Fifth programming exercise:

Connect-four

The idea for this assignment is to implement the two-player game connect four. The assignment is comprised of two parts:

1. Programming connect four on an m x n board (m the number of rows, and n the number of columns, both at least 1) and a simple AI computer opponent. The game is initially displayed in a very simple fashion.
2. Playing the game with fixed size 6x7 in a graphical layout made with QT.

You may, and it is encouraged to work in pairs of two on this assignment! The workload might be somewhat high, and learning to program in teams is also an important skill.

The game is played as follows: On an m x n board two players take turns to place a chip of their own color on the board. The chip is played in one of the 7 columns and falls down to the lowest spot of that column. You can not play a chip in a full column. The game stops when one of the players has 4 chips connected in a single line, horizontally, vertically or diagonally. The player that has connected those 4 chips wins. If none of the players manages to connect 4 and the board is full the end-result is a draw.
The game that we will initially program should be playable multiple times in a row. At the start of the game the player specifies the size of the board (maximum of const int maxrows/maxcols).
If it's the users turn he can make a legal move, or take back his last move (and the computer's move in between as well). The computer will either perform random actions (options 1), or play according to a certain strategy. The user can specify during the game which strategy the computer uses to play. It should also be possible that the computer plays against himself. In that case the user chooses a strategy for both computer players.
Finally, the player can also request the amount of ways the game can be continued from

there. Naturally, this only works if the number of next possible board states is somewhat restricted, the actual number of possible board stage is HUGE (note: this is not the amount of next moves! Some moves lead to a finished game, etc.
For the strategy of the player you can make something yourself. It is not required of you to make a tree structure of next board positions if you do not want to. At the very least you should consider all possible moves, from which you choose wisely somehow. The program should at the very least immediately win if it can win, or stop a win from happening if the situation arises for the opponent. You for example use a simple evaluation function for each of the next move's board states to choose which move to play. The user should be able to set at least one parameter to change the weighting of the moves in a certain way, or to determine the difficulty of the program. For example, if you look at the amount of connected 3's and the amount of blocked attempts of the opponent for the evaluation, the parameter can give the weighting between these. Or for tree searching, it can specify the depth of search (or certain factors in the heuristics, you could even optimize these by letting the computer play itself!).

First you show the board in a simple ascii format in the command line. Later on we will use QT to make this a bunch prettier. When writing your code, you should take into account the fact that we will make a graphical interface later on. Use proper functions and classes for this reason. For the graphical version, the user should be able to click on the columns to play a certain move.

Eisen aan de programmatuur

Read all input neatly with cin.get(), or the function 'readNumber'() from previous exercises, and make sure that wrong input is handled correctly.
Write a class 'board' which stores the current game state. This can be done with a large enough two-dimensional array for example (dynamic array?!). On top of this, it should also include the player to move and the actual size of the board. It also needs several member-functions (fill in the parameters and returns yourself):

- board (...): the constructor (initialize everything).
- beginPosition (...): user specifies the size of the board with this.
- print (...): print the current board to the command line.
- playRandomMove (...): make a random move, use the standard rand() functions.
- playStrategyMove (...): use a certain AI smart strategy to make a move.
- chooseColumn(...): The use chooses a column.
- isLegal(...): is this move legal?
- makeMove (...): voert een gegeven (toegestane) zet uit.
- isFinished (...): is the game over? not yet? who won? Is it a draw?
- calculateFollowingPossibleGames (...): Recursively calculate the following number of possible

games, restrict this to cut off at a certain maximum amount of possible games (to avoid overflow, just state that it's more than const int MAX in that case).

The human player should be able to take back moves. This means that all previous board positions (so not the moves itself, the boards!) in which the player is to move should be stored. This can nicely be done with a class 'game'. When a player makes a move, the old board position is stored in a heap (a member-variable of 'game', so NOT a member variable of board). Look up what a heap is on the internet, and write a nice class that represents this data structure. It should at least have the following member functions:

- putOnHeap (...)
- getFromHeap (...)
- heapIsEmpty (...)

Because we will couple the code to a QT implementation it is wise to make the code as modular as possible. Thus split your code in multiple files, each class having it's own .cpp and .h files. Don't forget the main() file.

Software tools

We will be using a combination of software tools for this assignment. First of all, all code should be stored online in a github repository. Read the github introduction. Github makes collaboration of programming easier, and automatically stores all your code's versions for you. This is amazing for when you make mistakes and want to revert them.

We will also use cmake. Read the cmake introduction. Cmake is used to be able to collaborate even across operating systems. It also makes project file creation a lot easier, as you can auto-include libraries like QT. Make sure that you out-of-source build everything, so that your code-base stays clean and easily shareable through github! (This includes the .cxx and moc files of QT)

Lastly we will be using QT for designing a layout for the program. Integration of QT with your compiler is easy through cmake. Read the QT install and usage instructions. You can use QT creator to generate simple layouts with buttons and displays. Each of these buttons and displays can be clicked and interacted with. When you click a button for example, it triggers a function "onButtonPressed" with the ID of that button. Now you can call your 'makeAMove(int column)' function from that function. It's that easy!

Turn in your code by giving a link to your github repository.