ThinkChess^{++}

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

Oliver Krischer

February 29, 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 status, and how to find a good next move.

To that goal we we'll also dive into data structures (*lists, 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 Main.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
1  #include <SFML/Graphics.hpp>
2
3  int main() {
4    sf::ContextSettings settings;
5    settings.antialiasingLevel = 8;
6    auto window = sf::RenderWindow{ {640u, 640u}, "ThinkChess++",
7                  sf::Style::Default, settings };
8    window.setFramerateLimit(10);
9
10   while (window.isOpen()) {
11     for (auto event = sf::Event{}; window.pollEvent(event);) {
12       if (event.type == sf::Event::Closed) {
13         window.close();
14       }
15     }
16     window.draw(bs);
17     window.display();
18   }
19 }
```

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.

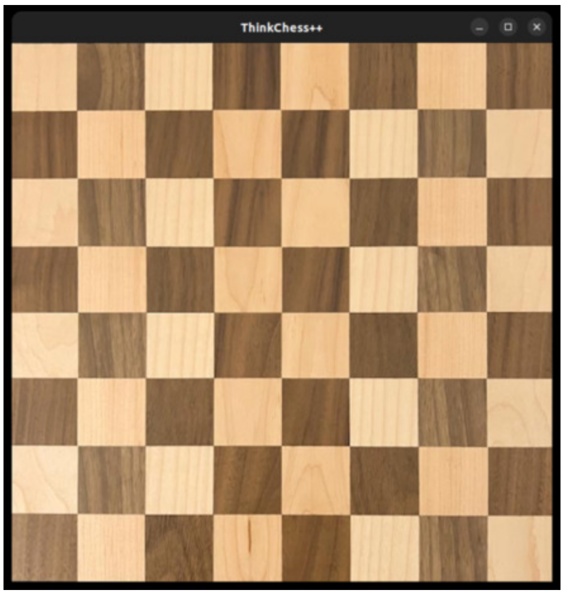
1.1.1 The board

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);
```

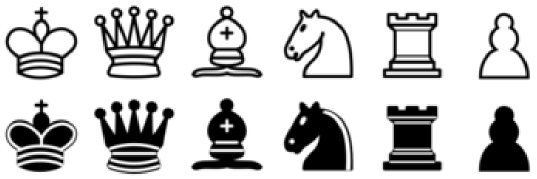
Here, we create a SFML texture (1) and load an image from the file system into that texture (2). Then we create a SFML sprite (3) and set its texture to that image (4).

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
sf::Texture figures;
figures.loadFromFile("../img/figures.png");

sf::Sprite wk;
wk.setTexture(figures);
wk.setTextureRect(sf::IntRect(0,0,60,60));

sf::Sprite bk;
bk.setTexture(figures);
bk.setTextureRect(sf::IntRect(0,60,60,60));
// --- 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:

```
#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;
};
```

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:

```
1  class King : public Piece {
2  public:
3    ~King() {}
4    King(bool w, int r, int c) : type{'K'}, white{w}, value{0},
5                                captured{false}, row{r}, col{c} {}
6
7    char getType() override { return type; }
8    char getValue() override { return value; }
9    bool isWhite() override { return white; }
10   bool isCaptured() override { return captured; }
11   int getRow() override { return row; }
12   int getCol() override { return col; }
13   bool isValid(std::vector<std::vector<Piece*>>& bd, int r, int c)
14     override;
15
16 private:
17   char type;
18   bool white;
19   short value;
20   bool captured;
21   int row;
22   int col;
23 };
```

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:

```
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

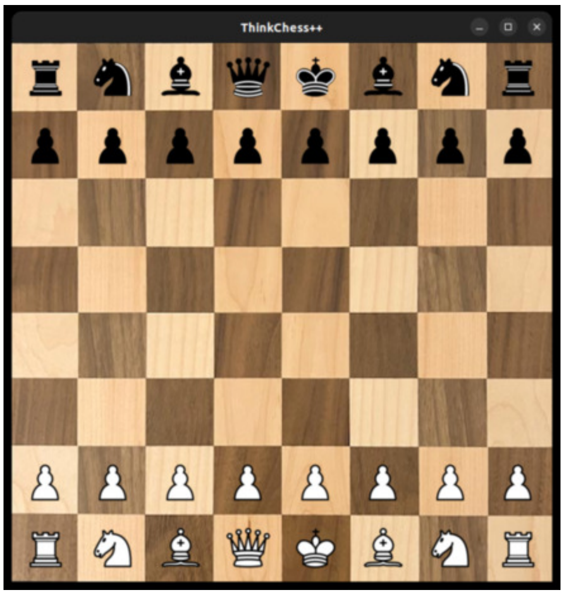
Now we can use the filled board matrix to actually draw the pieces within the game loop of the main function:

```
    // 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:

```
    // in event loop
    // mouse button pressed
    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, board[f.first][f.second]);
      }
    }
    // 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));
```

```
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:

```
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):

```
 1  // before game loop
 2  sf::CircleShape valid(20.f);
 3
 4    // inside game loop
 5    // draw valid moves
 6    for (int row = 0; row < 8; row++) {
 7      for (int col = 0; col < 8; col++) {
 8        if (validMoves[row][col] > 0) {
 9          valid.setPosition(col*80.f + 20.f, row*80.f + 20.f);
10          if (validMoves[row][col] > 1)
11            valid.setFillColor(sf::Color(0, 200, 0, 200));
12          else valid.setFillColor(sf::Color(100, 200, 0, 100));
13          window.draw(valid);
14        }
15      }
16    }
```

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).

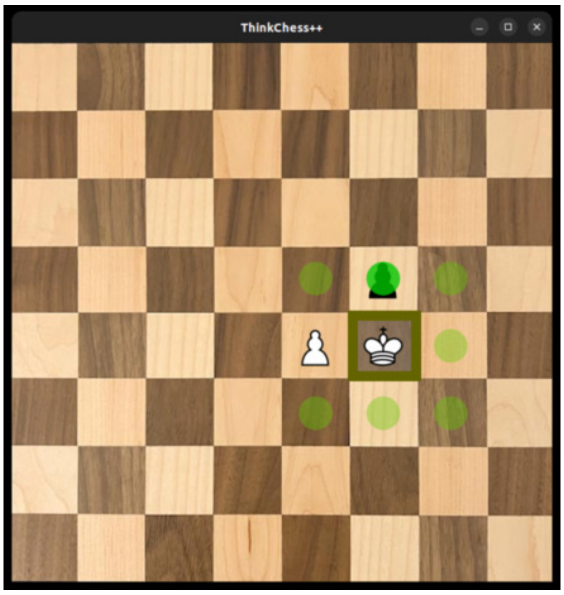
With that, all valid moves for a selected piece are displayed while pressing the right mouse button (with a green circular shape on the board). As soon as you release the mouse botton, the `validMoves` vector is reset, and there's nothing more to draw for the next frame.

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`:

```
1  #include "pieces.hpp"
2  #include <vector>
3  using namespace std;
4
5  bool King::isValid(vector<vector<Piece*>>& bd, int r, int c) {
6    bool valid = abs(r-row) <= 1 && abs(c-col) <= 1;
7    return valid;
8  }
```

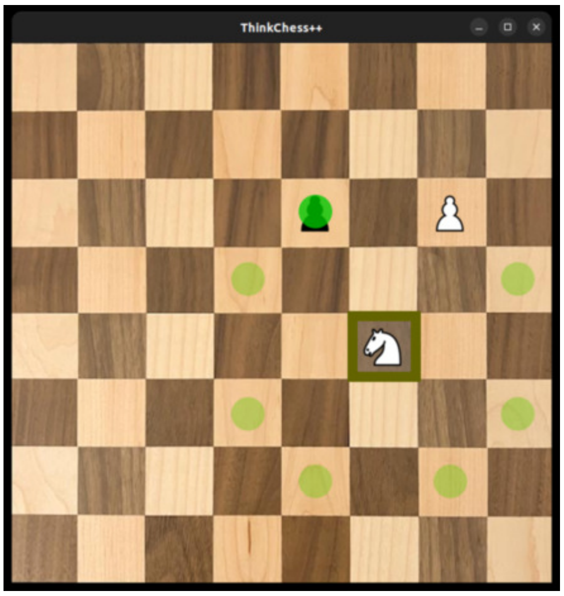
First, we have to include the header file located at `include/pieces.hpp` (1), in order to make the type definitions of the pieces available. Observe, that we don't have to spell out the complete path to that file, thanks to these instructions of our CMakeLists.txt file:

```
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:



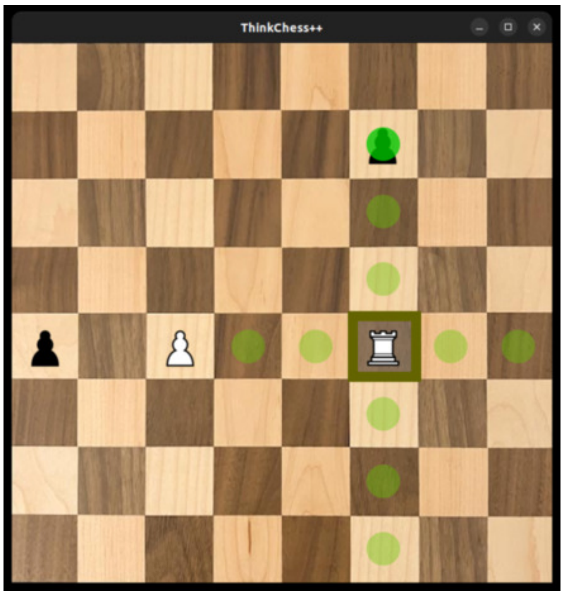
Observe that a knight, placed on a dark field, can reach only light fields, and vice versa.

Its isValid function is implemented like so:

```cpp
bool Knight::isValid(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:

```cpp
bool Rook::isValid(vector<vector<Piece*>>& bd, int r, int c) {
  bool valid = r == row || c == col;
  if (r == row && abs(c-col) > 1) { // same row
    if (c < col) { // left
      for (int cc = c+1; cc < col; cc++) {
        auto pc = bd[r][cc];
        if (pc) { valid = false; break; }
      }
    } 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:

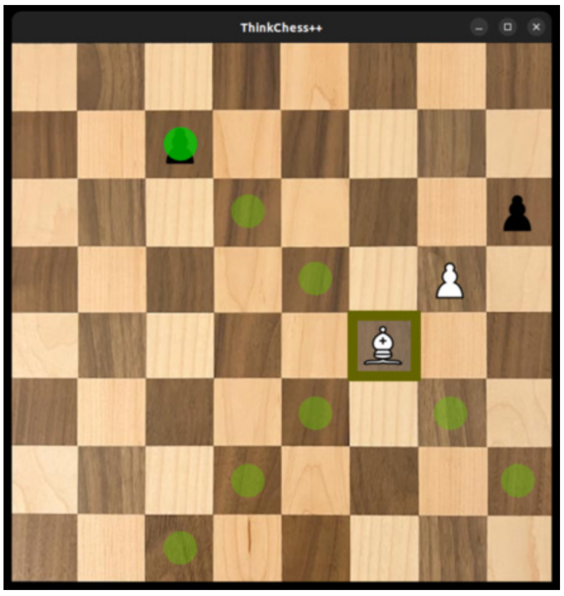
- the field is on the same row (3), or
- the field is on the same column (16).

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.

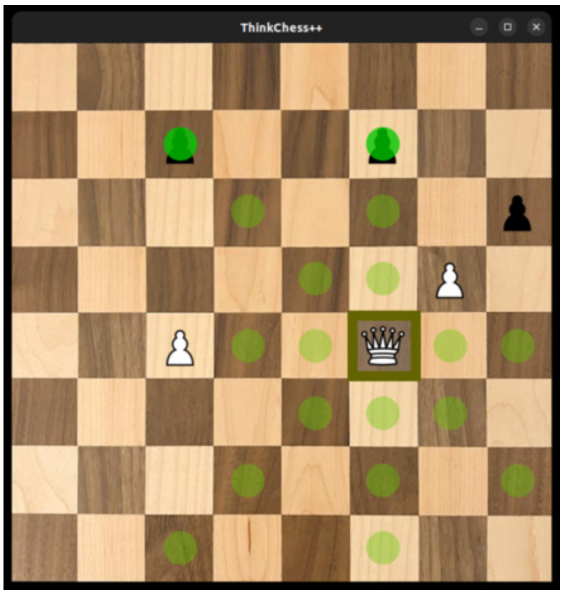
We use essentially the same logic as for the rook, but this time we check for pieces on the same diagonal:

```
bool Bishop::isValid(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:



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:

```
bool Queen::isValid(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.

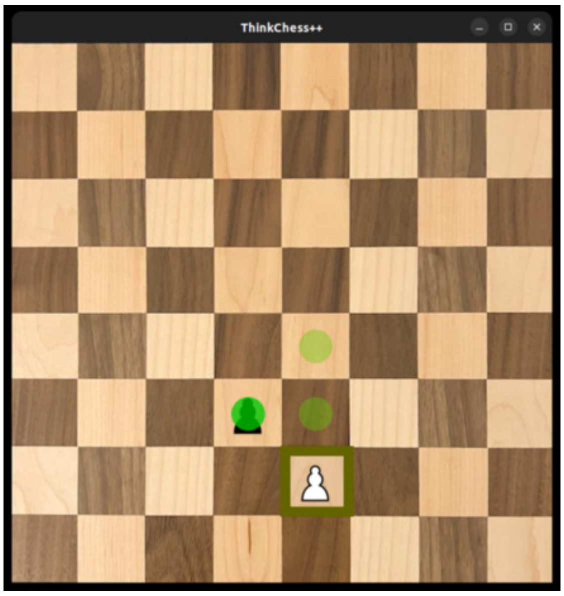
The Pawn

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(vector<vector<Piece*>>& bd, int r, int c) {
  bool valid = false;
  if (white) {
    if (r == row-1 && c == col) { // move
      auto pc = bd[r][c];
      if (!pc) valid = true;
    }
    if (row == 6 && r == 4 && c == col) { // initial move
      auto pc = bd[r][c];
      if (!pc) valid = true;
    }
    if (r == row-1 && (c == col-1 || c == col+1)) { // capture
      auto pc = bd[r][c];
      if (pc && pc->isWhite() != white) valid = true;
15      }
16    } else { // black
17      if (r == row+1 && c == col) { // move
18        auto pc = bd[r][c];
19        if (!pc) valid = true;
20      }
21      if (row == 1 && r == 3 && c == col) { // initial move
22        auto pc = bd[r][c];
23        if (!pc) valid = true;
24      }
25      if (r == row+1 && (c == col-1 || c == col+1)) { // capture
26        auto pc = bd[r][c];
27        if (pc && pc->isWhite() != white) valid = true;
28      }
29    }
30    return valid;
31  }
```

We have to consider white and black pawns separately, as they move in different directions (3, 16). If there's no piece in front of the pawn, it can move to that field (6, 19). If the pawn is on its initial position (8, 21), it also can move two squares ahead, if that field isn't occopied (10, 23). If there's a piece diagonal in front of the pawn (12, 25), it can be captured, if it is of the other color (14, 27).

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:

```
// matrix of pieces representing the board
vector<vector<Piece*>> board(8, vector<Piece*>(8));

// matrix of valid moves for display
vector<vector<short>> validMoves(8, vector<short>(8, 0));

// list of moves, used as a stack
auto* moves = new list::List<string>;

// list of captured pieces, used as a stack
auto* captured = new list::List<Piece*>;

// touched field for making moves
pair<int, int> touched{-1, -1};

```

```cpp
// player to turn, starting with white
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
void makeMove(vector<vector<Piece*>>& bd,
              list::List<string>* mv,
              list::List<Piece*>* cp,
              pair<int, int>& td,
              pair<int, int> to,
              bool& player)
{
  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;
  }
  if (pcf->isValid(bd, to.first, to.second)) {
    if (pct && pct->isWhite() != pcf->isWhite()) {
      cap = true;
      pct->capture();
      cp->push_front(pct);
    } else if (pct) {
      cout << "illegal move!\n";
      td = {-1, -1};
      return;
    }
    mv->push_front(convertFromBoard(cap, pcf, to));
    bd[to.first][to.second] = pcf;
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

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`:

```
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

```
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:

```
// 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.

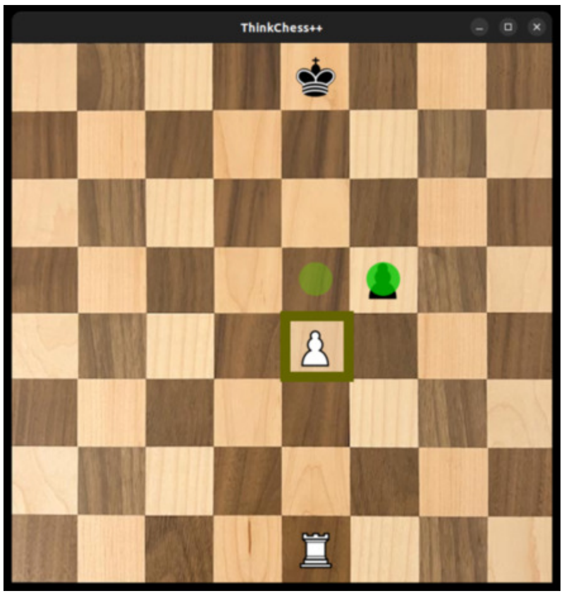
For that, we only need to iterate over all opponent pieces, and if any of them can reach the king with a valid move, we have a check:

```
bool check(vector<vector<Piece*>>& bd, bool white) {
  Piece* king;
  bool check = false;
  for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 8; col++) {
      auto current = bd[row][col];
      if (current && current->getType() == 'K' &&
          current->isWhite() == white)
      {
        king = current;
      }
    }
  }
  for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 8; col++) {
      auto current = bd[row][col];
      if (current && current->isWhite() != white) {
        if (current->isValid(bd, king->getRow(), king->getCol())) {
          check = true;
        }
      }
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:

```
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;
  }
  // 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;
    }
    // legal move
    string move = convertFromBoard(cap, pcf, to);
    bd[to.first][to.second] = pcf;
    bd[td.first][td.second] = nullptr;
    pcf->makeMove(to.first, to.second);
    // check?
    if (check(bd, player)) { // gives itself check
      cout << "illegal move!\n";
      bd[to.first][to.second] = pct;
      bd[td.first][td.second] = pcf;
      pcf->makeMove(td.first, td.second);
      td = {-1, -1};
      return;
    }
    if (check(bd, !player)) { // gives opponent check
      if (resolveCheck(bd, !player)) {
        move.append(1, '+');
      } else { // cannot get out of check
        move.append(1, '#');
        mv->push_front(move);
        cout << mv->peek(1) << "\n";
        checkmate = getKing(bd, !player);
        return;
      }
    }
    mv->push_front(move);
    cout << mv->peek(1) << "\n";
    td = {-1, -1};
    player = !player;
  // illegal move
  } else {
    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:

```
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
      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);
      }
    }
  }
```

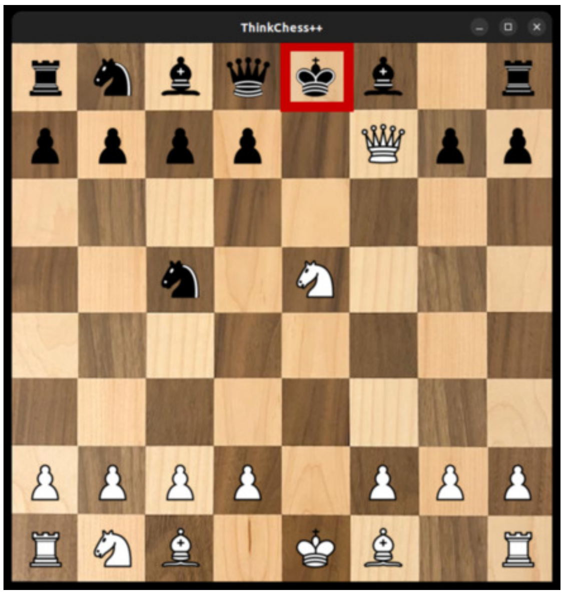
```
    // display frame
    window.display();

    // 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 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

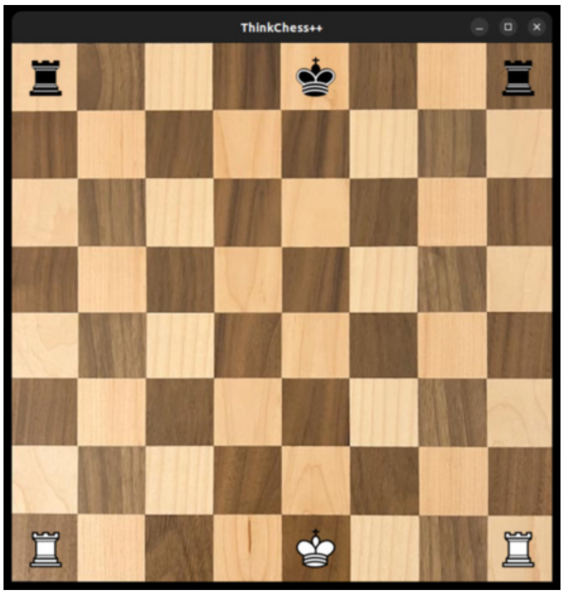
- the king: castling
- the pawns: en passant and promotion.

2.3.1 Castling

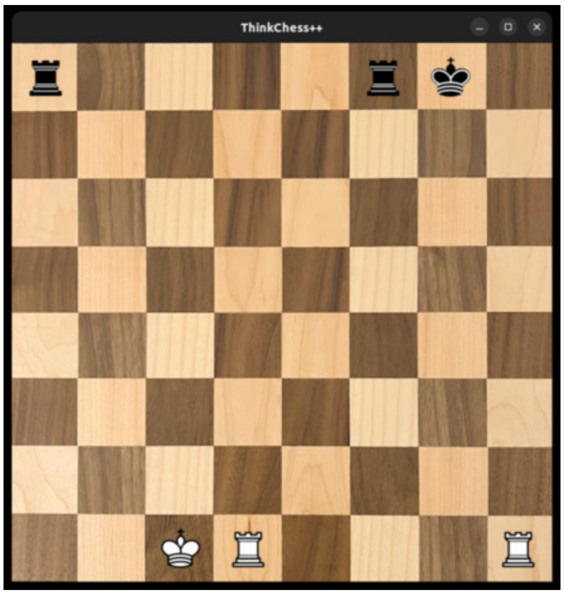
Castling is the only move, in which two pieces are moved at once (the king and one of the rooks of the same color). The king moves two squares towards a rook on the same rank and the rook moves to the square that the king passed over.

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`:

```
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;
      move = "O-O";
      // make king move
      bd[td.first][td.second+2] = pcf;
      pcf->makeMove(td.first, td.second+2);
      bd[td.first][td.second] = nullptr;
      // make rook move
      auto rook = bd[td.first][7];
      bd[td.first][td.second+1] = rook;
      rook->makeMove(td.first, td.second+1);
      bd[td.first][7] = nullptr;
      if (check(bd, !player)) { // gives opponent check
        if (resolveCheck(bd, !player)) {
          move.append(1, '+');
        } else { // cannot get out of check
          move.append(1, '#');
          mv->push_front(move);
          cout << mv->peek(1) << "\n";
          checkmate = getKing(bd, !player);
          return;
        }
      }
      // complete move
      mv->push_front(move);
      cout << mv->peek(1) << "\n";
      td = {-1, -1};
      player = !player;
      return;
    } else if (form == 'Q') {
      castled = castled > 0 ? 3 : cast;
      move = "O-O-O";
      // make king move
      bd[td.first][td.second-2] = pcf;
      pcf->makeMove(td.first, td.second-2);
      bd[td.first][td.second] = nullptr;
      // make rook move
      auto rook = bd[td.first][0];
      bd[td.first][td.second-1] = rook;
      rook->makeMove(td.first, td.second-1);
      bd[td.first][0] = nullptr;
      if (check(bd, !player)) { // gives opponent check
        if (resolveCheck(bd, !player)) {
          move.append(1, '+');
        } 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.

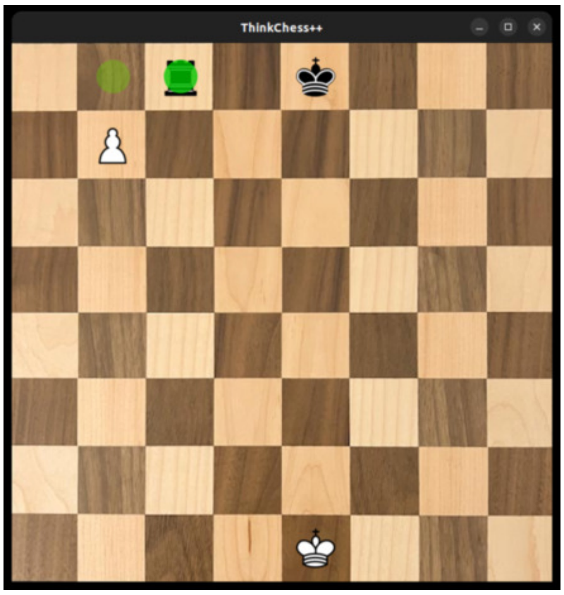
Promotion is almost always to a queen (a.k.a *queening*), as it is the most powerful piece. Since we don't have a sufficient GUI yet (so the player cannot choose a different piece), we will restrict promotion to queening for now.

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

3.1 Adding Analytics

3.2 Storing a Game