INTRODUCTION

This project involved implementing a Chess game in C++, providing both textual and graphical representations. The game features a standard 8x8 chessboard and follows the traditional rules of chess, including piece movements, checks, checkmates, and stalemates. The major components of the game include the Board, various Piece subclasses (such as King, Queen, Bishop, etc.), and displays (Text and Graphical). The Board is responsible for managing the state of the game, while the Piece classes define the behavior of each type of chess piece. The game supports both human and computer players, with different levels of computer difficulty.

In summary, we adhered closely to our original UML design with some adjustments: for example, we refactored the Piece subclasses to inherit from a base Piece class, ensuring better code reuse and maintainability.

OVERVIEW

Although the project did not employ any specific design patterns but focused on a clear and straightforward object-oriented design; demonstrating the application of course concepts in a practical setting, showcasing a well-structured, modular, and maintainable implementation of a Chess game in C++. Additionally, the separation of concerns is clear, with the Board managing game state, Pieces handling movement logic, and display classes providing visual output

Main.cc is the entry point to the program, and it controls which functions to call for particular command line arguments: game <white-player> <black-player> - Start a new game, move <from> <to> [promotion] - Make a move, resign - Resign from the game, setup - Enter setup mode, + <piece> <position> Add a piece in setup mode, - <position> - Remove a piece in setup mode, = <color> - Set the color of the player to go next in setup mode; done - Finish setup mode, quit - Quit the game. If no command line arguments are given, a default Chess Game is loaded from a file and if such arguments are given, the gameboard is initialized with those.

The following classes are used: Board (8x8 chessboard), Piece (abstract base class for all chess pieces), TextDisplay (printing to terminal and automatically updating Board), Graphical Display (provides a graphical visualization of the board) and Computer (represents an AI player).

The **Board class** is central to the game, handling the initialization of the board, managing the state of each square, replacing a piece, moving a piece and determining the validity of moves. It keeps track of the move history , undo and the last moved piece to handle special moves like enpassant. The Board class also includes methods for checking game conditions such as determining if the path between two pieces are clear (isPathClear), check, checkmate, and stalemate. This class demonstrates several key concepts from the course, including classes and encapsulation, inheritance and polymorphism, and design quality.

The **Piece class** is an abstract base class for all chess pieces. Each piece type (King, Queen, Bishop, Knight, Rook, Pawn) inherits from this class and overrides the `move` method to implement its specific movement rules allowing us to achieve a high degree of flexibility and extensibility. Furthermore, the class includes attributes for the piece's color and type and a reference to the board it belongs to. This approach ensures that the logic for each piece's movement is encapsulated within the respective subclass, adhering to the principle of separation of concerns.

The **TextDisplay class** provides a simple text-based visualization of the board. It updates the display whenever the board state changes, ensuring that the text representation is always current. Additionally, it keeps track of the score, enables the call resign command and checks the board for checkmate and statements.

The **Graphical Display class** uses the Xwindow library to provide a graphical visualization of the board. It updates the graphical display whenever the board state changes, providing a more visually appealing representation of the game.

The **Computer class** represents an AI player. Depending on the difficulty level, it uses different strategies to make moves, ranging from random move selection to more sophisticated algorithms like minimax. The Computer class interacts with the Board to generate possible moves and evaluate the best move based on the current game state. This class demonstrates algorithms and template functions, exception safety, and polymorphism.

DESIGN

The Chess game's design is structured around several key components, namely `Board`, `Piece`, `Player`, `Move`, and `TextDisplay`. Each component plays a crucial role in maintaining the modularity and extensibility of the codebase. The design principles applied here ensure a clear separation of concerns, making the code maintainable and easy to understand.

The Board class is the central component of the chess game, representing the 8x8 grid on which the game is played. It is responsible for initializing the board with pieces in their starting positions, managing the state of the game, and executing moves. Proper initialization and memory management were crucial given the dynamic nature of the game board. The `Board` class's `initialize` and `clear` methods ensure the board is correctly set up at the start of a game and properly cleaned up when needed. This careful management prevents memory leaks by ensuring all dynamically allocated pieces are appropriately deleted when no longer needed.

Moreover, to handle the complexity of piece movement and ensure moves are valid, several methods were developed and implemented in the board class. The `valid` method checks positional bounds, while the `isValidMove` method assesses whether a move adheres to the rules of chess. The `move` method integrates these checks, updating the board state and managing special cases like pawn promotion. The implementation of `isPathClear` ensures that pieces cannot move through other pieces, a critical rule for non-knight pieces. The board's `check` method determines if a king is in check by verifying if any opposing pieces can attack the king's position. Building on this, the `checkmate` and `stalemate` methods determine if the game has reached an end state - `checkmate` checks if the king is in check and if no legal moves can alleviate this, while `stalemate` checks for a lack of legal moves without the king being in check. Furthermore, managing move history and enabling the undo functionality required careful tracking of moves. The board's `undo` method reverses the last move by restoring the previous board state, accounting for standard moves and promotions. This functionality is essential for supporting features like player takebacks and debugging.

The Piece class serves as an abstract base class for all chess pieces. It provides common attributes and methods, such as position, color, and possible moves, which are shared by all piece types. The “Piece” class includes methods like “getColor()”, which returns the piece's color, and “getPosition()”, which returns the current position of the piece. Specific piece types, such as `King`, `Queen`, `Rook`, `Bishop`, `Knight`, and `Pawn`, inherit from the `Piece` class and override the move method to implement their unique movement rules. The use of virtual methods and inheritance in the “Piece” class is another significant design choice. The `Piece` class defines common behaviors for all pieces, such as movement rules, through virtual methods. Subclasses of `Piece`, such as `King`, `Queen`, `Rook`, `Bishop`, `Knight`, and `Pawn`, override these virtual methods to implement specific behaviors. This design promotes extensibility, allowing new piece types to be added by simply subclassing `Piece` and overriding the necessary methods. It also promotes encapsulation, where specific behaviors are encapsulated within subclasses, reducing code duplication and increasing maintainability.

Creating a graphical representation of the chessboard posed unique challenges. The `Graphical_Display` class utilizes an `Xwindow` object to render the board and pieces. The `draw_board` and `render_square` methods handle the rendering, updating the display after each move. This visual feedback enhances the user experience by clearly depicting the game state. The TextDisplay class is responsible for the visual representation of the board. It observes changes in the board state and updates the display accordingly. The `TextDisplay` class includes a method called render() which updates the display to reflect the current state of the board.

This design provides a comprehensive implementation of a chess game, with detailed handling of board state, piece movement, special conditions, computer player logic, and graphical display. Each component interacts with the others to provide a complete and functional chess experience.

RESILIENCE TO CHANGE

The design of the chess game project is crafted to accommodate various changes to the program specifications while ensuring flexibility and maintainability. This adaptability is primarily achieved through modular design and the use of abstraction.

The modular design of the system, with distinct classes such as `Board`, `Piece`, `Computer`, `Graphical_Display`, and `Text_Display`, allows for targeted changes. For instance, the `Piece` class employs inheritance and polymorphism to manage different piece types and their behaviors. This means that adding new pieces or changing existing ones can be handled by modifying or extending the `Piece` hierarchy without disrupting other components.

The clear separation between the core game logic and user interface components is a key factor in supporting change. The `Text_Display` and `Graphical_Display` classes serve as separate modules for user interaction. This means that alterations to the way the game is presented, such as updating graphics or changing text output formats, can be implemented independently of the core game logic. This separation allows for easy updates to the user interface or the addition of new display features without impacting the underlying game mechanics.

However, the undo functionality poses a challenge in terms of adaptability. The current implementation of the undo feature is tightly coupled with the specific structure and rules of the game. This means that changes to the game's rules or the introduction of new features may require significant rework of the undo mechanism. The `undo` method and its related tracking of move history are designed to revert the game to previous states, but they are not inherently resilient to changes in the game's state management or rule set. This could necessitate substantial adjustments to ensure that the undo functionality continues to operate correctly in light of new game rules or modifications.

In summary, while the overall design supports changes to various program specifications through modularity and clear abstraction, the undo functionality is less adaptable and may require focused attention and reengineering when significant changes are introduced to the game's rules or state management.

ANSWERS TO PROJECT SPECIFICATION QUESTIONS
(MOSTLY UNCHANGED FROM DD1, Q2 CHANGED)

QUESTION: *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. See for example https://www.chess. com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.*

Integrating a book of standard openings into the chess program involves creating a robust and efficient system for storing and accessing well-known chess openings. Without relying on databases, complex data structures like hash maps or trees, we can leverage the use of classes, iterators, and decorators to complete this task. To be specific, this method involves creating a well-structured class hierarchy to represent the opening moves and utilizing iterators to traverse and match these moves.

First, we need to create a foundational class to represent a single move. This `Move` class will encapsulate the move's notation and any additional sub data that might be useful, such as a description or move name. This class provides a clear and simple, modular way to handle individual moves, making it easy to manage and extend if needed. Next, we can create an `Opening` class to represent a sequence of opening moves. This class will contain a list (maybe through the use of a `std::vector`) of Move objects and provide methods to add moves to the sequence and retrieve them. By encapsulating the moves within an `Opening` class, we can manage different opening sequences and access them in an organized manner. Furthermore, each opening sequence will also have a name (e.g., "Sicilian Defense"), which will help in identifying and suggesting openings during gameplay.

With the `Opening` class in place, we can then create an `OpeningBook` class to hold a collection of these opening sequences. This class will provide a method to add new openings to the book and another method to find a matching opening based on the current sequence of moves in the game. The key functionality of this class is its ability to match the current game state (a sequence of moves made so far) with one of the stored openings. This can be achieved through a straightforward iteration over the stored openings and comparing each opening's moves with the current sequence.

To initialize the opening book, we need to populate it with standard openings. This is done by creating instances of the `Opening` class for each standard opening and adding the respective moves to these instances. Once all standard openings are created, they are added to the `OpeningBook`. This initialization can be done at the start of the program or during a specific setup phase.

During gameplay, we utilize the opening book to suggest moves based on the current state of the game. As moves are made, we keep track of the sequence of moves. After each move, we use the `OpeningBook` to find a matching opening sequence. If a match is found, the next move in the sequence is suggested to the player. This involves comparing the current move sequence with each opening in the book and finding the one that matches the longest prefix of the current sequence. The suggestion is then based on the next move in this matched sequence.

In summary, implementing a book of standard can be effectively achieved by creating a structured class hierarchy to represent moves and openings, and using simple iteration to match and suggest moves. By using iterators to traverse and match the moves, we maintain simplicity while ensuring that the opening

book functionality is robust and extendable. Additionally, this method allows for flexibility in terms of how openings are defined and managed, making it easy to add or modify openings as needed without the need for complex data structures or external dependencies.

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

To implement an undo feature that supports reverting the last move, and potentially an unlimited number of moves, we need to maintain a detailed history of game states. This approach involves using a stack data structure to manage these states, leveraging its LIFO (Last In, First Out) nature to easily revert to the most recent state.

In the current implementation, the undo mechanism uses a history stack (`move_history`) to track recent moves. The `Board::undo()` method retrieves the last move from this stack and reverses it to restore the game to its prior state. Here's a breakdown of how the undo functionality is implemented: The `move_history` stack keeps track of all executed moves. Each move is represented as a string where each part encodes specific details of the move. For example, `"a2a3P  "` indicates a move from `a2` to `a3` with a piece promotion, and `"a4b5Pp "` indicates a move from `a4` to `b5` with a pawn promotion and a special en passant move.

The `undo()` method starts by checking if there are moves in the history stack. If the stack is not empty, it retrieves and removes the last move. The method parses the move string to determine the source (`from`) and destination (`to`) squares involved in the move. For example, if the last move string is `"a2a3P  "`, it identifies `a2` as the source and `a3` as the destination. Based on the move details, it then calls `replacePiece()` to reverse the move. This method places the piece back on its original square and clears the destination square and returns the positions of the squares that need to be re-rendered, ensuring the graphical and text displays are updated to reflect the reverted game state.

The stack-based approach inherently supports an unlimited number of undos, as long as the system memory can handle the stack's growth. Each undo operation simply involves popping the most recent entry from the stack and restoring the game state. To handle potential memory issues in long games, it's important to manage the stack size and clear history if necessary, especially in scenarios involving multiple games or prolonged play sessions.

In summary, the undo feature leverages a stack to efficiently manage and revert game states. By maintaining a history of moves and applying them in reverse order, the implementation allows for undoing the last move or multiple moves as needed. However, special attention is needed for complex scenarios involving promotions or special moves to ensure accurate state restoration.

QUESTION: *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. (If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?)*

To adapt a two-player chess program into a four-handed chess game, modifications to the board representation, player management, and game logic are required. To be specific, it involves designing a larger board, managing multiple players, and updating the game rules to accommodate the four-player dynamic.

First, we need to modify the board structure to support a larger grid that can accommodate four sets of pieces. This involves redesigning the board layout and updating the logic to handle the extended size . We will create a Board class that represents the chessboard, using a std::vector to store the grid of pieces. By encapsulating the board in a class, we ensure modularity and ease of management. The board size will be increased to accommodate the additional pieces and players, ensuring enough space for all four players to place their pieces and make moves.

Next, we update the player management system to handle four players. This involves managing turn order and maintaining player-specific information. We will create a Game class that manages the players, using an enumeration to represent the four players and a vector to store the player list. The Game class will include methods to handle turn rotation, ensuring that each player gets their turn in the correct order ie instead of alternating between two players, the game must now cycle through four players, maybe in a clockwise fashion. By encapsulating player management in a class, we achieve a modular design that is easy to extend and maintain.

Additionally, the piece initialization process will also need to be updated to initialize pieces for all four players. This involves placing pieces on the board in distinct starting positions for each player. A possible initializeFourPlayerBoard method will be responsible for setting up the initial state of the board, placing pieces for each player in their respective positions. This method ensures that the board is correctly set up at the start of the game, with all players' pieces in place and ready for play.

Moreover, we need to adjust the move validation and game rules to accommodate the four player dynamic. This includes handling captures, checks, and checkmates for four players. The game logic will be updated to account for the additional players, ensuring that all rules are correctly enforced. This involves updating the move validation logic to handle moves from four different players, ensuring that captures and checks are correctly managed for all players.

Finally, the graphical display must be modified to visually differentiate between the four players' pieces. This can be achieved through color coding, labels, or distinct piece designs. The user interface should also include additional elements to indicate the current player, and provide a clear overview of the board state from a four-player perspective.

In summary, adapting a chess program for four-handed play involves significant modifications to the board representation, player management, piece initialization, and user interface. By carefully addressing these aspects, the program can provide a seamless and engaging experience for four players while maintaining the core mechanics and strategic depth of chess.

EXTRA CREDIT FEATURES:

Implemented undo function

FINAL QUESTIONS:

1. *What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?*

Working on this Chess game project taught us several important lessons about writing large programs. One of the most significant lessons is the necessity of thorough planning and design before diving into coding. At the beginning of the project, we focused on creating a comprehensive UML diagram and detailed design documents. This step was crucial as it provided a clear roadmap and helped in visualizing the relationships between different classes and components. Without a clear plan, it is easy to get lost in the complexities of the code and lose track of the overall objectives.

Another key lesson is the importance of breaking down the project into smaller, manageable tasks. By dividing the project into distinct modules such as `Board`, `Piece` subclasses, `Text_Display`, and `Graphical_Display`, we could focus on one aspect at a time, ensuring each part was well-implemented before moving on to the next. For instance, implementing the movement logic in each `Piece` subclass required careful attention to the specific rules for each piece, which was more manageable when tackled individually. This modular approach also made it easier to test and debug each component individually, leading to a more stable and maintainable codebase.

Moreover, regular testing and debugging were vital throughout the development process. By continuously testing each module as it was completed, we could catch errors early and ensure that the code remained functional and stable. For example, the setup command suite was imperative in testing the `move` function in each `Piece` subclass, check and checkmate functions to ensure they correctly handled various scenarios. This approach prevented the accumulation of bugs and reduced the time spent on debugging towards the end of the project.

Lastly, we learned the value of effective documentation. Documenting the code and keeping detailed records of design decisions and changes made the development process smoother and made it easier to revisit and understand the code after some time. For instance, maintaining clear documentation for the `Graphical_Display` and `Text_Display` classes helped in understanding how the visual components interacted with the `Board` class. Good documentation was especially important in the group, as it helps maintain clarity and consistency throughout the codebase.

In conclusion, writing large programs requires meticulous planning, breaking down tasks into manageable components, regular testing, and thorough documentation. These practices ensure a more efficient and organized development process, leading to a higher-quality final product.

2. *What would you have done differently if you had the chance to start over?*

Reflecting on our experience with the project, there are several areas where we could have approached things differently if given a chance to start over. Firstly, we would allocate more time to the initial planning and design phase. While we did create a UML diagram and design documents, spending more time on this step would have provided an even clearer roadmap and helped avoid some of the issues encountered during implementation. A more detailed and realistic timeline that accounts for potential challenges and unforeseen obstacles would also have been beneficial.

Secondly, we would incorporate more frequent and structured testing phases into my project plan. While we did test each module individually, more systematic and rigorous testing throughout the development process could have caught some issues earlier and reduced the time spent on debugging towards the end. For example, integrating automated tests for the `move` functions in `Piece` subclasses and the `check` and `checkmate` methods in the `Board` class could have ensured robustness and reliability in the game logic.

Additionally, we would prioritize better time management and avoid underestimating the complexity of certain features. There were moments when we underestimated the time required to implement certain functionalities, such as the `Board` class's checkmate, check methods. Allocating sufficient time for each task and building in buffers for unexpected delays would have helped maintain a more steady and less stressful workflow.

Finally, we would have focused more on refining and enhancing the graphical display component. While the textual display was functional, the graphical display could have benefited from additional features and improvements, such as smoother animations and a more user-friendly interface. Allocating more time and resources to this aspect would have resulted in a more polished and visually appealing final product.

In conclusion, if given the chance to start over, I would invest more time in detailed planning and design, incorporate structured testing phases, manage time more effectively, and focus on enhancing the graphical display. These changes would lead to a more organized development process, higher-quality code, and a more refined final product.