilation units into a public interface (the header file) and an implementation file. That allows to use the definitions of the header file in any other implementation file and gives us even more flexibility in structuring the source code.

`Piece` is declared as a pure *abstract* interface. That means, it doesn't have a *constructor*, and all the member functions are marked as `virtual`. Thus, you cannot instanciate it directly, and to make any use of it, we have to create *derived* classes (subclasses of `Piece`) for every piece, which will implement the virtual functions. The reason for doing so is: we need different implementations for performing moves for each piece, but also a common type for storing them in the board matrix.

The concrete types for `Piece` are implemented like so:

```cpp
class King : public Piece {
public:
  ~King() {}
  King(bool w, int r, int c) : type{'K'}, white{w}, value{0},
                               captured{false}, row{r}, col{c} {}

  char getType() override { return type; }
  char getValue() override { return value; }
  bool isWhite() override { return white; }
  bool isCaptured() override { return captured; }
  int getRow() override { return row; }
  int getCol() override { return col; }
  bool isValid(std::vector<std::vector<Piece*>>& bd, int r, int c)
    override;

private:
  char type;
  bool white;
  short value;
  bool captured;
  int row;
  int col;
};
```

The other pieces are implemented likewise; the only difference between them so far is the implementation of the member function `isValid` (13). I decided to leave all the class definitions in the header file, and moved only the implementation for `isValid` to the file `app/pieces.cpp`. We'll study these implementations in the next section 1.2.2.

For now, we are only interested in the type definitions:
each of them is derived from `class` Piece (1) and has a *destructor* (3), which will be called by the compiler when an instance of the class is destroyed. This will happen automatically, whenever an *automatic* variable gets out of scope. But, if a reference to an instance was created explicitly (using the keyword `new`), we have to `delete` that object explicitly in order to avoid *memory leaks*.

Next, we have the *constructor* (4-5):
it initializes a new instance with its type (which we need only for drawing the correct sprite), the color of the piece (white or black), its value, and the current position of the piece (the row and column of the piece in the board matrix).

The following member functions (7-12) are just *getters* to retrieve the `private` member types.

With the derived `Piece` types in place, we can initially fill the board matrix:

```cpp
void reset_board(vector<vector<Piece*>>& bd) {
  for (auto rank : bd) {
    for (auto piece : rank) {
      delete piece;
    }
  }
  // rank 8 (black)
  bd[0][0] = new Rook(0,0,0);
  bd[0][1] = new Knight(0,0,1);
  bd[0][2] = new Bishop(0,0,2);
  bd[0][3] = new Queen(0,0,3);
  bd[0][4] = new King(0,0,4);
  bd[0][5] = new Bishop(0,0,5);
  bd[0][6] = new Knight(0,0,6);
  bd[0][7] = new Rook(0,0,7);
  // rank 7 (black)
  bd[1][0] = new Pawn(0,1,0);
  bd[1][1] = new Pawn(0,1,1);
  bd[1][2] = new Pawn(0,1,2);
  bd[1][3] = new Pawn(0,1,3);
  bd[1][4] = new Pawn(0,1,4);
  bd[1][5] = new Pawn(0,1,5);
  bd[1][6] = new Pawn(0,1,6);
  bd[1][7] = new Pawn(0,1,7);
  // rank 2 (white)
  bd[6][0] = new Pawn(1,6,0);
  bd[6][1] = new Pawn(1,6,1);
  bd[6][2] = new Pawn(1,6,2);
```

```
    bd[6][3] = new Pawn(1,6,3);
    bd[6][4] = new Pawn(1,6,4);
    bd[6][5] = new Pawn(1,6,5);
    bd[6][6] = new Pawn(1,6,6);
    bd[6][7] = new Pawn(1,6,7);
    // rank 1 (white)
    bd[7][0] = new Rook(1,7,0);
    bd[7][1] = new Knight(1,7,1);
    bd[7][2] = new Bishop(1,7,2);
    bd[7][3] = new Queen(1,7,3);
    bd[7][4] = new King(1,7,4);
    bd[7][5] = new Bishop(1,7,5);
    bd[7][6] = new Knight(1,7,6);
    bd[7][7] = new Rook(1,7,7);
}
```

That seems tedious, but thankfully we have to do this only once. I've placed the initialization code within a function `void reset_board()`, just in case we want to reset the board later (e.g. when starting a new game). The board is passed as an automatic reference to the function (indicated by the & after the parameter type), such that we are able to modify the board directly, instead of working with a copy of the board.

The code in the first three lines actually resets the board by deleting all existing references to pieces. When setting the pieces, you have to be careful: every piece must be initialized with exactly the same coordinates from the board (and of course with the correct color: 0 for black and 1 for white).

The rows of a chessboard are called *ranks*, while the colums are called *files*. The ranks are indicated by the numbers 1 to 8, whereas the files are indicated by the letters `a` to `h`, both starting at the lower left square (the dark field `a1`):

```
8 . . . . . . . .
7 . . . . . . . .
6 . . . . . . . .
5 . . . . . . . .
4 . . . . . . . .
3 . . . . . . . .
2 . . . . . . . .
1 . . . . . . . .
  a b c d e f g h
```

With the initialization code above, the pieces will be placed like so on the board:

```
8 R N B Q K B N R
7 P P P P P P P P
6 . . . . . . . .
5 . . . . . . . .
4 . . . . . . . .
3 . . . . . . . .
2 P P P P P P P P
1 R N B Q K B N R
  a b c d e f g h
```

Observe that we abbreviate the *Knight* with the letter `N` to avoid confusion with the *King*. So, the white *Queen* is placed at `d1`, the black *Queen* at `d8`, and so forth.

1.1.4 Drawing the pieces

Now we can use the filled board matrix to actually draw the pieces within the game loop of the main function:

```cpp
// draw board
window.draw(bs);
// draw pieces
for (int row = 0; row < 8; row++) {
  for (int col = 0; col < 8; col++) {
    if (board[row][col]) {
      auto piece = board[row][col];
      sf::Sprite pc;
```

```
 9          switch (piece->getType()) {
10          case 'K':
11            piece->isWhite() ? pc = wk : pc = bk;
12            break;
13          case 'Q':
14            piece->isWhite() ? pc = wq : pc = bq;
15            break;
16          case 'R':
17            piece->isWhite() ? pc = wr : pc = br;
18            break;
19          case 'B':
20            piece->isWhite() ? pc = wb : pc = bb;
21            break;
22          case 'N':
23            piece->isWhite() ? pc = wn : pc = bn;
24            break;
25          case 'P':
26            piece->isWhite() ? pc = wp : pc = bp;
27            break;
28          }
29          pc.setPosition(col*80.f + 10.f, row*80.f + 10.f);
30          window.draw(pc);
31        }
32      }
33    }
```

We are iterating over all elements of the board matrix and get the piece at this position (7). Then we get the type and color of that piece by calling the appropriate getter functions (getType(), isWhite()) on it, and let a newly created sprite pc point to the corresponding figure sprite (9-28). Finally, we set the correct postion of the pc sprite on the board (29) and draw it to the current frame buffer (30). Observe that we have to use the pointer notation -> (instead of a dot) for calling those functions on a piece, as the pieces were actually defined as pointers.

And that's it: when you start the app now, you will see this screen, which shows a correct initialized chessboard with all pieces:

1.2 Showing valid moves

The goal for this section is to show the valid moves for any given piece on the board.

1.2.1 Game mechanics

For reaching our goal, we first need a way to process user input. With SFML this is done inside the event loop of the main function:

```
1   // in event loop
2   // mouse button pressed
3   if (event.type == sf::Event::MouseButtonPressed) {
4     if (event.mouseButton.button == sf::Mouse::Right) {
5       pair<int, int> f =
6         getField(event.mouseButton.x, event.mouseButton.y);
7       setValidMoves(board, board[f.first][f.second]);
8     }
9   }
10  // mouse button released
11  if (event.type == sf::Event::MouseButtonReleased) {
12    if (event.mouseButton.button == sf::Mouse::Right) {
13      validMoves =
14        vector<vector<short>>(8, vector<short>(8, 0));
```

```
15          }
16      }
```

Here, we check if a mouse button is pressed (3). If so, we check wether it's the right
mouse button (4). Then, we get the coordinates of the corresponding field on the board
(5-6), and fill another vector matrix with the computed valid moves for the piece under
the mouse cursor (7).

When the right mouse button is released (11-12), we reset the vector matrix of valid
moves (13-14).

In order for that to work, we need the following definitions in the main file, just before
entering the main function:

```cpp
1   vector<vector<short>> validMoves(8, vector<short>(8, 0));
2
3   void setValidMoves(vector<vector<Piece*>>& bd, Piece* pc) {
4     if (!pc) return;
5     for (int row = 0; row < 8; row++) {
6       for (int col = 0; col < 8; col++) {
7         if (pc->isValid(bd, row, col)) {
8           auto current = bd[row][col];
9           if (!current) validMoves[row][col] = 1;
10          else if (pc->isWhite() != current->isWhite())
11            validMoves[row][col] = 2;
12        }
13      }
14    }
15  }
16
17  pair<int, int> getField(int x, int y) {
18    int fx = x / 80;
19    int fy = y / 80;
20    auto field = make_pair(fy, fx);
21    return field;
22  }
23
```

The work is done inside the `setValidMoves` function (3):
if there's no piece at the given coordinates, do nothing (4). Otherwise, iterate over all
fields of the board (5-6), and check wether this position can be reached by the piece
under the mouse cursor (7). If so, get the piece of the current search position (8). If
there is no piece at this position, set this position to valid (9). Otherwise, check the

color of the current piece and if it's different from the piece under the cursor (i.e. it can be captured), set it to valid with the special marker 2 (11).

Our *object-oriented* design is starting to pay off, as we don't need to define any game logic inside the main application file!

The last thing to do, is to draw the content of the `validMoves` vector inside the main game loop (directly after drawing the pieces):

```
// before game loop
sf::CircleShape valid(20.f);

  // inside game loop
  // draw valid moves
  for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 8; col++) {
      if (validMoves[row][col] > 0) {
        valid.setPosition(col*80.f + 20.f, row*80.f + 20.f);
        if (validMoves[row][col] > 1)
          valid.setFillColor(sf::Color(0, 200, 0, 200));
        else valid.setFillColor(sf::Color(100, 200, 0, 100));
        window.draw(valid);
      }
    }
  }
```

We iterate over all fields of the board (6-7), and if `validMoves` contains an entry at this position (8), we set the marker `valid` on the board (9). As an extra feature, we set the marker to a brighter color, if the piece at this postion can be captured (10-11). Finally, we draw the marker on the board (13).

With that, all valid moves for a selected piece are displayed while pressing the right mouse button (with a green circular shape on the board). As soon as you release the mouse botton, the `validMoves` vector is reset, and there's nothing more to draw for the next frame.

That's all for the mechanics of the game for now. Of course, we still need to implement the `isValid` function for every piece, which leads us to the next section.

1.2.2 Movement of the pieces

The King

Let's start with the *King*: it can move one step in any direction (on ranks, files and diagonals), provided the target field is not occopied by a piece of the same color:
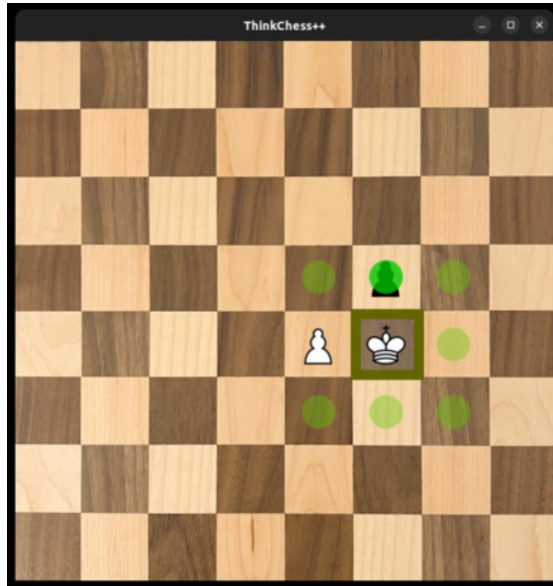
Observe, that the piece, for which the valid moves are shown (the king on f4), is marked with a dark green frame. We will learn how to set this marker in section 2.1, so when progamming along, you will not see this marker for now.

If the target field is occopied by a piece of the other color (the black pawn on f5), the king can capture that piece, indicated by a brighter color of the green marker.

But notice: the king is the only piece on the board that is not allowed to move to a field, where it is immediately attacked, as it would put itself into a *check* position, and the game would be over. We're not taking this special case into account yet, so the field g4, which is attacked by the black pawn, is still marked as valid.

Besides that, there's also a special move involving the king, called *castling*, which we will cover in section 2.3.

The function isValid for the king is implemented in the file app/pieces.cpp:

```cpp
#include "pieces.hpp"
#include <vector>
using namespace std;

bool King::isValid(const vector<vector<Piece*>>& bd, int r, int c) {
  bool valid = abs(r-row) <= 1 && abs(c-col) <= 1;
  return valid;
}
```

First, we have to include the header file located at include/pieces.hpp (1), in order to make the type definitions of the pieces available. Observe, that we don't have to spell

17

out the complete path to that file, thanks to these instructions of our `CMakeLists.txt` file:

```
add_executable(ThinkChess app/main.cpp app/pieces.cpp)
target_include_directories(ThinkChess PUBLIC include)
```

The function `isValid` is actually defined as `King::isValid` (5), which defines the function as a member function of the type `King`. It returns `true` only if the given coordinates can be reached within one step.

The Knight

The *Knight* is also a very special piece in one sense: it is the only piece, able to jump over any other piece on the board. It can move like so: either two fields on a rank and one on a file, or two on a file and one on a rank. That leads to an L shape for every move:

Observe that a knight, placed on a dark field, can reach only light fields, and vice versa.

Its `isValid` function is implemented like so:
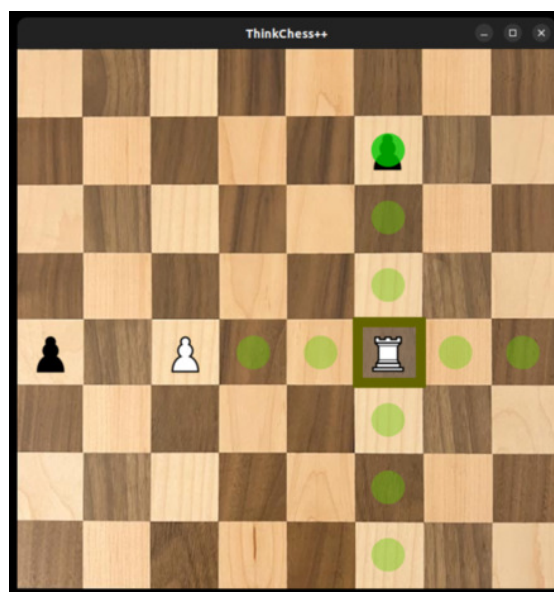
```
bool Knight::isValid(const vector<vector<Piece*>>& bd, int r, int c) {
  bool valid = (abs(r-row) == 1 && abs(c-col) == 2) ||
               (abs(r-row) == 2 && abs(c-col) == 1);
  return valid;
}
```

The Rook

The *Rook* moves any distance on its current rank or file, but cannot jump over any other
piece.

As the rooks movement is limited by other pieces on its way, we have to take those pieces
into account for implementing its `isValid` function:

```
1  bool Rook::isValid(const vector<vector<Piece*>>& bd, int r, int c) {
2    bool valid = r == row || c == col;
3    if (r == row && abs(c-col) > 1) { // same row
4      if (c < col) { // left
5        for (int cc = c+1; cc < col; cc++) {
6          auto pc = bd[r][cc];
7          if (pc) { valid = false; break; }
8        }
9      } else { // right
```

```
10      for (int cc = col+1; cc < c; cc++) {
11        auto pc = bd[r][cc];
12        if (pc) { valid = false; break; }
13      }
14    }
15  }
16  if (c == col && abs(r-row) > 1) { // same column
17    if (r < row) { // top
18      for (int rr = r+1; rr < row; rr++) {
19        auto pc = bd[rr][c];
20        if (pc) { valid = false; break; }
21      }
22    } else { // down
23      for (int rr = row+1; rr < r; rr++) {
24        auto pc = bd[rr][c];
25        if (pc) { valid = false; break; }
26      }
27    }
28  }
29  return valid;
30 }
```

First, we set the result to `valid` if the given field it is on the same row or column as the rook(2).

Then, we check for two cases:

- the field is on the same row (3), or

- the field is on the same column (16).

In both cases, we check wether there is another piece between the field and the rook by iterating over all those fields on that row, respective column. If so, we set `valid` to `false` and stop the search (7, 12, 20, 25).

We have four searches, for every direction the rook can move: up, down, left, right. But only one of them will ever be executed for any given field: either on the same row *or* on the same column. As the search-space is very small for each search (at most 6 checks), we can consider this as a *constant time* search, or in asymptotic notation $\mathcal{O}(1)$.

The Bishop

The *Bishop* can move any distance along the diagonals, on which it is placed, unless its way is blocked by another piece:

Notice, that a bishop, placed initially on a light field, will always stay on a light field, and vice versa. Thus, both players each have a *dark* and a *light* bishop (the white bishop on f4 is a *dark* bishop with its initial position on c1). Dark and light bishops can never attack each other.

We use essentially the same logic as for the rook, but this time we check for pieces on the same diagonal:

```cpp
bool Bishop::isValid(const vector<vector<Piece*>>& bd, int r, int c) {
  bool valid = r-c == row-col || r+c == row+col;
  if (r-c == row-col) { // same major diagonal
    if (r < row) { // upper
      for (int rr = row-1; rr > r; rr--) {
        for (int cc = col-1; cc > c; cc--) {
          if (rr-cc == row-col) {
            auto pc = bd[rr][cc];
            if (pc) { valid = false; break; }
          }
        }
      }
    } else { // lower
      for (int rr = row+1; rr < r; rr++) {
        for (int cc = col+1; cc < c; cc++) {
          if (rr-cc == row-col) {
            auto pc = bd[rr][cc];
            if (pc) { valid = false; break; }
```

```
          }
        }
      }
    }
  }
  if (r+c == row+col) { // same minor diagonal
    if (r < row) { // upper
      for (int rr = row-1; rr > r; rr--) {
        for (int cc = col+1; cc < c; cc++) {
          if (rr+cc == row+col) {
            auto pc = bd[rr][cc];
            if (pc) { valid = false; break; }
          }
        }
      }
    } else { // lower
      for (int rr = row+1; rr < r; rr++) {
        for (int cc = col-1; cc > c; cc--) {
          if (rr+cc == row+col) {
            auto pc = bd[rr][cc];
            if (pc) { valid = false; break; }
          }
        }
      }
    }
  }
  return valid;
}
```

Here, we also have four searches: two for the major diagonal and another two for the minor diagonal. The searches are now nested loops, as we need to get both coordinates for the current field. There are at most $6 \times 6 = 36$ tests per search, and on average only $3 \times 3 = 9$ tests for each. And, as before, only one of these searches will ever by executed for any given field. So, we may consider these searches as *constant time* searches as well.

The Queen

The *Queen* can move any distance in all directions (on its rank, file or diagonals), only limited by other pieces in its way. In the image below you can see, why the queen is the strongest piece on the board:

22
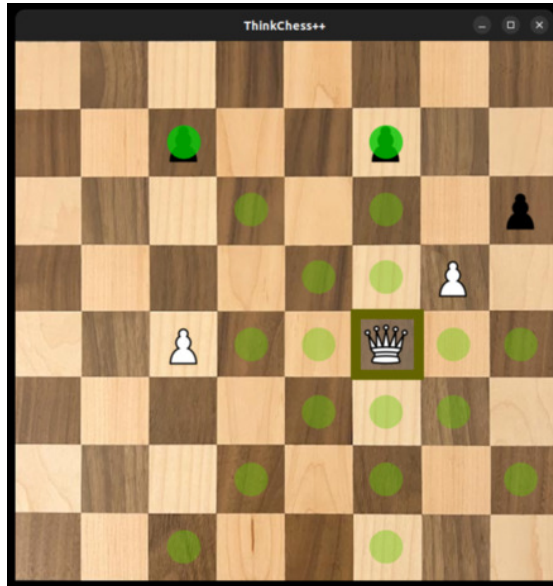
We use a little trick to get the queens valid moves: as the queen can move like a rook *and* a bishop, we just check for valid moves for *either* of them:

```cpp
bool Queen::isValid(const vector<vector<Piece*>>& bd, int r, int c) {
  auto rook = new Rook(white, row, col);
  auto bishop = new Bishop(white, row, col);
  bool valid = rook->isValid(bd, r, c) || bishop->isValid(bd, r, c);
  delete rook;
  delete bishop;
  return valid;
}
```

Observe that we have to delete those dummy pieces, in order to prevent any memory leaks.

The Pawn

The last piece to cover is the *Pawn*: it is the weakest piece on the board, but in exchange, both players have 8 of them. The movement of the pawns is the most complex of all pieces:

- **base case**: move straight forward one square on its file, if that square is vacant

- **capturing**: capture an opponents piece on eiher of the two squares diagonally in front of it

- **initial position**: if the pawn has not yet moved, it has the option of moving two squares straight forward, provided both squares are vacant.

Notice that the terms *forward* and *in front* always refer to the direction towards the opponents pieces: the *white* pawns moves to the higher ranks (e.g. `e2-e4`), while the *black* pawns move towards the lower ranks (e.g. `d3xe2`).
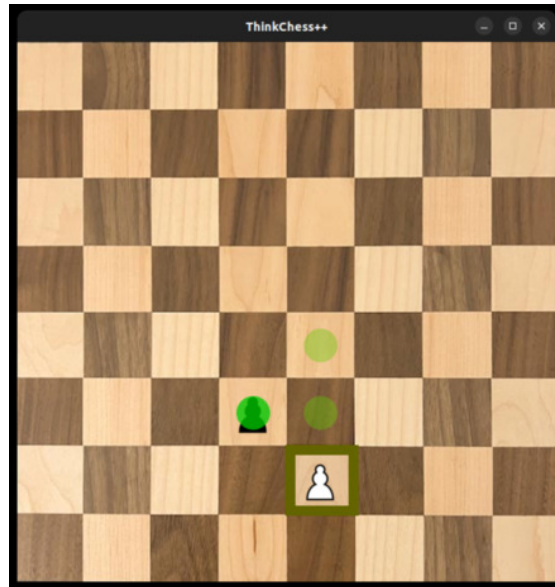
There are also two special moves involving pawns, called *en passant* and *promotion*, which we will cover in section 2.3.

Here's the code to put the rules in action:

```cpp
bool Pawn::isValid(const vector<vector<Piece*>>& bd, int r, int c) {
  bool valid = false;
  auto pc = bd[r][c];
  if (white) {
    if (r == row-1 && c == col) { // move
      if (!pc) valid = true;
    }
    if (row == 6 && r == 4 && c == col) { // initial move
      auto pc1 = bd[5][c];
      if (!pc1 && !pc) valid = true;
    }
    if (r == row-1 && (c == col-1 || c == col+1)) { // capture
      if (pc && pc->isWhite() != white) valid = true;
    }
```

```
15    } else { // black
16      if (r == row+1 && c == col) { // move
17        if (!pc) valid = true;
18      }
19      if (row == 1 && r == 3 && c == col) { // initial move
20        auto pc1 = bd[2][c];
21        if (!pc1 && !pc) valid = true;
22      }
23      if (r == row+1 && (c == col-1 || c == col+1)) { // capture
24        if (pc && pc->isWhite() != white) valid = true;
25      }
26    }
27    return valid;
28  }
```

We have to consider white and black pawns separately, as they move in different directions (4, 15). If there's no piece in front of the pawn, it can move to that field (6, 17). If the pawn is on its initial position (8, 19), it also can move two squares ahead, if both fields ahead aren't occopied (10, 24). If there's a piece diagonal in front of the pawn (12, 23), it can be captured, if it is of the other color (13, 24).

2 Play the Game

Now that we have a board with the pieces on it, we actually want to play the game.

To that goal, we will explore how to make moves and store them for later usage. Finally, we will consider special moves and positions to improve the game play.

2.1 Making Moves

The first thing, we need for making moves, is a way to process user input. We'll do that inside the event loop of the main fuction:

```cpp
// mouse button pressed
if (event.type == sf::Event::MouseButtonPressed) {
  if (event.mouseButton.button == sf::Mouse::Left) {
    pair<int, int> f = getField(event.mouseButton.x,
                                event.mouseButton.y);
    if (touched.first == -1) touched = f;
    else if (f == touched) touched = {-1, -1};
  }
}
// mouse button released
if (event.type == sf::Event::MouseButtonReleased) {
  if (event.mouseButton.button == sf::Mouse::Left) {
    pair<int, int> f = getField(event.mouseButton.x,
                                event.mouseButton.y);
    if (touched.first != -1 && touched != f)
      makeMove(board, moves, captured, touched, f, player);
  }
}

// get board coordinates
pair<int, int> getField(int x, int y) {
  int fx = x / 80;
  int fy = y / 80;
  auto field = make_pair(fy, fx);
```

```
25    return field;
26  }
```

We're using the *left* mouse button for making moves (3, 12); remember that we've used the right button for showing valid moves.

First, we need the coordinates of the board, where the mouse button was pressed (4), respectively released (13).

When *pressing* the button, and no field was touched yet (6), we set the variable `pair<int,int> touched` to those coordinates.

But if a field was already touched, and it's the current field, we reset `touched` (7).

When *releasing* the button, we check wether a field was touched (15), and if it's not the current field, we make a move from the touched to the current field (16).

With this mechanism, we can make a move in two ways:

- *drag-and-drop*: just move a piece from its current position to the target, while holding the left mouse button;

- *click twice*: click once on the piece to move, and a second time on the target field.

The `getField()` function (21-26) converts the coordinates of the mouse cursor to the coordinates of the board matrix. Observe that it returns the coordinates in that order: first the row index, second the column index.

In order to clean up the design, I decided to do some refactorings: I moved all function definitions, except the main function, to a speparate implementation file `app/moves.cpp`; and I moved all variable definitions into the main function, to get rid of global variables. So, we'll have the following definitions inside the main fuction:

```
1   // matrix of pieces representing the board
2   vector<vector<Piece*>> board(8, vector<Piece*>(8));
3
4   // matrix of valid moves for display
5   vector<vector<short>> validMoves(8, vector<short>(8, 0));
6
7   // list of moves, used as a stack
8   auto* moves = new list::List<string>;
9
10  // list of captured pieces, used as a stack
11  auto* captured = new list::List<Piece*>;
12
13  // touched field for making moves
14  pair<int, int> touched{-1, -1};
15
```

```
16    // player to turn, starting with white
17    bool player = true;
```

The variables `moves` and `captures` are *linked lists*, which I've implemented in a separate library called `datastructures` within a namespace `list`. I will not go into details here, but if you are interested in implementing a linked list for yourself, have a look at the file `include/list.hpp` in that directory (it's quite well documented, so you should find your way around).

But now, to the most important function of this section: `makeMove()`.

```cpp
1    void makeMove(vector<vector<Piece*>>& bd,
2                  list::List<string>* mv,
3                  list::List<Piece*>* cp,
4                  pair<int, int>& td,
5                  pair<int, int> to,
6                  bool& player)
7    {
8      auto pcf = bd[td.first][td.second];
9      auto pct = bd[to.first][to.second];
10     bool cap = false;
11     if (!pcf) {
12       cout << "no piece under cursor\n";
13       td = {-1, -1};
14       return;
15     }
16     if (pcf->isWhite() != player) {
17       cout << "it's not your turn\n";
18       td = {-1, -1};
19       return;
20     }
21     if (pcf->isValid(bd, to.first, to.second)) {
22       if (pct && pct->isWhite() != pcf->isWhite()) {
23         cap = true;
24         pct->capture();
25         cp->push_front(pct);
26       } else if (pct) {
27         cout << "illegal move!\n";
28         td = {-1, -1};
29         return;
30       }
31       mv->push_front(convertFromBoard(cap, pcf, to));
32       bd[to.first][to.second] = pcf;
```

28

```
33      bd[td.first][td.second] = nullptr;
34      pcf->makeMove(to.first, to.second);
35      td = {-1, -1};
36      player = !player;
37      cout << mv->peek(1) << "\n";
38    // illegal move
39    } else {
40      cout << "illegal move!\n";
41      td = {-1, -1};
42    }
43  }
```

First, we get the pieces of the start (`td`, the touched field) and target coordinates (`to`) of the move(8, 9).

If there's no piece at the start coordinates (11), reset the touched field (13) and abort the move (14).

If the color of the start piece doesn't match the player, whose turn it is (16), also abort the move (19).

Otherwise, check wether the move is valid (21), and wether the piece at the target position can be captured (22).

If so, capture the piece (24) and and push it onto the stack of captured pieces (25).

Otherwise, if there's a piece of the same color (26), abort the move (29).

So far, we've checked all the possible variations and have a valid move; thus, we add the move to the stack of moves (31).

Then, we set the piece under the cursor to its target position on the board (32), overwriting the old reference, and delete its old postion (33).

Finally, we tell the piece its new position (34), reset the touched field (35), and switch to the other player (36).

For pushing the move onto the moves stack, I used a function `convertFromBoard` (31), which is defined like so:

```
string convertFromBoard(bool cap, Piece* from, pair<int, int> to) {
  string move;
  char type = from->getType();
  if (type != 'P') {
    move.append(1, type);
  }
  move.append(1, colToFile(from->getCol()));
  move.append(1, rowToRank(from->getRow()));
  if (cap) {
    move.append(1, 'x');
```

```
  } else {
    move.append(1, '-');
  }
  move.append(1, colToFile(to.second));
  move.append(1, rowToRank(to.first));
  return move;
}


char colToFile(int col) {
  char file = 97 + col;
  return file;
}


char rowToRank(int row) {
  char rank = 56 - row;
  return rank;
}
```

The function takes a piece (`from`) and target coordinates (`to`) as parameters, and converts them into a string of algebraic chess notation.
It makes use of two helper functions `colToFile()` and `rowToRank()`, which convert the coordinates of the board matrix to chess coordinates.

Those functions use a little trick, based on the specification of the `char` type in C++: the actual value of a character is stored as a `short int`, based on the ascii code for that character. So, the letter 'a' is stored internally as 97, and the digit '8' as 56. We use this fact for a simple calculation.

The only thing left to do is to adjust the `resetBoard()` function to take care of the moves stack and the captured pieces:

```
void resetBoard(vector<vector<Piece*>>& bd,
                list::List<string>* mv,
                list::List<Piece*>* cp)
{
  // reset moves and captured pieces
  delete mv;
  mv = new list::List<string>;
  delete cp;
  cp = new list::List<Piece*>;
  // reset board
  for (auto rank : bd) {
    for (auto piece : rank) {
      delete piece;
```

```
    }
  }
  // --- snip ---
}
```

We have to explicitly delete the moves stack and the stack of captured pieces when resetting to board, in order to avoid memory leaks. But then, we have to create them again as empty lists, as we want to use them for the next game. Notice, that not only the references to those lists are deleted, but also all elements inside the lists, due to that destructor in the file `datastructures/include/list.hpp`:

```cpp
template<typename T>
class List : public LL<T> {
public:
  ~List() {
    Node* node = head;
    while(node) {
      Node* curr = node;
      node = node->next;
      delete curr;
    }
    delete node;
  }
}
```

The very last thing, we want to do in this section, is to add a marker inside the main function for the touched piece on the board

```cpp
sf::RectangleShape frame(sf::Vector2f(63.f, 60.f));
frame.setFillColor(sf::Color(200, 200, 200, 50));
frame.setOutlineThickness(12.f);
frame.setOutlineColor(sf::Color(100, 100, 0));
```

and draw it together with the pieces inside the game loop:

```cpp
// draw pieces
for (int row = 0; row < 8; row++) {
  for (int col = 0; col < 8; col++) {
    if (board[row][col]) {
      auto piece = board[row][col];
      // --- snip ---
      if (row == touched.first && col == touched.second) {
```

```
            frame.setPosition(col*80.f + 10.f, row*80.f + 10.f);
            window.draw(frame);
        }
        pc.setPosition(col*80.f + 10.f, row*80.f + 10.f);
        window.draw(pc);
      }
    }
  }
```

2.2 Special Positions

2.2.1 Check and checkmate

The most prominent position in a game is the check, where the king is under immediate attack. So far, we have no mechanism to respond to or even detect a *check*. Let's change that now.

For that, we only need to iterate over all opponent pieces, and if any of them can reach the king with a valid move, we have a check:

```
1   bool check(vector<vector<Piece*>>& bd, bool white) {
2     Piece* king;
3     bool check = false;
4     for (int row = 0; row < 8; row++) {
5       for (int col = 0; col < 8; col++) {
6         auto current = bd[row][col];
7         if (current && current->getType() == 'K' &&
8             current->isWhite() == white)
9         {
10          king = current;
11        }
12      }
13    }
14    for (int row = 0; row < 8; row++) {
15      for (int col = 0; col < 8; col++) {
16        auto current = bd[row][col];
17        if (current && current->isWhite() != white) {
18          if (current->isValid(bd, king->getRow(), king->getCol())) {
19            check = true;
20          }
21        }
```

```
22        }
23    }
24    return check;
25 }
```

First, iterate over all positions of the board (4-5) and check wether the piece at that position is the king of the given color (7-8). If so, store the position of the king (10).

Then, iterate again over all positions of the board (14-15) and check wether there's a piece of the opponent color (17). If so, check wether this piece can reach the king with a valid move (18), and if it can, set the position to *check* (19).

With that, we have two searches with 64 tests each, and we need to run them sequentially, resulting in 128 tests. Now you could think that's superfluous, as we could test only against the piece actually beeing moved, reducing the second search to a single test. But you would be wrong: a check can be given by any other piece, if a blocking piece is moved. Convince yourself with this diagram:
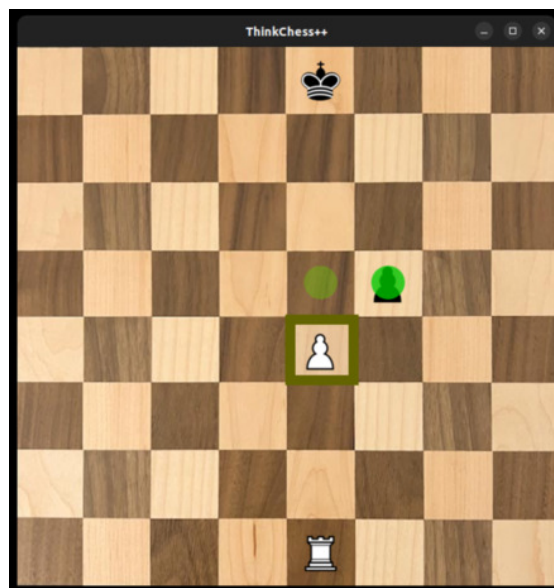
If white captures the black pawn on `f5` with `e4xf5+`, the white rook on the `e`-file will give the black king check.

Now, before making the move, let's look at the other special (and terminal) position: *checkmate*. If a player, given check in the last move, cannot respond to the check in the very next move, that players king is checkmate, and the game is over.

There are, in general, three ways to get out of check:

1. capturing the checking piece

2. moving the king

3. blocking the check.

It's the players duty to analyze the position and to find the best solution. For now, our app cannot do this, it's a quite complex task.

But thankfully, it doesn't have to: we only want to detect a checkmate and decide wether the game is over at this point.

For that, we dont't need any fancy algorithm or some kind of intelligence, we just use a brute-force search. Instead of finding the best solution, we only need to know if there is *any* solution:

```cpp
bool resolveCheck(vector<vector<Piece*>>& bd, bool white) {
  for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 8; col++) {
      auto current = bd[row][col];
      // get every piece of given color
      if (current && current->isWhite() == white) {
        for (int rr = 0; rr < 8; rr++) {
          for (int cc = 0; cc < 8; cc++) {
            if (current->isValid(bd, rr, cc)) {
              auto pct = bd[rr][cc];
              if (!pct || pct->isWhite() != white) {
                // make move and test for check
                bd[rr][cc] = current;
                current->makeMove(rr, cc);
                if (!check(bd, white)) {
                  bd[rr][cc] = pct;
                  current->makeMove(row, col);
                  return true;
                } else {
                  bd[rr][cc] = pct;
                  current->makeMove(row, col);
                }
              }
            }
          }
        }
      }
    }
  }
  return false;
}
```

Iterate over all fields of the board (2-3); if there's a piece at this postion with the color of the checked player (6), start another nested loop (7-8) to get the valid moves of that piece (9).

If the reachable fields are empty or the pieces at these positions are of the other color (i.e. they can be captured), make this move by setting the current piece to this position (13-14).

If this new position is not giving check (15), we have a solution and return `true`, indicating that the check could be resolved with that move (18).

But, before doing so, reset the board to its original state, as we don't want to actually make the move (16-17).

If the new position still is giving check (19), reset the board as well (20-21) and continue the search.

With that, we have four nested loops with $8 \times 8 \times 8 \times 8 = 64 \times 64 = 4096$ tests, which is the cost of *brute search*. But, real work is only done, if we have a piece of the given color (outer nested loops), and if this piece has any valid moves (inner nested loops).

A player has at most 16 pieces on the board, and assuming that each of them has 3 valid moves on average, the `check` function will be called mo more than 48 times on average. Thus, we may consider that search as of constant time as well.

With all that in place, we can finally make a move, while testing for *check* and *checkmate*:

```cpp
void makeMove(vector<vector<Piece*>>& bd,
              list::List<string>* mv,
              list::List<Piece*>* cp,
              pair<int, int>& td,
              pair<int, int> to,
              bool& player,
              pair<int, int>& checkmate)
{
  auto pcf = bd[td.first][td.second];
  auto pct = bd[to.first][to.second];
  bool cap = false;
  if (!pcf) {
    cout << "no piece under cursor\n";
    td = {-1, -1};
    return;
  }
  if (pcf->isWhite() != player) {
    cout << "it's not your turn\n";
    td = {-1, -1};
    return;
  }
```

```cpp
22    // valid move?
23    if (pcf->isValid(bd, to.first, to.second)) {
24      // can capture?
25      if (pct && pct->isWhite() != pcf->isWhite()) {
26        cap = true;
27        cp->push_front(pct);
28      } else if (pct) { // same color
29        cout << "illegal move!\n";
30        td = {-1, -1};
31        return;
32      }
33      // legal move
34      string move = convertFromBoard(cap, pcf, to);
35      bd[to.first][to.second] = pcf;
36      bd[td.first][td.second] = nullptr;
37      pcf->makeMove(to.first, to.second);
38      // check?
39      if (check(bd, player)) { // gives itself check
40        cout << "illegal move!\n";
41        bd[to.first][to.second] = pct;
42        bd[td.first][td.second] = pcf;
43        pcf->makeMove(td.first, td.second);
44        td = {-1, -1};
45        return;
46      }
47      if (check(bd, !player)) { // gives opponent check
48        if (resolveCheck(bd, !player)) {
49          move.append(1, '+');
50        } else { // cannot get out of check
51          move.append(1, '#');
52          mv->push_front(move);
53          cout << mv->peek(1) << "\n";
54          checkmate = getKing(bd, !player);
55          return;
56        }
57      }
58      mv->push_front(move);
59      cout << mv->peek(1) << "\n";
60      td = {-1, -1};
61      player = !player;
62    // illegal move
63    } else {
64      cout << "illegal move!\n";
```

```
65      td = {-1, -1};
66    }
67  }
```

We have an extra parameter `pair<int, int>& checkmate` for keeping track of *checkmate*. Besides, There's nothing new in lines (1-37).

Then test, wether the move gives the own king check (39), and if it does, abort the move (44-45), as this is not allowed. But, before that, reset the already made move (41-43). Notice, that we had to make the move in the first place in order to allow the test for check.

If the move gives the opponents king check (47), test wether the check could possibly be resolved (48), and if so, add the marker '+' to the move (indicating check, 49).
If the check cannot be resolved (50), add the marker '#' to the move (indicating checkmate, 51), push the move to the moves stack (52), and set the `checkmate` variable to the postion of the checkmated king (54). Then, proceed as before (58-67).

The only thing missing, is the implementation of the `getKing` function on line 54. It's essentially the same as the first loop of the `check` function, but only returning the coordinates of the king. Try to implement that function for yourself; if you get stucked, check my implementation in the file `app/moves.cpp`.

We use the `checkmate` variable to to stop the game and draw a corresponding marker on the board within the main function of the app:

```cpp
short state = 1;
pair<int, int> checkmate{-1, -1};

//marker for checkmate
sf::RectangleShape cm(sf::Vector2f(63.f, 60.f));
cm.setFillColor(sf::Color(200, 200, 200, 50));
cm.setOutlineThickness(12.f);
cm.setOutlineColor(sf::Color(200, 0, 0));

// game loop
while (window.isOpen()) {
  // event loop
  for (auto event = sf::Event{}; window.pollEvent(event);) {
    if (event.type == sf::Event::Closed) {
      window.close();
    }
    // game is running in play mode
    if (state == 1) {
      // mouse button pressed
```

37

```
        if (event.type == sf::Event::MouseButtonPressed) {
          if (event.mouseButton.button == sf::Mouse::Right) {
            pair<int, int> f =
              getField(event.mouseButton.x, event.mouseButton.y);
            setValidMoves(board, validMoves, board[f.first][f.second]);
          }
          if (event.mouseButton.button == sf::Mouse::Left) {
            pair<int, int> f =
              getField(event.mouseButton.x, event.mouseButton.y);
            if (touched.first == -1) touched = f;
            else if (f == touched) touched = {-1, -1};
          }
        }
        // mouse button released
        if (event.type == sf::Event::MouseButtonReleased) {
          if (event.mouseButton.button == sf::Mouse::Right) {
            validMoves = vector<vector<short>>(8, vector<short>(8, 0));
          }
          if (event.mouseButton.button == sf::Mouse::Left) {
            pair<int, int> f =
              getField(event.mouseButton.x, event.mouseButton.y);
            if (touched.first != -1 && touched != f)
              makeMove(board, moves, captured, touched,
                       f, player, checkmate);
          }
        }
      } // end play mode
    } // end event loop

    // draw pieces
    for (int row = 0; row < 8; row++) {
      for (int col = 0; col < 8; col++) {
        if (board[row][col]) {
          // --- snip ---
          if (row == checkmate.first && col == checkmate.second) {
            cm.setPosition(col*80.f + 10.f, row*80.f + 10.f);
            window.draw(cm);
          }
          pc.setPosition(col*80.f + 10.f, row*80.f + 10.f);
          window.draw(pc);
        }
      }
    }
  }
```

```
// display frame
window.display();

// stop game when checkmate
if (checkmate.first != -1) state = 0;
```

And here you have it: a fully playable chess app for two players!

Playing the moves

1. e2-e4 e7-e5
2. Ng1-f3 Ng8-f6
3. Nf3xe5 Nf6xe4
4. Qd1-f3 Ne4-c5
5. Qf3xf7#

will lead to this position, correctly identified as checkmate (a.k.a the *Scholar's mate*):

2.2.2 Stalemate

Stalemate is a situation in chess where the player, whose turn it is to move, is not in check but has no legal move. Stalemate immediately results in a draw (i.e. the game is over and both players are credited half a point). During the endgame, stalemate is a

39

resource that can enable the player with the inferior position to draw the game rather than lose.

Stalemate is mucher harder to detect than checkmate: as the king has not been given check, we cannot use our `check` and `resolveCheck` functions for this position. Of course, we could try to do something similar, but only with a much higher expense. But, as stalemates are actually much rarer than checkmates, I decided to not spend this effort for now.

So, when playing a game with the app, both players have to agree to stalemate (and with that to a draw), like they would have to in a tournament.

2.3 Special Moves

To complete this chapter, we will now implement the missing *special* moves for

- the king: castling
- the pawns: en passant and promotion.

2.3.1 Castling

Castling is the only move, in which two pieces are moved at once (the king and one of the rooks of the same color). The king moves two squares towards a rook on the same rank and the rook moves to the square that the king passed over.

Castling is permitted only if

- neither the king nor the rook has previously moved
- the squares between the king and the rook are vacant
- the king does not leave, cross over, or finish on a square attacked by an enemy piece.
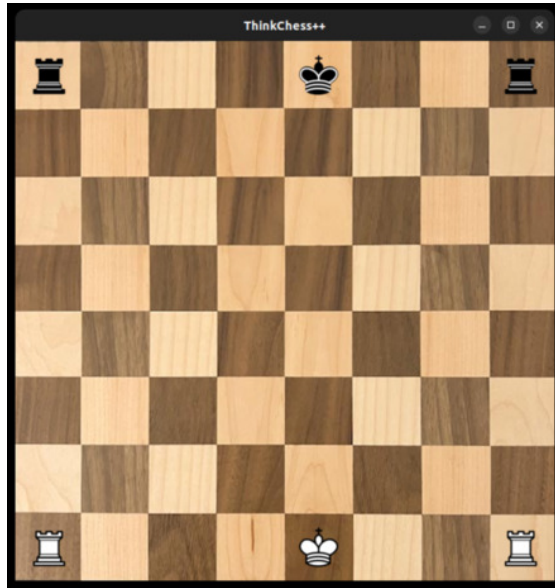
Assuming in the diagramm above, that neither of the players has casteld yet, they could move like so:
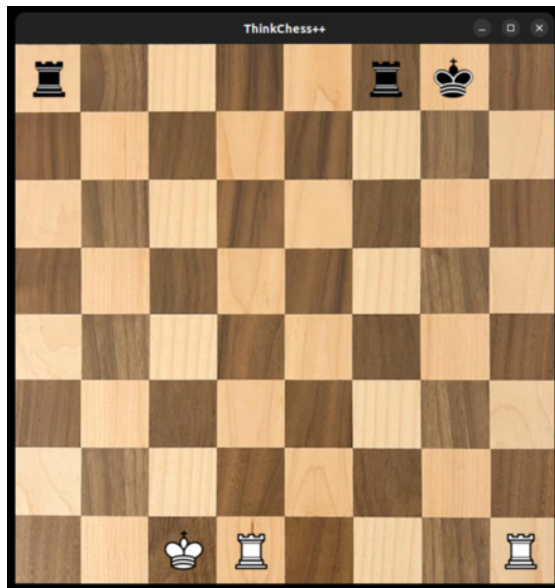
White has castled to the queenside (indicated with 0-0-0), while black has castled to the kingside (indicated with 0-0).

We implement the logic for *castling* with a function called `castling`:

```cpp
char castling(vector<vector<Piece*>>& bd,
              Piece* king, pair<int, int> to)
{
  if (check(bd, king->isWhite())) return 'N'; // king is in check
  if (king->isWhite()) { // white
    if (king->getRow() == 7 && king->getCol() == 4) {
      // TODO: king has not yet moved
      if (to.first == 7 && to.second == 6) { // kingside
        auto rook = bd[7][7];
        if (!rook || rook->getType() != 'R'
                  || rook->isWhite() != king->isWhite()) {
          // TODO: rook has moved
          return 'N';
        }
        auto pc1 = bd[7][5];
        auto pc2 = bd[7][6];
        if (pc1 || pc2) return 'N'; // fields occupied
        else {
          // test for check
          bd[7][5] = king;
          king->makeMove(7, 5);
          if (check(bd, king->isWhite())) {
            bd[7][4] = king;
            king->makeMove(7, 4);
            bd[7][5] = nullptr;
            return 'N';
          } else {
            bd[7][4] = king;
            king->makeMove(7, 4);
            bd[7][5] = nullptr;
          }
          bd[7][6] = king;
          king->makeMove(7, 6);
          if (check(bd, king->isWhite())) {
            bd[7][4] = king;
            king->makeMove(7, 4);
            bd[7][6] = nullptr;
            return 'N';
          } else {
            bd[7][4] = king;
            king->makeMove(7, 4);
            bd[7][6] = nullptr;
          }
```

```
44              return 'K';
45          }
46        } else if (to.first == 7 && to.second == 2) {
47          // --- snip queenside ---
48        }
49      }
50    } else {
51      // --- snip black ---
52    }
53    return 'N';
54 }
```

First, test wether the king is currently in check, and if so, abort with negative result (4). Then, test for the white king (5): we have to test the kings separately, as they sit on different ranks.

Next, test wether the king is on its initial position (6). Observe, that for now, this test is not complete: the king could have moved from and back to its initial position. Since we will learn to process stored moves only in the next chapter, there is no way to complete the test for now. But we must not forget to make this right, so I've left a `TODO` comment here.

Next, test for kingside castling (8): if there's no rook of the same color on its initial position (10-11), abort with negative result (13). Observe, that we have the same issue as with the king before, so I've added a `TODO` marker as well.

Then, test wether the fields between the king and the rook are occopied (17), and if not, continue with the tests for check for these fields (18). For that, we move the king to the first field (20-21) and test for check (22). If it is in check, abort with negative result (26), after undoing the king move (23-25), since we don't want to actually move the king. Otherwise reset the move as well (27-31), and test the second field for check (32-43), following the logic from the first field.

If there are no pieces at both fields, and the king would not be in check at those fields, return positive result `K` for kingside castling (44).

Then, repeat the process for queenside castling (46), and finally both processes for the black king (51). If no valid castling position was found, return negative result (53).

With that, we have a lot of redundant code, only differing in the ranks of the king. Of course, one could write that in a more compact way. But that could lead to an overly clever solution, which would be hard to understand for someone without implicit knowledge of what's going on, even for your future self. So, I decided to leave the code this way and to make every step explicit.

The compiler won't care about lengthy code anyway. On the contrary, when optimization is enabled (e.g. with the flag -O3), the compiler will create the most efficient machine code possible.

By the way, if you want to compile the project in release mode with optimization enabled, you can call <cmake .. -DCMAKE_BUILD_TYPE=Release> from the build directory, followed by <cmake -build .>.
This will recompile the whole project, and every time, you call cmake after that, the project will be compiled in release mode, unless you decide to delete the build directory and re-create it with cmake -B build from the root folder.

Having castling in place, we can use it inside our makeMove function:

```cpp
// already castled, 0 = no, 1 = white, 2 = black, 3 = both
short castled = 0;

void makeMove(vector<vector<Piece*>>& bd,
              list::List<string>* mv,
              list::List<Piece*>* cp,
              pair<int, int>& td,
              pair<int, int> to,
              bool& player,
              pair<int, int>& checkmate,
              short& castled)
{
  auto pcf = bd[td.first][td.second];
  auto pct = bd[to.first][to.second];
  bool cap = false;
  if (!pcf) {
    cout << "no piece under cursor\n";
    td = {-1, -1};
    return;
  }
  if (pcf->isWhite() != player) {
    cout << "it's not your turn\n";
    td = {-1, -1};
    return;
  }
  // castling
  short cast = player ? 1 : 2;
  if (castled < 3 && castled != cast && pcf->getType() == 'K' && !pct) {
    string move = "";
    char form = castling(bd, pcf, to);
    if (form == 'K') {
      castled = castled > 0 ? 3 : cast;
```

```cpp
33        move = "0-0";
34        // make king move
35        bd[td.first][td.second+2] = pcf;
36        pcf->makeMove(td.first, td.second+2);
37        bd[td.first][td.second] = nullptr;
38        // make rook move
39        auto rook = bd[td.first][7];
40        bd[td.first][td.second+1] = rook;
41        rook->makeMove(td.first, td.second+1);
42        bd[td.first][7] = nullptr;
43        if (check(bd, !player)) { // gives opponent check
44          if (resolveCheck(bd, !player)) {
45            move.append(1, '+');
46          } else { // cannot get out of check
47            move.append(1, '#');
48            mv->push_front(move);
49            cout << mv->peek(1) << "\n";
50            checkmate = getKing(bd, !player);
51            return;
52          }
53        }
54        // complete move
55        mv->push_front(move);
56        cout << mv->peek(1) << "\n";
57        td = {-1, -1};
58        player = !player;
59        return;
60      } else if (form == 'Q') {
61        castled = castled > 0 ? 3 : cast;
62        move = "0-0-0";
63        // make king move
64        bd[td.first][td.second-2] = pcf;
65        pcf->makeMove(td.first, td.second-2);
66        bd[td.first][td.second] = nullptr;
67        // make rook move
68        auto rook = bd[td.first][0];
69        bd[td.first][td.second-1] = rook;
70        rook->makeMove(td.first, td.second-1);
71        bd[td.first][0] = nullptr;
72        if (check(bd, !player)) { // gives opponent check
73          if (resolveCheck(bd, !player)) {
74            move.append(1, '+');
75          } else { // cannot get out of check
```

```
76          move.append(1, '#');
77          mv->push_front(move);
78          cout << mv->peek(1) << "\n";
79          checkmate = getKing(bd, !player);
80          return;
81        }
82      }
83      // complete move
84      mv->push_front(move);
85      cout << mv->peek(1) << "\n";
86      td = {-1, -1};
87      player = !player;
88      return;
89    } // else do nothing and continue with check for valid move
90  }
91  // --- snip processing other moves ---
92 }
```

We have a new variable `castled` inside the main function (2), and a respective parameter in the `makeMove` function (11) for keeping track of castling.

The new part starts on line (27): we define a local variable `cast` and set it to the player, whose turn it is. Then we test, wether that player has already castled, and wether the piece to move is a king and the target field is not occupied (28). Then, test wether castling is possible for that king (30), and if it's a kingside castling (31), do the following:

1. indicate, that the player has castled (32)

2. set the correct notation for the move (33)

3. make the king move by setting it 2 fields to the right (35-37)

4. make the rook move by setting it to the left of the king (39-42)

5. test wether the move gives the opponents king check (43)

6. if so, test wether the check could be resolved by the opponent (44)

7. if so, complement the move's notation with a `+`

8. otherwise, it is checkmate: complete the move and stop the game (46-52)

9. complete the move (55-59)

Then, repeat the steps above for queenside castling (60-89). I've left out the subsequent code, as it does not differ from the previous version.

With that, we can perform (almost) correct castling for both players; we'll revisit the code for the `castling` method in the next chapter.

2.3.2 Promotion

Promotion is the replacement of a pawn with a new piece when the pawn is moved to its last rank. The player replaces the pawn immediately with a queen, rook, bishop, or knight of the same color. The new piece does not have to be a previously captured piece.

Promotion is almost always to a queen (a.k.a *queening*), as it is the most powerful piece. Since we don't have a sufficient GUI yet (so the player cannot choose a different piece), we will restrict promotion to queening for now.

With that restriction, it is quite easy to implement promotion inside our `makeMove` function:

```
// valid move?
if (pcf->isValid(bd, to.first, to.second)) {
// can capture?
if (pct && pct->isWhite() != pcf->isWhite()) {
  cap = true;
  cp->push_front(pct);
} else if (pct) { // same color
  cout << "illegal move!\n";
  td = {-1, -1};
  return;
}

string move = convertFromBoard(cap, pcf, to);

// promotion
if (pcf && pcf->getType() == 'P') {
  if (pcf->isWhite() && pcf->getRow() == 1) { // white
    pcf = new Queen(1, pcf->getRow(), pcf->getCol());
    move.append("=Q");
  } else if (!pcf->isWhite() && pcf->getRow() == 6) { // black
    pcf = new Queen(0, pcf->getRow(), pcf->getCol());
    move.append("=Q");
  }
} // end promotion
```

```
25
26   // legal move
27   bd[to.first][to.second] = pcf;
28   bd[td.first][td.second] = nullptr;
29   pcf->makeMove(to.first, to.second);
```

We have to put the new code (15-24) inside the check for valid moves, just before making
the move (27-29). The code tests wether the piece to move is a pawn (16), and wether
it is currently placed on the last but one rank for its color (17, 20). If so, the pawn
is simply replaced with a queen of the same color (18, 21) and the moves notation is
completed accordingly (19, 22).
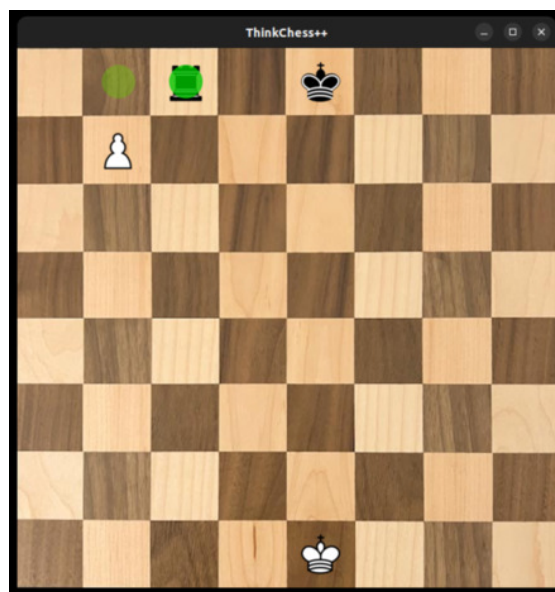
When the pawn on `b7` captures the rook on `c8` in the diagram above, our app will promote
the pawn to a queen and also detects the check, reporting this move: `b7xc8=Q+`.

2.3.3 En passant

The last special move, we want to cover, is a pawns move called en passant.

The move describes the capture by a pawn of an enemy pawn on the same rank and an
adjacent file, that has just made an initial two-square advance.
The capturing pawn moves to the square that the enemy pawn passed over, as if the
enemy pawn had advanced only one square.

The rule ensures that a pawn cannot use its two-square move to safely skip past an enemy pawn.

But, since capturing *en passant* is permitted only on the turn immediately after the two-square advance, we would need to investigate the moves history for this. As we have no means to do that for now, we'll implement that move only in a later chapter, marking it with `TODO` in the meanwhile.

2.4 Showing valid moves (revisited)

In section 1.2 we learned how to show the valid moves for any piece, by pressing the right mouse button on it.

So far, this will only display the valid moves in general, ignoring special positions (§ 2.2) and special moves (§ 2.3). So, we will not see the moves for castling when clicking on a king, but we'll see moves, which are not allowed in the current position (whenever the move would give check to the own king).

We could certainly resolve this, but this would also mean a major redesign of our display logic, and would not bear any other benefits for the current game behavior. So I decided, to leave it as it is for now.

On the bright side, this behavior has a positve didactic effect for learners: if you want to make a move, shown as valid, and the app refuses that move (for good reasons), you have to analyze the position for yourself and find the reason why. In most of theses cases, you'd have missed a given check.

3 Enhancing the GUI

In this chapter we'll add a dashboard to our app, including

- a *timer* for keeping track of the player's thinking time

- an indicator for *material evaluation*

- the display of *captured pieces* for each player

- a *move indicator* for the last move

- a display for the complete *moves history*, and

- an *info field* for displaying messages from the app.

We will also store the current game to a textfile for later use, and add a splash-screen, allowing to display the result of a game and to switch the game mode.

Furthermore, we will give a player the option to take back a move, to offer a draw, and to resign.

Finally, we'll revisit some *special moves*.

With all that in place, our app will look like so:

3.1 Refactoring

First of all, I created a new translation unit at `app/display.cpp` and its header file at `include/display.hpp`, which will contain all the display-oriented logic.

Next, I created another new unit at `app/position.cpp` and its header file at `include/position.hpp` which contains all the move-oriented logic.
I moved all the functions from the old file `app/moves.cpp` to this new file and deleted the old file.

This new file contains a new type `Position`, which holds all the position-relevant variables, so I could delete those from the `main()` function.

```cpp
class Position {
public:
  ~Position() {}
  Position(short gs) : gamestate{gs}, player{true},
                board{8, vector<Piece*>(8)},
```

```cpp
                mvCount{0}, castled{0}, checkmate{-1, -1},
                checked{false}, eval{0.f}
{}

// returns a material evaluation for both players
void evaluate();

// take back last move
bool takeBackMove();

// make a move
bool makeMove(pair<int, int> from, pair<int, int> to);

// gamestate: 0 = game over, 1 = 2-player, 2 = analyze
short gamestate;

// player on turn, starting with white
bool player;

// matrix of pieces representing the board
vector<vector<Piece*>> board;

// vector of moves, used as a stack
vector<string> moves;

// list of captured pieces, used as a stack
vector<Piece*> captured;

// total count of moves
int mvCount;

// castled: 0 = no, 1 = white, 2 = black, 3 = both
short castled;

// coordinates of the piece, which gave checkmate
std::pair<int, int> checkmate;

// last move gave check
bool checked;

// basic evaluation of positions
float eval;
```

```
  // infotext
  string info;
};
```

Observe, that I've made all the member types public, in order to avoid creating all the associated *getter* and *setter* functions. This would not be a good design decision, if we were to create a library, which was intended to be used by different clients, as those clients could harm the game logic. But, in our case, the only client is the `main()` function of our app, and I suppose that we know what we are doing.

With the new type `Position` in place, we can initialize all relevant variables with a single call from inside the main function: `Position position(0);` and call the `makeMove` function with just two parameters (which are the field coordinates from the board) like so: `position.makeMove(touched, to)`.

Finally, I extended the viewport of the app and made it not resizable:

```
auto window = sf::RenderWindow{ {870, 640u},
                                "ThinkChess++",
                                sf::Style::Close,
                                settings };

window.clear(sf::Color(100, 100, 100));
```

The last line just sets the background of the display area to gray (inside the game loop).

3.2 Timer

For displaying the *timer*, we first have to include the `chrono` library and define two time counters, as well as two time points:

```
#include <chrono>
using namespace chrono;

unsigned wTime = 0;
unsigned bTime = 0;
steady_clock::time_point last;
last = steady_clock::now();
steady_clock::time_point now;
```

53

For calculating the current thinking time for both player, we have this code inside the game loop, just before drawing anything:

```
1  if (position.gamestate == 1) {
2    now = steady_clock::now();
3    if (now - last > 1s) {
4      last = steady_clock::now();
5      if (position.player) wTime++;
6      else bTime++;
7    }
8  }
```

The code checks, wether we are in playing mode (gamestate is set to 1), and captures the current time point (2). If the difference between `now` and `last` is greater than one second (3), increase the time counter for the player whose turn it is (5-6). Remember, we have set the `sf::FramerateLimit` to 10, so this code will be called 10 times pers second, and we cannot simply increase the time counters for each pass.

In oder to actually draw the timer, we have this code inside the game loop (directly after the last code snippet):

```
1   string timer = "";
2   if (position.player) {
3     window.draw(wActive);
4     timer = getTime(wTime);
5     wTimer.setString(timer);
6   } else {
7     window.draw(bActive);
8     timer = getTime(bTime);
9     bTimer.setString(timer);
10  }
11  if (position.checkmate.first != -1) {
12    if (position.player) {
13      bActive.setFillColor(sf::Color(200, 0, 0));
14      window.draw(bActive);
15    } else {
16      wActive.setFillColor(sf::Color(200, 0, 0));
17      window.draw(wActive);
18    }
19  }
20  window.draw(wTimer);
21  window.draw(bTimer);
```

This code also draws the indicator for the active player (the white or black circle in front of the timer) in lines 3 and 7. If the app has detected a checkmate (11), the active player's indicator is set to a red color (13, 16).

For that to work, we need to have the respective `sf::Sprite`s to be defined (before entering the game loop):

```
1   // set timer
2   sf::Text wTimer;
3   wTimer.setFont(noto);
4   wTimer.setCharacterSize(24);
5   wTimer.setStyle(sf::Text::Bold);
6   wTimer.setFillColor(sf::Color::White);
7   wTimer.setString("00:00:00");
8   wTimer.setPosition(690.f, 10.f);
9
10  sf::Text bTimer = wTimer;
11  bTimer.setFillColor(sf::Color::Black);
12  bTimer.setPosition(690.f, 40.f);
13
14  // marker for aktive player
15  sf::CircleShape wActive(10.f);
16  wActive.setPosition(650.f, 15.f);
17  wActive.setFillColor(sf::Color::White);
18
19  sf::CircleShape bActive(10.f);
20  bActive.setPosition(650.f, 45.f);
21  bActive.setFillColor(sf::Color::Black);
```

In order to display text with SFML, we have to set a font to the `sf::Text` sprite (3), and we also must have a font already in place (just before the last snippet):

```
sf::Font noto;
if (!noto.loadFromFile("../img/NotoMono-Regular.ttf")) {
  cout << "failed to load the font\n";
  return 1;
}
```

I have chosen the *NotoMono-Regular* font, because it is a fixed-space font and quite compact: it only contains 897 glyphs (the images for actually drawing each character), in contrast to more than 5,000 glyphs for fancier fonts.

The last thing missing is the function `getTime()` to convert a time counter (an unsigned integer value) into a readable time format. This function is located inside the new `app/display.cpp` implementation file:

```cpp
string getTime(unsigned t) {
  string result = "";
  unsigned h = 0;
  unsigned m = t / 60;
  unsigned s = t % 60;
  if (m >= 60) {
    h = m / 60;
    m = m % 60;
  }
  if (h < 10) {
    result.append(1, '0');
    result.append(to_string(h));
  } else {
    result.append(to_string(h));
  }
  result.append(1, ':');
  if (m < 10) {
    result.append(1, '0');
    result.append(to_string(m));
  } else {
    result.append(to_string(m));
  }
  result.append(1, ':');
  if (s < 10) {
    result.append(1, '0');
    result.append(to_string(s));
  } else {
    result.append(to_string(s));
  }
  return result;
}
```

The logic is straight forward: just divide the total amount of seconds by 60 to get the minutes (4), and take the ramainder of that calculation to get the remaining seconds (5). If the resulting minutes are greater than 60 (6), repeat those steps to get the hours and remaining minutes (7-8). The rest of the code (10-29) is just for formatting: if hours, minutes, and seconds are less than 10, insert the digit 0 at the appropriate position.

3.3 Material evaluation

Material evaluation is the first step in evaluating the players positions on the board.

Evaluating positions is a copmlex task, for which one must have a deeper understanding of several factors, including the pawn structure, control of the center, space and initiative. We'll explore the basics of position evaluation in the next chapter, which will also be the first chapter mainly about *playing* chess.

For now, we are only interested in the first part of this process, which is straight forward: add up the values of the pieces of each player and compare them.

For that we need a new function, which needs access to the board; so it's a natural choice to define it as a member of the type `Position`:

```cpp
void evaluate() {
  pair<int, int> matEval = evaluateBoard(board);
  eval = float(matEval.first) / 100
       - float(matEval.second) / 100;
}
```

The function calls a helper function, which is defined in the corresponding implementation file:

```cpp
pair<int, int> evaluateBoard(const vector<vector<Piece*>>& bd) {
  int white = 0;
  int black = 0;

  for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 8; col++) {
      auto pc = bd[row][col];
      if (pc) {
        pc->isWhite() ? white += pc->getValue()
                      : black += pc->getValue();
      }
    }
  }
  pair<int, int> eval = make_pair(white, black);
  return eval;
}
```

Observe my design choice: I defined all functions, which need a reference to the board, but will not modify it, as helper functions outside the class `Position`, and I made the

reference to the board `const`. So the compiler will ensure that indeed no updates to the board will happen inside those functions.

I also decided to sligthly modify the value scheme for pieces of section 1.1.2: the queen now has a value of 9, and the rooks each a value of 5. To be able to do more accurate calculations later, the values are defined as *centipawns* by multiplying them with 100. With that, we'll be able to make all future calculations on the base of integer values, which should be faster than calculating with `float` representations of the value.

The current implementations of the pieces look like so:

```cpp
class Queen : public Piece {
public:
  ~Queen() {}
  Queen(bool w, int r, int c) :
    type{'Q'}, white{w}, value{900}, row{r}, col{c} {}

  char getType() override { return type; }
  int getValue() override { return value; }
  bool isWhite() override { return white; }
  int getRow() override { return row; }
  int getCol() override { return col; }
  void makeMove(int r, int c) override { row = r; col = c; }
  bool isValid(const vector<vector<Piece*>>& bd, int r, int c) override;

private:
  char type;
  bool white;
  int value;
  int row;
  int col;
};
```

Now we can use the evaluation function inside the game loop to actually draw the *evaluation meter*:

```cpp
if (moved) {
  if (position.mvCount % 2 == 0) {
    position.evaluate();
    eval = position.eval;
    if (eval > 10.0) {
      eval = 10.f;
      mb[2].color = sf::Color(200, 0, 0, 200);
```

```
 8        mb[4].color = sf::Color(200, 0, 0, 200);
 9        mb[5].color = sf::Color(200, 0, 0, 200);
10      }
11      if (eval < -10.0) {
12        eval = -10.f;
13        mb[0].color = sf::Color(200, 0, 0, 200);
14        mb[1].color = sf::Color(200, 0, 0, 200);
15        mb[3].color = sf::Color(200, 0, 0, 200);
16      }
17      mi[2].position = sf::Vector2f(750.f + eval*6.f, 80.f);
18      mi[3].position = sf::Vector2f(750.f + eval*6.f, 100.f);
19      if (eval > 0) {
20        mi[2].color = sf::Color::White;
21        mi[3].color = sf::Color::White;
22      } else {
23        mi[2].color = sf::Color::Black;
24        mi[3].color = sf::Color::Black;
25      }
26    }
27  }
```

When a move is complete, i.e. both players have moved (2), evaluate the material (3).
Then, set the local variable `eval` to the calculated value (4); we are doing so, because
we don't want to tinker with the member variables of `Position` from outside the class
(remember my caveat for this design decision).
For display purposes, we restict that value to the range $[-10, 10]$, and set the background
color of the evaluation meter to red, if the value is outside that range (5-16).
Finally, we set the position and the color of the evaluation indicator (17-24). We'll see
soon, how this works.

You may have noticed that this code will only be executed, if a move was made (1).
Remember, this code is placed inside the game loop, and we don't want to evaluate the
board for every update of the display. So I modified the `Position::makeMove()` function
to report success, and call it like so: `moved = position.makeMove(touched, to)`:

```
bool makeMove(pair<int, int> td, pair<int, int> to) {
  auto pcf = board[td.first][td.second];
  auto pct = board[to.first][to.second];

  if (!pcf) {
    info = "no piece selected";
    return false;
  }
```

```
  if (pcf->isWhite() != player) {
    info = "it's not your turn";
    return false;
  }
  // --- snip ---
}
```

The background of the evaluation meter (`mb`) is defined as an SFML primitive (a set of two unconnected triangles), and the evaluation indicator (`mi`) as a set of two unconnected vertical lines:

```
sf::VertexArray mb(sf::Triangles, 6);
mb[0].position = sf::Vector2f(690.f, 80.f);
mb[1].position = sf::Vector2f(690.f, 100.f);
mb[2].position = sf::Vector2f(810.f, 100.f);
mb[3].position = sf::Vector2f(690.f, 80.f);
mb[4].position = sf::Vector2f(810.f, 80.f);
mb[5].position = sf::Vector2f(810.f, 100.f);
mb[0].color = sf::Color::White;
mb[1].color = sf::Color::White;
mb[2].color = sf::Color::Black;
mb[3].color = sf::Color::White;
mb[4].color = sf::Color::Black;
mb[5].color = sf::Color::Black;

sf::VertexArray mi(sf::Lines, 4);
mi[0].position = sf::Vector2f(750.f, 80.f);
mi[1].position = sf::Vector2f(750.f, 100.f);
mi[2].position = sf::Vector2f(750.f, 80.f);
mi[3].position = sf::Vector2f(750.f, 100.f);
mi[0].color = sf::Color::Green;
mi[1].color = sf::Color::Green;
mi[2].color = sf::Color::Black;
mi[3].color = sf::Color::Black;
```

I've set the color of the triangle points to black and white, such that the meter's background color is interpolated to fill the background with a gradient from white to black. The green line of the indicator shows the neutral position (the evaluation value 0) and it's static. The black line ($mi[2]$ and $mi[3]$) is the actual indicator, which we have updated when processing a move.

With this setup, the indicator will move to the right when the evaluation value is greater than zero (i.e. white has an advantage), and move to the left when black has an advantage. This may seem counter-intuitive at first, but once you get used to it, you'll catch the evaluation at a glance.

3.4 Captured pieces

For displaying the captured pieces of each player, we only need two background areas:

```
sf::RectangleShape bcw(sf::Vector2f(210.f, 20.f));
bcw.setFillColor(sf::Color(30, 30, 30, 200));
bcw.setPosition(650.f, 115.f);
sf::RectangleShape bcb = bcw;
bcb.setFillColor(sf::Color(220, 220, 220));
bcb.move(0.f, 30.f);
```

and a way to draw miniature versions of the pieces figures onto those:

```
window.draw(bcw);
window.draw(bcb);
int wc = -1;
int bc = -1;
for (auto piece : position.captured) {
  sf::Sprite cp;
  switch (piece->getType()) {
  case 'K':
    piece->isWhite() ? cp = wk : cp = bk;
    break;
  case 'Q':
    piece->isWhite() ? cp = wq : cp = bq;
    break;
  case 'R':
    piece->isWhite() ? cp = wr : cp = br;
    break;
  case 'B':
    piece->isWhite() ? cp = wb : cp = bb;
    break;
  case 'N':
    piece->isWhite() ? cp = wn : cp = bn;
    break;
  case 'P':
```

```
      piece->isWhite() ? cp = wp : cp = bp;
      break;
  }
  cp.scale(0.3f, 0.3f);
  if (piece->isWhite()) {
    wc++;
    cp.setPosition(650.f + wc*15.f, 115.f);
  } else {
    bc++;
    cp.setPosition(650.f + bc*15.f, 145.f);
  }
  window.draw(cp);
}
```

Getting the captured pieces is easy, as we had already stored them in a separate data structure `captured` within the `position` object, while making a move:

```
// valid move?
if (pcf->isValid(board, to.first, to.second)) {
  // can capture?
  if (pct && pct->isWhite() != pcf->isWhite()) {
    cap = true;
    captured.push_back(pct);
  } else if (pct) { // same color
    info = "illegal move";
    return false;
  }
  // --- snip ---
}
```

I've rejected the idea of storing the captures pieces in a linked list; instead, I'm using a vector of pieces: `vector<Piece*>`. We can easily use a vector as a *stack* (a *LIFO* data structure) by using its member functions `push_back()` and `pop_back()` in a very performant way. Additionally, vectors provide an iterator, which we can use in the *range-for* statement `for (auto piece : position.captured)`, without actually *popping* the elements.

3.5 Move indicator

The *move indicator* is implemented with a similar pattern:

```
// current move background
sf::RectangleShape mvb(sf::Vector2f(120.f, 20.f));
mvb.setFillColor(sf::Color(200, 200, 0, 200));
mvb.setPosition(690.f, 190.f);

// current move indicator
sf::Text mvi;
mvi.setFont(noto);
mvi.setCharacterSize(16);
mvi.setFillColor(sf::Color::Black);
mvi.setPosition(720.f, 190.f);
```

As with the evaluation meter, we only update the move indicator when a move was made:

```
if (moved) {
  // --- snip ---
  position.mvCount > 0 ? mvi.setString(position.moves.back())
                       : mvi.setString("");
  position.checked ? mvb.setFillColor(sf::Color(200, 100, 0, 200))
                   : mvb.setFillColor(sf::Color(200, 200, 0, 200));
  position.checked = false;
  position.info.clear();
  itt.setString("");
  // --- snip ---
}
```

Additionally, if the `position.checked` flag was set when making the move, we set the background color of the move indicator to orange.

3.6 Moves history

The next feature of our GUI is the display of the complete moves history for a game, but we also want to persintently store the hsitory in a text file. This is done with the follwing code:

```
1  if (moved) {
2    // --- snip ---
3    if (position.mvCount % 2 == 0) {
4      game << position.moves.back() << "\n";
5      history += position.moves.back();
```

```
6      history += "\n";
7    } else {
8      game << (position.mvCount / 2) + 1 << ". "
9            << position.moves.back() << " ";
10     history += to_string((position.mvCount / 2) + 1);
11     history += ". ";
12     history += position.moves.back();
13     history += " ";
14   }
15   // trim history string
16   if (position.mvCount > 52 && position.mvCount % 2 == 1) {
17     auto pos = history.find("\n") + 1;
18     history = history.substr(pos, history.size());
19   }
20   hist.setString(history);
21   // --- snip ---
22 }
```

If it was a black move (3), write the move in algebraic notation to the file-stream `game` and add a newline (4). Also add it to the `history` string (5-6).

If it was a white move (7), write it to the file-stream as well (9), but only after prepending it with the current move number (8). Also add the move to the history string (10-13).

If there are more than 52 half-moves (i.e. 26 complete moves) in the history (16), we trim the history string from its beginning (17-18), as there's only space for 26 moves in the history display area. But, we'll never trim the history of the file stream, so there's no action needed in this case.

In order for this to work, we need to have the file-stream and the history field in place:

```
#include <fstream>

string lastGame = "../games/last.txt";
std::fstream game;

// moves history
sf::RectangleShape hisb(sf::Vector2f(180.f, 380.f));
hisb.setFillColor(sf::Color::White);
hisb.setPosition(660.f, 220.f);
sf::Text hist;
hist.setFont(noto);
hist.setCharacterSize(12);
```

```
hist.setFillColor(sf::Color::Black);
hist.setPosition(670.f, 230.f);
```

We're using an `fstream` to be able to read from and write to that stream. But that's not enough: we also need to make sure that the game file actually exists, and if not, to create it:

```
1   game.open(lastGame, std::ios::trunc);
2   if (!game.is_open()) {
3     game.clear();
4     game.open(lastGame, std::ios::out); // create file
5     game.close();
6     game.open(lastGame);
7   }
```

If the file already exists, open it in truncation mode, so that all contents will be cleared (1). If not (2), re-open the file in output mode to create the file (4), close and open it again in write mode (5-6).

We must not forget do synchronize the file stream with the actual file and to close the stream whenever a game is finished:

```
if (game.is_open()) {
  game.flush();
  game.close();
}
```

This is currently done in two cases: when the main window is closed (inside the event loop), and when a checkmate is detected (at the very end of the game loop).

3.7 Keyboard input

3.7.1 Splash Screen

First, we're going to add a splash screen to our GUI to be able to start and end a game properly:

```cpp
sf::RectangleShape bsplash(sf::Vector2f(280.f, 90.f));
bsplash.setFillColor(sf::Color(30, 30, 30, 200));
bsplash.setPosition(155.f, 235.f);
sf::Text welcome;
welcome.setFont(noto);
welcome.setCharacterSize(16);
welcome.setFillColor(sf::Color(220, 0, 0));
welcome.setStyle(sf::Text::Bold);
welcome.setPosition(170.f, 240.f);
welcome.setString("Welcome to ThinkChess++");
sf::Text spt;
spt.setFont(noto);
spt.setCharacterSize(16);
spt.setFillColor(sf::Color(0, 220, 0));
spt.setPosition(210.f, 270.f);
spt.setString("Start new game <S>\nLoad last game <L>");
```

Remember, we initialize the main position object like so: `Position position(0);`
Having the gamestate initially set to 0, allows us to distinguish between the game modes
and to draw the display, when starting a new game, like so:

```cpp
// inside game loop
if (position.gamestate == 0) {
  window.draw(bsplash);
  window.draw(spt);
  if (position.checkmate.first != -1) {
    string restart;
    if (position.player) restart = "      White ";
    else restart = "      Black ";
    restart.append("wins!");
    welcome.setString(restart);
  }
  window.draw(welcome);
}
```

This also allows us to distinguish between modes inside the event loop, and the player
is able to start a new game in splash mode:

```cpp
// inside event loop
if (position.gamestate == 0) {
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::S)) {
    position = Position(1);
```

```
 5        resetBoard(position);
 6        game.open(lastGame, std::ios::trunc);
 7        if (!game.is_open()) {
 8          game.clear();
 9          game.open(lastGame, std::ios::out); // create file
10          game.close();
11          game.open(lastGame);
12        }
13        bActive.setFillColor(sf::Color::Black);
14        wActive.setFillColor(sf::Color::White);
15        mi[2].position = sf::Vector2f(750.f - position.eval*3.f, 80.f);
16        mi[3].position = sf::Vector2f(750.f - position.eval*3.f, 100.f);
17        mvb.setFillColor(sf::Color(200, 200, 0, 200));
18        mvi.setString("");
19        wTime = 0;
20        bTime = 0;
21        last = chrono::steady_clock::now();
22        bTimer.setString("00:00:00");
23        history.clear();
24        hist.setString(history);
25      }
26    }
```

Here, we read input from the keyboard (3) and if the key 'S' is pressed, the position object is re-created with gamestate 1 (4), and the board is reset (5).

We also re-load the game file (6-12, which we have seen in the last section), and reset the active players indicators (13-14).

The evaluation meter is also reset (15-17) and the move indicator cleared (17-18).

Then, reset the thinking time for both players (19-20), start the timer (21) and reset the timers display (22). Finally, clear the moves history (23-24).

All this is necessary because of this: we use the splash screen not only when starting the app, but also whenever a checkmate during a game is detected and the game is ended:

```
// last statement in game loop
if (position.checkmate.first != -1) {
  position.gamestate = 0;
  if (game.is_open()) {
    if (position.player) {
      game << "\n";
      game << "1-0\n";
    }
```

```
        else game << "0-1\n";
        game.flush();
        game.close();
      }
    }
```

3.7.2 Draw and resign

In section 2.2.2 we learned that our app will not discover a *stalemate*. To make up for this, each player gets the opportunity to offer a draw, after a move has made.

```cpp
// inside event loop
if (position.gamestate == 1) {
  // draw offer
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::D) && !draw) {
    draw = true;
    itt.setString("Accept a draw? (Y/N)");
  }
  // respond to draw offer
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::Y) && draw) {
    position.gamestate = 0;
    draw = false;
    if (!position.player) game << "\n";
    game << "1/2-1/2\n";
    game.flush();
    game.close();
    welcome.setString("     It's a draw!");
  }
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::N) && draw) {
    itt.setString("Draw offer declined.");
    draw = false;
  }
  // --- snip ---
}
```

If the D-key is pressed during the game (4), a corresponding message is displayed (6). Now the other player has to decide if she's accepting the draw. If so, she'll respond with 'Y' (9), and the game is ended (10) and the result is recorded (12-16).
Otherwise, if the response is 'N' (18), the draw offer is declined (19) and the player, who made the offer, is still on turn.

Observe that the app will respond to the keys 'Y' and 'N' only if a draw was actually offered, in order to prevent unintentional draw (9, 18).

Similar rules are in place, when a player wants to resign:

```cpp
if (sf::Keyboard::isKeyPressed(sf::Keyboard::G) && !giveUp) {
  giveUp = true;
  itt.setString("Sure to give up? (Y/N)");
}
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Y) && giveUp) {
  position.gamestate = 0;
  giveUp = false;
  if (position.player) {
    game << "0-1\n";
    welcome.setString("    White gives up!\n");
  } else {
    game << "\n";
    game << "1-0\n";
    welcome.setString("    Black gives up!\n");
  }
  game.flush();
  game.close();
}
if (sf::Keyboard::isKeyPressed(sf::Keyboard::N) && giveUp) {
  giveUp = false;
  itt.setString("Okay, move on.");
}
```

But this time, the player who made the request is expected to confirm, again in order to prevent unintentional resign.

For both cases, draw and resign, the app will behave like this: if no answer is given to the request, the player whose turn it is, can proceed with a regular move, and the request is declined.

3.7.3 Take back a move

I wanted to add another feature for smoothing the game experience: while playing a regular game, each player should be able to take back the last move unless the other player has moved. This does not comply with the original rules, but since you can't argue with an app, I decided to add it anyway. The code for triggering a take-back is located inside the event loop, just as the triggers for *draw* and *resign* from the last section:

```
1   // inside event loop && gamestate == 1
2   if (sf::Keyboard::isKeyPressed(sf::Keyboard::T) && !takeback) {
3     takeback = true;
4     string lastMove = position.moves.back();
5     int sz = lastMove.size();
6     moved = position.takeBackMove();
7     if (moved) {
8       if (position.mvCount > 0) {
9         string newLast = position.moves.back();
10        mvi.setString(newLast);
11        newLast.back() == '+'
12            ? mvb.setFillColor(sf::Color(200, 100, 0, 200))
13            : mvb.setFillColor(sf::Color(200, 200, 0, 200));
14      } else {
15        mvi.setString("");
16      }
17      if (position.mvCount % 2 == 0) { // last move was white
18        if (position.mvCount >= 199) sz += 6;
19        else if (position.mvCount >= 19) sz += 5;
20        else sz += 4;
21      } else sz += 1; // last move was black
22      game.seekp(-sz, std::ios_base::end);
23      history = history.substr(0, history.size() - sz);
24      hist.setString(history);
25      moved = false;
26    }
27  }
```

When a player hits the T-key (2), get the last move and determine the length of that string (4-5). Then call the new function `takeBackMove()` and capture its success (6); this is the main task for this feature.

If the take-back was successful (7), we have to update the game history, as the move has already been recorded (8-25).

If the last move was not the very first move (8), set the move indicator to that last move (9-10) and set the backgound of the move indicator to orange if the last move gave check (11-13).

If the last move was white (17), calculate the offset for the new writing position (18-20), otherwise (black move) set the offset to the length of the `lastMove`-string and add 1 for a single whitespace (21). This is necessary, because we have added the move number in front of every white move, which is increasing with every move.

Finally, set the writing position for the file-stream to the calculated offset (22) and right-trim the history string (23).

With that in place, let's have a look on the new `takeBackMove` function:

```cpp
bool takeBackMove() {
  string move;
  if (moves.size() > 0) {
    move = moves.back();
    moves.pop_back();
    mvCount--;
  } else return false;
  vector<int> coords = parseMove(move);
  auto pc = board[coords[4]][coords[3]];
  // promotion
  if (move.back() == 'Q' || move[move.size()-2] == 'Q') {
    bool white = coords[4] == 0;
    pc = new Pawn(white, coords[1], coords[0]);
  }
  board[coords[1]][coords[0]] = pc;
  pc->makeMove(coords[1], coords[0]);
  if (coords[2] == 0) { // normal move
    board[coords[4]][coords[3]] = nullptr;
  } else { // captured
    auto cp = captured.back();
    board[coords[4]][coords[3]] = cp;
    captured.pop_back();
  }
  player = !player;
  return true;
}
```

If the last move was the first move, do nothing and return failure (7), otherwise get the move's algebraic notation string (4) and delete it from the moves vector (5-6).
Then get the coordinates of the last move by parsing its notation string (8), and get the piece of the target coordinates from the last move (9).
If the last move was a *promotion* (queening), replace that piece with a new *Pawn* (11-14).
Then set the piece to the starting coordinates from the last move (15-16).
If the move was a 'normal' move, delete the piece from the target position (17-18); otherwise (if it was a capture) place the last captured piece on the target coordinates and delete it from the vector of captured pieces (19-23).

The `parseMove` function from line 8 and its helper functions `fileToCol` and `rankToRow` are implemented like so:

```cpp
vector<int> parseMove(string move) {
  vector<int> coords(5, 0);
  if (isupper(move[0])) move.erase(0, 1);
  for (int i = 0; i < 5; i++) {
    if (i == 0 || i == 3) { // file
      coords[i] = fileToCol(move[i]);
    } else if (i == 1 || i == 4) { // rank
      coords[i] = rankToRow(move[i]);
    } else { // captured?
      move[i] == 'x' ? coords[i] = 1 : coords[i] = 0;
    }
  }
  return coords;
}

// convert files to colums
int fileToCol(char file) {
  int col = file - 97;
  return col;
}

// convert ranks to rows
int rankToRow(char rank) {
  int row = 56 - rank;
  return row;
}
```

The `parseMove` function converts the algebraic notation of the last move to a vector of board coordinates. If a piece was captured with the last move, we set the according flag on position 2 of that vector.

The helper functions just convert the characters of the move string to the according integer values.

This logic works fine for regular moves and promotion, but unfortunately it does not work for *castling*, since we have no coordinates for this move in the notation string (either `0-0` for kingside castling or `0-0-0` for queenside castling. So we have to solve this the hard way by undoing the castling manually. The following code snippet is placed inside the `takeBackMove` function, just after the first test:

```cpp
bool takeBackMove() {
  // --- snip ---
  if (move.back() == '0' || move[move.size()-2] == '0') {
    if (move.back() == '+') move.pop_back();
```

```cpp
  char form;
  move == "0-0" ? form = 'K' : form = 'Q';
  bool white = (mvCount % 2 == 0);
  if (white && form == 'K') {
    board[7][4] = board[7][6];
    board[7][6] = nullptr;
    auto king = board[7][4];
    king->makeMove(7, 4);
    board[7][7] = board[7][5];
    board[7][5] = nullptr;
    auto rook = board[7][7];
    rook->makeMove(7, 7);
    castled == 1 ? castled = 0 : castled = 2;
  }
  if (white && form == 'Q') {
    board[7][4] = board[7][2];
    board[7][2] = nullptr;
    auto king = board[7][4];
    king->makeMove(7, 4);
    board[7][0] = board[7][3];
    board[7][3] = nullptr;
    auto rook = board[7][0];
    rook->makeMove(7, 0);
    castled == 1 ? castled = 0 : castled = 2;
  }
  if (!white && form == 'K') {
    board[0][4] = board[0][6];
    board[0][6] = nullptr;
    auto king = board[0][4];
    king->makeMove(0, 4);
    board[0][7] = board[0][5];
    board[0][5] = nullptr;
    auto rook = board[0][7];
    rook->makeMove(0, 7);
    castled == 2 ? castled = 0 : castled = 1;
  }
  if (!white && form == 'Q') {
    board[0][4] = board[0][2];
    board[0][2] = nullptr;
    auto king = board[0][4];
    king->makeMove(0, 4);
    board[0][0] = board[0][3];
    board[0][3] = nullptr;
```

```
      auto rook = board[0][0];
      rook->makeMove(0, 0);
      castled == 2 ? castled = 0 : castled = 1;
    }
    player = !player;
    return true;
  }
  // --- snip ---
}
```

3.8 Special moves revistited

We have two open task from section :

- implement *en passant* for pawns

- verify that neither the king nor the rooks have moved for *castling*,

which we'll take care of in this section.

3.8.1 En passant

In order to show *en passant* as a valid move, I added this code to the `Pawn::isValid()` member function:

```
1  bool Pawn::isValid(const vector<vector<Piece*>>& bd, int r, int c) {
2    bool valid = false;
3    auto pc = bd[r][c];
4    if (white) {
5      // en passant
6      if (row == 3 && r == 2 && abs(col-c) == 1) {
7        if (!pc) {
8          auto pawn = bd[row][c];
9          if (pawn && pawn->getType()=='P' && pawn->isWhite() != white) {
10            valid = true;
11          }
12        }
13      }
14    } else { // black
15      // en passant
16      if (row == 4 && r == 5 && abs(col-c) == 1) {
17        if (!pc) {
```

```
18        auto pawn = bd[row][c];
19        if (pawn && pawn->getType()=='P' && pawn->isWhite() != white) {
20          valid = true;
21        }
22      }
23    }
24  }
25  return valid;
26 }
```

If a white pawn is placed on the 5th rank, respectively a black pawn on the 4th rank, and it wants to make a capture move (6, 16), check wether it's a regular capture (7, 17). If there's no piece at the target coordinates, check wether there's a pawn of the other color placed right next to it (9, 19). If so, mark the move as valid (10, 20).

Observe, that *en passant* will be shown as valid, even if the opponents last move wasn't an initial pawn move. There's no way of checking this inside the `isValid` function, since we don't have access to the main position object. I could have changed that, but this would also mean to change all the other `isValid` functions as well, and I didn't want to do that.

So, we're going to add the test for the last pawn move to the `Position::makeMove` function:

```
1  bool makeMove(pair<int, int> from, pair<int, int> to) {
2    auto pcf = board[from.first][from.second];
3    auto pct = board[to.first][to.second];
4    bool cap = false;
5    bool passant = false;
6
7    // valid move?
8    if (pcf->isValid(board, to.first, to.second)) {
9      // can capture?
10     if (pct && pct->isWhite() != pcf->isWhite()) {
11       cap = true;
12       captured.push_back(pct);
13     } else if (pct) { // same color
14       info = "illegal move";
15       return false;
16     }
17     // en passant
18     if (pcf && pcf->getType() == 'P') {
19       if (pcf->isWhite() && pcf->getRow() == 3) { // white
```

```
20          if (!pct) {
21            auto pawn = board[pcf->getRow()][to.second];
22            if (pawn && pawn->getType() == 'P' &&
23               pawn->isWhite() != player)
24            {
25              string lastMove = moves.back();
26              pair<int,int> epTo=make_pair(pawn->getRow(),pawn->getCol());
27              auto epPc = new Pawn(!player, 1, to.second);
28              string ep = convertFromBoard(false, epPc, epTo);
29              delete epPc;
30              if (ep != lastMove) {
31                info = "illegal move";
32                return false;
33              } else {
34                passant = true;
35                cap = true;
36                captured.push_back(pawn);
37                board[pcf->getRow()][to.second] = nullptr;
38              }
39            }
40          }
41        } else if (!pcf->isWhite() && pcf->getRow() == 4) { // black
42          if (!pct) {
43            auto pawn = board[pcf->getRow()][to.second];
44            if (pawn && pawn->getType() == 'P' &&
45               pawn->isWhite() != player)
46            {
47              string lastMove = moves.back();
48              pair<int,int> epTo=make_pair(pawn->getRow(),pawn->getCol());
49              auto epPc = new Pawn(!player, 6, to.second);
50              string ep = convertFromBoard(false, epPc, epTo);
51              delete epPc;
52              if (ep != lastMove) {
53                info = "illegal move";
54                return false;
55              } else {
56                passant = true;
57                cap = true;
58                captured.push_back(pawn);
59                board[pcf->getRow()][to.second] = nullptr;
60              }
61            }
62          }
```

```
63        }
64      } // end en passant
65      string move = convertFromBoard(cap, pcf, to);
66      if (passant) move.append("ep");
67      // --- snip ---
68    }
69  }
```

Observe, that we were able to put the new logic inside the test for valid moves (8), since the updated `Pawn::isValid` function will now report the move as generally valid.

If the piece to move is a pawn (18), and if it's placed on a rank allowing an *en passant* (19), test wether there is a piece placed at the target field (20). If not, check wether there's a pawn of the opposite color placed right next to the pawn (21-23). This is redundant to the logic of the `isValid` function, but that doesn't hurt, as it makes sure that we're dealing only with an *en passant move.*

Now we can actually check for the last move (26-37), and if it was the initial move for the other pawn, mark the move as valid (34-37). Otherwise abort the move (30-32).

Repeat that procedure for black (41-59), and if it is an *en passant* move, add `ep` to the moves notation in order to indicate that special move (66).

The only thing left to do is to take care about taking back an *en passant* move:

```
1  bool takeBackMove() {
2    string move;
3    if (moves.size() > 0) {
4      move = moves.back();
5      moves.pop_back();
6      mvCount--;
7    } else return false;
8    // en passant
9    if (move.back() == 'p' && move[move.size()-2] == 'e') {
10     move.pop_back();
11     move.pop_back();
12     vector<int> coords = parseMove(move);
13     auto pc = board[coords[4]][coords[3]];
14     board[coords[4]][coords[3]] = nullptr;
15     board[coords[1]][coords[0]] = pc;
16     pc->makeMove(coords[1], coords[0]);
17     auto cp = captured.back();
18     captured.pop_back();
19     bool white = (mvCount % 2 == 0);
```

```
20      if (white) {
21        board[coords[4]+1][coords[3]] = cp;
22      } else {
23        board[coords[4]-1][coords[3]] = cp;
24      }
25      player = !player;
26      return true;
27    }
28    // --- snip ---
29  }
```

If the last move was an *en passant* (9), remove the last two characters from it, in order to get the coordinates for a regular pawn move (12). Then, get the piece of the target position of the last move (13), delete it from that position (14), and set it to its original position before the move (15-16).

Then we have to deal with the captured pawn (17-24):
first, retrieve it from the captured pieces (17-18). Then check wether the last move was white (19). If so, move the captured piece one rank down and place it back there (21), otherwise move it one rank up and place it back there (23). Finally, complete the move and report success (25-26).

3.8.2 Castling

In order to make testing for *King* and *Rook* moves possible, I moved the helper function `castling` from its compilation unit `app/position.cpp` to its header file at `include/position.hpp` and made it a member function of the type `Position`.
This also fits my design choice of defining all functions with a non constant reference to the board as member functions.

With that, I was able to add the follwing code to the `castling` function:

```
1  char castling(Piece* king, pair<int, int> to) {
2    if (check(board, king->isWhite())) return 'N'; // king is in check
3    string history;
4    for (auto mv : moves) {
5      history += mv;
6    }
7    if (king->isWhite()) { // white
8      auto pos = history.find("Ke1");
9      if (pos != std::string::npos) return 'N'; // king has moved
10     if (king->getRow() == 7 && king->getCol() == 4) {
11       if (to.first == 7 && to.second == 6) { // kingside
```

```
12        auto rook = board[7][7];
13        if (!rook || rook->getType() != 'R'
14                 || rook->isWhite() != king->isWhite())
15        {
16          return 'N'; // no rook
17        } else {
18          auto pos = history.find("Rh1");
19          if (pos != std::string::npos) return 'N'; // rook has moved
20        }
21        // --- snip ---
22      }
23    }
24  }
25 }
```

I've created a history string (3), and filled it with all moves so far (4-6). Then, for the white king, the history is searched for the substring `Ke1` (8). If that substring is found in the history, the king has already moved, and castling is denied (9).

We use the same logic to test for moves of the kingside rook (13-20), and only need to apply it for kingside castling and the black king as well, and we are done here.