University of Central Lancashire UCLan Department of Computer Science	UCLan Coursework Assessment Brief		2022 - 2023
	Module Title: **Advanced Programming with C++** Module Code: **CO2402**		Level 5
	UCan't: The Winnowing	This assessment is worth 50% of the overall module mark	

Introduction

For this assignment, you will work towards implementing a simplified version of a classic trading card game. This is not an interactive game you can play, but rather a simulation of two (or maybe more) players taking turns over a set number of rounds. Play is to be automated - there is no artificial intelligence, and no user input. All output will be text based and directed towards the console – there are no graphical elements to this assignment.

In the game, prestigious Professors navigate their way through University life. Who can defend themselves against the admonition of the administrative system? Who will avoid falling foul of management's malevolence? Supported by their loyal cohort of students, which Professor will survive the semester with their academic prestige intact? Find out in UCan't: The Winnowing!

I expect you to complete this project in your own time outside of scheduled labs.

- This is an **individual** project and no group work is permitted.
- Do not diverge from the assignment specification. If you do not conform to the assignment specification then you will **lose marks**. Ask for clarification if you are unsure!

Avoiding Plagiarism

- You will be held responsible if someone copies your work - unless you can demonstrate that have taken reasonable precautions against copying.

Learning Outcomes Assessed (see module descriptor for full list)

- Make an informed choice of implementation method for a given problem
- Implement and document a structured program to meet a given specification
- Select and apply appropriate data structures and algorithms to a given problem

Deliverables, Submission and Assessment

- There are **three** deliverables – your **source code**, a **report**, and an **in-person demonstration**
    - To get a mark above 0 you must submit all 3 deliverables.
- Submission is electronic, via Blackboard and in-person in the lab.
    - For the **source code**, upload **ONLY** the **.cpp** and **.h** files. DO NOT submit the solution file!
    - Make sure you include your name as a comment on the first line of all your source code files. Also include your name at the top of your **report**.
    - See the section on page 8 for a summary of what to include in your report.
- Assessment will be in-person in your scheduled lab during the week following the deadline.
    - Failure to attend the lab demonstration will result in your work not being marked.
- You may use whichever IDE you prefer to develop your solution. I am only interested in the source code and output.

Overview

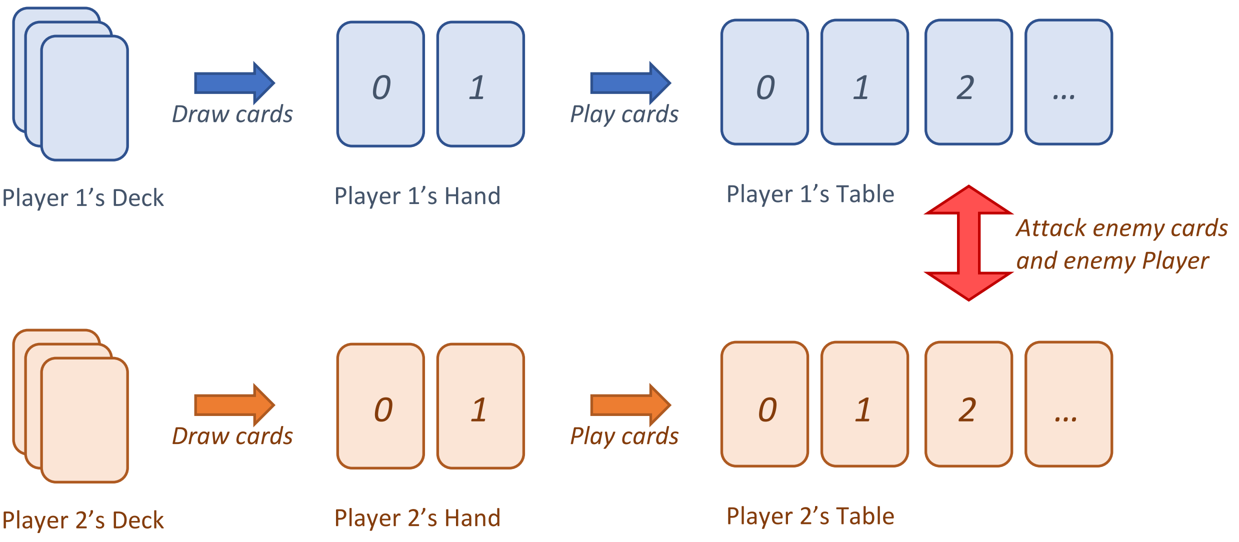
In your simulation, the game should (initially) have two players, each taking the role of a **Professor**. Player 1 is ***Professor Plagiarist***, and Player 2 is ***Professor Piffle-Paper***.

Each Professor has a Deck of cards, the details of which will be read from a data file at the start of the game: "plagiarist.txt" and "piffle-paper.txt".

The Game begins by each Professor drawing the first Card from their Deck to become their *starting Hand*. The game then proceeds as a series of **rounds**.
In each Round, players take it in turn to do the following:

- Draw one Card (in order, from their own Deck) into their Hand.
    - There are never more than 2 cards in a Hand.
- Choose a Card from their Hand to play to their Table.
    - There is no limit to the number of cards which can be active on the Table
    - You will represent choosing which card to by simulating tossing a coin (i.e. generating a random number between 0 and 1 using the code I have provided – *see appendix*).
- Use their Cards on the Table attack their esteemed colleague and their cohort.

Although there is no graphical element to the program, the following layout of a physical card game may be helpful for you to think about how the cards are arranged in various collections.



Cards have *power* and *resilience* points.
When one Card attacks another, it uses its *power* to reduce the other Card's *resilience*.
When a Card's resilience is exhausted, it is discarded from the Table.

In the full game, there are different *types* of Card, which behave differently when played.
To gain a "good" mark (above 60%) you need to implement the different behaviour of the Card types using polymorphism.

Professors also have *prestige*, which can be reduced by the Cards on the opposing Player's Table.

The game will be played for a maximum of 30 rounds. If, at any point, a Professor's prestige is reduced to zero, they are fired from the University, and their opponent has won. Otherwise, the most prestigious Professor after either 30 rounds (or the cards run out) is the winner.

Program Specification

You should implement the features described below **in order**. To be eligible for a mark within any classification, you must have at least *attempted* **all** the features for **all** the previous classifications.

Basic Scenario = bare pass (40% max)

If you implement only what is in this section and demonstrate it running to completion, you will get 40%. No more, no less.

- Create a class called CCard to represent the cards held by each player.
- Read in the data files plagiarist.txt and piffle-paper.txt (*see appendix*) and use it to set up an array of CCard objects for each player.
  - You do not need to use pointers or dynamic memory allocation at this level.
  - Each line in the file represents one card.
    - The first number in each line represents the card's **type**. For a bare pass, **Only store cards of type 1 in your Deck arrays** – skip over all other card types.
    - Following the type number is the cards's *name*, and then its *power.* You will need to store the name and power as data in the CCard objects as you create them.
- Use a structure SProfessor to represent the professors. You need to store their *name* and *prestige*. Each professor begins with 30 prestige points.

In your main program, simulate playing 30 rounds of the game as follows. Prof. Plagiarist always starts:

- When the game is started, a welcome message is displayed. The format of the message is:
  ```
  'Welcome to U-Can't. Let the winnowing begin...'
  ```
- After each player draws the card into their starting hand, output the message:
  ```
  '<Player> starts with <Card>'
  ```
- At the start of each round, the round number should be output thus:
  ```
  'Round <round number>'
  ```
- For each turn, the player draws (removes) a card from their Deck, and adds it to their Hand.
  - The name of the player and the card they draw is displayed. Output the message:
    ```
    '<Player> draws <Card>'
    ```
  - On the next line the name of the player and the name of the card they have "chosen" at "random" to play from their hand are displayed. Output the message:
    ```
    '<Player> plays <Card>'
    ```
  - In this simple version of the game, there is no Table, and the card played directly attacks the professor's colleague (=enemy!), and is then discarded.
  - Reduce the colleague/enemy player's Prestige by the Power of the card, and output:
    ```
    '<Card> attacks <Enemy>. <Enemy>'s prestige is now <prestige>'
    ```
- At the end of the game you should output the final amount of prestige each player has, and also which player has won. Output the following messages:
  ```
  'Game Over'
  '<Player1>'s prestige is <amount>
  '<Player2>'s prestige is <amount>
  '<Player> wins.'
  ```

Notes:

- The cards must be *drawn* from the decks *in the same order* they appear in the text file.
- Your code must follow the style guide precisely, and be well commented throughout.
- When the code runs, there should be no user interaction. Specifically, there should be no need to press a key in-between rounds. You may wish to introduce such a feature to help with development/debugging, but it should be disabled in the code you submit and demonstrate.

Pass Mark = third classification (40% +)

Implement the Bare Pass criteria, with the following additions/modifications:

- All objects (cards and professors) should be stored in *dynamic* memory
  - Cards in the Decks should be accessed via an array (or better still a vector) of *pointers*.
  - The Hands should also be (collections of) pointers.
  - Your code should have memory-leak detection.

- Ordinary Students form the rank and file of a Professor's cohort, represented by the cards on the player's Table. Declare a CStudent class for students.
  - A student is now *A Kind Of* card.
    - The student class should be derived from the card class.
  - In the .txt files, rows beginning with '1' denote ordinary student cards.
    - You should now read in ALL the cards (including those which are not ordinary students) and set up the correct object type – either a generic CCard or a CStudent.
  - As well as their type, name and *power*, students also have a *resilience* value.

  The details of ordinary student cards found in the input text files are given in the table below: All cards have a type number, a two-word name, and then some additional numbers.

Type	Name	Power	Resilience
1	Hard Worker	2	4
1	Smarty Pants	4	2
1	Lazy Complainer	2	4
1	Hung Over	2	1
1	Sneaky Cheat	2	4
1	Basket Weaver	1	1
1	Clever Dick	2	4
1	Teachers Pet	4	4
1	Scanner Scrammer	1	1
1	Bored Rigid	2	2

- The players should now be able to *draw* and *play* **all** cards (not just students) from the deck (with output as in the bare pass scenario), but only Student cards should activate to attack the enemy cohort.

- If a professor's prestige is exhausted before the end of the 30 rounds, the game should end immediately with the sacking of the professor.
  `'<Player> has no prestige and is sacked!'`

Lower second classification (50% +)

For this grade you need to *attempt* a **polymorphic** solution.

- You will need a *hierarchy* of classes for the different card types.
- When it comes to implementing the different behaviour of the cards as each one is played or activated, the functionality is devolved to the lower levels of the hierarchy:
  - You should have a collection of pointers of the *base class* type CCard*, but the bespoke implementation code should be written *in the derived classes*.
  - You should **not** call methods on the derived classes directly.

- You *must* use STL containers to store/access the CCard pointers, for the players' Decks, Hands and Tables.

- Most cards are Students and should now be played by moving them from the Hand to the Table (and NOT deducting prestige from the enemy professor).
    - Other cards, such as Admin duties, behave differently, and their rules are given separately.

- The cards on the table should be thought of as being in a kind of stack, with cards being pushed onto the top of the stack as they are played (although they are removed from anywhere).
  After playing a card, the name and resilience of each of the players cards on the table should be output (with the top of the stack output first):
  `'Cards on table: <Card> (<Resilience>), <Card> (<Resilience>),'` etc.

- The final operation of each players' turn is to activate the students in their professor's Table.
    - Each student card should activate in order, with the card most recently played first.
    - When a student activates, it will attack an enemy student if one is available. Otherwise, it will attack the enemy professor directly.
    - The enemy to attack is chosen at random, using the random function provided
        - Hint: Pass the length of the enemy's table as the parameter.
    - When a student is attacked, it's *resilience* is reduced by the *power* of the attacking student.
        - When a student's resilience reaches zero, it quits University, is removed from the table and destroyed(!)

  The ouptput after each activating card should be either
  `'<Card> attacks <Enemy Card>. <Enemy Card>'s resilience is now <resilience>'`
  Or if the enemy card's resilience reaches zero
  `'<Card> attacks <Enemy Card>. <Enemy Card> is defeated'`
  Or if the enemy professor is attacked directly
  `'<Card> attacks <Enemy>. <Enemy>'s prestige is now <prestige>'`

- Implement two additional classes derived from CCard for the Admin duty cards detailed below.
    - Admin cards all activate immediately they are played, and are then discarded.
    - They are not added to the table.

  Card Type 2 – Admin – Plagiarism Hearing
  Dealing with a case of suspected plagiarism brings the miscreant into disrepute… but is it the professor or one of their students who has been caught copying from the internet?
    - Causes 3 points loss of resilience to either a single enemy student OR 3 points loss of prestige to the enemy professor. The target entity is chosen at random.
        - Hint: number the professor one more than the greatest card to choose them…
    - Output a message in the usual format saying which card attacks what entity, and the outcome for the attacked entity.

  Card Type 3 – Admin – Course Accreditation
  The task of preparing documents for an accreditation visit is tedious and thankless, and in the end everybody suffers!
    - Causes 1 point loss of resilience to the entire enemy student cohort AND 1 point loss of prestige to the professor.
    - Output something suitable for all the entities which are attacked and their outcomes.

Type	Name	Power
2	Plagiarism Hearing	3
3	Course Accreditation	1

Upper second classification (60% +)

For this grade, the polymorphic aspect of your solution must be fully correct, and the polymorphism must be serving a genuine purpose. You will not be awarded marks if you just plonk the keyword "virtual" into your code, whilst the work within your code is still done in a procedural fashion.

- Players must be implemented as classes (not variables in main or structs).
- At this level there should be no global variables in your code. (Global constants are allowed.)
- There must be no memory leaks when you run your code.
  - Proof of this should be included in the documentation.

- Implement additional card types 4 and 5:

  Card Type 4 – Admin – Feedback Forum

  Attend the Feedback Forum to find out what students think about various modules this semester. Will it be congratulations from the boss and a boost for your team, or a telling-off for your esteemed colleague…?
  - Choose a target at random from all the students and professors – friend and enemy.
    - If the target is an enemy, their resilience/prestige is reduced by 2.
    - If the target is friendly, their resilience/prestige is *increased* by 2.
  - Output a suitable message.

  Card Type 5 – Student – Industrial Placement

  We all know that going on an industrial placement year helps students to grow in skill and confidence… so much so that the more they go to work against the enemy team, the more their resilience grows.
  - An extra special kind of student! The industrial placement student attacks in the same way as all the other students, but additionally boosts its own resilience by 1 after each attack!

Type	Name	Power	Resilience	Boost
4	Feedback Forum	2		2
5	Industrial Placement	2	3	1

First classification (70% +)

The implementation at this level should make use of object-oriented methods throughout (not just the polymorphic parts). This will mean the implementation of several classes.
  - One of the classes must be the game itself, which will act as a manager class.
  - All your classes should have their own header file and source file.

- The STL syntax for vectors is ugly and cumbersome.
  - Use typedef and auto to clean up any ugly syntax of vector declaration.

- Use smart pointers such that you do not directly invoke dynamic memory allocation anywhere in your code.
  - Use the C++14 syntax to avoid the keyword new.
  - The only allowable use of raw pointers is if you need to implement an *observer*.

- Add a 3rd and 4th player to the game.
  Player 3 is ***Professor Pointless*** and Player 4 is ***Professor Perdition***.
  - If you have designed your solution well, this should literally be adding 2 lines of code.
  - Read from the files perdition.txt and pointless.txt

- Implement card type 6:

  Card Type 6 – Student – PASS Leader
  PASS leaders are known to hang around in groups and their power grows in proportion to their number…
  - For each PASS Leader in the cohort, the power of all of them is increased by 1
    - i.e. if there were only one PASS Leader on the table, its power would be 1. If there were three, however, each of them would have power 3 etc.

Type	Name	Power	Resilience	Power Increment
6	PASS Leader	1	3	1

High First classification (85+)

This classification introduces Management Meddling cards, which change the characteristics of friendly students in the professor's own cohort!

- The management bestows its beneficence upon a specific student, who is chosen at random from the professor's cohort when the card is played.
- The management card stays with the student as long as they remain in the cohort, and modifies their behaviour in the way specified. If the student leaves, the management card leaves with them.
- If there is no student available, the card is effectively discarded as it is played, but the professor's prestige is increased by 2.

Be careful to maintain your object-oriented polymorphic program architecture.

Card Type 7 – Management – Research Funding
Research funding is allocated to a specific student, granting them
- When a student has research funding, their power is increased by 2.

Card Type 8 – Management – Mitigating Circumstances
All students want MCs. They shield you from the realities of life!
- When any student with MCs is attacked, the power of the attack is reduced by 1

Type	Name	Power Increment	Mitigation
7	Research Funding	1	
8	Mitigating Circumstances		1

Here is a further selection of Student cards you could add, which will take you up to full marks. Implement as many as you can!

Card Type 9 – Student – Easy Target
The student who draws attention to themselves and presents an easy target for other's jibes…
- The Easy Target is literally powerless.
- When students are activating, if there is an Easy Target in the enemy cohort, they MUST target them rather than choosing a random target.

Card Type 10 – Student – Serial Offender
Some types of student just won't go away! They keep on knocking on your door and sending messages…
- When a student who is a Serial Offender reduces an enemy student's resilience to 0 and has excess power, they move on to target another victim in the same turn.

        - Excess power is calculated by subtracting the target student's resilience from the attacking student's power.
        - If the excess power is greater than 0, a second target is selected at random, and attacked using the excess power.
        - The process repeats until the student has no excess power.
        - If the serial offender runs out of students to attack, they turn to the enemy professor.

Card Type 11 – Student – Graduate Student

Graduate students bring prestige to Professors… as long as they submit their thesis on time!

- Graduate students activate like any other student, but with the additional benefit of giving 2 prestige points to their Professor.

Type	Name	Power	Resilience	Boost	Power Increment
9	Easy Target	0	6		1
10	Irritating Idiot	3	5		
10	Scary Austrian	4	6		
11	Graduate Student	2	2	2	

Report Contents

For any grade:

- Include the functionality checklist (which will be available on Blackboard) as the first page of your report. (You can use it as a report template if you like) This will tell me which grade level you are working towards, and which sample output I should compere your output to.
- Copy the entire console output of your program running from the seed I supplied. Use `Consolas 10pt` font for this.

For a grade above 50%:

- Include a UML class diagram.

For a grade above 60%

- Take a screen-shot of the Visual Studio Debug Output window showing evidence that your program has no memory leaks.

Appendix

Resources

This assignment has an associated set of support files (text files for input) which can be found on Blackboard. You will also need to use the Style Guide.

A set of sample output files for the various levels of the assignment will also be provided for you to compare your own results against.

Reading the text files

In order to make reading the cards from the files straightforward, a code number is used.
At the beginning of each line there is a number which identifies the type of card stored on that line.
Use the code number in order to determine whether the data refers to an Ordinary Student - or for higher marks, a special type of student, a type of Admin duty, or a type of Management meddling.

Code	Meaning
1	Student
2	Admin Duty – Plagiarism Hearing
3	Admin Duty – Course Accreditation
4	Admin Duty – Feedback Forum
5	Student – Industrial Placement
6	Student – PASS Leader
7	Management Meddling – Research Funding
8	Management Meddling – Mitigating Circumstances
9	Student – Easy Target
10	Student – Serial Offender
11	Student - Graduate Student

The file will always be in the same format. You can assume that every Card has a name consisting of two words, followed by its power, and then other numeric properties according to its specific type.

Random number generation

Copy this function into an appropriate place in your code.
It generates a random number in the rage 0 – max (inclusive)

```
int Random(int max)
{
    return int(float(rand()) / (RAND_MAX + 1) * float(range));
}
```

Random number generators only generate a pseudo-random sequence of numbers. If the generator is seeded with the same number then an identical sequence of numbers is generated.

- The seed for the random number generator is read from the file "seed.txt".
- The use of a seed will allow me to check your work against a known number sequence.
    - I will supply examples of play using a particular seed.
- Seed the generator with the given number and the play will be exactly the same each time.

At an appropriate place in your code, before you make any calls to Random() include the following line, where seed is the number you read from the file:

```
srand(seed);
```