Chess Game

Group: Checkmate

Members:
Marko Illievski
Lukas Grasse
Brandon Robertson
Zack Shortt
Michael Wilson

Fall 2014

CheckMate Chess Game

The Project

The software that we have developed is a functioning chess game. What has been implemented is a text based board that has a real time cursor that can move the pieces. Gameplay occurs on a single computer. There is also a tournament option that is included and a leader board. Players can create a new user, load existing users and be ranked against other players within the database.

Features

The list of features implemented is as follows:

- Chess board and functioning pieces with text-based interface
- Game Saving and Loading
- Loading and Saving Users to Database
- Replay - Replay a game loaded from databases
- Different User types including:
    - Guest(who can play games but do not have stats saved)
    - Players(who can play games and have stats saved)
- Leaderboard, compares all the user scores in the database to each other
- Tournaments - single elimination style.

In the final project, there were various feasible features that ended up being moved to wish list and not implemented, as well as one wish list item being implemented. Prebuilt chess puzzles were not implemented due to time. Different game modes, such as checkers, were not implemented, also due to time constraints. However, replay mode from the wish list was implemented. We underestimated some of the complexity in implementing rules and undo.

Object-oriented design

There is a large difference between the original design and the final product. A major component of the code that was changed was the use of a GUI. A GUI is no longer being implemented and instead a text based display on the command line is used. some of the things that have been changed include the implementation of timekeeping. Time is no longer a feature and as such there is code that is no longer being implemented. Another feature that is no longer being used is the use of any mouse events. The reason why no mouse events are being used is because since we switched the game to text base we do not have any way of using the mouse. We used keys to move a cursor around and select a location. We created a new class called “status” that contained two registered players and a board so we could pass it around and use it in all the menus, this made it easy to login players and modify the board anywhere (ex. loading a board). status contained setter and getter functions. We made a new class called “chessGameGUI” which displays the chess Board that you play on.

For Players it was decided that there would be a base class called Player. Player now handles the memento functionality. RegisteredPlayer no longer inherited from Player but

uses the player class. Player is polymorphic. Now the RegisteredPlayer class defaults to a guest. AIPlayer and ReplayPlayer are no longer being used and the functionality of those two classes have been moved elsewhere.

For the rules we have decided to leave out rules such as Not Enough Pieces, 3 consecutive moves, as well as the rules associated with checkmate. Not Enough Pieces refers to a check that would have been completed to check if you have enough pieces to checkmate your opponent. If you did not have the required pieces the game would result in a StaleMate. We did not implement this due to the time constraints that we were faced during the end of this project. Consecutive moves should have checked if the last 3 moves where the same (i.e. White moves King a4-a5, Black Moves c1-c5, White moves King a5-a4, Black Moves c5-c1, White moves King a4-a5, Black Moves c1-c). This was not implemented because during our implementation we did not give this functionality to the rules classes. We also did not implement the rules associated with the program determining when the player is in checkmate. Instead we leave it up to the player capture the opponent's king.

Coding Conventions

The coding conventions that were used include the following:

User defined names

- Constants and enumerated type members are all upper case with the words separated by underscores.
- Exceptions have descriptive names all lower case words separated by underscores.
- Class names are lowercase except for the first letter of the class name and the first letter of each interior word in the name with are upper case.
- All other names are lowercase with interior words that begin with an uppercase letter.
- All pointer variable names end Ptr.

Function names:

- mutator and modifier functions will be named using verbs
- (except for previous and next functions which are actually adjectives)
- accessor functions will be named using nouns
- Private member names are descriptive
- Subclasses names will end with the name of the class from which they are derived.

Open curly braces for functions, loops etc begin on first line after the end of the name/control line, and have the line to themselves.

All loops and branching statements use {} even if they are not necessary

Error Handling

Custom error classes that inherit from runtime_error are stored in a central file called Errors.h. The two main error classes are database_load_error and tournament_error. These errors are very important for reading and writing files. If a file is missing and cannot be opened, database will throw an error which can be caught. For example, the function getTopPlayers in database, catches the database_load_error, and skips to the next available player to load. Tournament uses error handling to throw errors if the tournament object is initialized incorrectly, or setMatchWinner tries to set a winner with an incorrect name.

Default system errors, such as when the system memory is full, were not caught. Given more time, letting a user save their game before a crash would be something to implement in the future.

Test Strategies

Our testing strategy involved using CPPUNIT to unit test the tournament class. These unit tests were then set by cron to run at 1:30 am every morning. The results are stored in a file. Beyond this, we also made use of extensive manual testing as follows:

- Constant user testing among all the members of the project was a large part of testing, it was strongly discouraged to submit a broken piece of coding to the repository
- Pieces of code would be tested to the point of “working”, then committed where other members would end up badgering the code as well.
- The interdependence between some classes forced members to have to test other’s code in order to use their own.
- Giving running versions of the game to persons outside of the group and having them attempt to play and respond with feedback. This helped with giving unbiased input from an outside source, showing potential oversights by the members

Debugging and Optimization Issues

- GDB was a strong tool for debugging, however even with its implementation there were cases where a second person’s perspective was necessary for solving an issue.
- Valgrind was also used when looking for memory leaks, however as stated further on; there was still some leaks that could not be solved even with its help.
- Creating a board that was useful for testing any specific rule was a challenge

Lessons

While implementing this project, the group learned many lessons, such as:

- a major issue is teamwork and members progressively building code together, while the project was divided up, portions of it still required other members code to properly implement. This caused some waiting for peoples codes to be fully created and last minute changes
- Communication is also important, two pieces of the project that rely on each other at times were developed with different intentions resulting in code that needed to be changed when put together, an example of this being the development between Tournament and Tournament Menu
- A more thorough insight into the the exact functionality of every class would have been beneficial, at times there were functions between classes that became redundant and resulted in wasted work.
- Furthermore, there were oversights on certain input/output requirements on some classes that required members having to request functions be made by others.
- Another common issue was at times that members would not fully test before committing to the svn and then this resulted in a broken program for the other team members to try and use or wait for to be fixed.
- The original unit test and scripts did not compile. This was originally thought to be a problem with the class being tested, but ended up being a problem in the makefile of the tests itself. Better testing of the unit tests themselves would have revealed this earlier.

Known Defects

There are known defects in the final product that were not corrected due to time constraints and/or minor impact on the project.

- The single player changed from playing against a AI to playing against your self. This is due to the AI being a wishlist item and not being implemented
- Using a name too long in the Tournament menu will cause the display to become unaligned
- Tournament will has no functionality besides displaying a list of names and saving/creating that list, with more time a version which acts on those names would have been created
- Memory leaks based off of the status class; upon deleting the pointers and re-allocating them causes a segmentation fault.
- En Passant rule not implemented
- 3 moves consecutive rule not implemented

- Backspace in the tournament player name input is not implemented, instead results in an invalid character
- In some instances check is not shown when the king is in a check position
- Replay was not made functional for a game that has ended, this is due to the user not having the option to save the game from the End Game menu, thus not able to load that game for Replay
- Doing an undo on a Pawn promoted to Queen does not revert the Queen to a pawn
- You can load a game and it doesn't necessarily load in the players that were using the game, instead just uses the current players
- Caps-Lock Disables the ability to move through menu’s
- Once castling happens it screws up the replay if game is saved.