Charlie Bauchop - Last Card Internal

Planning:

Checkpoint 1 - 20th March:

By the 20th of March I will hope to have completed my functions for user input, the card structure and began working on the methods of my deck class. I will test the user input by trialling strings and other input types to see how it would handle them when it would expect an integer. I will test my card structure by setting and outputting the cards to see if any unexpected situations occur. And I will test my deck card methods by checking whether the deck is shuffling correctly and whether taking cards from the stack would work as I would intend it to.

Checkpoint 2 - 27th March:

By the 27th of March I hope to have completed the methods for playing and gaining cards, as well as having the functionality for special cards including jokers. I will test playing and gaining cards at the same time by enacting turns to see if the player can play a card as well as the discard pile gaining cards. I will also complete checks to confirm the special cards produce the intended outcomes

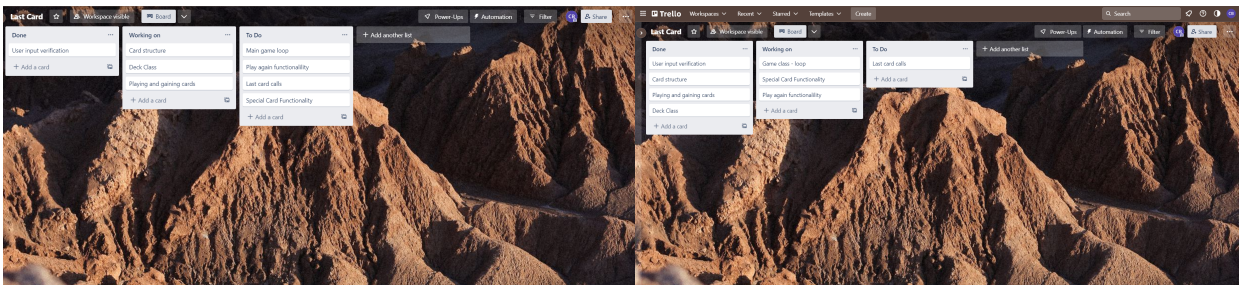
Checkpoint 3 - 3rd April:

By the 3rd of April my aim is to complete the programming aspect of the project, And would then hopefully begin further testing and debugging on any issues that may arise. I will run through multiple instances of the game as well as letting fellow peers and family members test the game to uncover any issues and bugs to further refine the project.

Evaluation of planning - 20/4:

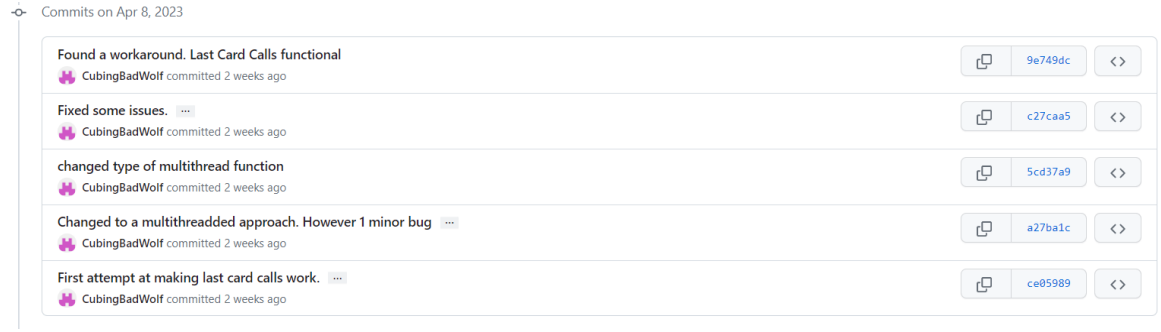
Whilst I made good progress on keeping to the first 2 checkpoints, I fell behind on the 3rd checkpoint, needing until the 8th of April to complete the last components. These components being the last card calls and chained + twos. Having these checkpoints were a good indication of how I was progressing with the project. I also used a Trello board to manage the individual tasks and to keep the progress flowing.

To record the trialling and testing that took place, I documented each step, and have collated this information into the table below. Further details can also be reviewed in the Version Control software of my github repository, which allowed me to maintain and use the testing to change details to my project. And keep track of these changes to identify where I may have introduced bugs.



- Snapshots of the trello board during the project working

For example, on the 8th of April I started working on the last card calls, the use of the version control software in my github repository shows the changes that I made between versions as the result of testing.



This shows the changes I made between versions and the tests that led to these versions being developed. The full commit history on the repository shows all the changes to the documents in an ordered list.

Trial and testing timeline:

Date	Action	Reason/Steps taken	Expected Outcome	Observed Outcome
14/3	Added header files for main components	I needed to organise the program into its key components before I could start programming them	No expected outcome	Nothing was run
14/3	Wrote code for user input and verifying that input was an integer	Checking whether someone enters an integer or not is an important step in error prevention. I also tested this by trialling both the one with a range of values and the default generic integer range	I would expect the program to tell me if I didn't enter an integer and ask for reinput	It worked as intended[1]. Returning the integer value and in the case of non integer it would ask for reinput

[1] I later found an issue and fixed this when identified

15/3	Add code for a struct to represent an individual card	Cards needed to be an object with both a value and a suit	When passing values into the constructor it would create a card with the values defined	Success
17/3	Add framework for deck class	I decided what methods a deck would need to have to function and added them as framework for a class	No expected output as code wasn't run	n/a
20/3	Wrote code for the shuffling, gaining and taking cards from a deck	These methods are important to the game and needed to be implemented	I had not tested these functions yet so no outcome was expected	n/a
20/3	Identified issue with user verifications and handling floats as integers	I decided to treat floats as a non int input so I rewrote the function to handle this	I tested this new function with an int, a float, a negative number and a string and checked how it would respond	It worked as intended returning the int value and asking for reinput for the other cases
21/3	Deck.cpp: Wrote code to print the current hand in the deck Main.cpp: Wrote code to initialise the 4 decks needed and deal cards, and test the print function	I wrote code to deal 5 cards to each players hand then used the print function defined in deck.cpp to test whether both methods were working	I expected to be greeted with the 5 cards in each hand.	This was the case, the cards were being dealt properly and then printed properly as well
22/3	I decided to add a new component to handle the game itself as a game class. I chose to do this because if I kept all the game functions in the main file it would have needed me to pass	I copied the code previously written in the main.cpp file into the game class for handling the game	I expected for the code to run the same as it did yesterday.	It ran the same as it did yesterday.

	the piles through all the functions, where if this was a class this isn't necessary.			
22/3	Wrote code for playing cards, and for the game loop itself	The program for playing cards would be run within a game loop function in the game class. Simulating a turn based system and allowing for drawing cards as well.	For both the player and computer to successfully play cards and draw cards if they could not play a card	After a couple of tweaking to the if statements this worked as intended with both the computer and playing playing cards based on input for the player and first available for the computer as well as drawing a card if unable to play (however would constantly draw cards even when able to play sometimes (I identified that it could play when the card being played was first in hand)
23/3	Added win condition when a player/computer has 0 cards. Fixed bug where computer would gain cards despite being able to play	When a player/computer has 0 cards remaining the gameloop should return true ending the game I wrote by error != instead of == meaning computer gained card when able to play instead of not gaining when unable to play	For when the game is over it should return true and break the main loop For the computer to only gain cards when unable to play	Success Success

24/3 & 25/3	Used a switch case and referenced booleans to implement the logic for special cards	Using booleans passed by reference I can allow the play card function to modify variables defined in the game class which allows me to implement logic in the gameloop for the outcomes of playing the special cards	Booleans changed in game class when special cards played	Success
28/3	Added jokers to deck and added their functionality to special card case	Set up an additional case for a card with numeric value 14 which creates a joker	Would create a card of value ‘j’	Didn’t work had a value of ‘>’ hence failed comparison tests that expected ‘j’
			Would result in picking up 6 and changing suit	Couldn’t test due to earlier error
				When a joker was the top card of the discard pile at the start of the game it would break as no cards could be played on it - Found after multiple tests
30/3	Fixing issues with jokers	I fixed the issue with jokers being represented by > instead of j by adding a missing break; statement that I missed after analysis	Errors fixed	Success + pick up 6 and suit worked as well using earlier code from 2s and aces
		To fix the other issue with jokers being unable to		

		played on when played initially I instead we wrote the constructor so jokers are added to the deck after the first card is taken from the draw pile		
1/4	Added play again feature	When the game concludes ask players whether they want to play again y/n	Once game complete ask for y/n if y play again if n terminate program	success
	Fixed players always starting with jokers	Jokers were not being shuffled before dealing - always on top of stack hence always dealt Add shuffle	Jokers no longer always being dealt to players	success
	Added delay between turn	When testing often I was confused as to what was happening especially when I went long periods without being able to play cards and having 5 cards immediately added to hand	Wait for 500 ms between turns. Have effect when chained pickups occurring - not instantly gaining cards	Success - gained cards with delay so was clearer to understand what was happening
4/4	Add chained plus twos	Add a new flag called mustPlay2 to allow for playing of 2s on top of 2s - check this when code would normally give you two	Would allow for twos to be chained onto each other making the penalty increase by 2	ERROR - wouldn't compile

		cards. I used code previously used for playing a two on its own and put it behind an if statement checking if mustPlay2 was true		
5/4	Tried fixing the error above	I identified through looking through my code that the error was comparing the card played with a constant card with a value of 2. In order to fix this I manually overloaded the == operator in the object to allow for this to work	Code would now compile	Code compiled however did not work as intended
6/4	The issue I was having above I identified as being a looping error that I would be correctly asked to play a two if I had one but would not stop asking once it was played	After testing I noticed that this was only happening when the player was playing, not when the computer was. I discovered this by debugging my code and stepping through line by line to see what was occurring. I then checked the code for differences and found that I had a mustPlayTwo = false; line in the computer section but not	Chained twos would work	Success

		the player section. I then added that to the player's section		
8/4	Add last card calls	This needs a whole paragraph to discuss the trials, tests and subsequent changes (See below in main component description section)	When one card entering any input would call last within a time range	Eventually worked (see description on last card for more details)
17/4	Fix bug with play again	During further testing I discovered that when entering longer edge cases when playing again instead of y/n it would check characters one at a time until it found a y or a n	I fixed this by changing the input variable to a string from a character this allowed me to check the whole string instead of just one character at a time	Success now correctly asked for reinput when longer string was inputted instead of checking whether a y or n was in the string
19/4	Fix a new bug with play again.	Sometimes rarely I would have to enter an input twice because the first input wouldn't register	I fixed this by clearing the input buffer as sometimes it would contain leftover input so it wouldn't correct check the first input after asking	Success
20/4	Fixed bug with joker suit being valid suit when changing	Entering 4 as an edge case when 3 was meant to be the max value	4 would now correctly trigger input range check	Success - see video_1 for when I discovered this bug

This table only covers the main functional changes, not including any changes made for grammar and code clarity reasons - These types of changes can be seen when viewing the commit history in my github repository. This version control software allowed me to keep track of my changes which would allow me to reverse any changes that may have caused issues.

FullGameTest.mkv shows me running through the entire game testing edge cases for inputs.
Video_1.mkv shows me testing and discovering a bug.

Components descriptions + changes made due to testing:

User Input - Verification:
The component that deals with verifying user input is located in my VerifyUserInput.hpp header file. It is located in a header file so I can store it and include it in all the files that need it. This file contains two different variations of the input verification function which uses overloading to define the function. This is so it can deal with not only checking if the input is purely an integer, but can also check if said integer is within range, if this range is defined as a function parameter. It does this by iterating through all the characters of the input and checking if any of them are not a digit. If this is the case it would return this to the user and ask for re-input. If the range is found it will then check if the number is within this range, and if it is not then it will return to the user and ask for re-input.
When I was testing this function I found that the range would not deal with the edge case of trying one more than the range. This is because I had used a <= instead of <. By correcting this issue the function works as intended. I also tested its handling on strings and negative numbers and it dealt with this as intended. However, I had to make additional changes when testing with floats, because by default when converting say “5.2” to an int in c++ it will succeed, but instead truncating, so instead of doing a convert to int and if fails this conversion, I would ask for re-input I would instead check for any non digit characters which fixed this issue.

Card Struct:
In order to make a card game, you need to have a way of storing and displaying the individual cards in the game. In order to do this I used the ‘complex programming technique’ of using classes and object oriented programming to store the values of the card. The card is a struct which has 2 values for the picture value and the card suits. This card has a constructor which takes two numerical values (ints) and uses those values to generate a card with a character for the picture value {A,2,3,4,5,6,7,8,9,T,J,Q,K} and a character for a suit {H,S,D,C}. When making this card struct I included a default constructor which makes a blank card in case the necessary values were ever omitted to ensure safety.

When I realised that I would need to add jokers to the decks I had to account for this in the card struct. I did this by adding an additional input for my switch case which assigns the picture values with a value of 14 and a suit value of 4 which would correspond to the joker: j.
I did have an issue with the joker not working properly at first and was instead showing the character ‘>’ instead of ‘j’. This was because I had missed a break; statement in my code for assigning jokers so instead of just setting the picture value to ‘j’ it would then cause undefined behaviour by reassigning the number again using the default statement of setting picture value

to the character version of the number inputted. I fixed this by testing and identifying the missing break; statement and adding it.

Deck Class:

The deck class is the way I am representing all the groups of cards that exist in a game of last card. For a 2 player game there are 4 unique piles of cards. The draw pile, discard pile and as well as each player's hands (one for the computer and one for the player). The deck class must be capable of handling all the actions that a real card deck can before hence I will be using methods in the class to handle this. The deck class also contains a std::vector which contains all the cards currently located within the deck. I used an std::vector because it has dynamic sizing but also includes inbuilt garbage collection so I don't have to deal with any advanced memory management by using dynamic arrays.

Deck Class - constructor:

The constructor is what initialises the class and handles any logic required at initialisation. The constructor takes in a boolean value {true or false}. This boolean determines whether the constructor needs to create all the cards inside the deck. It does so by running a for loop to iterate for all numbers 1 - 13 and all suits 1 - 4 corresponding to the 4 suits in a pack of cards.

Deck Class - shuffle:

The shuffle method simply calls std::random_shuffle to randomly shuffle all elements of the vector containing the cards. This is an important feature as without the ability to shuffle a deck you would have the same gamestate over and over again which would become predictable

Deck Class - Take Card:

The take card function is the method that allows for the selection of a card to be removed from the vector of cards and return it from the function so future functions and variables can store these values and use them to advance the game.

I used function overloading to define two versions of this function, one takes an index position on the vector of cards to select that card and removes it from the vector and returns that card. And the other function takes no input and instead will remove the card from the stack and return that value instead. A stack is a 'complex programming technique' which operates on the basis of "first on, first off" meaning that the card at the back of the pile would be the card removed from the vector and returned from the function. When thinking about this, this is how a real life pile of cards would work as you have a stack of cards that you are removing from during the game.

Deck Class - Play Card:

The play card method is what allows both the player and the computer to take cards from their decks (hands) and play them onto the discard pile and execute any of the special actions from the power cards, It does this through use of several boolean variables passed to the function via reference from the game class. I chose to pass them via reference because that allows for the value to be changed inside the deck class function, and the changed value to take effect in the game class. These variables are drawTwo for when a 2 was played, jokerPlayed for when a

joker was played and the next player must pick up 6, skipNextTurn for when a Jack or an Eight were played. These values when passed by reference allow the two classes to communicate to each other, and to ensure that when I play a special card the next player will face the result of the special cards action.
There is another variable being passed into the function by reference as well, this being the nextSuit variable. This ensures that whenever an ace or a joker changes the suit from whatever the original suit was, it would then be conveyed correctly, and that the new suit will be what is played upon, not the original suit. I also pass a boolean called computer which indicates whether the computer or the player is playing, which allows for the computer's software to make decisions independently from the player.

Now I have all these variables defined and passed by reference, I can play cards. First I check for whether the player can play a card by checking all cards in their hands for any matches between card and the facing card of the discard pile's number and suit. I also check whether a player has a joker which can be placed on anything. If they can, I run an user input on which card to play if it's a player or pick the first available card if it is the computer, and call a switch case which carries out the actions for the special cards and defaults to just returning the card if it is not a special card.

When testing this I realised that I did not have any functionality for playing 2s on top of eachother which is defined in the rules of the game. In order to achieve this I added another boolean passed by reference called mustPlayTwo which means that if a 2 was previously played they must only play a two. I had a bug when testing this because I was not setting this value to false after playing a two on top of a two, which meant I was being asked to play a two even when I didn't have any. I fixed this by identifying where this error was occurring and adding the line mustPlayTwo = false; to correct this issue.

Deck Class - Gain Card:
This was a very simple method to introduce. The gain card function only has to take a card as a parameter for the function and add it to the vector of cards. This uses the stack to call std::vector.push_back() to add the card to the back of the vector and hence on top of the stack. By putting it on top of the stack we can clearly see what card is being added to the player's hand when printed because it would be the last when printed. Placing it onto a stack is also important for the discard pile because you need for the last played card to be placed on top of the discard pile, so you can see which card was last placed, and use the values to check whether a card can be played on top of it or not.

Deck Class - Print Cards:
The print card function uses a for loop to iterate through every card in the classes vector of cards to output the cards to the terminal. It also prints the index of the card to allow the user to easily identify the index to select whenever they are to play a card.

Game Class:

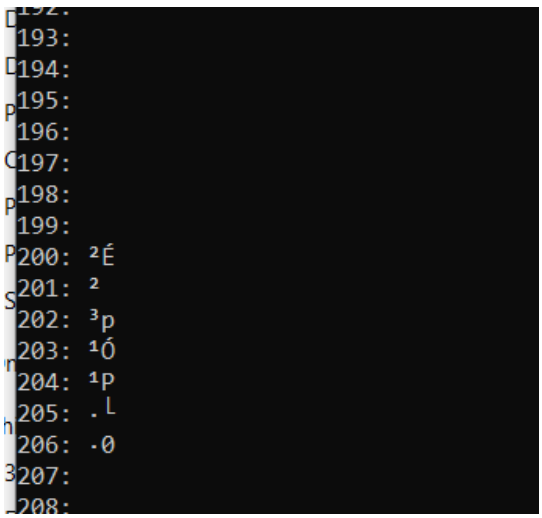
When I originally started making this card game I wasn't intending on making a separate class for the game, and instead was going to run this inside the main function of the file. But after considering that I was going to have to pass all decks into every function for the game to run, I decided that it would be easier to instead use a single class with the decks as member variables to handle this. I believe I made the right decision as it not only improved the legibility and flowingness of my code, but also reduced the number of variables needed to be passed from function to function in my code.

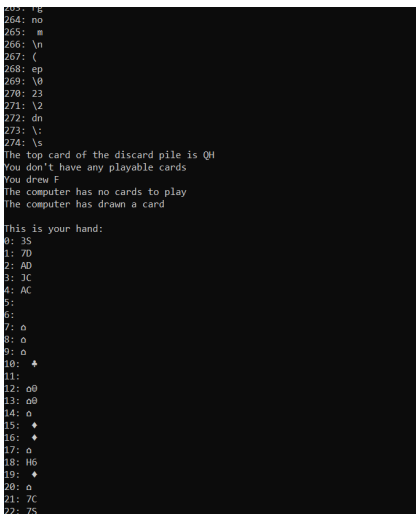
Game Class - Constructor:

The constructor of the game class initialises the four decks needed for the game (Draw Pile, Discard Pile, Player Hand, Computer Hand), as well as shuffling the draw pile and taking the top card to be the first facing card of the discard pile. When testing this I discovered an issue that in very rare circumstances when a joker was the first card in the discard pile. The program would be unable to continue because a joker technically has no suit hence only another joker could be played upon it. To fix this I made this card be taken and added to the discard pile then added the jokers to the deck, shuffled it and took 10 cards from the draw pile adding 5 to each player to start the game.
This all takes place inside the constructor so it is automatically done whenever the game is initialised.

Game Class - Flip Discard Pile:

When testing my code I realised that in cases where the game lasts a long time, especially when jokers are used to make players pick up 6 cards, we would potentially run into cases where there would not be enough cards left in the draw pile to perform actions. This resulted in segmentation faults, as I was trying to access memory that I shouldn't have access to, which caused a lot of errors to occur {see images}. I then decided to implement a function to mimic the behaviour of flipping the discard pile to create a new draw pile that occurs when cards are running low in a physical game of cards.





In order to prevent this from happening constantly I decided to limit this to only occurring when there are 2 cards or fewer in the draw pile. If this is the case, then it will iterate through the discard pile from the front of the vector taking the card and adding it back to the stack of the draw pile until only one card remains, which will

remain as the face up card of the discard pile to continue to be played upon.
This prevented any future segmentation faults caused by accessing memory I shouldn't have accessed due to running out of available cards in the draw pile.

Game Class - Last Card Input:
When a player has only one card in a game of last card they must call last card to be able to place the card else they have to draw a card instead. This was a challenge for me to implement as I needed to account for not only taking input to receive a last card call, but also timeout after enough time had elapsed to account for a player forgetting to call "last card". I at first tried using a while loop taking input until a variable increasing overtime would reach a certain amount of time. However because std::cin (the standard input in c++) is blocking, this would not timeout as it prevented the timeout variable from increasing until input was taking in effect giving the user infinite time to call last card. After thinking about the problem, the only solution I came up with was to use multithreading to handle the two separate tasks which would occur at the same time. So I wrote a thread to handle taking input whilst the main thread counts 3 seconds. If the input was received before 3 seconds was elapsed it would return true but if the 3 seconds elapsed before the input was received it would tell the user and make them pick up a card and then continue on with the program. I did have issues with the program not continuing correctly but after consideration of this problem I realised it was due to having to rejoin threads post completion, which would once again wait for input. This was an issue that after some research could not be solved using only the standard library, which I wanted to use because of its platform independent nature. In order to minimise this issue I changed some details to tell the user to press a key to continue once they are informed of the timeout. Whilst this isn't ideal it is the best I could come up with, without resorting to using platform dependent code which would have restricted the program to a specific operating system which goes against what I wanted to achieve with this program.

Game Class - Gameloop:
The gameloop function is what ties all the previous components together. It operates in two phases, one for when it is the players turn to play a card, and when it is the computers turn to play a card. During each of these phases it checks for all the flags for when a special card was played (draw2, draw6, skipTurn), calls the necessary functions for the player to play a card, and also checks for whether the player has one or zero cards left. If they have one card left the last card call function is called, and if they have zero they have won the game. Then it does all this for the computer as well. I have defined the logic for player/computer separately because it allows me to handle the different grammar and outputs that change for when the computer plays as opposed to when the human player is playing. After each pair of turns it sleeps for 500 milliseconds so it doesn't instantly jump through when no cards could be played and instead shows the user what is happening turn by turn rather than all at once. Once a player has zero cards it will return true so the main function knows to stop the game.

Main File:
The main file is a small file which only contains the minimal amount of logic to run the game. It provides the starting welcome/information to the user then it initialises the game class which

runs the game constructor creating all the information needed for the game. It then runs a while loop until the gameloop function returns true after which it asks the user whether they want to play again.

Main File - Play Again:
The play again component is the final component in the project. It uses a while(true) loop to take user input and check whether it is equal to y or n (case insensitive). If y then it uses recursion to rerun the main class which begins the game again. If n then it will return 0 indicating that the program has exited successfully which exits the program. If neither of these then it will inform the user that it didn't match what was accepted and asks for reinput.

Relevant Implications:

Accessibility;
Accessibility is the measure of whether a program is accessible to a wide range of users across multiple backgrounds. In order for a program to be accessible it has to be available to any user who'd want to use it despite their platform or requirements. In a c++ program you might begin to have some issues when checking whether the code is compatible with all operating systems and cpu architecture. This is because all platforms use different primary programming, for example, Windows often uses x86 assembly, but this will not be compatible with other architectures that cannot run x86 assembly. In order to ensure the program is platform independent for a c++ program you can ensure that you only use the standard library. By using the standard library only, you ensure that the program is able to be compiled on any operating system on any computer. For example there were additional things I would have liked to implement into my program but couldn't because there was not a platform independent way of doing it. This includes clearing the terminal after each game when playing again. Whilst this would be an improvement to my code and its aesthetics, because there is not a way of doing it for all operating systems, it was not worth the cost of potentially limiting the software to a single operating system which would have decreased the accessibility of my program.
.
The program itself also needs to be designed and checked in a way that will make the final product accessible to any user, I have addressed this by providing necessary information at the start of all instances of the program, which not only acts as a welcome to the game, but also provides instructions and clarifications on the individual visual components. It also provides further resources to the rules of the game. For example, I assumed that the way I have represented cards could be confusing to some users hence I provided a description at the start of the function explaining that the cards are depicted as {number}{suit}, so as an example the 3 of spades would be represented as 3S in my program. I provided this information at the beginning of all games to allow the user to receive the necessary information inside the game window, increasing the accessibility of the program.

When I started testing my program, often when no cards could be played in multiple consecutive turns, the program would instantly perform all the card pick ups in very little time. This resulted

in the user going from having 2 cards to having 7 in what appeared to be no time. To make this more accessible to the user and make the game not overwhelming and confusing, I implemented a small delay of 500ms to show turn by turn what is happening.
I also asked volunteers to test and feedback on the playing experience. This led me to making adjustments based on their feedback especially with the visuals of the program. This improved the accessibility of the program to a wide audience.

By taking in account all these factors in my program I have made adjustments to ensure that the final product is as accessible as possible to the end-user.

Future Proofing:

The implications of future proofing a programme are important to consider. Primarily, a future proof programme will ensure the long term viability of the product, and ensure it remains functional and relevant. As c++ is a compiled language it is dependent on the existence of a c++ compiler to be able to run. The cons of this is that it requires additional software to be run, but the pros of this is that the because the language is used for such a wide range of purposes and has been and will be used for years if not decades to come. Because of this we can assume that as the operating systems evolve and computers change beyond recognisability, the compilers such as clang or gcc or any new compilers written will continue to evolve and change to match these system changes. This means that the language used will be futureproofed hence meeting the requirement to consider and solve the issues relating to future proofing.

The second challenge would be to ensure that the code itself will survive the test of time. This is another reason that only using the c++ standard library is a good idea. As spoken above I discussed how only using std:: improves the accessibility of the program but it also helps with the future proofing of the software as the standard library is pretty well maintained by c++ maintainers and are practically ensured to have defined behaviour across systems and across different compilers.

The third thing to consider when future proofing software is to ensure the program is easily able to be updated if there are any issues that come up. Having the main components of the program in separate header files and then separated into several methods of a class makes it really easy to change only one section of the code at a time. By doing this and maintaining the connection to the github repository it allows me to view and perhaps revert any changes made to the program overtime which could allow me to fix any potential errors and bugs introduced when making changes to the program. This improves how I can update specific sections of the program which can make it more future proof.

Useability:

Useability is the measure of how useable the software is to the end user, for a program to be useable it must satisfy the useability heuristics, an example of these are: consistency, real world matches, error prevention, flexibility, etc. By taking these factors into account we can ensure that the program will be useable to the end user.

In order to make my card game consistent I used the same technique for taking user input for multiple choice options. For example, I used the same input function for taking input when the user wanted to play a card from their hand and when they wanted to change suit after playing an ace or joker. By using the same function for both you build up familiarity with the software which results in less thinking for the end user which makes it more useable.
By making the software replicate a real life game of last card it made it more useable for the end user as it replicates what they are familiar with and makes it easier for them to understand what is being shown to them. I used text based inputs and showed them minimal detail, which would replicate what a physical hand of cards would appear in their hands as a list. I told the user what cards were being played by the computer and what card was on top of the discard pile to allow them to continue with the game instead of having to track the cards played in their minds. I also inform the user in the case of a suit being changed by an ace or joker when the suit they can play on doesn't match with the suit of the facing card. This improves useability by informing the user in cases where the required knowledge mightn't have been obvious without being explicitly told.

Another important factor in making software useable for the end user is error prevention. If a program breaks unexpectedly or performs in unusual ways or doesn't handle incorrect input properly, it would make the end user experience worse. In order to address this my function for handling user input checks for any non numeric characters to tell whether a non integer value was entered which would break my code. This provides a layer of security by informing the user that they have made a mistake, and gives them a chance to correct this mistake without breaking the code and ruining their experience of their game forcing them to restart.

Conclusion:

For this project I was instructed to make the card game last card using complex programming techniques. For these complex techniques I used the concept of object oriented programming to create classes to handle the piles, cards and even the game itself. I also used the concept of stacks and queues. The cards being drawn by players into the hand used the concept of the stacks to take the top card of the pile and place it into the hand. Also when flipping the discard pile into the draw pile it used a stack based method to take cards from the discard pile and add them to the stack. I used the concept of queues to manage the computer's playing cards. It takes the leftmost card available to play and plays it and gains cards at the end of the queue.

I created a refined program by repeatedly testing and debugging the program often. This also allowed me to address the relevant implications of accessibility, useability and the future-proofness of the program. The outcome of this was a high quality program.