Rock Paper Scissors Design

Tool class
Abstract class

Members:

int strength - holds the strength of the tool
char type - holds a character to identify the type of the tool (r, p or s)

All members are private.

Functions:

void setStrength(int strength) - used to set the strength member of tool class.
int getStrength() - returns the strength of the member of tool class.

void setType(char type) - used to set the type member of tool class.
char getType() - returns char of the type member of tool class.

Accessor and mutator functions should be protected - we don't need to change these from outside the class, but they need to be used in derived classes.

Int halfStrength() - returns a value that is half the tools strength to compare when fighting.
Int doubleStrength() - returns value that is double the tools strength to compare when fighting

Tool() - default constructor sets strength to 1
Tool(int strength) - sets strength member to the parameter strength

Virtual ~Tool() - destructor shouldn't need to do anything here? But we should explicitly define it

virtual int fight(Tool *opponent) = 0

Pure virtual function - this must be defined in all derived classes. Since the advantage is different for each. The return value is a int. There are three possible situations, the tool calling the member fight function can lose, win, or tie. We will return 0 for a loss, 1 for a win and 2 for a tie.

Example:

If fight is called as the member function of a Rock tool and you are fighting paper, the call would be:

rockobj.fight(<pointertoPaperObject>)

This call will return 0 -> rock loses to paper.

If a scissors pointer was passed in place of a paper pointer the function will return 1.

If a rock pointer is passed, the function will return 2

Rock class

Derived from Tool

Rock(): default constructor sets type to r
rock (int strength) : constructor sets strength to parameter strength using the base class constructor.

int fight(tool*) - implementation of fight for rock see pseudocode below

- check type of tool* parameter
- If parameter is paper:
  - if this->halfstrength() is greater than tool->getstrength() then return 1
  - *The rock has won!*
  - Else return 0- *paper has won.*
- Else if parameter is scissors:
  - If this->doublestrength() is greater than tool->getStregnth then return 1
  - *The rock won!*
  - Else return 0 - *scissors has won.*
- Else - *the passes in tool is rock*
  - Return 2 - *this is a tie*

Paper class

Derived from Tool

Paper(): default constructor sets type to p
Paper(int strength): constructor sets strength to parameter strength using the base class constructor.

Int fight(tool*) - implement in the same way as the rock class but with correct comparisons for tools

Scissors class

Derived from Tool

Scissors(): default constructor sets type to s

Scissors(int strength): constructor sets strength to parameter using the base class constructor.

Int fight(tool*) - implement in the same way as the rock class but with correct comparisons for tools

RPSGame Class

Members

tool* humanChoice
tool* computerChoice
Vector<tool*> humanChoicesHistory - *vector of pointers of the history of the human choices.*
Int humanWins
Int computerWins
Int ties

Functions

round() - plays one round of RPS game, gets user choice, computer decides what to play, the user's choice object fights the computer's choice, the result is added to the correct integer counter, results are displayed on screen. *also needs to delete allocated memory for the humanChoice and ComputerChoice pointer and set them to nullptr.

accessors/mutators
SetUserChoice
Getuserchoice
setComputerChoice
getComputer choice

RPSGame() - constructor should only need to set counter variables to 0.
**I think that the other functions will handle the other members.*

void userChoice() - gets input from user for their RPS choice. This needs to set the humanchoice pointer to a tool object, this should use new keyword. This should also add a new element to the humanChoicesHistory vector of the current human choice. Needs to use dynamic memory allocation

void computerChoice() - this will be a recursive search function that will search the humanHistoryChoices vector for matching patterns of the user's most recent past 4 choices. If non are found it will search for the user's most recent past 3 choices and so on until it has searched for choices of two. When it find a match, it will add the human choice that occurs directly after the pattern match to another

vector of, matches. When the entire humanHistoryChoices has been searched, the matches vector will be searched for the choice that was made the most times. If there are two choices that have happened an equal number of times, then a one of them will be chosen randomly. This choice is what the computer “thinks” the human will choose. This function will set the computerChoice pointer to the tool that has the advantage over what it thinks the human will choose. This will then free memory for the matches vector.

void addToolToVector(char type) - this function will be used in computer choice and userchoice. It will add a tool to a vector with dynamic memory allocation.

pointer to tool analyzeMatches(vector<tool pointers>) - this function will look at the matches vector created by the computer choice class and analyze what the computer should play based on what is in that matches vector. This will return a tool* so the member of the class computer choice can point to the desired tool.

Void analyze result(int result) - pass the result of a fight to this function and it will increment the correct result counter.

void displayRound()- displays what computer chose, and the current win statistics.

~RPSgame() - needs to free memory of the userChoiceHistory array.

play_game.ccp file;

This really should just make a RPS object and have a do-while loop that will play the game until user enters e-Exit. Then displays the final tally of the score one last time.

Other notes (extra thoughts):

The specifications say “You must have the proper constructors, destructors, and assignment operator overload for the Tool, Rock, Paper, Scissor, and RPSGame classes.” I think we won’t need to explicitly define copy constructors for the Tool class and its derived classes, since these themselves aren’t dynamically allocating memory. But maybe we should just to practice? But we will need to define one for the RPSGame Class.

The destructor for the tool class I believe needs to be defined as virtual, I forget exactly why at the moment but I know I did that in project 2 for my animals, and it was important.

Testing

Test	Expected Result	Actual
Input Validation	Program should only accept r, p, or s.	The program would only accept r,p, or s, however it would accept rr, pp, ss, and then just take the first letter and use that. We decided to force users to re enter their choice in these cases.
Standard r v r	Tie, no winner	Verified
Standard p v p	Tie, no winner	Verified
Standard s v s	Tie, no winner	Verified
Standard r v p	Paper wins	Verified
Standard r v s	Rock wins	Verified
Standard s v p	Scissor wins	Verified
Augmented r v r	Higher strength wins	Verified
Augmented p v p	Higher strength wins	Verified
Augmented s v s	Higher strength wins	Verified
Augmented r v p	Higher strength wins	Verified
Augmented r v s	Higher strength wins	Verified
Augmented s v p	Higher strength wins	Verified
halfStrength test	Verify strength is cut in half	The decimal was being truncated on odd numbers… so since the base strength is 1, and the project called for int, we were left with an inaccurate number. We removed this function and focused on the doubleStrength() function instead to determine the winner.

doubleStrength()	Verify strength is accurately doubled and the correct winner is declared	Verified
AI test	After a series of patterned choices, the computer should anticipate your next turn.	Verified when a pattern was entered at least twice.

Reflection

Working as a group was both fun and challenging. We were able to take a few key steps to ensure our success. We started with a group meeting to get to know each other and to talk about the project. We decided on a direction and had a draft of the design written. This design was then reviewed and commented on by all group members. After that iteration of review we met again to talk about design changes and the best way to accomplish what the design suggested. We broke the responsibilities into two groups, one group to work on the Tool classes and the other to work on the RPSGame class. Both groups had mutually agreed due dates to get everything done by. With splitting up the groups we were able to each take smaller pieces to accomplish which worked very well because we had a great outline of how the pieces needed to work together. Like most projects there were some design changes after we started implementing our program. We all had an understanding of the overall design so it was easy to communicate changes to other group members so that the changes would still work together. The following parts of the reflection will outline some of these changes and how we were able to work them out as a group.

Tool Class Changes:
In the fight() function, the compiler was showing errors when trying to access the getStrength() and getType() functions from the opponent Tool class that was being passed to the Tool that was running the fight function. The errors stated that those function were private. These function were initially set to protected in the Tool Class, so these two functions were changed to public, and that eliminated the error messages.

The original design declared a tie in the fight() function if both objects were of the same type. However, this did not take into account that two objects of the same type could have different strengths. Since the strength was to be taken into account, we needed to compare the strength of two objects that are the same type instead of just calling it a tie, so an a third else if{} block was added to the fight() function comparing the strengths of two objects of like type.

We originally include both a halfStrength() and a doubleStrength() function in the tool class. The halfStrength() too class would return half the strength of the tool, and the doubleStrength() function would return double the strength of the tool. The functions were called in the fight() function to compare the strengths of two tools. The halfStrength() function was used to half the strength of the testing Tool subclass when it had disadvantage, and the doubleStrength() was

used to double the strength of the opposing tool when it had advantage. However, since the strength variable was an integer, this resulted in the decimal place being truncated while performing integer division in the halfStrength() function. This resulted in inconsistent results depending on which Tool was doing the testing and which was passed to that testing tool as the opponent. To correct this problem, we decided to just use the doubleStrength() function to always double the strength of the tool with advantage, and eliminated the halfStrength function so that no integer division was performed.

RPSGame Class Changes:

The computer choice section of the program was mainly the same as the design but there were a few things that were overlooked. First we used a few extra helper functions to implement computer choices. This was to break of the function into more manageable pieces. Second the original plan was to have the function void and the computer choice pointer would be set within the function but we decided that it would be better to return a pointer from the function. This made the recursion make more sense to us because void doesn't return anything and with recursion the whole idea is returning a call to another function until the problem is solved and the correct answer is returned, this change just made more thematic sense. Another thing was changed was the vectors that we were actually comparing. At first the idea wa that we would have vectors of Tool objects. As we started implementing this the overhead of managing that memory started to seem like a lot. Instead of comparing vectors of objects we just saved the type member the elements of the human choice vector into char vectors. These char vectors could be made and compared easily.

The final program also had some memory issues. These were all due to memory allocation in the RPSGame Class. The issue was with the addition of pointers to the humanChoiceHistory Vector. We were just pushing a pointer of the humanchoice each round onto the vector, then deleting the humanChoice before the human makes a different choice. When the human choice was deleted this would also delete the object in the vector. When the human choice vector needed to be analysed, we would be accessing freed memory. Another memory issue was that in the computer choices algorithm we were attempting to access the element at the position of the *size* of the vector. Because vectors are indexed starting at 0 this read was outside of the vector leading to a segmentation fault. This was easy to fix, we just had to start looking at the vector at size - 1.

Another area that differed a bit from the initial design was the userChoice function. Originally, this function was designed to handle dynamic memory allocation when the user selected a tool by inserting a new pointer to the user history vector. However, during implementation, it made more sense to handle the inserts at the end of the game function. This way, we were able to ensure that the AI did not have an unfair advantage in the game by reading the user's choice before the computer selected its tool.

While we originally were going to have each round be its own function, we realized that i was much easier to incorporate instead the entire game in one function. At the beginning of the game we had the user select the strengths for the tools, and the rounds were played as a loop handled by the userChoice function. We also created a new function to print the tool that the computer chose while removing that functionality from the result displaying function. It made more sense to separate these two out because we first needed to analyze if the computer's choice beat the user's choice before printing the results.

One problem that we ran into is that our character input validation function did not work correctly in a few specific cases, such as the program would accept inputs with two or more valid characters in a row. We handled this problem by first making sure the user's input had a size of 1, otherwise we would have the user reenter their input.