Data Structures Lab 4 Report

Nick McClorey, Daniel Wood, Kyle Van Blaricom February 11, 2019

Project Overview:

We experimented with polymorphism, inheritance and virtual functions. We created an abstract class that others inherited from. Inheritance is important because many frameworks allow you to inherit from classes and create your own implementation. The framework then uses that class you created. This works because the framework knows what member functions to expect. Overriding allows people to create multiple implementations of the same base class. Abstract classes can act like templates for other classes, allowing designers and users of the class on the same page.

Task 1:

We had two class attributes, “playerOneScore” and “playerTwoScore”. Both of these variables are private integers. They are not available to inherited classes or other functions. However, derived classes can use the getters and setters to access these values. They are only available to the classes member functions. All of the member functions will be available to base classes unless they are overwritten.

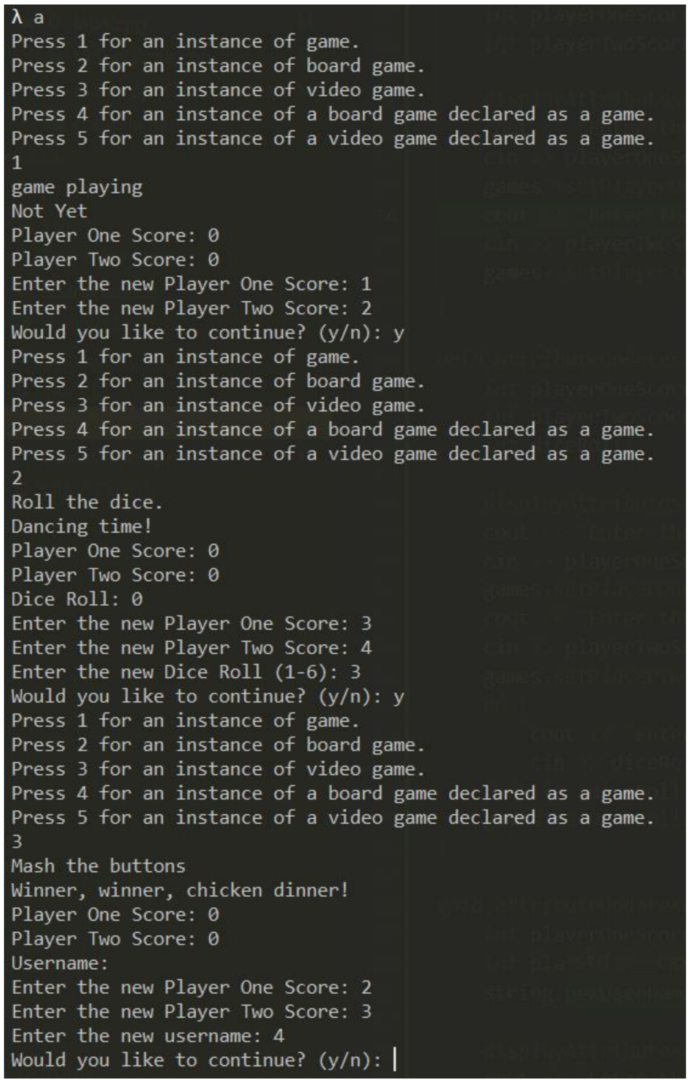
Task 2:

We created the BoardGame and VideoGame class as described in the assignment. Each class derived from the base class. Instances of these classes have access to all of the member functions in the base class except for the Play() and Winner() function. These functions were overwritten. Instances of these classes declared as the “game” class have access to all the member functions that weren’t overwritten. The Winner() function didn’t use the “virtual” keyword so when this method is called, it defaults to the base class’s version of the function. The Play() function was declared using the virtual keyword so every instance of the BoardGame and VideoGame class will have access to the Play() method. This is seen when selecting option 4 or 5 in the program. The base class of Winner() will be called because the “virtual” keyword wasn’t used.

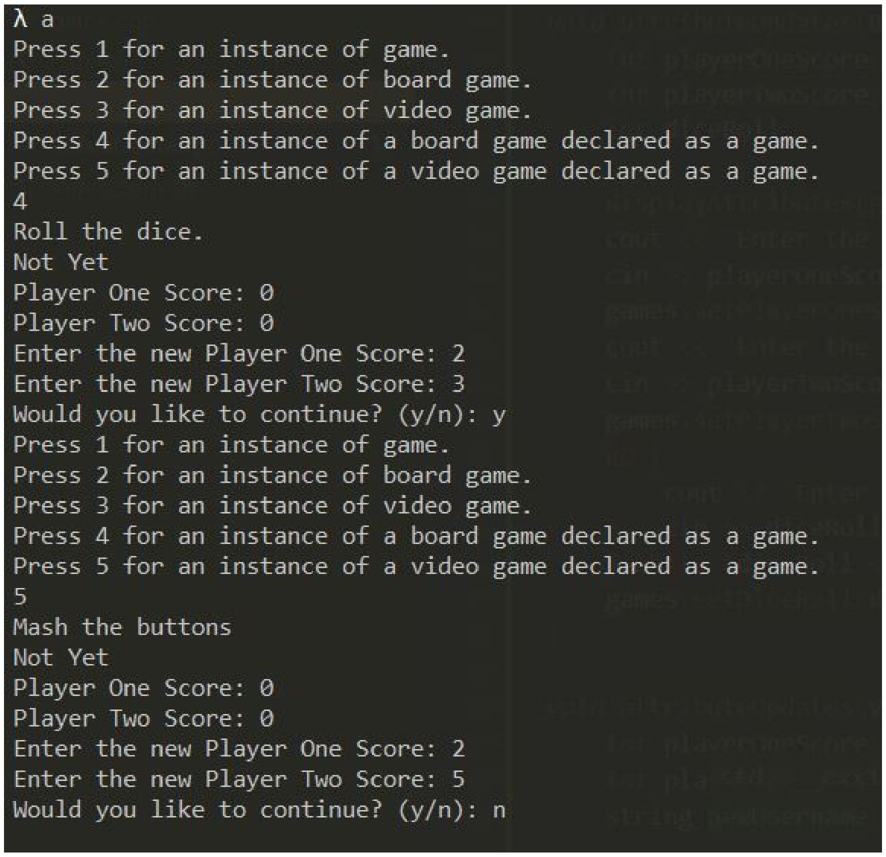
Task 3:

The main() function allows the user to create objects from any of the classes previously declared. They can also create a Game pointer that points to a derived class. When creating the classes normally, the methods work as expected. When a Game pointer points to a derived class, the Play() function will correspond the the instance of the class created. The Winner() function will always default to the base class’s version. The Winner() function doesn’t have the virtual keyword and the pointer is a Game pointer so the Game version of the method is called. If we add the “virtual” keyword, the Winner() function called will always correspond the class of the object. When updating the games, we called an overloaded function. This function acted normally for the first three options. However, passing a Game declared as one of the sub classes called the overloaded function for the Game, not the VideoGame or BoardGame. This meant the new attributes added by the inheriting classes weren't displayed. This could be fixed by passing a variable declared as a VideoGame or BoardGame.

Screenshots:



```
λ a
Press 1 for an instance of game.
Press 2 for an instance of board game.
Press 3 for an instance of video game.
Press 4 for an instance of a board game declared as a game.
Press 5 for an instance of a video game declared as a game.
1
game playing
Not Yet
Player One Score: 0
Player Two Score: 0
Enter the new Player One Score: 1
Enter the new Player Two Score: 2
Would you like to continue? (y/n): y
Press 1 for an instance of game.
Press 2 for an instance of board game.
Press 3 for an instance of video game.
Press 4 for an instance of a board game declared as a game.
Press 5 for an instance of a video game declared as a game.
2
Roll the dice.
Dancing time!
Player One Score: 0
Player Two Score: 0
Dice Roll: 0
Enter the new Player One Score: 3
Enter the new Player Two Score: 4
Enter the new Dice Roll (1-6): 3
Would you like to continue? (y/n): y
Press 1 for an instance of game.
Press 2 for an instance of board game.
Press 3 for an instance of video game.
Press 4 for an instance of a board game declared as a game.
Press 5 for an instance of a video game declared as a game.
3
Mash the buttons
Winner, winner, chicken dinner!
Player One Score: 0
Player Two Score: 0
Username:
Enter the new Player One Score: 2
Enter the new Player Two Score: 3
Enter the new username: 4
Would you like to continue? (y/n):
```



```
λ a
Press 1 for an instance of game.
Press 2 for an instance of board game.
Press 3 for an instance of video game.
Press 4 for an instance of a board game declared as a game.
Press 5 for an instance of a video game declared as a game.
4
Roll the dice.
Not Yet
Player One Score: 0
Player Two Score: 0
Enter the new Player One Score: 2
Enter the new Player Two Score: 3
Would you like to continue? (y/n): y
Press 1 for an instance of game.
Press 2 for an instance of board game.
Press 3 for an instance of video game.
Press 4 for an instance of a board game declared as a game.
Press 5 for an instance of a video game declared as a game.
5
Mash the buttons
Not Yet
Player One Score: 0
Player Two Score: 0
Enter the new Player One Score: 2
Enter the new Player Two Score: 5
Would you like to continue? (y/n): n
```

Contributions:

Kyle Van Blaricom wrote the base class, Daniel Wood created the BoardGame class and Nick McClorey made the VideoGame class. Everyone pitched into the main function to test the classes.