8: Polymorphic Dice

CSCI 6626 / 4526 March 2018

1 Goals

- To use derivation and ctors.
- To use interaction between a method in a base class and the method that overrides it.
- To understand the form and meaning of an interaction diagram.
- To create a non-random version of the Dice class so that we can test the program logic.

2 Dice Changes

In this assignment, you will rework Program 1: the Dice class. The new version will have three classes, each with one constructor and a different roll() function. Much of the code for this assignment is already written and debugged; your task now is to separate the parts into three classes that form an inheritance hierarchy. Please put all three of these classes in one pair of .hpp and .cpp files.

A note on design: Normally, all user interaction would be initiated by the primary class, Game, which is the controller class in this application. However, because this is an academic course, we need to use all the major features of the C++ language. One of those features involves the way polymorphic functions work; another is the interplay between two related functions in a class hierarchy. Therefore, I am choosing a slightly worse design; your CantStopDice class will interact directly with the human users.

The Dice base class. Members include:

- The number of dice
- An int* for the dynamically allocated array of dice values
- The 1-argument constructor, with a default parameter and and a ctors.
- The roll() function will continue to do the simple random calculation in a loop.
- The print() function.
- The `operator<<` extension.

3 The CantStopDice class.

This will be derived from the Dice base class. Components include:

- An array of two ints to store the totals of the two dice-pairs.
- Constructor and destructor functions. The constructor should have no parameters and it must have a ctor to call the Dice constructor to create four six-sided dice.
- Override the roll() function:
  - Call the base class roll() function to get the random values,
  - Display the dice values to the human player. (Suggestions: you can use a,b,c,d to label the four dice values or display the values without labels. The user must decide how to pair-up the four dice. If they are labelled (a,b,c,d) it is fairly easy to validate two single-keystroke choices. If the values are not labelled, the user can enter the actual dice values, but this is probably harder to validate[1]. You must ensure that the user choses two different valid dice. Call fatal if that fails.

[1] In a GUI-based display, the user would use the mouse to click on the selected dice.

- In addition to specifying the pairs, it is also essential that the user specify which pair to use first. When there is only one tower left, and both pairs denote a column without a tower, only the first pair can be used. (The other is wasted.) We must therefore ask the user to select and name the one pair with the greatest priority. Prompt the player to choose ONE pair and validate his two choices. Store the sum of those two dice in slot 0 of the pair-value array. In the slot 1, store the total of all four dice minus the two selected dice.
- Return a const int * to the pair-value array instead of the original dice array.

Your code must **use** the polymorphic functions. The Game class should contain a member that is a Dice*. **Do not change that type**. However, instead of allocating a set of Dice, allocate a set of CantStopDice and store the result in your Dice*. Then perform the unit test for that class.

Test the CantStopDice. Test the CantStopDice class first, since the FakeDice class depends on it. Make a copy of the unit test for your Dice class and modify it so that it only creates a set of four dice with six sides. Make sure that the dice pairs are created properly, and that correct pair-totals are returned. Test for user-input errors and handle them with fatal for the time being. Later this will be replaced by exceptions.

4 The FakeDice class.

This class will be derived from the CantStopDice class. Components include:

- An ifstream.
- A zero-argument constructor that opens an input file stream. Put the name of your test file in a #define at the top of dice.hpp or use a command-line argument.
- Override the roll() function to read four dice values from the file instead of computing them randomly. Compute two pair totals (first+second and third+fouth) and store in the pair-totals array of the CantStopDice class. Return a pointer to the pair-value array.

  This will permit you to single-step through the game without thinking about pairs or selecting pairs. Put the sequence of moves that you want to test into the input file. Put enough lines in this file to completely test starting and stopping for two players enough times for one player to win the game. You will need to think carefully about a sequence of events that will thoroughly test your code.

  If you reach the end of file before the game is over, use fatal() to terminate the program. (Note: the fake dice are ONLY for testing. We do not need to implement full error recovery here. We will get to exceptions later.)

Test the FakeDice.. When the CantStopDice work, allocate a set of FakeDice in your Game class and store the new object in your Dice*. To test the FakeDice, make a test plan that contins a sequence of several dice rolls (4 numbers per line). Put the test plan into a file. It does not matter yet what those dice values are, only that they are reproducible and predictable. A full game test is not necessary to test the derived Dice.