# SOFTWARE PROJECT

In the final project of the course, you will create an interactive chess program that applies graphical user interface (GUI) and artificial intelligence (AI) based opponent.

## Introduction

The program has two modes:

**Console mode** - The console mode will operate in a similar way to your implementation for connect4 game in assignment 3. You will develop the console mode so that it would be easier later to integrate your code with the GUI implementation.

**graphical mode (Gui mode)** - The graphical mode presents the user with visual menus and controls enabling the user to play Chess, choose the game's players (user vs. user or user vs. computer, and set the game's difficulty level in case the user is playing against the computer.

For AI, the Minimax algorithm will be used. The Minimax algorithm is mostly the same as the algorithm in ex. 3, except we use **pruning as shown in tutorial 3** to improve its efficiency.

## High-level description

The project consists of 5 parts:

- Console user interface for the Chess game
- Minimax algorithm for AI.
- Graphical User interface for the Chess program.
- (Bonus)
  - 4 pts. Chess player with improved scoring function and minimax depth selection which outperforms the trivial one (see later in the bonus section).
  - 2 pts. Highlighting all possible moves in GUI mode when the user clicks on the right click mouse (see later in the bonus section).
  - 2 pts. Allow pawn promotion
  - 2 pts. Allow castling move

The executable for the program will be named "*chessprog*". The graphical mode is specified as a command line argument:

./chessprog  -c – will start the program in console mode.

./ chessprog  -g – will start the program in GUI mode.

./ chessprog  – will execute the program; with the default execution mode - console

# Chess

A brief introduction on chess is given in this section. You can find further details in the following link: http://en.wikipedia.org/wiki/Chess.

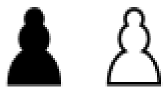
It is important to note that for simplicity, we will ignore some of the rules regarding the regular chess game.

## Rules:

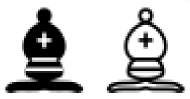
Each Player begins the game with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The game board is an 8X8 grid, in which each of the pieces are located in predetermined order (look online for more information). Each of the six piece types moves differently. Pieces are used to attack and capture the opponent's pieces. Capturing is performed by moving a piece to a position occupied by an opponent's piece. Besides the pawn, all pieces capture in the same direction they move.

A victory is achieved when a player captures (or kills) the opponent's king. Victory is declared before the actual move is executed. Each opponent has his own color (either black or white). As it is accepted worldwide, the white player plays first. Further notice that the player may not move any of its pieces to a state where the king is threatened in the next round.

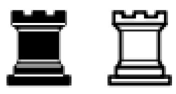
The pieces' types



- Pawn:
  - Movement: a single move forward in the same column. The user may move the pawn two steps forward only if the pawn is located at its starting position.
  - Capturing: one diagonal step forward
  - Promotion - A pawn that advances all the way to the opposite side of the board (the opposing player's first rank) is promoted to another piece of that player's choice, other than a king. This move is optional, you may choose to implement it in order to obtain extra credits (see bonus section).
  - Console representation: m/M (lower case for white, capital letter for black).
  - There is no need to implement *en passant* moves. This move will not be checked, so implementing it will **not** provide you with extra credit.

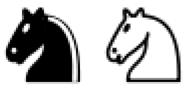


- Bishop
  - Movement: can move any number of squares diagonally (backward and forward), but may not leap over other pieces.
  - Capturing: diagonally in any direction
  - Console representation: b/B

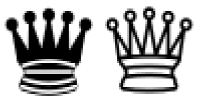


- Rook:
  - Movement: can move any number of squares along its line or column on the board, but may not leap over other pieces. (horizontal  moves)
  - Console representation: r/R

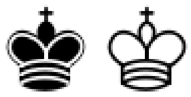
- Knight:
  - Movement: "L"-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The knight is the only piece that can leap over other pieces.
  - Console representation: n/N



- Queen:
  - Movement: combines the power of the rook and bishop and can move any number of squares along rank, file, or diagonal, but it may not leap over other pieces.
  - Console representation: q/Q



- King:
  - Movement: moves one square in any direction
  - Console representation: k/K
  - You are not required to implement the *castling* move (but can for extra credit, see bonus section)



## Check, Mate and Tie

### Check

- When player's king is threatened by the opponent, it's called "check". A king is threated, if there is an opponent's piece that can capture (kill) the king in the current settings of the gameboard. A response to a check is a legal move if it results in a position where the king is no longer under direct attack
- The player must move one of its pieces so that the king is not threatened anymore. Other moves are illegal.
- **Notice that the current player may not perform any movement that will result in its king being in check state.**

### Checkmate

- When the king is threatened by the opponent ("check") and it king cannot be saved (there are no legal moves)
- Mate terminates the game (In Chess, victory is achieved before the actual capturing of the king is executed)

### Tie

- When the player doesn't have any legal moves, but the king is not threatened by the opponent (no "check").
- A tie terminates the game.
- Note that in real chess there are more option to reach a tie, but you are not required to implement them.

# Implementation details

## Game setting

The game setting depends on the following elements:

- ***The game mode:*** the game has two operating modes, either 1 player or 2 players mode (the modes are represented by 1 or 2 respectively). In a '1-player' mode, the user plays against an AI opponent. In a '2-player' mode the game is played with two different opponents. ***The default value is 1.***
- ***The difficulty level of the game:*** If the game is operating in a '1-player' mode, then there are 4 difficulty levels; 1, 2, 3 and 4. These numbers represent the following levels: noob, easy, moderate and hard. To get bonus, you need to implement level 5 which represents an expert level. ***The default value: 2.***
- ***User color:*** in a '1-player' mode, the user may specify her color. The color can be either black or white and in console mode these colors are represented by 0 and 1 (0 represents black and 1 represents white). ***The default value: 1***

In both the command line mode (console mode) and in graphical user interface mode (GUI mode) the user may choose the game setting. The proper way to set the game for each mode is specified in later sections.

## Console mode

The program may be in two states during the execution in the console mode, either settings state or game state. Initially, the game starts with the settings state, where the user may change the setting of the game prior to the game itself.

### Settings state

Once the user is in the settings state, the following message is printed:

```
“Specify game setting or type 'start' to begin a game with the
current setting:\n"
```

The above message is printed once whenever the game enters the setting state. This can occur either at the beginning of the program in the console mode, or when the user explicitly enters the setting when the program is in the game state (see the *reset* command in the game state section).

Immediately after the message is printed, the program repeatedly prompts for the user's commands. Once the user enters the *start* command, the game switches to the game state with the specified settings.

The commands in the settings state are shown below:

1.
    - Command: *game_mode x*
    - Description: this command sets the game mode. x can be wither *1* or *2*:
        i. 1 – one player mode (a player vs. AI)
        ii. 2 – two players mode
    - After executing this command, the program prints the message: `"Game mode is set to XXX\n"`, where XXX can either be `"2 players"` or `"1 player"`
    - In case a wrong game mode is entered, the program prints `"Wrong game mode\n"` and the command is not executed
2.
    - command: *difficulty x*
    - Description: this command is only legal when the game is set to 1 player mode. Otherwise, you must treat it as any other invalid command. This command sets the difficulty level of the game. The value of *x* is can be 1,2,3 or 4. If you decide to implement the bonus section then you also need to support difficulty level 5. More details:
        i. If x is 1,2,3, 4 or 5 – This command sets the difficulty level to noob, easy moderate, hard and expert, respectively. In case $x \in \{1,2,3,4\}$, this defines the depth of the MinMax algorithm, that is, the maximum level we can reach while applying the MiniMax algorithm. Notice that if $x = 5$, then setting the depth of the MinMax algorithm to 5 will not help, that is because for depth 5 the MinMax algorithm is time consuming and non-applicable. Thus for $x = 5$, you need to think how to improve the MinMax in more creative ways.
        ii. If the expert level not supported, then you need to print `"Expert level not supported, please choose a value between 1 to 4:\n".`
        iii. If the user enters a value that is not in the range 1 to 5, then the message `"Wrong difficulty level. The value should be between 1 to 5\n"` will be printed, and the difficulty value will not be set.

3.
    - command: *user_color x*
    - description: This command is valid only if the game is set to a '1-player' mode. The user's color will be set accordingly (either black or white). The value of x is can be *0* or *1, where 0 and 1 symbolize the black and white colors*, respectively.
    - Notice that like the difficulty command, this command is only allowed in a '1-player' mode. For the '2-player' mode, this command should be treated as an illegal command.

4.

- command: load x
- description: this command loads the game setting from a file with the name "x", where x includes a full or relative path of the file.
- In case the file does not exist or cannot be opened, the programs prints `"Eror: File doesn't exist or cannot be opened\n"` and the command is not executed.
- You may assume that the file contains valid data and is correctly formatted. More on the input files format appears in **appendix B**.

5.

- Command: *default*
- Description: Resets the game setting to the default values as appeared in the game settings parameters.

6.

- Command: *print_setting*
- Description: Prints the current game settings to the console in the following format:
  If the game is in 1-player mode:
  "SETTINGS:\n
  `"GAME_MODE: 1\n"`
  Otherwise (in 2-players mode):
  "SETTINGS:\n
  `"GAME_MODE: 2\n"`
  `"DIFFICULTY_LVL: X\n"`
  `"USER_CLR: Y\n"`
  where X is the difficulty level (an integer between 1 and 5) and Y is the user color (either the string BLACK or WHITE).
  Notice that when this command is invoked, then you need to print the updated settings (some of the settings may be default).
  (ELI: WHAT DOES IT MEAN? MOAB I clarified things)

7.

- Command: *quit*
- Description: Terminates the program. All memory resources must be freed. Once the user invoke this command the program prints:
  `"Exiting...\n"`

8.

- Command: *start*
- Description: Starts with the current settings.

**Special Remarks:**

- If the user enters two commands that affect the same setting parameter then the latest update is taken:
  - For example: if the user first sets the difficulty level to 1 and afterwards she changes it 3 then the game difficulty level is set to 3.

- If a game setting parameter is not specified, then the default value is taken as appears in the game settings parameters section.

## Game state

Once the user enters all settings and invokes the start command, the game begins by switching to the game state.

The board is printed to the screen at the beginning of each user's turn in the following way: If the game is in a '2-player' mode then board is printed in each turn. Otherwise, i.e. the game in a '1-player' mode, then the game board is printed only at the beginning of the user's turn. Notice that this means that the board is printed at most once in each turn. The format we use to represent the game board appears in Appendix A.

After the board is printed, the game asks the relevant user to enter her move by printing the following message:
`"XXX player - enter your move:\n"`, where XXX stands for the current player's color - black or white.
This message will be printed each time a user has to make a move. In case an error occurs (for example the user enters invalid move) then the above message is printed again.

Notice that in a '1-player' mode, the message is printed only at the user's turn and in a '2-player' mode, the message is printed at each turn.

The following commands can be executed by the user at each turn.

1.

- Command: *move <x,y> to <i,j>*
- Description: This command executes the user turn by moving the piece at <x,y> to the <i,j> location. *x* and *I* represent the row number, which can be between 1-8. The values of *y* and *j* represents the column letter, which can be between A-H (upper case). For example:
  *move <2,A> to <3,A>*
  You need to handle the following errors:
  (1) If either one of the locations in the command is invalid, the program prints the following message `"Invalid position on the board\n"`.
  (2) If position <x,y> does not contain a piece of the user's color, the message `"The specified position does not contain your piece\n"` is printed.
  (3) If the move is illegal for the piece in the position <x,y>, the message `"Illegal move\n"` is printed.
- You must print only one error message for each command. In case more than one error occurs, you need to print the error with the lowest ID as specified above.
- If no errors occur, the board is updated, and the following messages may be printed, depending on the case:
  a. If any of the player's king is threatened, the message `"Check: XXX King is threatend!\n"` will be printed, where XXX is either black or white. This test should only be performed if there is no Checkmate.
  b. If there is a checkmate or a tie :
     - In case there is a "checkmate" - the message `"Checkmate! XXX player wins the game\n"` is printed, XXX stands for the winner's color.
     - In case there is a tie - the message `"The game is tied\n"` is printed.
     - In both cases (mate and tie) the program terminates.
  c. If you implement a pawn promotion, then you program should behave as it appears in the bonus section.
  - Notice that there still could be a check (or checkmate) due to pawn promotion, thus you still need to handle this case.
  - In case the game is still on, the next turn is played.

2.

- (Bonus) Command: *get_moves <x,y>*
- Description: This command is optional, if you decide to implement it then you need to refer to the bonus section for information on the printing format (see **all possible moves feature** in the bonus section).

3.

- Command: *save x*
- Description: the command save the current game state to the specified file, where x represent the file relative or full path.
- If the file cannot be created or modified, the program prints the following message
  `"File cannot be created or modified\n"`
  and the command is not executed.
- If no error occurs, then the current game setting and the board status is saved to the specified file. The syntax of the specified file appears in Appendix B.

4.

- Command: *undo*
- Description:Undo previous moves done by the user. This command is available only in a '1-player' mode. The user may undo up to 3 moves. Thus, you need to store the previous 3 moves for the user and the AI opponent. The game saves the last three moves done by the user while least recent moves will be forgotten. A move that is "forgotten" cannot be undone.
- If the user invokes this command in a '2=player' mode, the following message is printed `"Undo command not avaialbe in 2 players mode\n"` and the command is not executed.
- If the user is trying to invoke the *undo* command and the history is empty, the program prints `"Empty history, move cannot be undone\n"`
- In case the *undo* command can be executed, then the program will undo the previous two moves (one for each player) and prints the following messages:
  ```
  "Undo move for player XXX : <x,y> -> <w,z>\n"
  "Undo move for player YYY : <i,j> -> <m,n>\n"
  ```
  Where XXX , <x,y> and <w,z> represents the color of last player to play, the position of the piece after the last move executed and the original position of the piece. Similarly for the second message. The player's colors are representing by the strings "black" or "white" (without quotation).

5.

- command: *reset*
- description:The program state is switched to the setting state. The following message is printed after the execution of this command
  `"Restarting...\n"`

6.

- Command: *quit*
- Description: Terminates the game. Prior to terminating,the following message is printed: `"Exiting...\n"`.

See next page for the behavior of the program in the computer's turn.

Computer's turn

Once If the AI opponent performed a regular move (with no pawn-promotion or castling), the program prints the computes move in the following format:

`"Computer: move [pawn|bishop|knight|rook|queen] at <x,y> to <i,j>\n".`

- Each move is represented by 2 positions: the current position of the piece <x,y>, and the destination <i,j>.

If you allow a pawn promotion, then the computer may promote the pawn as well. See the bonus section for more information. Further if castling is allowed, then the computer may perform castling move as well. You need to refer to the printing formats for both cases in the bonus section.

Once the board is updated the following happens:

1. Check for checkmate or tie :
   a. If the user will not be able to make any move in the next turn then:
      - In the case of "mate": the message `"Checkmate! XXX player wins the game\n"` is printed, `where` XXX stands for the winner's color ("black" or "white" without quotation).
      - In the case of a tie: the message `"The game ends in a tie\n".`

      Notice that in both cases (checkmate and tie) the program terminates.

   b. If the AI is threatening the opponent's king, the message `"Check!\n"` will be printed.
2. In case the game is not over, the game moves to the next turn (the user's turn).

## Computer AI

The AI used in the project is the minimax algorithm from the 3rd assignment, the depth of the MiniMax (or the number of steps to look-ahead) is defined in the difficulty level. Thus in difficulty 3, the depth of the minimax tree is 3 as well.

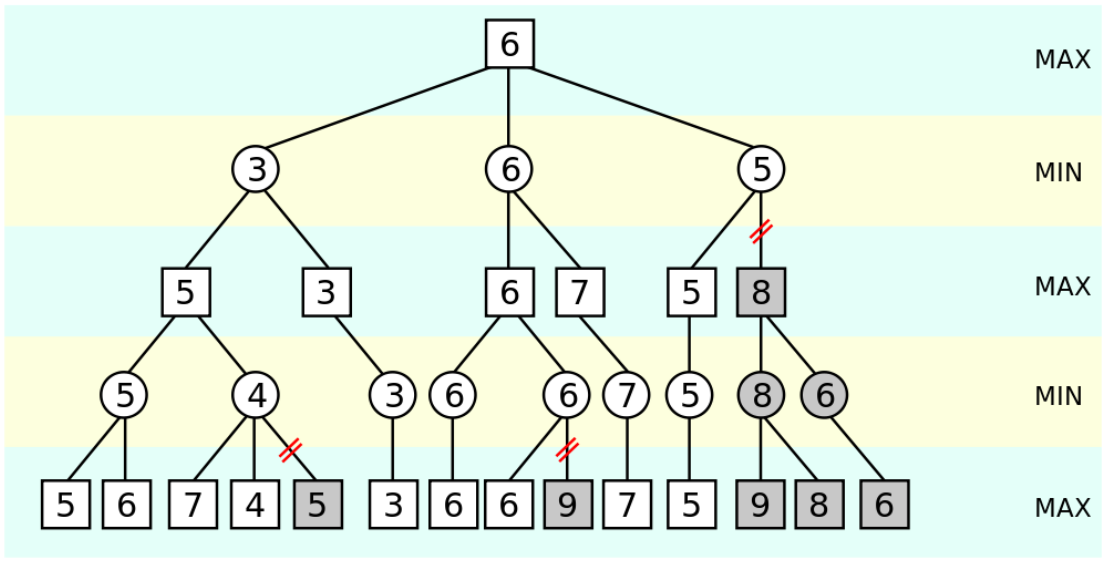
The minimax algorithm's basics stays the same – however, in the final project you are not required to construct the minimax tree in a different step, and can perform your calculations while creating the tree (notice you don't need to create an actual tree structure doing so will result in points deduction). In addition, we add **pruning** to the algorithm as described next.

The minimax algorithm scans all possible moves for each game – sometimes more than necessary.

For example, suppose a MAX node has a child node for which we finished calculating values, getting a value of 5. When we proceed to calculate the values for its next node, suppose we found a node with a value of 4 under it (a grandchild of our MAX node). The parent node of it (a MIN node, a child of our MAX node) will always choose the lowest possible value, and thus won't choose a value which is greater than that 4.

This means we can stop evaluating that branch of the tree – it will contain a value of at most 4, and thus smaller than the 5 we already found, and we can proceed to the next nodes for calculation.

The following image shows an example of pruning, where the minimax tree is scanned from left to right. You should understand why the grayed-out nodes need not be evaluated at all:



The pruning algorithm is left for you to understand and implement accordingly. A description of pruning along with pseudo-code can be found here: http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

## Scoring function

The naïve scoring function goes over the board and outputs a score according to the pieces on the board. The scores for each piece are given below:

pawn = 1 , knight = 3 , bishop = 3 ,  rook = 5, queen = 9, king=100

## SDL

Simple DirectMedia Layer (SDL) is a cross-platform, free and open-source multimedia library written in C that presents a simple interface to various platforms' graphics, sound, and input devices. It is widely used due to its simplicity. Over 700 games, 180 applications, and 120 demos have been posted on its website.

SDL is actually a wrapper around operating-system-specific functions. The main purpose of SDL is to provide a common framework for accessing these functions. For further functionality beyond this goal, many libraries have been created to work on top of SDL. However, we will use “pure” SDL – without any additional libraries.

The SDL version we’re using is **SDL 2.0.5**, a documentation can be found in the following link: https://wiki.libsdl.org/FrontPage, you can also use google for

Following are a few examples of common SDL usage. However, this isn’t a complete overview of SDL, but only a short introduction – it is up to you to learn how to use SDL from the documentation, and apply it to your project.

To compile a program using SDL we need to add the following flags to the GCC calls inside our makefile (including the apostrophes):

- For compilation we need to add ‘sdl-config-cflags’
- For linkage we need to add ‘sdl-config--libs’

In our code files we need to include $SDL.h$ and $SDL\_video.h$, which will provide us with all of the SDL functionality we’ll be using.

To use SDL we must initialize it, and we must make sure to **quit SDL** before exiting our program.

Quitting SDL is done with the call $SDL\_Quit()$. To initialize SDL we need to pass a parameter of flags for all subsystems of SDL we use. In our case the only subsystem is VIDEO, thus the call is $SDL\_Init(SDL\_INIT\_VIDEO)$.

A brief introduction to SDL was given in the tutorial, and you may find an example on moodle website.

**Note:** SDL calls can fail! Check the documentation and validate each call accordingly.

## User Interface

In this section we describe the graphical user interface. For further details refer to the slides on moodle. You may choose to design the interface as you wish, however it must follow the requirements which appear in the attached slides and in this document.

### Resolution

The resolution of the game window should be at least 800x600 (width x height). That is the game window width and height is at least 800 and 600, respectively. All other windows must have a resolution of at least 400x400.

### Main Window

When starting up, the program will display the **main window** to the user, which will present the user with the following options:

- **New Game**
  Start a new Chess game by showing
- **Load Game**
  Allows the user to load previously saved games
- **Quit**
  Frees all resources used by the program and exits.

The main window will also be displayed to the user whenever choosing to return to the main menu from within any game.

#### New game

After pressing the new game button, a new dialog will open to enable the user to set the following options:

1. Game mode: the user may specify to play in a '**1-Player'** or '**2-players'** mode.
2. Difficulty level: in a '1-player' mode, the user may specify the difficulty level of the game. The difficulties are: noob, easy, moderate, hard and expert (No need to show expert level if not supported). User color: in a '1-player' mode, the user may specify the color of its pieces.

In addition, add a $Back$ button that should take us back to the main menu.

After including all game setting parameters, a new game will be loaded.

### Load game

The user may load previously saved games by pressing the Load game in the menu button. This will open a dialog were previously saved games appears. The program presents the 5 most recently saved games. Notice that in the case that there aren’t 5 previously saved games, the program will display as much slots as needed.

In addition, add a $Back$ button that should go back to the window we came from (not necessarily the main menu).

## Game Window

We do not provide UI instructions for the game window – it's up to you.

However, the game window must provide the user with the following options:

- **Restart game**
  Restarts the game with the current game settings. The game initializes a standard chess board (all pieces are placed as in the standard chess game).
- **Save Game**
  Saves the game in slot 1, game previously saved in $slot\ i$ will be saved in $slot\ i + 1$. The game saved in $slot\ 5$ will be forgotten and the user cannot load it again.
- **Load Game**
  The user may load previously saved games using the load button. Pressing the load button will open the **load window** (further description later).
- **Undo**
  The user may undo previous modes if allowed. The history size for the undo button is the same as in the console mode, i.e at most 3 moves for each. The behavior of program when the undo button is pressed is the same as in the console mode.
- **Main Menu**
  Frees all resources used by the game and returns to the main window.
- **Quit**
  Frees all resources used by the program and the game, and exits.

Notes:

- If the user didn't save the current game when pressing **Main Menu** or **Quit**, then a message box will pop up asking the user if she wants to save the current game or not (you need to provide three buttons options, Yes, No and cancel). You may want to use the module SDL_ShowMessageBox.
- You need to indicate if there was a check/mate or a tie.

## Load Window

The requirements for the loading window appears below, for further information refer to the slide show on Moodle:

- You need to present to the user the most 5 recently saved games.
- The order in which the games appears should be sorted according to their save date such that the most recent game appears first.
- The selection must be visible at any time.
- The user may return back the previous window by pressing the 'back' button.
- The user loads the game only when she presses the 'load' button. If the user didn't select any game slot, then the load button is not active.

## Saving a game state to Files

Each game in the program can be saved at any point, and then later loaded to resume playing from that point. To save a game, the user selects the save button.

Games can be loaded from the main menu. To load a game, the user selects the relevant option from the main window, and a dialog is presented asking the user where to load the game from.

Since SDL doesn't directly provide us with the ability to output text conveniently, and to prevent further complications, the user won't be able to directly select a file to save or load.

The program will use **5** "slots" for saving and loading (use a constant in your code, don't assume it's always 5!).

Each slot will be associated with a pre-determined file, and the user always saves on the first slot. The file name is not visible to the user from the program, but the file itself should be easy to find (somewhere in the project's folder).

## Error Handling

Your code should handle all possible errors that may occur (memory allocation problems, safe programming, SDL errors, etc.).

Files' names can be either relative or absolute and can be invalid (the file/path might not exist or could not be opened – do **not** assume that since they're not provided by the user, they're valid – treat them as if they were user input).

Make sure you check the return values of all relevant SDL functions – most SDL functions can fail and should be handled accordingly (as usual - do not exit unless you must).

Throughout your program do not forget to check:

- The return value of a function. You may excuse yourself from checking the return value of any of the following I/O function ( $printf$, $fprints$, $sprint$, and $perror$)
- Any additional checks, to avoid any kind of segmentation faults, memory leaks, and misleading messages. Catch and handle properly any fault, even if it happened inside a library you are using.

Whenever there is an error, print to the console – which is still available even if you are running a program with a GUI. You should output a message starting with "ERROR:" and followed by an appropriate informative message. The program will disregard the fault command and continue to run.

Terminate the program only if no other course of action exists. In such a case free all allocated memory, quit SDL, and issue an appropriate message before terminating.

## Submission Guidelines

This assignment will be submitted in Moodle.

You should submit a zip file named ***id1_id2_finalproject.zip***, where id1 and id2 are the ids of both partners.

The zipped file will contain.

- All project related files (sources, headers, images).
- Your makefile
- Partners.txt file according to the pattern provided in the exercise zip file.

## Coding

Your code should be gracefully partitioned into files and functions. The design of the program, including interfaces, functions' declarations and partition into modules is entirely up to you, and is graded. You should aim for modularity, reuse of code, clarity, and logical partition.

In your implementation, also pay careful attention for use of constant values and use of memory. Do not forget to free any memory you allocated, and quit SDL.
Especially, you should aim to allocate only necessary memory and free objects (memory and files) as soon as possible.
Source files should be commented at critical points in the code and at function declarations.
In order to prevent lines from wrapping in your printouts – please avoid long code lines.

## Compilation

You must make your own make file, which compiles all relevant parts of your code and creates an executable file named 'chessprog'. Use the flags provided for in the SDL tutorial to link the SDL library properly. Your project should pass the compilation test with no errors or warnings, which will be performed by running **make all** command in a UNIX terminal.

## Bonus

- **Expert Difficulty (up to 4 pts):** Up to 4 points will be provided for extending the AI component for the Chess player with **<u>expert difficulty</u>** that would beat the trivial player in a set of pre-designed games.
    - The trivial player uses the trivial scoring function we defined in this exercise. Its difficulty, i.e the depth of the MiniMax, is defined according to the board and the maximum depth will not exceed 4. Your job is to implement an opponent that is hard to beat, this can be done in many aspects.
    - Our trivial scoring function takes into account only the number of pieces on the board. Thus you can improve the scoring function by considering other factors, such as, the number of threatened pieces, the type of pieces that are threatened (if a queen is threatened then it's not good right?). Use your imagination and knowledge in chess.
    - Improve pruning (ordering the moves) by choosing the order you explore the MinMax algorithm. This will allow you to explore the MiniMax tree with larger depth search.
    - For some game board statuses, you can look further than 4 moves ahead. Thus you need to come up with an optimal way to decide the depth of the Minimax for each board.
    - The run time of your algorithm should not exceed 10 seconds. You are also limited in memory, that is, your MiniMax tree should not exceed $10^6$ nodes.
    - Notice that if you choose to change the scoring function, this change should be applied <u>only to the maximum difficulty</u>. For every other difficulty, the scoring function should be the one that is defined in this document.

- **All possible moves feature (2 pts):** 2-points bonus will be provided for implementing all possible moves feature. This feature is only supported in 1 player mode and difficulties 1 and 2 (noob and easy), and it presents to the user all possible moves that can performed by a specified piece.
  - **In console mode:** You need to support the command *get_moves <x,y>*
    i. Description: this command prints all possible moves of the piece located at position <x,y>.
    ii. (1) if the position <x,y> in the command is invalid, the program prints the following message `"Invalid position on the board\n"`.
    iii. (2) if the position <x,y> does not contain a piece of the user's color, the message `"The specified position does not contain XXX player piece\n"` where XXX stands for the current player's color (white/black).
    i. If the position is correct and contains one of the player's pieces, each move will be printed in a new line in the format specified at Appendix C. A move is represented by the position of the piece and the move itself. If there are no possible moves for this piece, nothing is printed.
  - **In GUI MODE this feature** is called highlight moves feature. A description is given below (further details given in the GUI presentation):
    - When pointing to a user piece, the user may click on the right mouse button which will highlight all possible squares for the highlighted piece.
    - You should include three different highlight colors to indicate the following: a threatened square, that is, a square that is threatened by an opponent piece. A capture square, that is, a square that is occupied by an opponent piece. If a square is both a capture and a threatened square then you should highlight it as a threatened square.
- **Pawn Promotion (2 Pts):** If you decide to implement the pawn promotion move, then your program should behave as follow:
  - Console mode: when the user enters a ***move*** command such that a pawn reaches the 8th rank, that is, it reached row number 8 from its opponent side, then a pawn promotion occurs.
    In case a pawn promotion occurs then you need to print the following message `"Pawn promotion- please replace the pawn by queen, rook, knight, bishop or pawn:\n".` Once the message is printed, the user must specify the promotion of the pawn by entering the piece type (queen, knight, rook, bishop or pawn). If the user enters an invalid string then then the program `“Invalid Type\n”` and the prints previous message is printed again and the program waits for the user to enter a valid string. If the AI performed a pawn promotion move then the following message is printed:
    `"Computer: move pawn <x,y> to <i,j> and promote to [queen|rook|knight|bishop|pawn]\n"` For example:
    `Computer: move pawn <7,A> to <8,A> and promote to queen\n”`

- GUI mode: When the user moves a pawn to the highest rank, you need to display a message box, asking the user to promote the pawn. This can be done using SDL_ShowMessageBox (notice that you need to add 5 buttons, one for each piece). If the computer performed a pawn promotion, then you need to specify to the user what the promotion was.

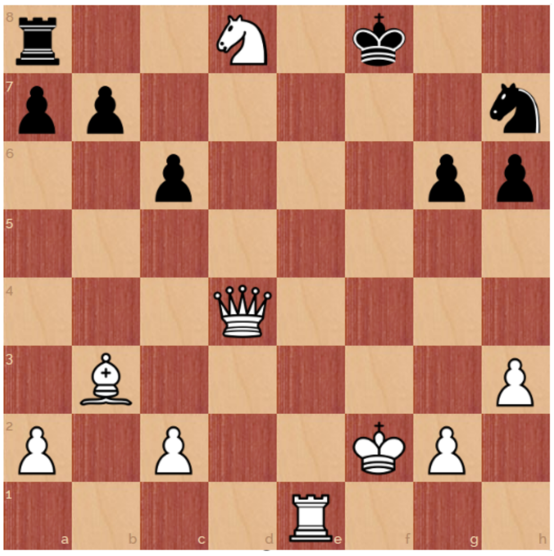
- **Castling Move (2pts):** The requirements for castling (see online for more information):
    - The king and the chosen rook are on the player's first rank (row).
    - Neither the king nor the chosen rook have previously moved.
    - There are no pieces between the king and the chosen rook.
    - The king is not currently in check.
    - The king does not pass through a square that is threaten by an enemy piece.
    - The king doesn't end up in check (true for any legal move).
    - Casteling consists of moving the king two squares towards a rook on the player's first rank, then moving the rook to the square over which the king crossed, which is the square on the other side of the king.
    - In console mode the following requirements must be implemented:
        - i. Command: *castle <x,y>*
        - ii. Description: <x,y> representing the rooks position. If <x,y> is illegal position on the game board, or <x,y> is not occupied buy a piece that belongs to the player, you must print the same error messages as in the *move* command.
        - iii. If <x,y> does not contain a rook, the computer should print the following message `"Wrong position for a rook\n"`
        - iv. Otherwise, if <x,y> is occupied buy a rook but casteling is not possible, the message `"Illegal castling move\n"` should be printed.
        - v. In the AI opponent turn, if the computer performed a castling move you need to print:
          `“Computer: castle King at <x,y> and Rook at <w,z>\n”`
          Where <x,y> is the position of the king and <w,z> is the rook's position.
        - vi. In get_moves command, the castle move should be printed last (if there are two possible castle moves then you first print the castle move that correspond to the rook at the left, i.e the rook in the lowest column number). The format for printing the castle move in this case is:
          `“castle <x,y>\n”`
          where <x,y> are the rook's position.
    - In GUI mode:
        - i. If you implement the highlight feature, then you need to choose a different color for the castling move.
        - ii. If you right click on the King and a castling move is possible then you need to highlight the rook with the new color of the castling move. If you right click on the rook, then the king needs to be highlighted.

# Appendix

## A Console mode - Game board format

The game board is printed as described in the following format:

1- The board is an 8x8 grid, in which rows are numbered from 1-8 and columns are numbered from A to H.
2- Black pieces are represented by upper case characters where white pieces are represented by lower case letters.
3- Each piece is represented by its first letter, except for the knight piece, which is represented by the letter **n or N** (for white and black pieces respectively).
4- Blank squares are represented by an underscore '_'.
5- Rows are numbered from 1 to 8 in upward order.
6- Columns are lettered from A to H in left to right order.
7- Pieces (and blank squares) are separated by one space.
8- Each row starts with its corresponding number followed by the '|' character
9- Each row ends with a '|' character.
10- Pieces (or blank square) lying on the boarder of a row are separated from the '|' character by one space.
11- After printing the first row a line is printed in the following format:
`"  -----------------\n".`
`Notice that the line starts with 2 spaces followed by 17 dashes.`
12- Last row starts with 3 spaces followed by one space separated characters A to H.



```
8| R _ _ n _ K _ _ |
7| P P _ _ _ _ _ N |
6| _ _ P _ _ _ P P |
5| _ _ _ _ _ _ _ _ |
4| _ _ _ q _ _ _ _ |
3| _ b _ _ _ _ _ p |
2| p _ p _ _ k p _ |
1| _ _ _ _ r _ _ _ |
  -----------------
   A B C D E F G H
```

## B Files Format

As explained in the instruction file, you need to be able to save the game status of the game. To do so, you will save the game status into a file with the format to be presented shortly.

Each file contains the following information:

- Current turn – which player should play (*black* or *white*)
- Game mode – 1 player mode (player vs AI) or 2 players mode.
- Users color and difficulty (relevant for player vs. AI mode)
- The board's state

The input/output files are in XML format. XML is stands for Extensible Markup Language (XML) - a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. Xml files can be easily viewed in most browsers, by typing the path in the address line. They can be easily manipulated using notepad++.

The first line in the file is the XML decleration and should be the same in each file you create. All the stored data is represented with markups and their content.
In the following example: <game_mode>2</ game_mode >, 2 is the content, and the markup contains 2 tags: opening: < game_mode > and closing </ game_mode >. In our XML files we will use the markups to represent the feature name, and the content will be the value. An XML file for the program has the following structure (the order of the tags should be the same as in the example)

```
<?xml version="1.0" encoding="UTF-8"?>

<game>

      <current_turn>current</current_turn>

      <game_mode>mode</game_ mode >

      <difficulty>difficulty</difficulty>

      <user_color>color</user_color>

      <board>

            <row_8>8 characters that represent the row's content</row_8>

            <row_7>8 characters that represent the row's content</row_7>

            <row_6>8 characters that represent the row's content</row_6>

            <row_5>8 characters that represent the row's content</row_5>

            <row_4>8 characters that represent the row's content</row_4>

            <row_3>8 characters that represent the row's content</row_3>

            <row_2>8 characters that represent the row's content</row_2>

            <row_1>8 characters that represent the row's content</row_1>

      </board>

</game>
```

You must replace all the values in boldface with their actual values. Notice that user_color and difficulty are optional.

The possible values are:

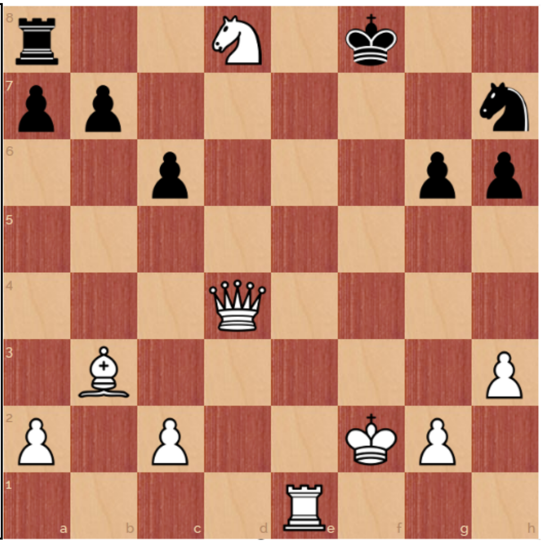
- mode: 1 or 2
- difficulty: the numbers 1 – 5, For example <difficulty>1</difficulty>. If you are trying to load a game with difficulty 5, and your game doesn't support this difficulty then the following error should be printed: `"Expert level not supported\n"`
- user_color/current_turn: is either 0 or 1 for black or white respectively.
- In "2 players" mode, the tags <difficulty> and <user_color> should not contain any value and can also be discarded from the file.

The representation of each row is the console representation of each piece in the row, except for the empty squares, which will be represented with "_".

For example

The following board is represented by:

```
<board>
        <row_8>R__n_K__</row_8>
        <row_7>MM_____N</row_7>
        <row_6>__M___MM</row_6>
        <row_5>________</row_5>
        <row_4>___q____</row_4>
        <row_3>_b_____m</row_3>
        <row_2>m_m__km_</row_2>
        <row_1>____r___</row_1>
</board>
```



An example of an input file is attached to the project.

This picture is taken from: http://www.ideachess.com/chess_tactics_puzzles/checkmate_n/

Additional information saved in XML:

- All additional information you choose to add to the XML should be saved under <game><general> (<general> is a new label under <game>). You can add as many labels as you need inside <general>.
- <general> tag should appear after the <board> tag

## C Get moves command

As appeared in the bonus section, if you support the *get_moves* command then you need to present all legal moves of a certain piece in the board. Each possible move should be printed in a separate line in the following format:

1- Each possible move is specified by the appropriate location on the board. The format for presenting the possible move is the following: <*x,y*> where x is number from 1-8 and y is a letter from A-H representing the appropriate square on the board.

2- Your result should be sorted in ascending order, where $< x, y >$ appears before $< z, w >$ if $x < z$ or $x = z$ and $y < w$.

In difficulties 1 and 2 (noob and easy), we also specify if a move is threatened by the opponent and if a move captures an opponent's piece.

1. If the move is threatened by the opponent, then a star will be printed after the move location. For example, if the move <7,a> is threatened by the opponent then we will print "<7,a>*"
2. If the move captures an opponent's piece then we will print "<x,y>^"
3. If the move satisfies both 1 and 2 then then we will print: "<x,y>*^"

**Note:** If you implement the castling move then you need to print $castle < x, y >$ to indicate if a castling move is possible. Refer to the castling move in the bonus section for more information.