Developers:

Caleb Cargill – cargilch@mail.uc.edu

John Notogiacomo – notorgjv@mail.uc.edu

C++ Final Project: Blackjack

Project Description:

For the final project, the game of Blackjack will be created using proper and efficient C++ coding practices. The game will be coded so that it fully follows the rules of Blackjack, and so that it gives the user a game experience that is both engaging and fun. All topics covered within this course will be implemented into the code of this game, with a major focus on classes and functions being implemented into the code so that it is as effective as possible. Work will be evenly split up between partners, so that one person is not doing more work than the other. Ensuring to be working on this as a team, ideas will be bounced off one another throughout the scope of the project to be working effectively as a team and not too individually.

Original Design and Specification:

A general outline of the code is as follows: A class that will contain private members and functions within the public area. The class will have a constructor that would set private members (such as score of user, score of computer, etc.) to zero. Some functions within the class could be but are not limited to: a set function for score of user and a set function for score of computer, a get function that returns the score of the user and a get function that returns the score of the computer, a function that draws the next card, a function that adds to the score of each player, and a function that checks to see if the score of either player is equal or over twenty one. Somewhere within the code, whether as a function inside/outside of the class, or within the main function of the script, there should be a while loop with a switch statement that provides a “menu” for the user to be interacting with during their turn throughout the game, allowing them to “draw a new card”, “not draw a new card”, or “quit playing”. Throughout the script, there will be clear and concise comments that show a clear purpose for each part/section of the code. This is common and proper practice when coding, to make code tracing/debugging easier though out the scope of the project. Proper practices such as commenting, whitespace, and indenting will be used when coding to have efficient and organized code.

Program Feature List:

The actual code consists of three classes with functions within them, and a main function that refers to one of the classes. The program we wrote contains the following features:

1. Basic rules of Blackjack
   - 1.1. AI Dealer
   - 1.2. Dealer’s second card “in the hole”, or hidden
   - 1.3. Hit
   - 1.4. Stand
   - 1.5. Bust
   - 1.6. Blackjack

    1.7. Winning/Losing with score relative to the dealer’s
    1.8. Push
    1.9. Payout
    1.10. Round Tracking
    1.11. Fluid Ace Value (1 or 11 as is best for each player)
2. Advanced:
    2.1. Betting
    2.2. Fund Tracking
    2.3. Insurance against Dealer Blackjack
    2.4. Surrendering
    2.5. Doubling Down

Testing Methodologies:

As we worked on this project, we began by coding the game to follow the basic rules of Blackjack. We did this using two classes and a function within the main. After this, we added more features to the game one by one, testing as we went. Therefore, as we worked on the project, the game gained more and more features. We tested as we went along, rather than simply adding several features at once and then testing because we wanted to ensure that each feature (such as insurance bet, doubling down, etc) would work perfectly well. Another bonus of testing like this is that if the script fails, then we could know where the error occured because we had only edited one part of the script since the last successful run. Using version control, we saved each new iteration of this game as a new file. At the end, each win or lose condition was tested for each feature by hardsetting the card draw values to ensure that everything was working properly in a time efficient manner. Further, the code was tested to make sure that, no matter the player interaction, the game would continue on without failing or causing a compiler error.

Program Usage Guidelines:

As a user uses this program, they should understand and follow the rules of Blackjack. The program gives instruction to the user, and they should follow said instructions based upon what they would like to do in that moment. Everything in the game can be done with keys: 1,2,3,4, y and n. The numbers are for making decisions about the user’s hand, while the y character is used to make decisions regarding continuing to play or making an insurance bet; the n character is listed because, when prompted to decide “yes or no” only the y character will confirm a positive answer while any other key will act as a no.

The program is coded so that the user is unable to make any invalid inputs, and the user will exit the game if they chose to quit or run out of funds. A player who decides to quit before losing all funds is thanked and told their ending fund amount, or score.

Lessons Learned (from Caleb):

Throughout this project I learned about several things, including proper formatting using whitespace and indentation, proper commenting practices, version control, linking header files containing classes to a main function file, communication with a team member, and referring to other classes within another class. Many of these things were covered in class and in various labs, but I learned more about the actual practice of such things as I applied them to this project. I believe that the reason that our script works is because these things were applied well to our code, and without these things, our code would have been a mess.

Proper formatting using whitespace and indentation, and proper commenting practices made tracing our code much easier and made debugging our code go much more smoothly than it would have without such practices. In labs in the past, I had gotten lazy about commenting, and that simply would not have worked with such a large project like this. In many cases, proper commenting saved myself so much time, as when I ran into an error, it was much easier to find the location of the error and fix it.

Before working on this project, I did not know much about version control at all, and my partner John had to teach me why it was important. Without version control, we would not be able to look back and see the process that we had gone through. On top of this, editing the same file over and over again could cause issues, since a crucial part of the code could accidentally be edited or deleted, causing the code to not work. Saving new versions means that this could not happen, and we would be able to trace the history of our code. Version control also allows us to go back to a previous version, if, when adding an extra feature to our code, the code decided to stop working. Finally, version control demonstrates the design process that we went through during this project.

In past labs, we had worked with multiple files that were linked together using the "#include "filename"" command. For this project, our code included three header files and one main function file, and we had to link all of these files to work together for our program to work. We could have simply put all of this code into one file, having a file that was over 600 lines long, but this would have been improper coding practice. During this project, I learned how to write Classes in different files, and then link them to the main function, while keeping the complexity and length of the main function file at a minimum. With this, I also learned how to refer to other Classes (in different files) from one specific class. I thought that this would be difficult at first, but I quickly realized that C++ allows developers to do this quite easily.

Throughout the project, much communication was required with my partner. We had to be constantly updating and debriefing one another on the work that we were doing, as well as teaching one another as we went through the project together. This is another point when commenting was necessary. If I were tracing some of his code, for example, him making comments on various parts of his code would be very helpful. Since we likely had differing visions for the project, communication made it so that we were not going off and doing our own thing, but that we were actually working together on the project.

Going into future projects, I hope to maintain these proper coding practices, and apply these things that I have learned. I know that I will be working in teams a lot in the future, so I must continue to apply communication skills, and I will probably also need to develop leadership skills. I will need to continue to maintain proper formatting and commenting practices, to make debugging and tracing easier for myself and others. And finally, I will eventually be working on larger projects where I will continue to need to have multiple files linked together to make a program work.

Lessons Learned (from John):

Doing this project was an eye opening experience. I’ve written smaller scale, simple “games” before, usually revolving around a die roll, but never something quite like this. Throughout the course of this project, I learned and came to appreciate C++ best practices, version control and code reusability. Most importantly, it really took my understanding of how to implement a class to a whole new level.

Prior to working with Caleb on this project, I was a little dismayed about having to use the class feature. To me, it seemed like a lot of extra work for not very much reward, in terms of for just one project. Sure, I understood that, for code reusability, any class I made could be implemented again elsewhere, but I didn’t appreciate the power a class would allow me to wield. The first, and biggest lesson I learned, and it was from Caleb, was that class functions could access private class data without having to use a get or set function and without having to be in any kind of sequential order. Before knowing that, I thought the class feature was just a good organizational tool. Now, having written most of the game in one class but with two supporting classes, I can really appreciate just how much more flexibility I have using a class rather than typing everything in the main. Not only that, but having everything declared in the top of the class both served to help plan what we were going to use (and how) as well as serving as a guide for where to look for things later, almost like an index in the front of a textbook.

One of the most memorable moments I had while working on this project was when Caleb first cleaned up the code. The previous evening, I had just finished working out the logic for the basic features and betting; the game could now be actually played and would end when the user either quit or ran out of the coins we gave them to start with. When I loaded up the file the next day, after he had told me he had cleaned it up a bit, I was shocked. Everything was gone; everything we had done so far had been done in the main file (not in the main, in a function declared before it and defined below it, but still in the main file). Caleb had implemented a 3rd, main class that referenced the other two classes we had been using, one for the deck and one for the players and it made the code so much more neat and easy to read. This experience actually had a tri-fold impact. For one, it showed me just how much better looking the code is when broken down this way, as well as how much easier it was to trace through and debug. Secondly, the code then lent itself to much greater reusability. Caleb had taken all the logic we had written thus far and turned every identifiable section into a function; this too made reading and tracing the code much easier, as now every line’s purpose was easily identifiable. Lastly, this is the part where I came to really appreciate version control. When Caleb made the move from everything in the main to using three classes, he did so in separate file from the one I had finished the night before. While making the move, he decided to read and type in what we had written thus far to make sure he had a good understanding of what was happening in the code. However, with over 600 lines of code to read and type, some errors were sure to ensue. Thus, having the previous version to reference while debugging was super useful and without it, we would have spent hours trying to figure out where we went wrong and how to fix it.

Lastly, it quickly became clear to me that good organizational and stylizing habits are key to writing easy to understand, trace and debug code. Before we updated the stylization and organization of the code, I would often find myself getting lost in it, scrolling through it looking for a section, only to forget what I was looking for and have to start again. Afterwards, I knew exactly where to look for things and was able to easily tell what was doing what in each line of the code.

I know and plan to implement these lessons in future projects, be it for school or for myself. I would really like to build a battle simulator this summer for a game that I play and, before this project, I actually felt more comfortable attempting to do so in Matlab than in C++. Now I realize how much easier, with class and header linking as well as better function usability, this will be in C. I plan to take these lessons and best practices on with me to wherever I end up coding next, be it at co-op, my current job or at home.