Plan of Attack

Group: Harsh Patel (h334pate), Jaimil Dalwadi (jdalwadi), Mayank Mehra (m4mehra)

Project Chosen: Chess

Project Breakdown:

Chess has been all of our favourite pastimes and so we wanted to take this chance to build it with what we have learned in the course. Having experience playing this game, we hope to apply the concepts taught in the course to the level that demonstrates our understanding the best as well.

Possible Extra Features:

- **Go into the future:** Allows a user to select x number of moves for which they can try their strategies and then reverts them back to the last spot if they wish to. This mimics going into the future to see what might happen and then coming back.
- **Print Move History:** Prints the past moves made by the user and the computer.
    - Animated move history: Dynamically shows the game from the past x moves being rendered one by one.
- **Timer:** Many players play timer-based chess games and it has become a popular mode for competitive tournaments. We can introduce a timer that is constantly displayed while the player is making their move. If the timer ticks down to 0 for any of the players before the game has ended it will end the game and reward the other player with a win.
- **Dark Mode** (dynamic render based on where pieces are): The user can only view the board for which they have pieces nearby (like fighting an enemy in the dark). The rest of the board appears hidden (X for text-based and something similar for graphic display). This mode will only apply to the Player vs Computer games.

Task Deadline Breakdown:

We will attempt to complete the project in 3 phases. In the first phase (done by November 27), we plan to get a basic version of the chess game running (with input handlers, levels 1 and 2 for the computer engine, basic text-based output, and all of the other basic required parts of the program). We will allocate one member to solely focus on the computer levels for this time period because they will take the majority of the time, while the other two (Harsh and Jaimil) focus on getting the rest of the classes set up and the game running in a basic format. In the second phase (done by December 2), we will get everything needed to submit a working project for most of the marks (including a fully compiled program, level 3 and 4 computer engine, scoreboard, graphic display, and the final report + UML). We have also allocated some time for phase 3 (done by December 6) for working on additional features. This will also act as a backlog for any work that hasn't been done in the first two phases. Keeping these goals and timelines will give us enough time to spare and hopefully get the whole project completed on time. Below is a list-based version of the schedule, with everyone's tasks and deadlines.

Detailed Calendar breakdown: Format → `Task @person @deadline/duedate`

- Phase 1 (November 24-27): Basic Setup
  - [x] ~~Set up Github repo + workflow @Jaimil Dalwadi @November 25, 2022~~
  - [ ] Command interpreter (input handling) basic set up @Jaimil Dalwadi @November 26, 2022
  - [ ] Set up pieces (abstract and subclasses) @Harsh Patel @November 26, 2022
  - [ ] Display engine (text) @Jaimil Dalwadi @November 26, 2022
  - [ ] Create move class @Mayank Mehra @November 26, 2022
  - [ ] Create Chessboard superclass and tile class @Harsh Patel @November 26, 2022
  - [ ] Create computer engine (levels 1 and 2) @Mayank Mehra @November 27, 2022
- Phase 2 (November 29 - December 2): Testing and Submission Prep
  - [ ] Makefile @Harsh Patel @November 29, 2022
  - [ ] Test simple moves @Harsh Patel @November 29, 2022
  - [ ] Scoreboard and Endgame (winning/losing/etc) @Jaimil Dalwadi @November 29, 2022
  - [ ] Move types (castle, enpassant, etc) @Mayank Mehra @November 29, 2022
  - [ ] Create computer engine
    - [ ] Level 3 @Mayank Mehra @November 29, 2022
    - [ ] Computer Engine Level 4 @Jaimil Dalwadi @Harsh Patel @December 1, 2022
  - [ ] Display engine (graphical) @Harsh Patel @December 1, 2022
    - [ ] images and visual
  - [ ] Testing and Final touches @everyone @Next Friday
  - [ ] dd2 design.pdf and uml.pdf @everyone @December 2, 2022
- Phase 3 (Dec 3-6): Additional Features
  - [ ] Work on additional features (if time permits)
    - [ ] go into the future @Jaimil Dalwadi @December 5, 2022
    - [ ] print move history @Harsh Patel @December 5, 2022
    - [ ] Timer @Mayank Mehra @December 5, 2022
    - [ ] Look at parts of the board @Jaimil Dalwadi @December 6, 2022

Questions:

1) **Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.**

The book of standard opening move sequences can be implemented by using the map standard template library in C++ which is a list of key-value pairs. Each element would contain a key, which in this case stores the current state of the board. The corresponding value for each of these keys would be an array of possible responding moves associated with the current board state. Note that the keys are unique, hence the map cannot contain the same board state (current state of the game) multiple times. However, there can be the same opening moves (values) associated with different board states stored in each of their possible move arrays. When it is a turn, the program can search through this book of moves by using the current board as a key on the map and retrieve a set of viable next moves. Then either the user can make the decision on what move to play from the associated responding move list or one move can randomly be chosen. This method leverages hardcoding in the most famous set of possible opening moves that are played by GM's or most players. As such the method is not the most efficient way of implementing a database of opening sequences and would only be able to function properly for the first few moves. Afterwards, the program would be significantly slower as the time required to search through the map would increase as the number of possible responses increase exponentially. Furthermore creating a map that stores all possible current board states based on turns would also become extremely difficult as turns increase.

2) **How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?**

In our UML, we have a class called Move. We can keep a pointer to the last move that was made in the chessboard superclass. We can create a reverse_move method that basically takes a Move and reverses it or undos it by reversing the pointers for where the piece was supposed to go and come from. If a piece was captured, it would be returned. If no pieces were captured, an empty square would be kept. The reverse_method would be called on this last move pointer.

However, a better implementation would be to create a vector (standard library template) of moves. By keeping track of all the moves as the user plays the game, we can keep a history. We can simply apply reverse_method on each Move by iterating over the vector of moves. This would remove any possibility of a leak as we would use vectors instead of specific pointers.

Another way can be to implement a simple linked list which only goes backwards and each element holds a pointer to the previous node (back field) and a Move (value field). One can only access the back, so the reverse_method could be called on the back and the back can be popped iteratively as many times as requested.

Keeping a vector would be more efficient and perhaps also an index of where the game is currently at. This could allow us to implement both redos and undos. For example, if a person undoes 3 times, the vector would remain the same, but the reverse_method would be called on the last 3 moves to alter the state of the chessboard, and the current index of the vector would be reduced by 3. Then if the player wants to redo, they can simply call the move at the current index + 1.

3) **Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.**

For this version of chess to be implemented, we would require more space on the chess board and instead of having 64 tiles, it would have 96 tiles as seen below. We would have to add two more player instances/classes, two more colours, and additional logic to check the validity of moves. This is due to the addition of two more players and changes to the size of the board (since there are 4 corners (3x3 squares) cut off on the whole board). Furthermore, the input handler (command interpreter) would have to account for the two new players when starting a new game with the “game” command and setting up the game with “setup”. When implementing the chess board, we can expand the 2D vector from 8x8 to 14x14, but set pointers to corner tiles to be “nullptr” and add logic to ensure that they are not accessed. There would also have to be a mode where the players can play in a team (2 v 2), in addition to the option of playing individually. The final checks for checkmate and end-game results would also have to be altered to account for the two new players. The image shown below is sources from wikipedia.