**CIS Project 2: Old Maid**

Louis Harbauer

Riverside City College

CIS-17C-48784: C++ Programming - Data Structures

Dr. Mark E. Lehr

December 17, 2023

Table of Contents

[Starter Overview 3](#_Toc153741351)

[Program Overview 3](#_Toc153741352)

[Rules 4](#_Toc153741353)

[General Rules & Setup 4](#_Toc153741354)

[Card Ranks & Pairing 4](#_Toc153741355)

[Score Counts & Rematches 5](#_Toc153741356)

[Development Commentary 6](#_Toc153741357)

[Pseudocode 8](#_Toc153741358)

[Documentation 12](#_Toc153741359)

[Cards, Master Deck, and Player Deck 12](#_Toc153741360)

[card (struct) 12](#_Toc153741361)

[deckMaster (class) 12](#_Toc153741362)

[deckPlayer (class) 12](#_Toc153741363)

[Common Deck Functions 13](#_Toc153741364)

[crtCard(card) 13](#_Toc153741365)

[retCard(varies) 13](#_Toc153741366)

[ersCard(int) 13](#_Toc153741367)

[prntDeck() 14](#_Toc153741368)

[Scoreboard 14](#_Toc153741369)

[top\_plyr (struct) 14](#_Toc153741370)

[score (class) 14](#_Toc153741371)

[Main Space Functions 15](#_Toc153741372)

[main() 15](#_Toc153741373)

[game(deckMaster&, deckPlayer&, deckPlayer&) 16](#_Toc153741374)

[rules() 16](#_Toc153741375)

[testyyy() 17](#_Toc153741376)

[Game Functions 17](#_Toc153741377)

[turn\_plr(deckMaster&, deckPlayer&, deckPlayer&) 17](#_Toc153741378)

[turn\_cpu(deckMaster&, deckPlayer&, deckPlayer&) 17](#_Toc153741379)

[retPair\_AI(deckMaster&, deckPlayer&) 18](#_Toc153741380)

[pwdCard(int, int, deckPlayer&, deckMaster&) 18](#_Toc153741381)

[random\_rank(int) 18](#_Toc153741382)

[createPlr(deckMaster&) 19](#_Toc153741383)

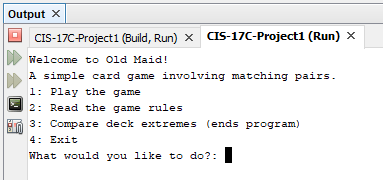
[drawCard(deckPlayer&, deckPlayer&) 19](#_Toc153741384)

[drawCard(deckMaster&, deckPlayer&) 19](#_Toc153741385)

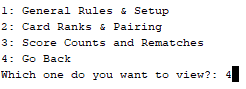
# Starter Overview

## Program Overview

**Old Maid** is a card game simulator written in the C++ programming language. It can be started through its separate executable in the “build” folder, or its source code can be run through an IDE such as NetBeans or Visual Studio. When starting the program, the user will be prompted with a menu that looks like this:



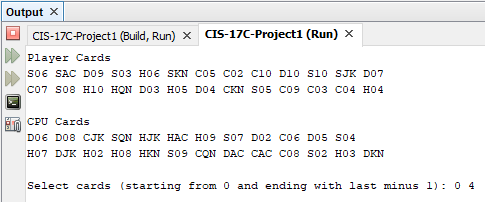
Selecting an option is rather straightforward. Inputting 1 starts the game and initializes the decks, 2 reads the game rules, 3 compares extreme options found in a deck, and 4 simply exits the game. Selecting 3 will also end the program, requiring the user to restart it if they want to return to the menu or select a different option. The Rules option allows the user to return to the main menu once they have understood how the game works, either after reading the final overview option, or in the main menu, which they can also return to after reading options.



## Rules

All content as follows is from the “Rules” option in the menu, with appropriate accompanying visuals:

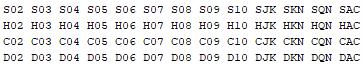
### General Rules & Setup



The master deck has 52 cards. Each player gets one half of the deck, or at least as much of the half as possible. Each player removes a pair from their deck as the game goes along. After removing a pair, they draw a card from the other player's deck. After both players have drawn their cards, the player will be prompted to swap decks.

### Card Ranks & Pairing

The master deck typically looks like this:



One Queen card is removed from the deck to make the deck uneven when distributing. A pair of cards includes two cards of the same rank, such as pairing C02 and H02. If a player has more than two cards of a specific rank, they only return two of those cards, leaving only the second pair if they had four of a kind, or a single unpaired card if they had three of a kind. The player who is left with the odd Queen card is considered the Old Maid, and loses the game. If a player is left with three unpaired cards instead of one, they also lose the game. Invalid card pairings will be noted by the interface, as the text "Invalid pair!" will show up if a pair has non-equal rankings.

### Score Counts & Rematches

When a game ends, the loser and their method of losing will be declared by the interface, and the total score between the two players will be displayed. The interface will ask if the player wants a rematch; if they answer with 'Y', a rematch will begin, and the decks will be redefined. If they answer with 'N', they will go back to the main menu, where they can exit or view the rules. Starting a new game without declaring a rematch will reset the score between both players.

# Development Commentary

I started Old Maid back in October or so, making the lone project at the start of the month and implementing features just before I started work on my midterm. I had started out by making a list for the master deck and a list for two player decks, but because I learned that lists could not be shuffled, I wound up making a deque for the player decks based on an external suggestion. Because I was preoccupied with the midterm, I was not able to start work on the project again until a lot later than I had hoped, really only giving me about a week to work on it to the best of my abilities. I was able to get a decent chunk of it done before realizing some flaws in my AI workings, more specifically how it handled recognition of pairs, more specifically calls for ones of a random ranking. I wanted it to recognize how much was actually available, but at times it would fail to work within the limits of the given deck, stalling about halfway through a typical game.

I initially thought converting Old Maid to the C++ workings would be rather simple as it is a game that involves elimination and pattern recognition, but it proved to be a lot more work to perfect. My initial estimation was that players would constantly eliminate pairs and try gaining new cards from the master deck, with the player who receives the odd queen card, or removing an invalid pair, losing the game. This proved to be somewhat impractical, even with remnants left in the code, so I reread the rules and bettered my estimate, with gameplay then becoming the player and the AI drawing from each other’s decks (rather than the master deck) after removing a pair, and swapping decks when given the opportunity. This seemed faithful enough to the rules available on the internet; I only had the human player and the CPU, which was only about two players for the game in total, so there was not really much to work with.

The new approach to gameplay definitely helped my personal workflow. I was able to figure out some problems faster than others, like distributing the cards from the master deck into the player and CPU decks, and adding onto what was already established became faster as I kept working. I was able to work with my AI well enough that it would start to recognize pairs better rather than calling two of one card or two unrelated cards, although I still ran into the problem of having it stall for the time it did, and may want to fix it in a future revision. I was able to get the game into a somewhat playable state, even if there were some hiccups with how the AI was managed, and how well I implemented my structures and algorithms. It allowed me to work with the game better, and helped make work more efficient.

Working on Old Maid was an interesting challenge. It was a game that I thought would be quite simple to convert, but wound up being more advanced than I anticipated. There were a lot of hoops I had to jump through in order to make it work in the way that I did, but it was worth learning how to do it properly, even if some parts may have not been up to par. Once I learn more about coding, I may want to look back at what I did wrong and see if I can fix it, and how I can improve my work in the future. I did not expect it to turn out this way, so I hope others can enjoy it or help make it better than I did.

# Pseudocode

Some of this is very different from the final product.

**Creating deck:**

* A two-dimensional list. Yes I'm sure this is something that can really be done. Stop asking me
* Read (blank) sublist element from master list
* Determine card type:
  + If element 1, set to Spade
  + If element 2, set to Heart
  + If element 3, set to Club
  + If element 4, set to Diamond
* Determine card ID:
  + If under 8, create stringstream with any number above 2, accompanied by a fake 0
  + If 8, set ID string to 10
  + If 9, set ID string to JK (Jack)
  + If 10, set ID string to KN (King)
  + If 11, set ID string to QN (Queen)
  + If 12, set ID string to AC (Ace)
* Push card element back to sublist

**Creating player/CPU deck:**

* Read cards from master deck
* Draw one random card (no queens) from master deck and search if it is already in player deck
  + If not in player deck, add card to deck
  + If in player deck, skip
* Shuffle final deck
* Return final deck

**Main:**

* Greet player ("Welcome to Old Maid!")
* Three choices for player:
  + 1 starts up a game
  + 2 provides a basic explanation of the rules (as it can get with these CLI thingies)
  + 3 exits the program

**Playing the game**

* Initialize master deck
* Remove one queen from master deck (51 card deck)
* Initialize player deck
* Initialize CPU deck
* Set turns to 0
* Loop until end of game as needed:
  + Show player cards
  + Show CPU cards
  + Call player turn function if even number
  + Call CPU turn function if odd number
  + Increment turn value
* Declare winner:
  + If player only has odd queen, CPU wins
  + If CPU only has odd queen, player wins
  + If player has three unpairable cards, CPU wins
  + If CPU has three unpairable cards, player wins
* Ask to play again
  + If yes, start game again
  + If no, return to menu

**Typical turn (player)**

* Ask player for cards to remove (from beginning of deque)
* Remove specified cards, or don’t if the same card is called
* Draw card from opponent
* Ask to swap
  + If yes, swap player deck and CPU deck
  + If no, leave decks as-is
* End turn

**Typical turn (CPU)**

* Run CPU turn logic (loop checks for pairs)
* Remove specified cards, or dont if the same card is called
* Draw card from opponent
* Randomize swap outcome with low chances
  + If wants to swap, swap player deck and CPU deck
  + If does not want to swap, leave decks as-is
* End turn

# Documentation

## Cards, Master Deck, and Player Deck

### card (struct)

Each **Card** is a structure with a character for **types** (Suits) and a string for **IDs** (Ranks). The Card structure has a variety of comparative operators; equal and non-equal operators compare **types** and **IDs**, while greater than and less than operators only compare **IDs**. When overriding a card, it swaps the **type** and **ID** of a card with another card specified in the operation.

### deckMaster (class)

The **deckMaster** class houses the 52-card deck the player decks initially pull from. The cards are housed in a two-dimensional list **(IntDeck)**, with 13 cards for each suit housed in 4 dimensions; **dimension 0** is reserved for spades, **dimension 1** is reserved for hearts, **dimension 2** is reserved for clubs, and **dimension 3** is reserved for diamonds. The master deck is initialized through defining cards through its **defCard(i,j)** function, with each defined card being pushed back to the sub-list in the current dimension. The sub-lists featuring the cards for each suit get pushed back to the **IntDeck** list.

### deckPlayer (class)

The **deckPlayer** class houses a double-ended queue **(PlrDeck)** that stores as many cards from the master deck as possible. Unlike **deckMaster**, it is initialized outside of its own class, but still has flexibility through uniqueness in functions. Player decks of the **deckPlayer** class can shuffle their own contents through **shfDeck()**, find a specific card through **srcDeck(card)**, swap with another **deckPlayer** deck through **swpDeck(deckPlayer&)**, and determine if a player is the Old Maid through **maid()**, which compares the maximum ID to a predefined queen card. Also available is **meido\_un()**, which compares values in a three-card deck to see if they are unpaired, which similarly declares a player the Old Maid. **srcDeck**, **swpDeck**, **shfDeck**, and the minimum/maximum comparisons of **maid()** all use their respective STL algorithms to carry out their jobs.

## Common Deck Functions

### crtCard(card)

**crtCard** creates a new card for the deck. In the **master deck**, it pushes the card back to the dimension of its suit; it examines the suit (the **type**) of the provided card and checks an iterator to see if it has reached the desired dimension before pushing. In a **player deck**, it simply pushes the card back to **PlrDeck**, as **PlrDeck** is a one-dimensional deque as compared to **IntDeck**’s two-dimensional list..

### retCard(varies)

**retCard** returns the card listed at the provided index. The **master deck** and **player decks** share similar functionalities when it comes to returning cards; they cycle through an iterator before reaching the desired index, and return the card listed at that index. The master deck uses two iterators, **it** and **it2**, due to the fact that **IntDeck** is a two-dimensional list, and requires a nested **for** loop. Player decks are much simpler and use one iterator, **it** by itself, due to **PlrDeck** being one-dimensional.

### ersCard(int)

**ersCard** looks for a card in the provided index and simply erases it. Similar to **retCard**, both function similarly with minor differences: they cycle through an iterator before reaching the desired index, and erase the card at that index from their decks. As before, the master deck uses two iterators to look for the card as **IntDeck** is two-dimensional, while player decks use one iterator due to **PlrDeck** being one-dimensional.

### prntDeck()

**prntDeck** does not serve as much of a purpose as the other functions, as it does not add, remove, or return any value from the decks. It simply prints what cards are available in the deck via the necessary iterators; master decks use two iterators due to **IntDeck** being two-dimensional, while player decks use one iterator due to **PlrDeck** being one-dimensional. Both use the iostream **cout** function to print the values, printing the suit **(type)** of the card before printing the rank **(ID)** of the card.

## Scoreboard

### top\_plyr (struct)

All top players in a scoreboard follow a structure that includes their initials and their playing score. If the player gets enough wins in a game to surpass the player with the highest score, they can enter their initials into the board, and a new player of this structure will be created, surpassing the last.

### score (class)

The **score** class is a graph that keeps track of the scores of top players. It has three functions for entering a player into the board, two private and one public; **hash\_inits** simply hashes a name for placement on the board, **make\_plyr** simply makes a player based on their wins against the CPU, and **add\_plyr** adds the player to **board**, the array of lists where top scores and players are kept. There are also options to display the board (which is also seen in the main menu) and to save the contents of the board to an external file (usually after the user is finished with the program). There is also a constructor for the class, which takes scores from a pre-existing file and puts them into the new board.

The scoreboard constructs itself by loading scores from a file and placing it into its graph. It hashes the initials using **hash\_inits** before pushing them back to the board lists, until the end of the file is reached. **max\_scor()** determines the highest possible score in the graph, which is then used to tell the player if they got a higher score than the last winner, which is then saved to the scores.txt file. **disp\_board()** and **printGraph()** display all of the high scores found on the board, with **printGraph** showing where they are in the graph, and **disp\_board** showing them based on their current standings.

## Main Space Functions

### main()

The **main** function is the primary frontend for the game itself. If the user wants to play a game, it initializes the master and player/AI decks before calling the **game** function, which returns a value based on the winning player. After the game is finished and scores are recorded, the function prompts a rematch from the user, to which they can choose to accept or decline. If the user wants to read the rules, it calls the **rules** function, which does not return a specific value, but allows the user to read through the rules as they are presented. The third choice returns **testyyy**, a debugging function that reads out the minimum and maximum ranks available in a master deck (but also ends the program), and the fourth choice simply ends the program. Entering an invalid value in any prompt will prompt the interface to print *“Please enter a valid option and try again,”* before repeating the prompt and the input for the prompt itself.

### game(deckMaster&, deckPlayer&, deckPlayer&)

The function that works as the game itself, **game**, sets itself up by initializing values that determine the validity of pairings, who is the Old Maid, the current turn of a game, deck sizes, and the indexes to locate for a pair. It constantly looks to these values to continue the game, but will also print the player and AI decks before prompting a turn function from either. Once both players have taken their turn, the player is prompted with the ability to swap decks, to which they can accept or decline. Once some kind of endgame has been reached, the interface will print who won and why, alongside return a value based on the winning player to the calling function. If the endgame is invalid, the interface will instead print *“Somebody goofed. whoops,”* before returning a predestined negative value that indicates the game failed to work out.

### rules()

The function that provides the player with the rules, **rules**, only initializes the user’s choice of rules to view, but later initializes confirmations as their specified rules get loaded into the interface. A majority of the rule code is simply printed output except for user choice prompts; if the user makes a choice outside of specifications, it prints *“Please enter a valid option and try again,”* and repeats itself similarly to the **main** and **game** functions. It does not return any values, and simply ends on user demand.

### testyyy()

**testyyy** is a debugging function that compares the minimum and maximum values of a hypothetical player and CPU deck. It creates its own master deck, player deck, and AI deck, alongside erasing one of the **Queens** of the master deck. It uses the maximum and minimum functions of the player and AI decks to determine their values, and outputs the cards those functions call. It does not return any value, but the program ends when it is called in **main** to avoid memory leaks. I created this function to test out if the minimum and maximum functions worked, and they did because the **Queen** cards were usually the maximum in both decks.

## Game Functions

### turn\_plr(deckMaster&, deckPlayer&, deckPlayer&)

**turn\_plr** is the primary function for a player turn. It prompts the player to select the cards they want to return; it repeats the prompt if they only return one card, and gives an invalid pairing warning if the pair they returned was not legitimate. After returning the cards, it asks them if they want to draw a card, which calls the two-player version of **drawCard** if they oblige. The turn ends once they have returned and drawn their cards, returning 1 to indicate a successful turn.

### turn\_cpu(deckMaster&, deckPlayer&, deckPlayer&)

**turn\_cpu** is the primary function for an AI turn. It uses a variable, **plr\_true**, to call **retPair\_AI** to indicate if the AI has made a successful pair return. If the AI made a successful return, it draws a card from the human player using **drawCard**, and ends its turn as the card is drawn.

### retPair\_AI(deckMaster&, deckPlayer&)

**retPair\_AI** is a function that allows the AI to return a pair. It uses a **while** loop to call for a specific rank to return a pair of using the **rank** variable and the **random\_rank** function, before using the **ringos** map to determine the indexes of the cards to return. If the rank is between the minimum and maximum ranks of the player deck, it starts to mark down the cards to return using **ringos** and a **for** loop; it first marks down the first value of **ringos** by the virtue of the index at the loop’s counter matching the rank defined by **rank**, before checking to see if any other values share the rank and marking the first one down. If both indexes of **ringos** share the same rank, the cards get returned; if they do not, the **while** loop repeats. On a successful pairing, the interface notes which cards the AI returned; if not, it simply notes it returned no cards.

### pwdCard(int, int, deckPlayer&, deckMaster&)

**pwdCard** is the primary function used to return pairs. It uses a variable, **truedo**, to determine if the pair is invalid or not, and sets the variable to 1 if the indexes are not equal, and both cards at the indexes share the same rank. If the first integer given is greater than the second, it sends the card at the first number before sending the card at the second number; if the second is greater than the first, it sends the card at the second before the card at the first. The function returns the value of **truedo** for its caller to verify if the return was valid.

### random\_rank(int)

**random\_rank** simply returns a rank string based on the integer it is given. If anything below 8 is given, it simply adds 2 to the provided number alongside an extra 0, and 8 itself returns a 10. Giving 9 returns a **Jack**, 10 returns a **King**, 11 returns a **Queen**, and 12 and anything above returns an **Ace**.

### createPlr(deckMaster&)

**createPlr** creates a player by obtaining cards from the provided master deck and looking to see if they already exist in the new player deck **(newdick)**. If a card is not already in the given player deck, it creates the card in the player deck; the process of verifying repeats until the size is below half of the master deck, or the total size of the master deck is nonexistent. The new player deck is returned to the caller.

### drawCard(deckPlayer&, deckPlayer&)

**drawCard** is a rather simplistic function used to draw a card from the opponent’s deck. It looks for a card within a range of 12 (or less depending on the deck size) and searches for the card in the drawing player’s deck. If it does not find the card, it adds the card to the drawing player’s deck by erasing the card from the opponent deck. If not, it reports an unsuccessful test. The card that was added to the player deck gets returned to the caller.

### drawCard(deckMaster&, deckPlayer&)

This older version of **drawCard** was used to draw cards from the master deck rather than the opponent’s deck. It looks for a card within the 52-card range (or less depending on the deck size) and searches for the card in the drawing player’s deck. If it does not find the card, it adds the card to the drawing player’s deck by erasing the card from the opponent deck. If not, it reports an unsuccessful test. The card that was added to the player deck gets returned to the caller. This was used in older revisions, where drawn cards would come from the master deck rather than the opposing player’s deck, but was eventually retired when re-thinking how the game worked.