Esthergen Poker

Software Specifications

Version 1.2

Authors
Esther Anaya, Noah Mathew, Trang Nguyen, and Artin Tammadon

Affiliation
UC Irvine Electrical Engineering and Computer Science (EECS) Department

Table of Contents

Glossary

All-in: A poker move where a player bets all their remaining chips.

API (Application Programming Interface): A set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.

Bet: To place a wager in the game.

Call: To match the current bet in a round of poker.

Check: To pass the action to the next player without betting, only possible if no bet has been made in the current round.

Fold: To discard one's hand and forfeit the round, losing any bets made.

Flop: The first three community cards dealt face-up in the game of Texas Hold'em.

Function Prototype: A declaration of a function that specifies the function's name, parameters, and return type, but not the function body.

HTTP (Hypertext Transfer Protocol): The protocol used for transmitting web pages over the Internet.

Raise: To increase the current bet in a round of poker.

River: The fifth and final community card dealt face-up in Texas Hold'em.

Socket: An endpoint for sending or receiving data across a computer network.

TCP (Transmission Control Protocol): A standard that defines how to establish and maintain a network conversation through which application programs can exchange data.

Turn: The fourth community card dealt face-up in Texas Hold'em.

WebSocket: A computer communications protocol providing full-duplex communication channels over a single TCP connection.

1. Poker Client Software Architecture Overview

1.1 Main Data Types and Structures

The header file defines several key data types and structures essential for playing a game of poker. Firstly, enumerations are employed to categorize card attributes and player actions. The `Suit` enumeration encapsulates the four traditional suits found in a standard deck of cards: clubs, diamonds, hearts, and spades. Similarly, the `Rank` enumeration encompasses the ranks of cards, ranging from two to ace. Another enumeration, `Color`, represents the color of cards, distinguishing between black and red. Additionally, `Combo` enumerates various poker hand combinations, such as straight, flush, and full house. The `PlayersMove` enumeration outlines permissible player actions during gameplay, including folding, checking, calling, and raising.

The structures defined in the header file serve to organize and manage game elements. The `Card` structure represents an individual playing card, encapsulating attributes like suit, rank, and color. A `Deck` structure models a standard deck of 52 cards and maintains the index of the top card. The `Hand` structure depicts a player's hand, comprising two cards. Furthermore, the `Player` structure embodies a player in the game, featuring attributes like name, points, hand, and move status. The `CommunityCards` structure captures the communal cards, also known as "the board," in a poker game. Seats at the poker table are represented by the `Seat` structure, indicating occupancy status and player details if taken. Additionally, the `Blind` structure encompasses information regarding blinds, including the small blind and big blind. The `Table` structure provides an overarching representation of the poker table, incorporating seats, community cards, and the pot. Lastly, the `RaiseTracker` structure monitors the seat and the amount of the highest bet during a round of betting.

In GUI.h important data types and structures include `SeatPosition`, representing the position of each seat, `Position` representing x and y coordinates used for placing widgets, `CardPosition` holds the position of a card in the sprite map, `CardData` contains the image data and position for a card to be drawn, and `PlayerLabel` representing the label and name for each player at the table.

Lastly, the header file declares function prototypes to support various game functionalities. These functions are responsible for tasks such as initializing, shuffling, and dealing cards, converting enumerations to strings for display purposes, managing hands, setting up players and the table, handling blinds, and orchestrating turn progression logic in a poker game. Together, these data types, structures, and function prototypes form the foundation for implementing and executing the rules and mechanics of a poker game.

1.2 Major Software Components

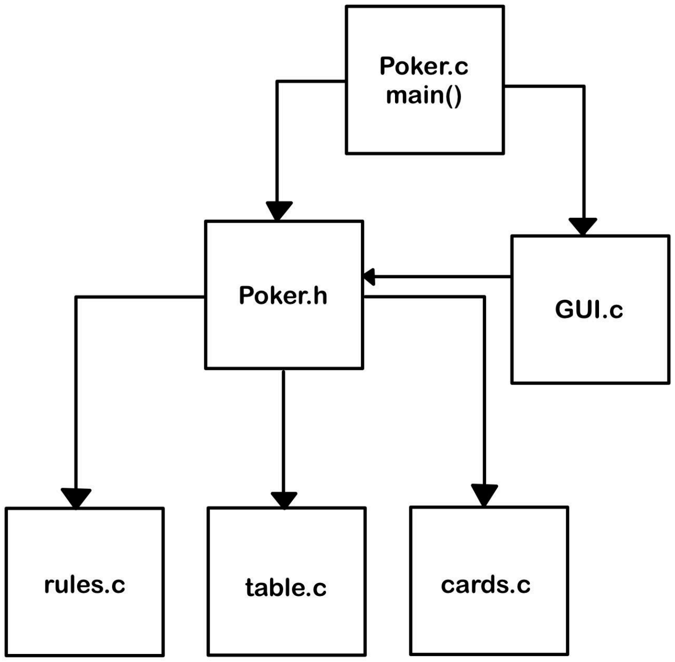


Figure 1: Diagram of client module hierarchy.

The client module hierarchy, shown in Figure 1, includes:

- Arrows indicate dependencies (i.e., an arrow from Poker.c to Poker.h means Poker.c depends on Poker.h).
- The \`poker.c\` source file contains the implementation of the main poker game functions declared in \`poker.h\`. It manages the initialization and shuffling of the deck, the setup of the poker table, and the assignment of blinds. The game flow includes dealing cards to players, displaying the table state, and handling player actions such as betting during the buy-in turn. It also includes logic for round progression and incrementing blinds, ensuring a complete poker game cycle.
- The \`poker.h\` header file contains declarations for the core functions and data structures needed to run a poker game. It includes enumerations for card suits, ranks, colors, and player actions, as well as structures for cards, decks, players, hands, tables, and blinds. The file also declares function prototypes for deck operations, hand management, table setup, blind handling, and game progression. This organization provides a clear framework for implementing and managing the game's logic and flow.
- The \`rules.c\` file implements functions for determining various poker hand rankings as declared in \`rules.h\`. It includes logic to check for specific hands such as royal flush, straight flush, four of a kind, full house, flush, straight, three of a kind, two pair, one pair, and high card. The functions use player hand and community cards, applying sorting and

comparison logic to verify the presence of these hands. The `cmpfunc` function is used for sorting cards, aiding in the detection of straights and flushes.

- The ‘table.c’ file implements a poker game logic with functions for initializing and displaying the table, handling blinds, and managing turns. It sets up player seats, manages community cards, and handles bets, checks, calls, and raises with appropriate error handling for user inputs. Blinds are assigned and incremented, and a buy-in turn process is implemented to manage the sequence of player moves. The code ensures smooth gameplay by validating player actions and maintaining the game state across turns.
- The `cards.c` file defines functions for initializing, shuffling, and dealing cards in a poker game. It includes utility functions for converting card ranks and suits to strings and for displaying hands and community cards. Additionally, it contains functions to clear hands, manage community cards, and create players. These implementations ensure the game logic for handling cards and players is encapsulated within well-defined functions.
- In the `gui.c` file, the graphical user interface (GUI) components for the poker game are implemented. This file handles the rendering of various elements such as the deck, community cards, and players' hands, ensuring they are displayed accurately on the screen. Additionally, it manages user interactions by capturing input actions like betting, folding, or selecting cards, providing players with an intuitive and engaging interface to interact with the game. `gui.c` interfaces seamlessly with the underlying game logic, updating the visual representation of the game state in real-time as the game progresses, thus enhancing the overall gaming experience for the players.

1.3 Module Interfaces

1.3.1: initializeTable(Table *table):

- Description: Initializes the poker table and players.
- Parameters: ‘Table *table’ - Pointer to the table struct.
- Returns: Void.

1.3.2: displayTable(const Table *table):

- Description: Displays the current state of the table and players.
- Parameters: ‘const Table *table’- Pointer to the constant table struct.
- Returns: Void.

1.3.3: preRoundInitializations(Table *table):

- Description: Performs pre-round initializations for each player.
- Parameters: ‘Table *table’- Pointer to the table struct.
- Returns: Void.

1.3.4: preTurnInitializationsTable *table):

- Description: Performs pre-turn initializations for each player.
- Parameters: ‘Table *table’- Pointer to the table struct.
- Returns: Void.

1.3.5: assignBlinds(Blind *blinds, Table *table):

- Description: Assigns blinds for the poker game.
- Parameters:
    - ‘Blind *blinds’- Pointer to the blind struct.
    - ‘Table *table’- Pointer to the table struct.
- Returns: Integer- 1 if an error occurs, 0 otherwise.

1.3.6: incrementBlinds(Blind *blinds, Table *table):

- Description: Increments blinds for the next round.
- Parameters:
    - ‘Blind *blinds’- Pointer to the blind struct.
    - ‘Table *table’- Pointer to the table struct.
- Returns: Void.

1.3.7: placeBlindsBets(Table *table, Blind *blinds):

- Description: Places blind bets at the start of a round.
- Parameters:
    - ‘Table *table’- Pointer to the table struct.
    - ‘Blind *blinds’- Pointer to the blind struct.
- Returns: Void.

1.3.8: findNextAvailableSeat(Table *table, int currentSeat):

- Description: Find the next available seat for betting.
- Parameters:
    - ‘Table *table’- Pointer to the table struct.
    - ‘int currentSeat’- Current seat number.
- Returns: Integer- Next available seat number.

1.3.9: checkBuyInMoveValiditiy(int currentSeat, Table *table, RaiseTracker *trackPtr, int move):

- Description: Checks the validity of a player’s move during buy-in.
- Parameters:
    - ‘int currentSeat’- current seat number.
    - ‘Table *table’- Pointer to the table struct.
    - ‘RaiseTracker *trackPtr’- Pointer to the raise tracker struct.
    - ‘int move’- Player’s move.
- Returns: Integer- 1 if the move is not valid, 0 otherwise.

1.3.10: excecuteBuyInPlayersMove(int currentSeat, Table *table, RaiseTracker *trackPtr):

- Description: Executes a player’s move during buy-in.
- Parameters:
    - ‘int currentSeat’- current seat number.
    - ‘Table *table’- Pointer to the table struct.
    - ‘RaiseTracker *trackPtr’- Pointer to the raise tracker struct.

- Returns: Integer- 1 if a raise is found, 0 otherwise.

1.3.11: create_main_window():

- Description: Initializes and sets up the main window of the game, including the table image and other GUI components.
- Parameters: none
- Returns: ‘GtkWidget*’- The main window widget

1.3.11: *create_quit_button(GtkWidget container):

- **Description**: Adds a "Quit" button to the specified container that exits the application when clicked.
- **Parameters**: `GtkWidget *container` - The container to which the button will be added.
- **Returns**: Void

1.3.12: **create_name_entry(GtkWidget container)*

- **Description**: Adds an entry field and label to the container for entering a player's name.
- **Parameters**: `GtkWidget *container` - The container to which the entry field and label will be added.
- **Returns**: Void

1.3.13: display_hole_cards(int seat_number)

- **Description**: Displays the hole cards for a specified player's seat.
- **Parameters**: `int seat_number` - The seat number for which the hole cards are displayed.
- **Returns**: Void

1.3.14draw_rotated_text(GtkWidget *widget, GdkEventExpose event, gpointer user_data)***

- **Description**: Draws text rotated on a widget, used for displaying player names on their seats.
- **Parameters**: `GtkWidget *widget`, `GdkEventExpose *event`, `gpointer user_data` - Standard GTK+ parameters.
- **Returns**: gboolean - False after the drawing operation.

1.4 Overall Program Control Flow
The overall program control flow typically follows these steps:

1. **Initialization:** The program starts by initializing any necessary data structures, variables, and configurations. This may include setting up the game environment, initializing the deck of cards, and preparing the GUI if applicable.

2. **Game Setup:** The program proceeds to set up the game, which involves tasks like creating player profiles, assigning seats at the table, dealing cards, and determining initial conditions such as blinds and starting chips.

3. **Game Loop:** The core of the program is a game loop that iterates through the various stages of gameplay. This loop continues until the game is over. Within this loop, the program manages actions such as dealing community cards, accepting player moves, updating the game state, and determining winners.

4. **Player Actions:** During each iteration of the game loop, the program prompts active players to make decisions based on the current game state. Players may choose actions such as folding, checking, calling, raising, or betting.

5. **Updating Game State:** After each player action, the program updates the game state to reflect changes such as updated chip counts, community cards, and player moves.

6. **Winning Condition Check:** At various points within the game loop, the program checks for winning conditions. This involves evaluating hands, determining winners, and awarding chips or ending the game if necessary.

7. **End of Game:** Once the game loop concludes, the program handles end-of-game tasks such as displaying the final results, declaring winners, resetting the game environment, and possibly prompting the players for another game.

8. **Cleanup and Termination:** Finally, the program performs any necessary cleanup tasks and gracefully terminates, releasing resources, closing connections, and exiting the application.

1.5 Automated Client: Poker Bot

The automated client, or poker bot, is a software program capable of playing poker games autonomously. It interacts with the poker server through an API, sending and receiving game data and executing actions based on its programmed logic. The bot analyzes game situations using sophisticated algorithms, considering factors such as hand strength, pot odds, and opponent behavior. It then makes decisions such as folding, calling, raising, or betting accordingly. The bot's strategies may range from simple rule-based approaches to more complex algorithms that adapt and learn from gameplay. However, it's important to note that using such bots on online poker platforms may violate their terms of service and can lead to penalties.

2. Poker Server Software Architecture Overview

2.1 Main Data Types and Structures
The header file defines several key data types and structures essential for playing a game of poker. Firstly, enumerations are employed to categorize card attributes and player actions. The `Suit` enumeration encapsulates the four traditional suits found in a standard deck of cards: clubs, diamonds, hearts, and spades. Similarly, the `Rank` enumeration encompasses the ranks of cards, ranging from two to ace. Another enumeration, `Color`, represents the color of cards, distinguishing between black and red. Additionally, `Combo` enumerates various poker hand combinations, such as straight, flush, and full house. The `PlayersMove` enumeration outlines permissible player actions during gameplay, including folding, checking, calling, and raising.

The structures defined in the header file serve to organize and manage game elements. The `Card` structure represents an individual playing card, encapsulating attributes like suit, rank, and color. A `Deck` structure models a standard deck of 52 cards and maintains the index of the top card. The `Hand` structure depicts a player's hand, comprising two cards. Furthermore, the `Player` structure embodies a player in the game, featuring attributes like name, points, hand, and move status. The `CommunityCards` structure captures the communal cards, also known as "the board," in a poker game. Seats at the poker table are represented by the `Seat` structure, indicating occupancy status and player details if taken. Additionally, the `Blind` structure encompasses information regarding blinds, including the small blind and big blind. The `Table` structure provides an overarching representation of the poker table, incorporating seats, community cards, and the pot. Lastly, the `RaiseTracker` structure monitors the seat and the amount of the highest bet during a round of betting.

In GUI.h important data types and structures include `SeatPosition`, representing the position of each seat, `Position` representing x and y coordinates used for placing widgets, `CardPosition` holds the position of a card in the sprite map, `CardData` contains the image data and position for a card to be drawn, and `PlayerLabel` representing the label and name for each player at the table.

Lastly, the header file declares function prototypes to support various game functionalities. These functions are responsible for tasks such as initializing, shuffling, and dealing cards, converting enumerations to strings for display purposes, managing hands, setting up players and the table, handling blinds, and orchestrating turn progression logic in a poker game. Together, these data types, structures, and function prototypes form the foundation for implementing and executing the rules and mechanics of a poker game.

2.2 Major Software Components

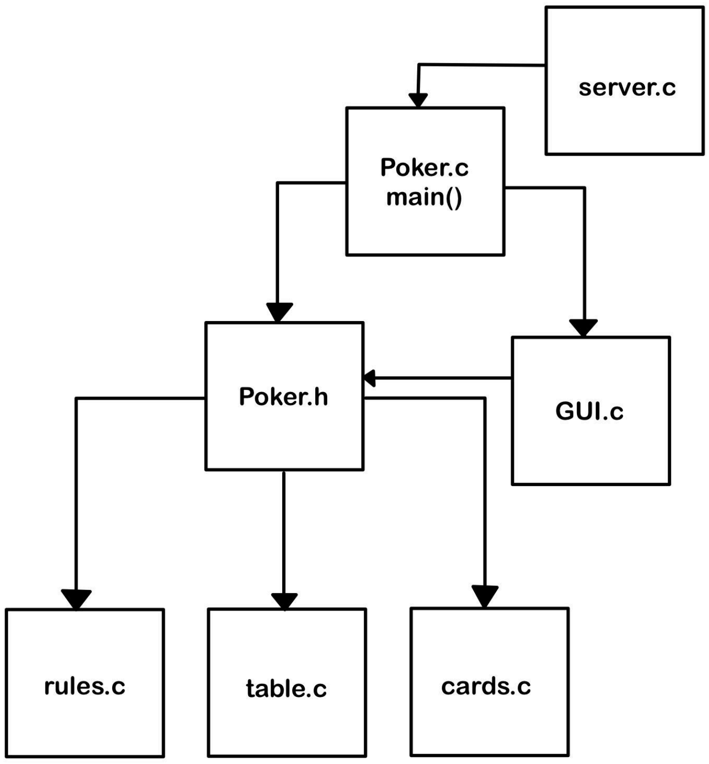


Figure 2: Diagram of server module hierarchy.

The client module hierarchy, shown in Figure 1, includes:

- Arrows indicate dependencies (i.e., an arrow from Poker.c to Poker.h means Poker.c depends on Poker.h).
- The `server.c` file is the core component of our client-server system, managing client connections, processing requests, and facilitating communication. It initializes sockets, binds to a port, and listens for incoming connections. When a connection is established, it handles each client interaction concurrently, parsing requests, executing actions, and sending responses. Robust error handling ensures stability, and logging functionality aids in monitoring and debugging. In summary, `server.c` forms the backbone of our application, enabling smooth client-server communication.
- The `poker.c` source file contains the implementation of the main poker game functions declared in `poker.h`. It manages the initialization and shuffling of the deck, the setup of the poker table, and the assignment of blinds. The game flow includes dealing cards to players, displaying the table state, and handling player actions such as betting during the buy-in turn. It also includes logic for round progression and incrementing blinds, ensuring a complete poker game cycle.
- The `poker.h` header file contains declarations for the core functions and data structures needed to run a poker game. It includes enumerations for card suits, ranks, colors, and player actions, as well as structures for cards, decks, players, hands, tables, and blinds. The file also declares function prototypes for deck operations, hand management, table

setup, blind handling, and game progression. This organization provides a clear framework for implementing and managing the game's logic and flow.

- The `rules.c` file implements functions for determining various poker hand rankings as declared in `rules.h`. It includes logic to check for specific hands such as royal flush, straight flush, four of a kind, full house, flush, straight, three of a kind, two pair, one pair, and high card. The functions use player hand and community cards, applying sorting and comparison logic to verify the presence of these hands. The `cmpfunc` function is used for sorting cards, aiding in the detection of straights and flushes.
- The 'table.c' file implements a poker game logic with functions for initializing and displaying the table, handling blinds, and managing turns. It sets up player seats, manages community cards, and handles bets, checks, calls, and raises with appropriate error handling for user inputs. Blinds are assigned and incremented, and a buy-in turn process is implemented to manage the sequence of player moves. The code ensures smooth gameplay by validating player actions and maintaining the game state across turns.
- The `cards.c` file defines functions for initializing, shuffling, and dealing cards in a poker game. It includes utility functions for converting card ranks and suits to strings and for displaying hands and community cards. Additionally, it contains functions to clear hands, manage community cards, and create players. These implementations ensure the game logic for handling cards and players is encapsulated within well-defined functions.
- In the `gui.c` file, the graphical user interface (GUI) components for the poker game are implemented. This file handles the rendering of various elements such as the deck, community cards, and players' hands, ensuring they are displayed accurately on the screen. Additionally, it manages user interactions by capturing input actions like betting, folding, or selecting cards, providing players with an intuitive and engaging interface to interact with the game. `gui.c` interfaces seamlessly with the underlying game logic, updating the visual representation of the game state in real-time as the game progresses, thus enhancing the overall gaming experience for the players.

2.3 Module Interfaces

2.3.1: initializeTable(Table *table):

- Description: Initializes the poker table and players.
- Parameters: 'Table *table' - Pointer to the table struct.
- Returns: Void.

2.3.2: displayTable(const Table *table):

- Description: Displays the current state of the table and players.
- Parameters: 'const Table *table'- Pointer to the constant table struct.
- Returns: Void.

2.3.3: preRoundInitializations(Table *table):

- Description: Performs pre-round initializations for each player.
- Parameters: 'Table *table'- Pointer to the table struct.

- Returns: Void.

2.3.4: preTurnInitializationsTable *table):

- Description: Performs pre-turn initializations for each player.
- Parameters: ‘Table *table’- Pointer to the table struct.
- Returns: Void.

2.3.5: assignBlinds(Blind *blinds, Table *table):

- Description: Assigns blinds for the poker game.
- Parameters:
  - ‘Blind *blinds’- Pointer to the blind struct.
  - ‘Table *table’- Pointer to the table struct.
- Returns: Integer- 1 if an error occurs, 0 otherwise.

2.3.6: incrementBlinds(Blind *blinds, Table *table):

- Description: Increments blinds for the next round.
- Parameters:
  - ‘Blind *blinds’- Pointer to the blind struct.
  - ‘Table *table’- Pointer to the table struct.
- Returns: Void.

2.3.7: placeBlindsBets(Table *table, Blind *blinds):

- Description: Places blind bets at the start of a round.
- Parameters:
  - ‘Table *table’- Pointer to the table struct.
  - ‘Blind *blinds’- Pointer to the blind struct.
- Returns: Void.

2.3.8: findNextAvailableSeat(Table *table, int currentSeat):

- Description: Find the next available seat for betting.
- Parameters:
  - ‘Table *table’- Pointer to the table struct.
  - ‘int currentSeat’- Current seat number.
- Returns: Integer- Next available seat number.

2.3.9: checkBuyInMoveValiditiy(int currentSeat, Table *table, RaiseTracker *trackPtr, int move):

- Description: Checks the validity of a player’s move during buy-in.
- Parameters:
  - ‘int currentSeat’- current seat number.
  - ‘Table *table’- Pointer to the table struct.
  - ‘RaiseTracker *trackPtr’- Pointer to the raise tracker struct.
  - ‘int move’- Player’s move.
- Returns: Integer- 1 if the move is not valid, 0 otherwise.

2.3.10: excecuteBuyInPlayersMove(int currentSeat, Table *table, RaiseTracker *trackPtr):

- Description: Executes a player's move during buy-in.
- Parameters:
  - 'int currentSeat'- current seat number.
  - 'Table *table'- Pointer to the table struct.
  - 'RaiseTracker *trackPtr'- Pointer to the raise tracker struct.
- Returns: Integer- 1 if a raise is found, 0 otherwise.

2.4 Overall Program Control Flow

The overall program control flow typically follows these steps:

1. **Initialization:** The program starts by initializing any necessary data structures, variables, and configurations. This may include setting up the game environment, initializing the deck of cards, and preparing the GUI.

2. **Game Setup:** The program proceeds to set up the game, which involves tasks like creating player profiles, assigning seats at the table, dealing cards, and determining initial conditions such as blinds and starting chips.

3. **Game Loop:** The core of the program is a game loop that iterates through the various stages of gameplay. This loop continues until the game is over. Within this loop, the program manages actions such as dealing community cards, accepting player moves, updating the game state, and determining winners.

4. **Player Actions:** During each iteration of the game loop, the program prompts active players to make decisions based on the current game state. Players may choose actions such as folding, checking, calling, raising, or betting.

5. **Updating Game State:** After each player action, the program updates the game state to reflect changes such as updated chip counts, community cards, and player moves.

6. **Winning Condition Check:** At various points within the game loop, the program checks for winning conditions. This involves evaluating hands, determining winners, and awarding chips or ending the game if necessary.

7. **End of Game:** Once the game loop concludes, the program handles end-of-game tasks such as displaying the final results, declaring winners, resetting the game environment, and possibly prompting the players for another game.

8. **Cleanup and Termination:** Finally, the program performs any necessary cleanup tasks and gracefully terminates, releasing resources, closing connections, and exiting the application.

3. Installation

3.1 System Requirements

compatible with Linux distributions that support Linux Kernel 4.14 or newer. Minimum recommended distributions include:

- Debian 9
- Ubuntu 18
- Fedora 27
- Red Hat Enterprise 7

Minimum recommended system hardware requirements:

- 10 MB disk space
- 1+ GB RAM

3.2 Unpacking and Configuration

1. Unpack the Poker_V1.0.tar.gz file using gunzip or a comparable utility.
2. Run the ./ESTHERGEN_Installer script to install the game in the /usr/local/bin/ESTHERGEN_Poker folder on your computer.
3. Run the ./ESTHERHEN_Poker app from the installation path folder to launch the game.
4. If using a Windows environment, ensure Xming or a similar X server is installed and running to display the graphical user interface properly.
5. Each run of the game exports a transcript of moves to the move_log.txt file in the game directory.

3.3 Building, Compilation, Installation

This Poker program is built using a directory structure containing all necessary resources, including the C files necessary for compiling the program and a custom installation script to streamline this process for the user.

The directory structure, created from the extraction of the package (**Poker_V1.0.tar.gz**), contains the following directories and content:

- **(root):** Contains the Makefile, README.md, LICENCE.txt, and Bash script files to install or uninstall the game (*install.sh*).
- **bin:** *ESTHERGEN_Poker* Application file (after its compiled)
- **doc:** Includes the user manual (*Poker_UserManual.pdf*).
- **images:** Holds bitmap files used with the GUI.

- **src:** Contains C header and module files used to compile the game application including files for socket-based client-server communication.

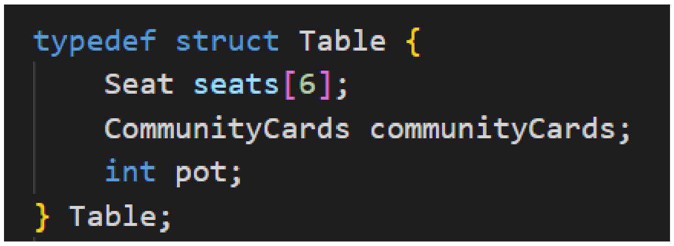
The installer script (install.sh) is a Bash script that uses case switch logic to offer the user install and uninstall options and carry out the corresponding actions. The 'install' option compiles the ESTHERGEN_Poker program, utilizing the Makefile in the root directory and the C header and module files in the src directory, including those required for socket setup. The 'uninstall' option removes all local files, including the installation script itself.

After unpacking the package (**Poker_V1.0.tar.gz**), type the following command to run the installer: `bash install.sh`

4. Documentation of Packages, Modules, Interfaces

4.1 Detailed Description of Data Structures

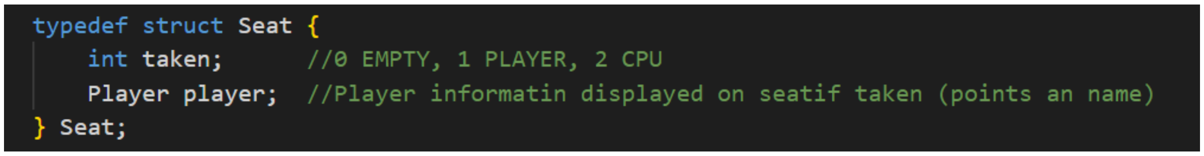
1. Table Structure:
   - Description: Represents the poker table, including seats, players, and community cards.
   - Code snippet:



```c
typedef struct Table {
    Seat seats[6];
    CommunityCards communityCards;
    int pot;
} Table;
```

Figure 4.1.1: Table Data Structure code snippet.

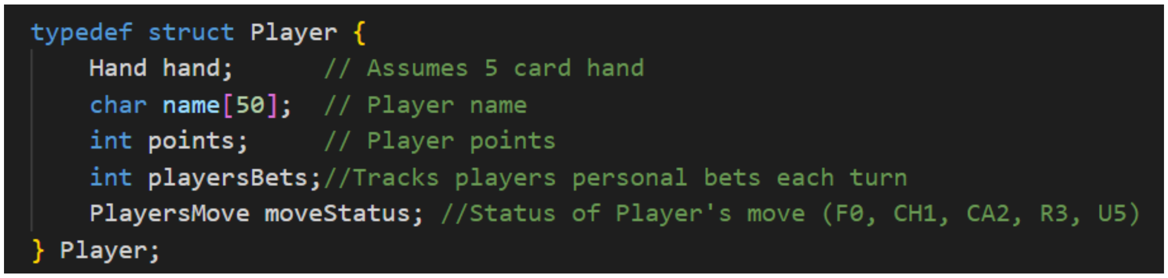
2. Seat Structure:
   - Description: Represents a seat at the table, containing information about whether it's occupied and the player sitting there.
   - Code snippet:



```c
typedef struct Seat {
    int taken;      //0 EMPTY, 1 PLAYER, 2 CPU
    Player player;  //Player informatin displayed on seatif taken (points an name)
} Seat;
```

Figure 4.1.2: Seat Data Structure code snippet.

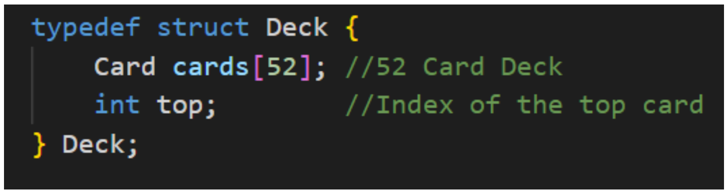
3. Player Structure:
   - Description: Represents a player, including their name, points (chips), hand of cards, and move status.
   - Code snippet:



```c
typedef struct Player {
    Hand hand;      // Assumes 5 card hand
    char name[50];  // Player name
    int points;     // Player points
    int playersBets;//Tracks players personal bets each turn
    PlayersMove moveStatus; //Status of Player's move (F0, CH1, CA2, R3, U5)
} Player;
```

Figure 4.1.3: Player Data Structure code snippet.

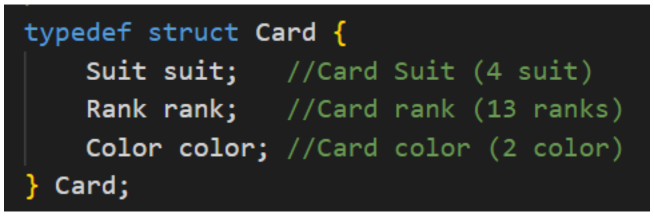
4. Deck Structure:
   - Description: Represents a deck of cards, including an array of cards and a pointer to the top card.
   - Code snippet:



```
typedef struct Deck {
    Card cards[52]; //52 Card Deck
    int top;        //Index of the top card
} Deck;
```

Figure 4.1.4: Deck Data Structure code snippet.

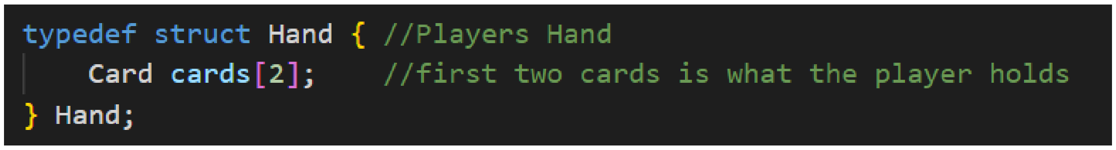
5. Card Structure:
   - Description: Represents a playing card, including its rank, suit, and color.
   - Code snippet:



```
typedef struct Card {
    Suit suit;   //Card Suit (4 suit)
    Rank rank;   //Card rank (13 ranks)
    Color color; //Card color (2 color)
} Card;
```

Figure 4.1.5: Card Data Structure code snippet.

6. Hand Structure:
   - Description: Represents a player's hand of cards, containing an array of cards.
   - Code snippet:



```
typedef struct Hand { //Players Hand
    Card cards[2];    //first two cards is what the player holds
} Hand;
```

Figure 4.1.6: Hand data Structure code snippet.

4.2 Detailed Description of Functions and Parameters

1. initializeDeck

- Function Prototype: void initializeDeck(Deck *deck);
- Parameters:
  - deck: A pointer to the Deck structure representing the deck of cards to be initialized.
- Brief Explanation: Initializes the deck of cards by assigning each card a suit, rank, and color.

2. shuffleDeck

- Function Prototype: void shuffleDeck(Deck *deck);
- Parameters:
  - deck: A pointer to the Deck structure representing the deck of cards to be shuffled.
- Brief Explanation: Shuffles the deck of cards using the Fisher-Yates shuffle algorithm to randomize the order of the cards.

3. printDeck

- Function Prototype: void printDeck(Deck *deck);
- Parameters:
  - deck: A pointer to the Deck structure representing the deck of cards to be printed.
- Brief Explanation: Prints the contents of the deck of cards, displaying each card's rank and suit in a 4x13 matrix format.

4. clearCommunityCards

- Function Prototype: void clearCommunityCards(Table *table);
- Parameters:
  - table: A pointer to the Table structure representing the poker table.
- Brief Explanation: Clears the community cards on the table by setting their ranks to EMPTY.

5. clearPlayersHand

- Function Prototype: void clearPlayersHand(Player *player);
- Parameters:

    - player: A pointer to the Player structure representing the player whose hand needs to be cleared.
- Brief Explanation: Clears the player's hand by setting the ranks of their cards to EMPTY.

6. displayCommunityCards

- Function Prototype: void displayCommunityCards(const Table *table);
- Parameters:
    - table: A pointer to the Table structure representing the poker table.
- Brief Explanation: Displays the community cards on the table, showing each card's rank and suit or indicating if a slot is empty.

7. displayPlayerHand

- Function Prototype: void displayPlayerHand(const Player *player);
- Parameters:
    - player: A pointer to the Player structure representing the player whose hand needs to be displayed.
- Brief Explanation: Displays the player's hand, showing each card's rank and suit or indicating if the hand is empty.

8. dealPlayersHands

- Function Prototype: void dealPlayersHands(Table *table, Deck *deck);
- Parameters:
    - table: A pointer to the Table structure representing the poker table.
    - deck: A pointer to the Deck structure representing the deck of cards.
- Brief Explanation: Deals hands to players at the table by distributing cards from the deck to each player's hand.

9. create_main_window

- Function Prototype: `GtkWidget* create_main_window();`
- Parameters: None
- Description: Initializes and sets up the main window of the poker game, including loading the table image and adding GUI elements for player interaction.
- Returns: A pointer to the main window widget.

10. create_quit_button

- Function Prototype: `void create_quit_button(GtkWidget *container);`

- Parameters: `GtkWidget *container` - The container to which the Quit button will be added.
- Description: Adds a button labeled "Quit" to the container, which exits the application when clicked.
- Returns: Void

11. create_name_entry

- Function Prototype: `void create_name_entry(GtkWidget *container);`
- Parameters: `GtkWidget *container` - The container to which the name entry field and label will be added.
- Description: Adds an entry field for the player to input their name and a label to prompt the player, to the specified container.
- Returns: Void

12. display_hole_cards

- Function Prototype: `void display_hole_cards(int seat_number);`
- Parameters: `int seat_number` - The seat number for which the hole cards are displayed.
- Description: Displays the hole cards for a player seated at the specified seat number.
- Returns: Void

13. draw_rotated_text

- Function Prototype: `gboolean draw_rotated_text(GtkWidget *widget, GdkEventExpose *event, gpointer user_data);`
- Parameters:
    - `GtkWidget *widget` - The widget on which to draw the text.
    - `GdkEventExpose *event` - The expose event for redrawing the widget.
    - `gpointer user_data` - Pointer to additional data required for drawing, such as player label details.
- Description: Draws text rotated on the widget, used for displaying player names on their seats.
- Returns: `gboolean` - Returns FALSE after the drawing operation

4.3 Detailed Description of the Communication Protocol

The communication protocol facilitates communication between the server and clients in a structured manner. It ensures that data exchange occurs efficiently and reliably. Here are the key components:

1. Function Calls:
    - sendData(socket, data): Sends data over the network using the specified socket.
    - receiveData(socket): Receives data from the network using the specified socket.
    - establishConnection(address, port): Establishes a connection to the server at the given address and port.
2. Protocol Flags/Settings:
    - Handshake Protocol: When a client connects to the server, a handshake protocol is initiated to establish the connection. This involves exchanging handshake messages to confirm the connection.
    - Data Encoding: Data exchanged between the server and clients is encoded using a specific format (e.g., JSON, binary) to ensure compatibility and readability.
    - Error Handling: The protocol defines how errors are handled during communication, including error codes and messages sent between parties to indicate issues such as connection failure or data corruption.
    - Security Measures: Security features such as encryption and authentication may be incorporated into the protocol to protect data integrity and confidentiality.

Explanation of Communication Protocol Function Calls:

1. sendData(socket, data):
    - This function is used to send data over the network.
    - Parameters:
        - socket: The socket through which the data will be sent.
        - data: The data to be sent, encoded according to the protocol.
    - This function ensures that the data is transmitted reliably to the intended recipient.
2. receiveData(socket):
    - This function is used to receive data from the network.
    - Parameters:
        - socket: The socket from which data will be received.
    - It retrieves data from the network buffer and returns it to the calling function for processing.
3. establishConnection(address, port):
    - This function is responsible for establishing a connection to the server.
    - Parameters:
        - address: The IP address or hostname of the server.

        - port: The port number on which the server is listening.
    - It creates a socket and connects it to the specified server address and port.

Explanation of Communication Protocol Flags/Settings:

1. Handshake Protocol:
    - During the handshake, the client sends a connection request to the server.
    - The server responds with an acknowledgment to confirm the connection.
    - Once the handshake is complete, data exchange can begin.
2. Data Encoding:
    - Data exchanged between the server and clients is encoded using a standardized format such as JSON or binary.
    - This ensures that both parties can interpret the data correctly.
3. Error Handling:
    - The protocol defines error codes and messages to handle communication errors.
    - Error codes indicate the type of error (e.g., connection refused, data corruption).
    - Error messages provide additional details about the error to aid in troubleshooting.
4. Security Measures:
    - Security features such as encryption and authentication may be implemented to secure the communication channel.
    - Encryption ensures that data is encrypted before transmission and decrypted upon receipt to prevent unauthorized access.
    - Authentication mechanisms verify the identity of clients and servers to prevent unauthorized access to the system.

5. Development Plan and Timeline

5.1 Partitioning of Tasks

In order to best manage our time, Esthergen plans on having weekly goals of what tasks need to be done to ensure a complete and bug-free project by the deadline. Our goals include:

Week 6:

- Complete User Manual and submit by May 12th.
- Set up git repositories and set up folders.
- Create necessary .c and .h files.
- Start basic functions.

Week 7:

- Complete Software Specification and submit by May 19th.
- Create test environment.
- Create rules functions.

Week 8:

- Complete Poker Alpha Version and submit by May 26th.
- Socket connection.
- Create GUI.
- Unit test and debug.

Week 9:

- Complete Poker Beta Version and submit by June 2nd.
- Unit test and debug.
- Extra features.

Week 10:

- Software release complete by June 9th.

5.2 Team Member Responsibilities

The responsibilities for Esthergen Poker will be split approximately:

Artin: Game Design, Rules support, Server Program support, Poker Bot, Testing
Esther: Client Program, GUI, Documentation, User Manual, Software Spec, Testing
Noah: Server and Client Program, Poker Bot, Testing
Trang: Rules, Client Program, GUI, Testing

Copyright

References

No external sources were referenced in the creation of this document.

Index