Final Project List — CSI 32

Prof. Luis Fernandez. Spring 2020.

Deadline: Thursday May 14th (last day of class)

Instructions

Every student must do one of these projects. Remember that the final project counts a 25% of the final grade. The projecs are divided in three categories, easy, medium and advanced.

- You must **follow** the detailed explanation and **instructions** written in the projects.
- You must **use the names provided** in the project description for the **variables, functions, classes**...
- It is OK to look for code in the internet, understand it and adapt it to your needs. **It is not OK to copy-paste code found online.**
- Projects with code copied from the internet **will receive a total negative grade of** -10.

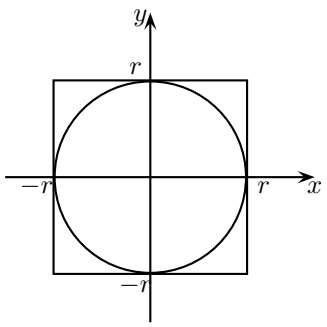
Grading rubric

- 45 to 75 points (depending on the difficulty of the chosen project): the program works well and does what it is supposed to do
- 20 points: well-commented (inner comments are present, correct and meaningful).
- 10 points: clear input/output and intuitive interface
- 10 points: reasonably fool proof.

Project descriptions

Easy:

1. Write a program that does the following simulation: Consider the following square of side length $2r$ with an inscribed circle of radius r on the plane:



Pick n random points with coordinates (x, y) inside the square (that is, x and y are `double` numbers with coordinates between $-r$ and r). Then count how many of these points lies inside the square (that is, their distance to $(0, 0)$ is less than r) and find ratio of the number of points that lie inside the circle to the total number of points.

Use this ratio to estimate the value of π: Since the area of the square is $4r^2$ and the area of the circle is πr^2, the ratio of the number points that fall within the circle to the total number of points will approximately be

$$\frac{\pi r^2}{4r^2} = \frac{\pi}{4}.$$

Write an interactive `main` part where the user is prompted for the radius and the number of points and outputs the approximate value of π obtained. The user should be able to run the program until "Q" is entered.

The name of your variables should be as follows:

- Number of points: `nPoints` (a `const int`).
- Radius of circle: `radius` (a `const int`).
- x and y coordinates: `xCo, yCo` (both `double`).

2. Write a program that a list of integers separated by spaces from a file (or from the keyboard) and outputs a file (or prints out with `cout`) with the integers sorted in increasing order (see 14.3 and 14.4 in the book for input/output of files). You will have to first read the input and store it in a `vector inputData`. Then define a function `simpleSort` that takes a vector as input and sorts it (it is better to pass the vector argument by reference; otherwise if the input is long, it may take time).

The algorithm works as follows: if the array has size n, the algorithm does $n - 1$ passes over the array. In each pass, it starts from the first element and checks if it is greater than the second. If it is, it swaps the values, otherwise it does not change them. Then checks if the second element is greater than the third and

swaps them if it is. Then checks the third and the fourth, etc., until it reaches element $n-1$ and compares it with element n.

The pseudocode for the algorithm in the function `simpleSort` is as follows:

```
void simpleSort( vector<int> & list)
{
  For i := 1 to n-1
    For j := 1 to n-i
      If (list[j] < list[j+1]) , interchange aj and aj+1
    End-for
  End-for
}
```

In this program, the user should be first asked whether the numbers should be read from a file or from the keyboard, and whether the output should be to the screen or to a file. Then the user must enter either the name of a file for inputs or the list of numbers, and the name of a file for output, if it applies.

3. This program allows a bunch of housemates to organize the sharing of expenses. Any housemate can enter his name, an item he bought, the quantity of the item, and the amount of money he spent. You need to define a class `Purchased` with at least the following data members: `name`, `item`, `amountItem`, `moneySpent`. Besides constructor and set and get functions for each of the data members, it must also have various member functions:

- The inventory can be listed with item name and quantity.
- The total amount that a person has spent has to be calculated and listed.
- Anyone can see how much they owe are owed.

Medium:

4. Write a program that simulates a Galton board. You can read about it, for example, in the webpage `https://mathworld.wolfram.com/GaltonBoard.html`. To do this, define a class `Ball` with one data member `posB` initially set to 0, and member functions `void setPos(int &po)` and `int getPos()` to set and get the position of the ball, and `void move()` that increases or decreases the value of `posB` by 1 according to a random number taking values 0 or 1.

In the `main` part of the program, create a `Ball` object and use a loop that runs the function `bump()` a number of times (corresponding to the number of pegs on the Galton board). At the end of all the loops, store the position of the ball in an array of integers (call it `frequency`).

Use another loop to repeat the last paragraph with many balls, and print the results using *ćharacters. If you use many balls and many pegs, the result will look like

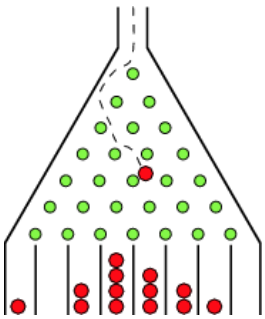
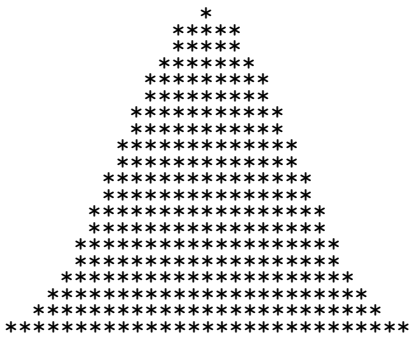


Diagram for Galton Board



Result of the program
(3 million balls, 140 pegs)

5. Write a program that defines two classes, `class Cat` and `class Mouse`. Objects of these classes move inside a rectangular grid of size `xBoard` by `yBoard` (which are global integer constants that should be defined at the very beginning of the program as `const`; you can play using different values for these). Each of the classes has two int data members `xPos` and `yPos` that are the x and y coordinates of the position of the object. `xPos` takes integer values between 1 and `xBoard` and `yPos` takes integer values between 1 and `yBoard`. Besides

`get` and `set` functions, the class has a member function `move` that moves the object one step up, down, right, left, or leaves it at the same square, each with equal probability (20%). Make sure to define the `move` function so that if a move takes the object out of bounds, the function `move` is repeated until it moves the object within the board.

In the `main` part of the program, define an object `Cat cat1;` and an object `Mouse mouse1`, with the initial positions you choose. By moving randomly on the board, the idea is to see whether the cat can catch the mouse, and in how many steps. That is, use a `while` loop that runs until the x and y positions of `cat1` and `mouse1` are equal, and at each loop executes the function `move` for `cat1` and `mouse1`. Once the cat catches the mouse, print `‘‘Yum!’’`, and output the number of moves that were made to make the catch.

6. Write a program that defines class `fraction` and does the usual operations with fractions: simplifying (that is, writing in lowest terms), adding, subtracting, multiplying, dividing. Its data members must be `int numer` and `int denom` (the numerator and denominator of the fraction). It must have the following function members:

- Constructor and default constructor.
- `setNumer` , `setDenom` , `setFrac` to set the numerator, denominator, and both at once. Make sure to check for valid input so that `denom` is not 0, and quit the program if it is (or, more advanced, throw an exception, see examples in Chapter 9 or a more detailed treatment in Chapter 17). Also, fractions should be stored in lowest terms, so you have to make use the function `simplify` in the set functions and in the constructor.
- `simplify` that takes a fraction object and rewrites it in lowest terms.
- `getNumer` , `getDenom` whose output are the numerator and denominator, respectively, of the calling object.
- `printFr` that outputs the fraction as, for example, 5/6.
- `addFrac` , `subtFrac` , `multiply` , `divide` that do the corresponding operations. Make sure to simplify the function after performing the operation.
- Write a `main` part of the program to show how the class works. You may ask the user for input or write and example of each function.
- In a second phase, enhance the class by overloading the operators $+, -, *, /$ (Chapter 10).

<u>Advanced</u>:

7. Write a program defining class `infinteger`. Its objects are vectors of arbitrary length (you can use the `vector` built in class for this) with short integer entries between 0 and 9. This allows to perform arithmetic with arbitrarily long integers. Define addition, subtraction, multiplication, division, and remainder functions (%). Do this by overloading $+$, $-$, $*$, $/$, and %. Also, overload these operators so that int + infinteger (and all the other operations) work properly (that is, ints should form a subset of infintegers).

8. Write a program to implement a version of peg solitaire. It is played on a rectangular grid of holes. Some of the holes will have pegs in them and some will not. On a turn, the player can jump a peg A over another peg B (in a horizontal or vertical fashion), if there is a space just after B and this space is unoccupied. Peg B is then removed from the board (but peg A stays). The user does this till she has no moves, the goal being to have as few pegs left as possible (having just one left is the best possible under some configurations).

You must 1) create a class named `PegSolitaire`, which will be described below, 2) create other classes as needed, which will all be known by the top level class PegSolitaire, 3) write a `main` program that uses the class `PegSolitaire`, to create an interactive game of Peg Solitaire.

In the following discussion, by a position we mean a tuple (X, Y), where X and Y are positive integers; a position is meant to refer to a location on the board. The class `PegSolitaire`, must have at least the following member functions (it may have more; also, some functions may be implemented as friends instead of members):

- The constructor takes two positive integer arguments `xSize` and `ySize`. This creates a `xSize` by `ySize` grid of holes as the playing board.

- `placePeg`: Takes integer arguments X and Y, and places a peg at row X and column Y.
- `removePeg`: Takes integer arguments X and Y, and removes a peg at row X and column Y.
- `jumpPeg`: Takes two positions and attempts to move the peg at the first position to the location indicated by the second position. If the move is legal, it is carried out, along with all appropriate adjustments to the board, and True is returned. If there is anything that forbids this move, the the method does not change the board, and returns False.
- `isStuck`: Takes no inputs, and returns True if there are no possible moves left, and False if there are possible moves.
- `display`: This creates a text output of the current state of the board. It should just be a grid of letters with the character "`P`" for a hole occupied by a peg, and "`o`" for a hole not occupied by a peg. An example of a possible result of display on a 2 by 3 board:

The member data for this class is a two-dimensional array `board` of the size of the board (that is, `xSize` rows and `ySize` columns) containing boolean values (0 or 1).

For example, for the following output one would have defined an object `PegSolitaire P{2,3};`, and set its data member board (via the member function `placePeg`) as follows: `board[0][0]` is 0, `board[0][1]` is 1, `board[0][2]` is 1, `board[1][0]` is 1, `board[1][1]` is 0, `board[1][2]` is 1:

o P P
P o P