# CompSys202/MechEng270
# Data Structures and Algorithms
# Assignment #3 (20% of final grade)
# Due: 11:00am, 21 October 2016

## Learning outcomes

The purpose of this assignment is to target the following learning outcomes:

- Demonstrate a basic understanding of C++ development.
- Continue to develop and practice your skills in OOP design.
- Practice using data structures and algorithms effectively.
- Apply good coding practices such as naming conventions and code style.

## 1 Connect Four!

The game of Connect Four is described by Wikipedia as follows:

> [Connect Four] is a two-player connection game in which the players first choose a color and then take turns dropping colored discs from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The objective of the game is to connect four of one's own discs of the same color next to each other vertically, horizontally, or diagonally before your opponent.

In this assignment you will construct a simple game of Connect Four. This game will be played between two human players, and will be played until one player wins or a stalemate is reached. A simple demonstration of the Connect Four game is displayed in figure 1.

## 2 Tasks

### 2.1 Player Information [Task 1] (15%)

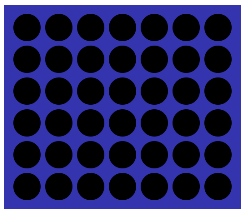
#### 2.1.1 Implement the Player Class

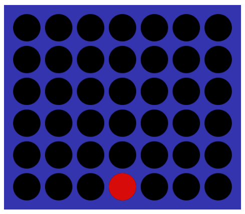
The `Player` class will store information about each player such as their name and color. A player has three properties:

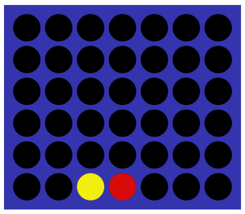
- The player's name, e.g. "Nick".
- The player's score.
- The total number of games won by the player.

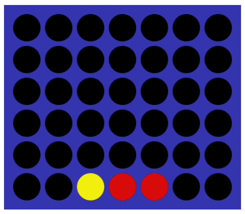
The player's name can only be set in the constructor, while the score should be initialized to 0 and can only be increased by 1 or reset using the methods of the `Player` class. The total number of games won by a player is initialized to zero when the Player is constructed, and then can only be increased by 1.

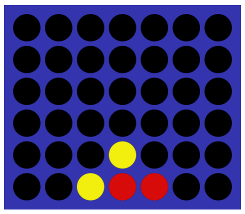
Complete the implementation of the `Player` class as described in the file Player.hpp by adding code to Player.cpp.

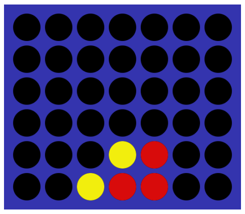

(a) The game is created with a standard 6 row by 7 column grid. At the beginning of the game the game grid is empty. The top row of the grid is row 0 and the left-most column is column 0.

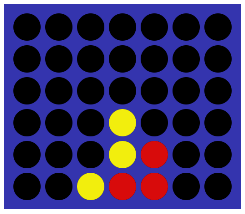

(b) The game starts with player one's turn. In this case player one has inserted a disc (red) into column 3. The disc falls to the bottom of the grid and stops at row 5.

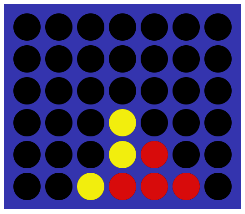

(c) Player two plays their first turn by inserting a yellow disc into column 2. The disc falls to the bottom of the grid to come to rest next to the disc placed by player one.

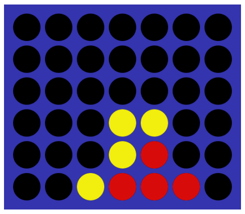

(d) Player one plays their next move, placing a red disc in column 4.


(e) Player two plays their next move, inserting a yellow disc in column 3. This disc falls towards the bottom of the grid and stops at row 4, where it is stacked on top of the first disc placed by player one.


(f) Player one plays their next move, inserting a second red disc into column 4. This disc falls towards the bottom of the grid, stacking on top of a previously placed red disc.


(g) Player two plays their third move, inserting a yellow disc into column 3, where it comes to rest in row 3 on top of two other discs already inserted into that column.


(h) Player one plays their fourth move, inserting a red disc in column 5 that falls to the bottom of the grid (row 5).


(i) Player two plays their fourth move by inserting a yellow disc into column 4 which stops at row 3 due to other discs previously inserted into that column.

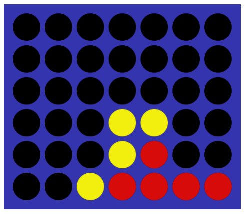

(j) Player one plays their fifth move, inserting a disc into column 6. This disc falls to the bottom row (row 5) and completes a horizontal row of four red discs. Connect four! Player one wins!

Figure 1: A demonstration of the Connect Four game in action.

### 2.1.2 Testing the Player Class

Testing code in this assignment works in a similar way to Assignments 1 and 2. When you are ready to test your code, you can compile and run the tests by executing the command `make test` from the command-line console. The tests for the `Player` class are already enabled in test.cpp.

If you need further information on how to test your code, refer to the testing sections of the assignment briefs for assignments 1 and 2.

## 2.2 Tracking Player Moves [Task 2] (20%)

### 2.2.1 Implement the Grid Class

The `Grid` class is represents the vertically suspended grid into which players insert their colored discs. Each cell of the grid can be represented by a value from the `Cell` enum declared inside the `Grid` class. Each cell will either be empty, or hold a disc inserted by one of the two players. The number of cells is determined by the number of rows and columns in the grid, and your `Grid` class constructor should allow the creation of a game `Grid` with any size greater than 4 rows by 4 columns. In the case where one of the initialized dimensions is smaller than 4, that dimension should be modified to be exactly 4. For example, if a grid is initialized with 2 rows and 6 columns, it should be automatically corrected to 4 rows and 6 columns inside the `Grid` constructor.

During a game, discs may be inserted into the `Grid` by either player using the `insertDisc` method, which takes a `Cell` value representing the new disc and a column of the grid that the disc should be inserted into. When a disc is inserted into the grid it should be inserted on top of the other discs in the grid, or at the bottom of the grid if there are no other discs in that column. This emulates the physical effect of inserting the disc into the grid in a physical game of Connect Four.

To implement the game, other classes will need to be able to access elements in the grid. This information will be accessed using the public `cellAt` method, which returns a value from the `Grid` given a specific row and column. This method should return `GC_EMPTY` for any row or column outside the bounds of the grid. The bounds of the grid will be returned by the `rowCount` and `columnCount` methods.

Once the game is over, the game grid needs to be emptied of all inserted player discs so a new game can be started. In the `Grid` class this is done using the `reset` method.

Complete the implementation for the `Grid` class in Grid.cpp based on the declarations already created for you in Grid.hpp. Make sure to read the comments above the class and each method in the class for more information on how to implement them properly.

### 2.2.2 Testing the Grid Class

The tests for the `Grid` class can be enabled by uncommenting the following line at the top of test.cpp:

```
// #define ENABLE_T2_TESTS
```

Once the Task 2 tests are enabled, run `make test` to execute all of the currently enabled tests. Remember that not all aspects of the `Grid` class may be tested in the tests you have been given, so **don't assume the task is complete once all of the tests pass**.

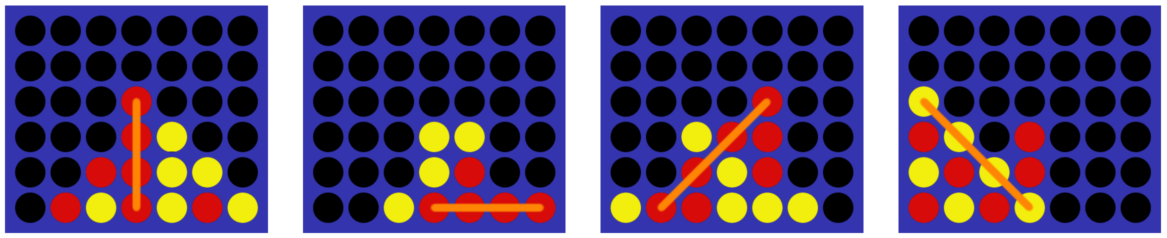
## 2.3 Creating a Connect Four Game [Task 3] (30%)

### 2.3.1 Implement the Game Class

The `Game` class *composes* the `Grid` class, *aggregates* multiple instances of the `Player` class together and performs the game logic. A game is created using the default constructor for the `Game` class. The `Game`'s status method will return `GS_INVALID` until two `Player`s are assigned to the `Game` using the `setPlayerOne` and `setPlayerTwo` methods and a `Grid` is assigned using the `setGrid` method.

The game will be executed one turn at a time using the `playNextTurn` method, with the `nextPlayer` method providing (read-only) information about which player's turn it is next. Once the game is complete, the status method will return `GS_COMPLETE`, and return the winning player via the `winner` method. Note that if the game is a draw, the `winner` method will return a null pointer (`0`), but the Game's status will still be `GS_COMPLETE`. If a new game is requested, the state of the current game can be cleared using the `restart` method.

You will need to create code to calculate the status of the game by traversing the grid and identifying 4 or more discs from the same player in a row in any vertical, horizontal or diagonal orientation. Once one player gets **at least** 4 discs in a row in any direction, they win the game. When a player wins the game, the `Game` state should be updated accordingly so that the game status is set to `GS_COMPLETE` and the corresponding `Player` is returned via the `winner` method. When a `Player` wins a game, their score and total number of wins should be increased by 1 using the appropriate methods of the `Player` class (a tie does not count as a win for either player). When the game is restarted (using the `restart` method), each `Player`'s score should be reset. The possible directions discs can be connected in to win the game are shown in figure 2 below.



(a) A vertical connect-four for the red player (player one).

(b) A horizontal connect-four for the red player (player one).

(c) A diagonal-up connect-four for the red player (player one).

(d) A diagonal-down connect-four for the yellow player (player two).

Figure 2: Completed games showing a winning combination in each possible direction. The winning combination of discs has been highlighted in each example.

Complete the implementation for the `Game` class in Game.cpp based on the declarations already created for you in Game.hpp. Make sure to read the comments above the class and each method in the class for more information on how to implement them properly.

### 2.3.2 Testing the Game Class

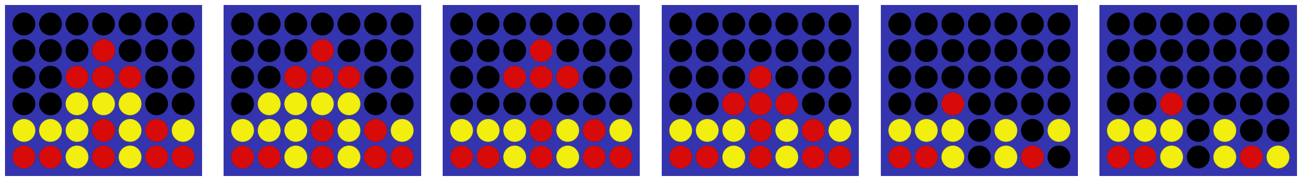
The tests for the `Game` class can be enabled by uncommenting the following line at the top of test.cpp:

```
// #define ENABLE_T3_TESTS
```

Once the Task 3 tests are enabled, run `make test` to execute all of the currently enabled tests. Remember that not all aspects of the `Game` class may be tested in the tests you have been given, so **don't assume the task is complete once all of the tests pass**.

## 2.4 Super Connect Four [Task 4] (15%)

In this task you will implement a variant of the classic Connect Four game we will call "Super Connect Four". In this variant of the game when a player connects four or more discs in a row, those discs will be removed from the grid. This will cause any discs above the removed discs to move down to fill the space in the grid left by the removal of the connected discs. One point will be added to the score of the player who connected the discs. If another line of four or more discs belonging to one player exists after all affected discs have been moved, then this process is repeated. This process continues until the grid reaches a stable state where no more lines of four discs exist and no more cells need to be moved. One turn of such a game is demonstrated in figure 3.



(a) The initial game board state for this example. Both players have already made several moves.

(b) Player two plays the next move, forming a chain of four discs horizontally in row 3. Player two's score is increased by 1.

(c) The connected discs are removed from the game grid, leaving blank spaces in the grid.

(d) The remaining discs above the ones that were removed are moved down vertically to fill the empty space in the grid. Moving the discs forms two new chains of discs belonging to player one.

(e) The grid is scanned, and two new chains belonging to player one are found. Player one's score is increased by 1 and the discs are removed from the grid.

(f) The disc that was above the removed discs is moved to the bottom of the grid. The grid is re-scanned but no new chains of discs have been formed and no discs need to be moved down. Player two's turn ends.

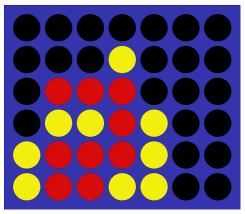
Figure 3: A demonstration of one turn in the "Super Connect Four" game.

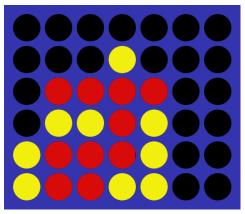
The process for each turn of the game should be as follows:

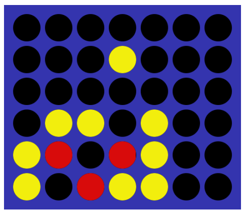
1. Make the player's move by inserting their token into the grid.
2. Check to see if any chains of 4 or more discs in any direction have been formed.
3. Increase the score by 1 for any players whose discs have formed chains of 4 or more. Each player's score may only increase by 1 each time the grid is checked (even if players get multiple simultaneous chains in a single check).
4. Remove all discs in all chains from the grid.
5. Fill all spaces in the grid created in the previous step by moving the discs above the spaces down.
6. Repeat from step 2 until no more chains of discs are found and the grid has reached a stable state.

Make sure that disc the disc removal and movement steps are fully completed before the grid is re-scanned. Failing to do so may cause additional matches while the grid is still between stable states. This is important for scenarios such as those shown in figure 4 and figure 5.

Just as in normal Connect Four, players take alternating turns until the game is complete. This game will end once there is no more space in the grid for either player to insert discs. The winner is the player with the highest score at the end of the game.


(a) The initial game board state for this example. Both players have already made several moves.


(b) Player one plays the next move, forming a chain of four discs horizontally in row 2 as well as a diagonal chain of four discs. Player one's score is increased by 1.


(c) The connected discs are removed from the game grid, leaving blank spaces in the grid.

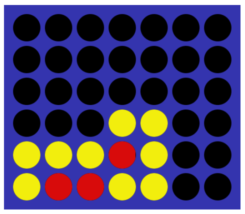
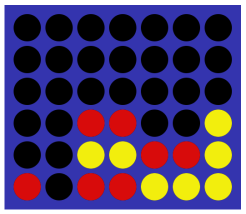
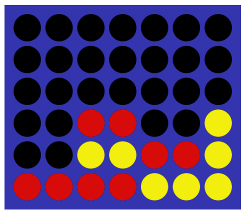
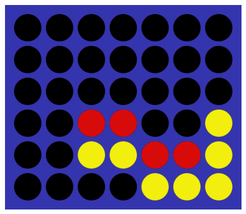
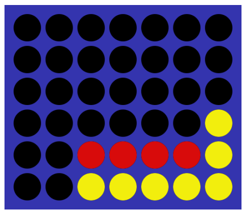

(d) The remaining discs above the ones that were removed are moved down vertically to fill the empty space in the grid. The grid is stable, so this is the end of player one's turn.

Figure 4: A demonstration of a turn in the "Super Connect Four" game where multiple chains of discs are formed for a player in the same move.


(a) The initial game board state for this example. Both players have already made several moves.


(b) Player one plays the next move, forming a chain of four discs horizontally in row 5. Player one's score is increased by 1.


(c) The connected discs are removed from the game grid, leaving blank spaces in the grid.


(d) The remaining discs above the ones that were removed are moved down vertically to fill the empty space in the grid. Moving the discs forms new chains of discs for both players.

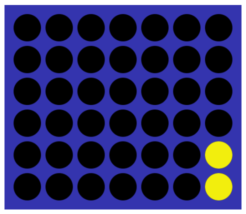

(e) The grid is scanned, and new chains belonging to both players are found. Both player's scores are increased by 1 and the discs are removed from the grid. There are no more discs to move down. Player one's turn ends.

Figure 5: A more complex example of the behavior of the "Super Connect Four" game.

### 2.4.1 Implement the `SuperGame` class

Implement the `SuperGame` class in SuperGame.hpp and SuperGame.cpp by deriving from the `Game` class, therefore reducing the amount of code you need to write by inheriting some of the functionality of the `Game` class. The `SuperGame` class should use the interface defined by the `Game` class so that it can be substituted in place of the `Game` class to provide a new game experience with the same user interfaces.

The scoring for the SuperGame is different to the original Game class. When a player gets a chain of four discs their score still increases by 1. However, the game does not end until the game grid has been filled, so the "winner" of the game is determined by the Player with the highest score once the game grid has been filled and no more moves can be played. Once the game is over the winner has their win counter increased by 1. Both player's scores are still reset when the game is restarted.

### 2.4.2 Testing the SuperGame class

The tests for the `SuperGame` class can be enabled by uncommenting the following line at the top of test.cpp:

```
// #define ENABLE_T4_TESTS
```

Once the Task 4 tests are enabled, run `make test` to execute all of the currently enabled tests. Remember that not all aspects of the `SuperGame` class may be tested in the tests you have been given, so **don't assume the task is complete once all of the tests pass**.

You have also been given a command-line interface (main.cpp) which can be used for advanced testing of your code by playing a Connect Four or Super Connect Four game. To build and run the command-line interface, use the command `make run` from the command-line console.

You may share any tests you write for this assignment with classmates on Piazza.

### 2.5 Code Inspection (20%)

Your code will be inspected for:

- Effective use of OOP
- Consistent coding style
- Code commenting
- Memory management

Marks will be allocated for each by your marker.

## Important: how your code will be marked

- Your code will be marked using a semi-automated setup. If you fail to follow the setup given, your code **will not be marked**. All submitted files must compile without requiring any editing. Use the provided tests and `Makefile` to ensure your code compiles and runs without errors on the university Ubuntu lab PCs. Any tests that run for longer than 10 seconds will be terminated and will be recorded as failed.
- Although you may add more to them (e.g. member variables, `#include` statements, or helper functions), you must not modify the existing interface of classes defined in the following files (e.g. do not delete or modify the existing functions declared):
  - `ConnectFour/Player.hpp`
  - `ConnectFour/Grid.hpp`
  - `ConnectFour/Game.hpp`
- Do not move any existing code from the `ConnectFour` directory, and make sure all of your new code files are created inside the `ConnectFour` directory.
- You may modify `test.cpp` as you please (for your own testing purposes); this file will not be marked at all. Be aware that your code must still work with the original `test.cpp`. Do not change the location of this file. You are free to share any additional tests you create with classmates on Piazza.
- Your code will also be inspected for good programming practices, particularly using good object-oriented principles. Think about naming conventions for variables and functions you declare. Make sure you comment your code where necessary to help the marker understand **why** you wrote a piece of code a specific way, or **what** the code is supposed to do. Use consistent indentation and brace placement.

## Submission

You will submit via Canvas. Make sure you can get your code compiled and running with test.cpp (`make test`) on the university Ubuntu computers. Submit the following, in a single ZIP archive file:

- A **signed and dated declaration** stating that you worked on the assignment independently, and that it is your own work. Include your name, ID number, the date, the course and assignment number. You can find this Cover Sheet on Canvas. All code will be checked against other submissions. Submissions detected as being similar to others will ensure that the students involved are forwarded to the Misconduct Committee.
- The entire contents of the src_for_students folder you were given at the start of the assignment, including the new code you have written for this assignment. Ensure you **execute `make clean` before zipping the folder** so your submission doesn't include any executable files (your code will be re-built for marking).

## Academic honesty

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help. Under no circumstances should you take or pay for an electronic copy of someone else's work.
- All submitted code will be checked using software similarity tools. Submissions with suspicious similarity will result in an Investigative Meeting and will be forwarded to the Disciplinary Committee.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never show or give another person your code.
  - Never put your code in a public place (e.g. Reddit, Github, forums, your website).
  - Never leave your computer unattended. You are responsible for the security of your account.
  - Ensure you always remove your USB flash drive from the computer before you log off.

## Late submissions

Late submissions incur the following penalties:

- 15% penalty for zero to 24 hours late
- 30% penalty for 25 to 48 hours late
- 100% penalty for over 48 hours late

**You must double check that you have uploaded the correct code for marking!** There will be no exceptions if you accidentally submitted the wrong files, regardless of whether you can prove you did not modify them since the deadline. No exceptions.