Name: Program Design

Author: Nathan Johnson

Date: 04.20.2019

Description: Program design for Lab 3

PROGRAM DESCRIPTION:

The main program implementation is inside the 'Game' class, which uses objects from the 'Die' and 'Loaded Die' classes, and helper functions 'getInt' for validation, and 'menu' to display the menu options. The main 'warMain' simply sets the random seed and creates and plays a game class instance.

Descriptions are below:

Game Class: Contains logic & implementation of War game

Member variables:

int roundsTotal // # rounds should be between 1-1000

int roundCurrent // stores current round in the game

int menuChoice // stores users menu choice

int dieType_p1 // 1 = normal die, 2 = loaded die

int dieType_p2 // 1 = normal die, 2 = loaded die

int dieSides_p1 // Die should have sides from 4-120

int dieSides_p2 // Die should have sides from 4-120

Member Functions:

Game() // Default constructor that initializes variables

void play() // Contains logic & implementation of War Game

Pseudocode:

Display menu

    If choose play, proceed

    If choose exit, exit program

Prompt for input for game initialization

    Input rounds played (w/ validation)

    Input type of die for player 1 (w/ validation)

Input # of sides of die for player 1 (w/ validation)

Input type of die for player 2 (w/ validation)

Input # of sides of die for player 2 (w/ validation)

Start game

Initialize variables

Create Player 1 die

Create Player 2 die

For each round

Print result of round

Print round #

Print Player 1 type/sides of die, and roll result

Print Player 2 type/sides of die and roll result

Print winner of round

Print Player 1 score

Print Player 2 game

At end of game

Print winner of game

Print Player 1 final score

Print Player 2 final score

Return to menu

Die Class: Creates a die that returns a random # depending on # of sizes

Should set rand seed in main before creating objects

Member variables:

int N // # of sides of the die

Member funcions:

Die() // Default constructor, creates 6-sided die

Die (int nIn) // Sets N through parameter

int roll() // Returns random # b/t 1-N

int getSides() // Returns N

LoadedDie Class: Creates a die that returns a random # depending on # of sizes. Inherits from die class. This die is loaded, which means it returns an average of a higher number than a normal die. This is achieved by limiting the return value from a minimum of 1 to a minimum of the total sides divided by 2.

Should set rand seed in main before creating objects

Member variables:

Inherited from Die class

Member funcions:

Constructors/functions inherit from Die class

int roll() // Overridden, returns random # b/t N/2-N

menu() Function:

Displays the menu and returns the user inputted choice. Uses integer validation function to prevent errors. Modular design to enable easier editing for future projects.

getInt() Function: Prompts for int input, returns int when successful

while valid input flag is false

    Prompt for int input

    Store input to string

    For each char in string

        Set error flag to true if non-int char

    If error is not flagged

        Set the input to int

        Set valid input flag to true

    Else

        Display error message

        Clear input

Return input

Overloaded **getInt(int minVal, int maxVal)** Function:

while valid input flag is false

Prompt for int input

Store input to string

For each char in string

Set error flag to true if non-int char

If error is not flagged

Convert input to int

If input is between min & max values

Store the input

Set valid input flag to true

Else

Display out of range message

Else

Display error message

Clear input

Return input

Test Case	**Input**	**Expected Output**	**Actual Output**
Run program	Run program	Program displays Menu	Program displays Menu
Select 1st menu option to start the game	1	Starts prompting parameter input	Starts prompting parameter input
Select 2nd menu option to exit the program	2	Exits the program	Exits the program
Select something other than either menu option	char, string, 1>#>2	Re-prompts for correct input	Re-prompts for correct input
Input correct range for # rounds	1≤#≤1000	Moves to next input prompt, stores # rounds in Game class	Moves to next input prompt, stores # rounds in Game class
Input incorrect range for # rounds	Char, string, 1>#>1000	Re-prompts for correct input	Re-prompts for correct input
Input correct range for type of die (p1 & p2)	1≤#≤2	Moves to next input prompt, stores type of die in Game class	Moves to next input prompt, stores type of die in Game class

Input incorrect range for type of die	Char, string, 1>#>2	Re-prompts for correct input	Re-prompts for correct input
Input correct range for # sides of die (p1 & p2)	4≤#≤120	Moves to next input prompt, stores # sides in Game class, creates p1 & p2 die & starts game	Moves to next input prompt, stores # sides in Game class, creates p1 & p2 dies & loaded dies & starts game
Input incorrect range for # of steps	Char, string, 4>#>120	Re-prompts for correct input	Re-prompts for correct input
Running Game	None	Displays detailed result of each round (side and type of die used for each player, the number each player rolls, and the score result) for each round, then the overall winner & player scores	Displays detailed result of each round (side and type of die used for each player, the number each player rolls, and the score result) for each round, then the overall winner & player scores
Finish Game	None	Displays menu again, starts loop over	Displays menu again, starts loop over

Reflection:

Modifications I made included overhauling my validation function, changing the way I handled the composed die classes in the Game class, figuring out how to truly randomize the die rolls, and how to use the overridden loaded die roll function. Debugging took a bulk of the time I spent on the project, I could have avoided hours of hassle if I had been more careful writing the code.

In previous assignments I had used cin.fail as the primary tool for validating integer input. It worked in most cases, except where a string such as “12e3” was entered, in which case it would accept “12” as valid input and not trigger for an error message. Using the tools we learned in Lab 2, I reworked the function to accept the user input as a string, and then parsed through each character in the string to make sure it was a number, triggering an error flag if it was outside the numerical chars. This made me realize that it wouldn’t work with negative numbers, as the negative sign would flag an error, so I had to make an exception statement for the first character if it was a negative sign. The other tricky thing about the validation was converting string output to an int if it was correct, but I was able to by using a string stream as an go-between. The int validation function should no work in every case.

By far the largest issue I had while debugging was an error that would occur as soon as the die were created while running the game. It would through an overload exception and crash the game. I searched through meticulously to see if there were any errors in how I was creating the die inside the class, or problems with the die class, all to no avail. I used the linux debugging tool to trace the cause for the crash, and saw it occurred in the roll function of the die class. After much testing, I realized something was wrong with one of the variables in the code for the randomization. I went back to the code in the game class and realized I had forgotten to change one of the input names for the # of sides from player 1 to player 2, so that variable had been left as 0 and threw an error when trying to do modulus of 0. It

was a simple fix which could have been avoided by being more careful when I was copying lines of code, but at least it wasn't because I had completely not understood composition, which is what I had initially feared.

For my roll function, I had initially tried seeding the rand function before using, as I had every other time I had used it. However, when I went to test the die class (before making game class) I realized it was rolling the same instead of being random. I saw a thread on Piazza expressing the same issue, and saw a fix where they seeded in main instead of the function. After doing more research to understand the issue, it seems to be because the loop calling all the rolls happened so quickly in the processer, the time function for randomization was using the same seed since the processor was so fast. I also saw that it was best practice to seed only once, and thankfully seeding in main instead of the function solved the issue.

One of the other largest issues I ran into was in creating the die objects in the Game class. I initially had an if/else statement that would either create a die or loaded die named d1 depending on user input. However, after testing, the rolls would always be from a standard die, it wouldn't use the overridden function in the LoadedDie class. After much testing, I tried creating the die outside the function, using pointers, and using dynamic memory, none of which worked. Eventually I tried creating a loaded die outside of the if/else statements which worked, and I realized that creating the die inside these statements probably led to a scope issue, where the die would be destroyed outside the enclosing brackets. The best way I could think of to have variables determine which die to use, was by creating four die, one of each type for each player. The switch statement would then roll either the standard or loaded die for the player depending on their choice. It might not be the most efficient solution, but it worked without any issues.

Planning out my classes and functions in detail helped me write the code much faster this time, and it would have been much easier if I had been more careful when copying code blocks to change every variable as needed. Thankfully I was able to debug all the issues eventually, and I am happy with the finished product.