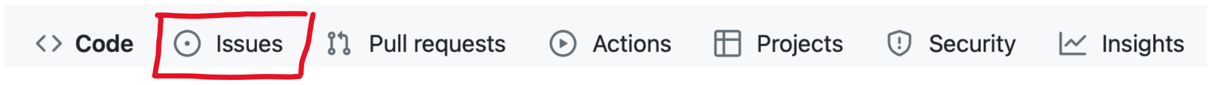
PEP C++ Tasks

Due to variation in C++ compilers it's important to note that your code will be considered working (compiling/linking/functioning) if it compiles and passes the tests on the module server (gcc C++ 11 under Ubuntu 22). To check this, you can either run your code in one of the labs using additional tests that you write, or you'll need to commit your code via Github (as per instructions on Keats). The code you commit through Github will be sent to our module server (automatically) and get tested with test files that are included as part of the code template. The feedback (e.g. which tests passed and which failed) will be generated (automatically) and sent back to your Github account. You'll be able to find the feedback under the 'Issue' tab (see below). It may take up to 30 minutes for the server to generate the feedback. During busy times (when 300 students are busy committing their code) it can also take a few hours. To ease the pressure in thelead up to the deadline, it would be helpful if students test their code locally at first, and only then commit it to Github.



The CW for the C++ section includes several tasks outlined below. You don't have to implement the tasks in any particular order or indeed in full. That is to say that you can start and stop working on any part as you see fit. You will still get feedback when you commit your code.

Throughout the implementation of this CW, please make sure that you don't commit any compiled code (.exe) to your GitHub repository. If you choose to use an IDE, ensure it doesn't commit project directories to your github. The feedback file will detail which file/s needs to be removed – please don't ignore this. There is no issue with these files on your local machine, but please don't \`commit\` or \`add\` them to your Github repository -- this isn't what git is designed for. If in doubt, raise a query on Keats forum and I will respond.

Final note. Some of the tasks are limited with regards to the libraries that students can use in order to solve the tasks (marked in red) and the time your code takes to solve them. This is not designed out of malice or to annoy. Rather, this was put in place to ensure that students are able to dissect a problem and manipulate the data / structure without the aid of libraries. Reinventing certain wheels helps us assess the level of coding mastery you exhibit. The alternative to this, is to design an end of term exam to account for 40%-50% of the mark for C++, which has quite a few shortcomings. If you require clarifications, please see me during my office hours. The time limit was put in place to ensure that the 1 server we have in the module can provide feedback to students in good time. There are 300+ students in this class and each student will submit their code multiple times. If there wasn't a time limit – feedback would be very slow. The same applies to when your code will be marked – times are indicated below.

[Part 1 – 10 Marks in total – 240 seconds /500,000KB limit]

# Solving the Countdown Numbers Game

In this part of the CW, we will solve Countdown number game using some of the basic building blocks of C++ programming, such as: vectors, classes and recursive functions.

- When solving this task your code must be in C++. Do not use any of the 'C' language functions for working with strings (the functions defined on the following page <https://en.cppreference.com/w/cpp/header/cstring>, or those on the following page in the 'Numeric string conversion' section --<https://en.cppreference.com/w/cpp/header/cstdlib>).

In addition to the above, you can only elicit the help of the following libraries: string, sstream and vector.

If you haven't seen or played Countdown conundrum (numbers) you can watch an episode on youtube in order to get the general gist of how it is being played. The number part we will be implementing in this section starts at 12 minutes and 19 seconds in the video.

The Countdown numbers game is specified as follows:

- Players are given six integer numbers and a target integer number

- The six numbers should be combined using the 4 mathematical operators: plus, minus, times and divide (+ - * and /) in a way that results in the target number or as close as possible to it.

-Each number can only be used once at most, but it is okay not to use any given number/s.

-Divisions by zero or any other number that doesn't result in an Integer **needs** be discarded (i.e. the expression building can be existed and the next expression in the search is evaluated).

An example of a countdown conundrum is the following 6 numbers (100,4,17,9,3,2) that needs to reach 37 as the target number. One possible answer is: (100 / (3 + 2)) + 17)

Evaluating numeric expressions [5 mark]

In `Countdown.h`, where it says `TODO: write code here`, implement a function `evaluateCountdown` that takes a `string` containing a mathematical expression written in Reverse Polish Notation (RPN) and returns the result of that expression as a `double`. For more details of on RPN see: https://en.wikipedia.org/wiki/Reverse_Polish_notation. In short, when using Reverse Polish Notation (RPN), operators are written *after* rather than between what they act upon. For instance:

`3.0 4.0 +` ...is the same as the usual (infix) notation `(3.0 + 4.0)`.

`3.0 4.0 + 2.0 *` ...is the same as `(3.0 + 4.0) * 2`.

Using the above example, we can write `(100 / (3 + 2)) + 17)` in RPN as: 100 3 2 + / 17 + (note the spaces between the numbers and operators).

The advantage of RPN is that there is no need to worry about brackets and order of precedence.

How to implement RPN in code? An expression (the string) can be evaluated by making a stack of numbers (using e.g. a `vector<double>`) splitting the input string by spaces into tokens, and working through the calculation one token at a time:

* If the token is a `+`, pop two numbers a and b off the stack, and push (b + a) onto the stack
* If the token is a `-`, pop two numbers a and b off the stack, and push (b - a) onto the stack
* If the token is a `*`, pop two numbers a and b off the stack, and push (b * a) onto the stack
* If the token is a `/`, pop two numbers a and b off the stack, and push (b / a) onto the stack
* Otherwise, the token is a number: push it onto the stack

After going through all the tokens, the answer should be the only number on that stack. Because you only need to be able to evaluate numeric expressions for 'countdown' problems, you can safely assume your stack won't be bigger than six elements -- the worst case is an expression with six numbers (all put on the stack) followed by five arithmetic operators, e.g. `100 4 17 9 3 2 + + + + +`.

To test your code, compile and run `TestEval.cpp`. This will evaluate four expressions, and check they give the right answer. As a reminder, to compile using g++ at the command line, with debugging turned on using (`-g`):

`g++ -std=c++11 -g -o TestEval TestEval.cpp`

then run `./TestEval`

Notes:

- `std::stod(s)` will convert the string `s` into a double, and return its value

## Solving countdown problems [5 marks]

The file `Countdown.h` contains a small class `CountdownSolution` for storing the solution to the Countdown number problem (e.g. a string containing the 'winning' RPN numeric expression and an int containing the value that that expression evaluates to (std::string solution; int value; in the code).

In `Countdown.h` implement a function `solveCountdownProblem` that takes a `vector<int>` containing 6 numbers, and a target number; and returns a `CountdownSolution` object containing the solution to the problem that gets as close as possible to the target.

To test your code, compile and run `TestCountdown.cpp`. This will evaluate some Countdown sets (6 numbers and their associated targets) and check they give the right answer.

Ensure that evaluateCountdown returns the exact same value when evaluating the stored RPN expression stored in CountdownSolution. Any difference will result in a failed test.

Notes:
-The suggested approach to solve this task is by writing code that constructs different RPN expressions (using each number once at most with the 4 available mathematical operators).

- The given code `intToString` may be helpful in the process of building RPN expressions: it takes an int, and returns a string.

Countdown Rules:

- Each number can only be used once.
- No negative numbers or Zeros will be fed into the code (Countdown.h).
- Once the code reaches the target it can stop.

[Part 2 – 5 marks in total-10 seconds/500,000KB limit]

String construction [10 marks]

This part of the assignment considers a string construction problem defined as follows:

- You are given as input a target string. Starting with an empty string, you add characters to it, until your new string is same as the target. There are two options in which you can add characters to the empty string:

1. You can **append** an arbitrary character to your new string, with cost **x**

OR

2. You can **clone** any substring of your new string (as it stands at that point), adding it to the end of your new string, with cost **y**

- For a given target you'll be given an **append** cost **x** and a **clone** cost **y**. At the end of the process, we want to know what is the *cheapest cost* of building the target string

For some simple examples:

- Target "aa" that has **append** cost 1, **clone** cost 2: the cheapest cost is 2:
  - Start with an empty string, ""
  - Append 'a' (cost 1), giving the string "a"
  - Append 'a' (cost 1), giving the string "aa"

- Target "aaaa" that has **append** cost 2, **clone** cost 3: the cheapest cost is 7:
  - Start with an empty string, ""
  - Append 'a' (cost 2), giving the string "a"
  - Append 'a' (cost 2), giving the string "aa"
  - Clone "aa" (cost 3), giving the string "aaaa"

- Target "xzxpzxzxpq", append cost 10, clone cost 11: the cheapest cost is 71:
  - Start with an empty string, ""
  - Append 'x' (cost 10): "x"
  - Append 'z' (cost 10): "xz"
  - Append 'x' (cost 10): "xzx"
  - Append 'p' (cost 10): "xzxp"
  - Append 'z' (cost 10): "xzxpz"
  - Clone "xzxp" (cost 11): "xzxpzxzxp"
  - Append 'q' (cost 10) : "xzxpzxzxpq"

In the file `StringConstruction.h` write a function `stringConstruction` that takes the **target** string, the **clone** cost, and the **append** cost (**in that order!**) and returns the cheapest way of constructing the target string. It doesn't need to return *how* to do it, just the cost.

To test your code, TestStringCons.cpp contains some test cases. To compile and run at the command line:

`g++ -std=c++11 -o TestSC TestStringCons.cpp`

`./TestSC`

Note: when your work is marked, it will be tested with target strings of a range of sizes. To get full marks, it will need to work for the largest of these target strings, which is 30,000 characters. Your code will be run on a modest desktop PC and allowed 60 seconds, although during the formative test, only 10 seconds will be given (this is to save time and due to the fact that the server does not include a test with a lengthy string).

[Part 3 – 10 marks in total- 60 seconds /500,000KB limit]

For these tasks you will be making a Linked List template class.

Please ensure you don't commit any compiled code to your GitHub repository; or if you choose to use an IDE, any large project directories created by your IDE. You can make these on your machine, but don't `commit` or `add` them to your repository -- this isn't what git is designed for.

# LinkedList basics [3 marks]

## a) Making a list node

Note that in answering this task, you should NOT add any `#include` or `using` statements. You must implement the functionality yourself, without using any data structures from the Standard Template Library (STL). The template already includes `#include <iostream>` if you want to print values to the screen for debugging purposes.

In the file `node.h` implement a template class `Node` that represents a node in a doubly linked list. It should have three `public` member variables:

- The `data` stored in that Node. The type of this should be a template argument.
- A pointer to the `next` Node in the list
- A pointer to the `previous` Node in the list

Make a constructor that takes an item of data, stores it in the node, and sets the two pointers to `nullptr`.

## b) Making an iterator for list nodes

The file `node.h` contains an incomplete `NodeIterator` class, that contains a pointer to a `Node`. It is a template class -- a `NodeIterator<T>` object contains a pointer to a `Node<T>`.

Complete the definition of the `NodeIterator` class, so that:

- We can increment the iterator, using ++, which makes it point to the next node in the list
- We can see if two iterators are the same, using the == operator. Two iterators are the same if they point to the same Node.
- We can see if two iterators are different, using the != operator

To test your code, compile and run TestNode.cpp. A Makefile has been provided, run: `make TestNode` ...at the command line. This makes two Nodes, one linked to the other, then it makes an iterator.

If you don't have make, you can always open the Makefile and copy-paste the command that will be used to compile TestNode:

`g++ -Wall -g -std=c++11 -o TestNode TestNode.cpp`

# Making a LinkedList class [7 marks]

Note that in answering this task, you should not add any `#include` or `using` statements. You must implement the functionality yourself, without using any data structures from the Standard Template Library (STL). #include "node.h" and #include <utility> have already been added.

In the file `linkedlist.h` implement a template class LinkedList. This should use the ` #include "node.h"` class you have written so far. As member variables you should have:

- A pointer to the head of the list
- A pointer to the tail of the list
- A count of how many elements are in the list

The functions in LinkedList should be:

- A constructor, that creates an empty list (head and tail are `nullptr`, zero elements in the list)

- A `push_front` function that takes an item and pushes it onto the front of the list, i.e. it becomes the head of the list. (Note this should *not* take a Node. For a LinkedList<T>, we should be able to pass a T to push_front.)

- A `front()` function, that returns a reference to the data inside the head node
- A `push_back` function that takes an item and pushes it onto the back of the list, i.e. it becomes the tail
- A `back()` function, that returns a reference to the data inside the tail node
- A `size()` function that returns the count of how many elements are in the list
- A `begin()` function that returns an iterator pointing to the head of the list
- An `end()` function that returns an iterator pointing to `nullptr` (NB it doesn't point to the tail: it points *off the end* of the list -- and the Node after the tail is `nullptr`.)
- A destructor, that `delete`s every node in the list.
- A `reverse()`function that reverses the order of the nodes in the list (NB it doesn't make new nodes, it should re-order the existing nodes.)

Recommendation: The above list of functions is in order of difficulty. It is therefore recommended you implement them in this order.

To test your LinkedList at this point, compile and run TestList.cpp. You can do this using:

`make TestList`

It may help you to comment out tests that use code you haven't written yet (as it won't compile). Alternatively, if you push your code to github, the automated tests will be run individually and should work even if you have not completed all parts.

Note: when your work is marked, deductions will be made for memory leaks. Take care to ensure that you have deleted every object you created with new, exactly once.

Once you're happy with your work so far, it's time to make the class a bit more useful.

First, extend your class to have a constructor that takes a :

[std::initializer_list<T>](http://en.cppreference.com/w/cpp/utility/initializer_list) as its argument, and makes a list containing these. You can `#include <initializer_list>` for this. This will allow lists to be created using the handy syntax:

`LinkedList<int> numbers {4,6,8,10,12};`

When we write this, the numbers in braces are put in the initializer_list and passed to the constructor.

Next, inserting and erasing elements from linked lists is a classic interview question. Add functionality as follows:

- `insert()`, taking as arguments a NodeIterator pointing to where to insert the new element in the list; and the element itself. The element should be inserted in this position, and an iterator to the new element returned. For instance, in the list: `[3,5,7]`

...if we have an iterator pointing to element `5` and call insert of element `4`. The new list would become: `[3,4,5,7]` returning an iterator pointing to 4.

Note you may edit your iterator class to provide a function to access the Node pointer from inside the iterator.

- `erase()`, taking as argument a NodeIterator pointing to an element to erase; and returning what is now in its place -- after removing element *i*, it should return a NodeIterator to what is now element *i*. For instance, if the list was: `[3,5,7]`

...and an iterator pointing to the 2nd element `5` was erased, that would update the list to be: `[3,7]` and return an iterator pointing to 7. If you are deleting the last item off the list, it should return an iterator pointing to nullptr.

Again, as above, make sure your implementation avoids memory leaks.

- Additional overloads of `begin()` and `end()` that are const methods -- i.e. can be used on a `const LinkedList`, to iterate through and provide read-only access to the elements. To go with this, you will need to make another iterator class with a different name of your choosing.

To test your LinkedList now, compile and run TestListD.cpp. You can do this by using: `make TestListD` , which is provided.

[PART 4- 20 marks in total- 60 seconds /500,000KB limit]

In this part you will be solving a puzzle.

No additional libraries are allowed to be used apart from <vector>, <iostream>, <memory> and <string>.

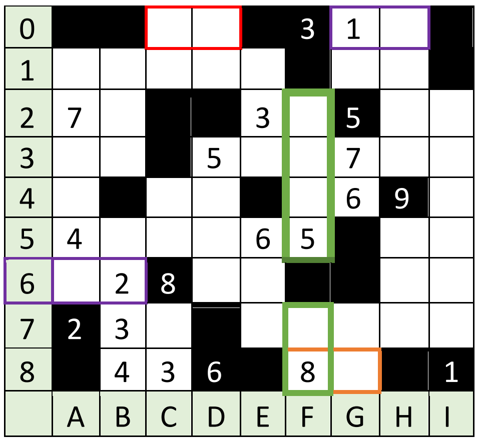
#The Puzzle

NumberSets is a numbers puzzle game, which is very similar to Sudoku. It is played on a 9x9 board that contains white and black cells. The aim of this task is to fill the empty white cells with the correct numbers so that they create sets. The black squares separate the white compartments, which can be both horizontal or vertical (see figure 1 below compartments marked in green and orange boundary). White squares in compartments must be filled with single numbers to complete a set that has no gaps, but can be in any order (e.g. 6,7,8 or 3,2,4,5).

Like in Sudoku no single number can repeat in any row or column. Black squares may contain numbers. These act as a ‘clue’ to say that the specified number cannot be an option (and therefore needs to be removed) from that row and column.

In this task you will write a program that fills the board’s compartments (white cells that are grouped horizontally/vertically) by taking into account all the rules that have been described below.

Table 1- Board 1



0	⬛	⬛			⬛	⬛ 3	1		⬛
1						⬛			⬛
2	7		⬛	⬛	3		⬛ 5		
3			⬛	5			7		
4		⬛			⬛		6	⬛ 9	
5	4				6	5	⬛		
6		2	⬛ 8			⬛	⬛		
7	⬛ 2	3		⬛					
8	⬛	4	3	⬛ 6	⬛	8		⬛	⬛ 1
	A	B	C	D	E	F	G	H	I

Table 2- Solution to Board 1

0	■	■	6	7	■	3	1	2	■
1	6	7	5	8	4	■	2	3	■
2	7	8	■	■	3	4	5	1	2
3	8	9	■	5	2	6	7	4	3
4	5	■	1	2	■	7	6	9	4
5	4	1	2	3	6	5	■	8	7
6	3	2	8	4	5	■	■	7	6
7	2	3	4	■	7	9	8	6	5
8	■	4	3	6	■	8	9	■	1
	A	B	C	D	E	F	G	H	I

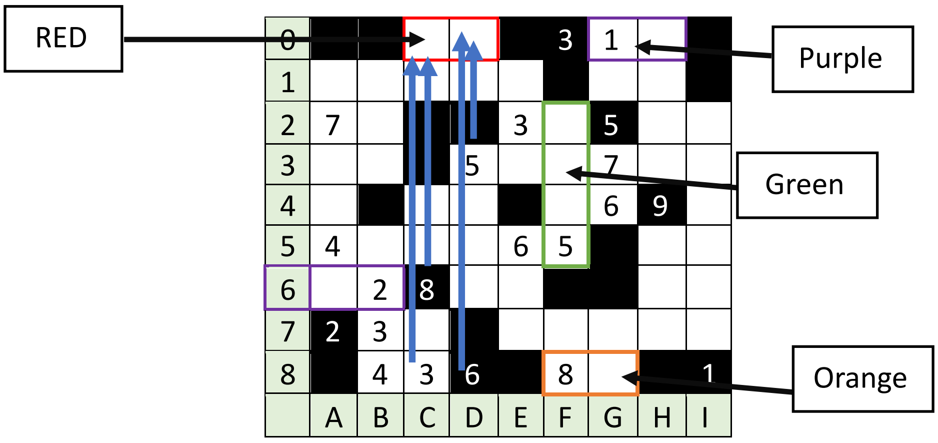
Here is a short summary of the rules:

1. Rows and columns are divided into **compartments**.
2. A **compartment** is a series of uninterrupted white cells. For example, the above column F is made up of two vertical compartments marked in green border). Similarly, row 8 is made of 2 horizontal compartments at B8, C8 and F8, G8.
3. Each **compartment** needs to be filled with numbers that **may** be unordered but **must** be uninterrupted (i.e. no gaps).
4. It's important to stress that consecutive **compartments** (horizontal or vertical) do not necessarily need to continue each other's number sequence. See above column B (marked in purple) which compartments run from 1-4 and 7-9.
5. Possible numbers are 1-9
6. By uninterrupted we mean there are no gaps in the middle (e.g., 3, 1, 2 or 5, 9, 6, 8, 7).
7. Like Sudoku, no single number can repeat in any row or column (irrespective of compartments).
8. The template code uses vectors that indexes cells from zero to 8.
9. Squares that contain numbers cannot be changed. The program should only seek to manipulate/populate empty white squares. The test files will check that populated squares contain the same numbers as in the beginning.

As mentioned above, a black square is a blocked square, which means that your solver cannot input a number into it. If a black square contains a number, it helps you by providing a clue that this number can effectively be removed from the list of possibility for any cell in that row and column.

How you might go about solving this puzzle.

As an example, the board below is showing different compartments in red, purple and green (labels provided for students who are visually impaired or colour blind). Both the red and purple squares cannot contain the number 3 as it appears in 0,F. The same applies to the number 1 in 0,G. It is also the case that the red compartment cannot contain the numbers 5, 6, 8 and 3 as they already appear in columns C and D (see arrows).



This knowledge can be used to eliminate and deduce which number need to be removed as option for these compartments (and their white squares). Looking above at the orange compartment (8F and 8G) it already contains the number 8 (in 8F). This means that the empty white square to its right (8G) can only contain a 7 or a 9 as a compartment can only contain uninterrupted numbers. As such, because column G contains the numbers 7, it can be removed as possibilities, leaving 9 as the only viable option.

SetSolver.h contains a 2dimensional vector (board), which resembles the mental structure we have for the NumberSet board. Each cell in that board is of type SetSolverSquareSet which is a vector in its own right, giving you the option to remove any numbers that a given white cell cannot contain.

0	■	■	6	7	■	3	1	2	■
1	6	7	5	8	4	■	2	3	■
2	7	8	■	■	3	4	5	1	2
3	8	9	■	5	2	6	7	4	3
4	5	■	1	2	■	7	6	9	4
5	4	1	2	3	6	5	■	8	7
6	3	2	8	4	5	■	■	7	6
7	2	3	4	■	7	9	8	6	5
8	■	4	3	6	■	8	9	■	1
	A	B	C	D	E	F	G	H	I

Table 3- solution to board 1

Representing a board in code

To represent the board, a vector of strings (vector <std::string>) will be provided by the .cpp test file (as shown below). The vector represents an empty **black** square as **Zero**, an empty **white** square as an **asterisk** and a black square with a number as a **negative** of that number. We're only using negative numbers for black squares that have numbers in order to distinguish them from a white cell with number. For example, the number 3 in row 0 (0,F) will be represented as -3 in the string. For clarity, empty white squares can only contain positive numbers 1 through to 9.

As an example, the above board will be represented in code in the following way:

```
vector<string> easyBoard =    {"00**0-31*0",
                               "*****0**0",
                               "7*003*-5**",
                              "**05**7**",
                              "*0**0*6-9*",
                              "4***650**",
                              "*2-8**00**",
                              "-23*0*****",
                               "043-608*0-1"};
```

The template includes two .h files. **SetSolver.h** and **SetSolverSquareSet.h**. The latter is a class designed to help you keep track of what number are possible for any given square. Every square that remains with only 1 single option causes a snowball effect on other squares in its row and column. Continually run those check to whittle down the possibilities until the board is solved.

Easier boards can be solved by process of elimination. More difficult boards might leave your code in a position whereby a few cells are left with the same options, and you'll need to devise a mechanism to break the deadlock. For example, a board may be left with 2 white cells both containing the options (2,3) that could go in either of those squares without breaking the rules. For that, a simple process of elimination won't work. More difficult boards simply reach the deadlock much earlier in the process and require a lengthier process in order to be solved.

You need to write a constructor that initialises the vector in **SetSolverSquareSet.h** to contain the numbers 1 – 9. After that when the populateBoard function is called, options can be removed.

In the file **SetSolver.h** a SetSolver class has been created with two member variables: a 2D vector of SetSolverSquareSet and the boardSize.

Your task is to write the following functions:

#PopulateTheBoard [1 mark]
- void **PopulateBoard(vector<string>skeletonBoard).** The function takes a vector of strings and translates/converts them to a vector of integers based on the following rules.

1. A zero remains a zero (denoting an empty black square)
2. A string of “-x” (negative integer) remains a negative number as an int (denoting a black square with a number)
3. An asterisk is converted and stored as the number 99. (e.g. it denotes an empty white square, which may hold any number of values from 1 to 9).

#ReturnCellValue [2 mark]
-int **ReturnValue (int row, int col)**, which returns the value for a particular cell. It should return -x for a negative number (denoting a black square with a number), x for a white square with a number, a zero for an empty black square and 99 for an empty white square.

#SolveBoards [17 mark]
- **void Solve()** this function will solve the board so that each empty white square has been filled with the correct value. This will be checked using the ReturnValue function described above. Below you’ll find 4 additional boards to practice/test you code with. You can use NumberSetSolveTest.cpp and NumberSetPopulateTest.cpp as a boilerplate to add these additional boards.

Once you have committed your final version of the code and after the deadline for the C++ CW has passed I will test your code and award [2 marks] for an easy board, [3 marks] for an Intermediate board, [4 marks] for a Difficult board and [8 marks] for very Difficult boards.

Attached below are additional example of boards along with their solutions. These examples were not included in the template test files and can be used for local testing.

Table 4- Board 2

6		3	■	■			■	■
4	1		3	■		8		6
		■			■			
			2				7	■
■	**9**			6			■	■
■					■			
		8	■	4		■	6	
		9		**1**	2			
■	■	5	6	■	**4**	2		■

Table 5 - Board 2 with Solutions

6	4	3	■	■	8	7	■	■
4	1	2	3	■	7	8	9	6
3	2	■	4	5	■	6	8	7
5	3	4	**2**	8	6	9	7	■
■	**9**	7	8	6	5	4	■	■
■	8	6	9	7	■	5	4	3
9	7	8	■	4	3	■	6	5
8	6	9	7	**1**	2	3	5	4
■	■	5	6	■	**4**	2	3	■

Board 3

■				7	■	■		1
5			7		■			
	3	**1**						■
4	2		■	**8**				■
				■		5		
■	4			■	■			6
■	7					■		
			9				3	■
	9	**7**	■	1				■

Board 3 Solution

■	5	8	6	7	■	■	2	1
5	8	9	7	6	■	3	1	2
2	3	**1**	8	5	6	4	7	■
4	2	3	■	**8**	7	6	5	■
3	1	4	2	■	8	5	6	7
■	4	2	3	■	■	7	8	6
■	7	6	4	3	5	■	9	8
7	6	5	**9**	2	4	1	3	■
8	9	**7**	■	1	3	2	4	■

Board 4

■	■1			9	■			■
						4		■
			7	■5	4		■	■
■		5	■	2		■		6
						6		
5		■			■9	7		■
■3	■			■4				
■					5			
■			■			5	■	■2

■	■1	7	8	9	■	2	3	■
7	5	6	9	8	3	4	2	■
6	8	9	7	■5	4	3	■	■
■	4	5	■	2	1	■	7	6
4	7	8	3	1	2	6	9	5
5	6	■	2	3	■9	7	8	■
■3	■	2	1	■4	6	8	5	7
■	2	3	4	7	5	9	6	8
■	3	4	■	6	7	5	■	■2

Board 5

■						■1		
■								
		■	■9				6	
	■				■			■5
■	■4			■			■	■
■2	6		■				■	3
					■	■		2
					3			■
	7	■						■

Board 5 Solution

■	3	7	4	5	6	■1	9	8
■	1	8	3	2	4	5	7	6
3	2	■	■9	4	5	8	6	7
4	■	2	1	3	■	7	8	■5
■	■4	3	2	■	7	6	■	■
■2	6	5	■	7	8	9	■	3
9	5	6	7	8	■	■	3	2
7	8	4	6	9	3	2	5	■
8	7	■	5	6	2	3	4	■

Board 6

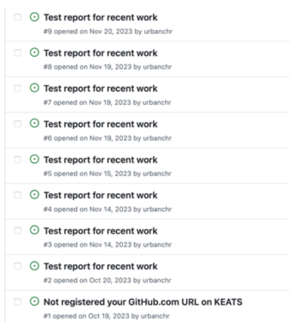
■	2			6				■
				■			6	8
		4				■		
■								
■	1		2				■	■
						1		9
		■				■		
7				■				
2	7			■			3	■

Board 6 Solution

■	2	3	1	6	7	8	9	■
5	3	2	4	■	9	7	6	8
6	5	4	3	1	2	■	8	7
■	4	5	9	3	8	2	7	6
■	1	6	2	4	5	3	■	■
8	6	7	5	2	3	1	4	9
9	8	■	7	5	6	■	1	2
7	9	8	6	■	1	4	2	3
2	7	9	8	■	4	5	3	■

Questions that were raised last year

- Can I use auto (variable type)?
  - Yes
- In the Countdown task, if the target is 78 but I'm getting 78.3 what should I do?
  - The code should discard any RPN expression as soon as it yields a fractional or an invalid (division by zero) result.
- Just to confirm, is it true that as long as I can properly receive reports in Github Issues, that means I have set up the Github repository correctly, and that means it can be marked correctly, right?
  - That's right. While a Github repo is not registered the feedback generation script will return a 'not registered you:



Test report for recent work
#9 opened on Nov 20, 2023 by urbanchr

Test report for recent work
#8 opened on Nov 19, 2023 by urbanchr

Test report for recent work
#7 opened on Nov 19, 2023 by urbanchr

Test report for recent work
#6 opened on Nov 19, 2023 by urbanchr

Test report for recent work
#5 opened on Nov 15, 2023 by urbanchr

Test report for recent work
#4 opened on Nov 14, 2023 by urbanchr

Test report for recent work
#3 opened on Nov 14, 2023 by urbanchr

Test report for recent work
#2 opened on Oct 20, 2023 by urbanchr

Not registered your GitHub.com URL on KEATS
#1 opened on Oct 19, 2023 by urbanchr

- What exactly is #PopulateTheBoard asking us to do? It asks for a vector<int>, whereas the tests ask for a vector<vector<int>>, and the code itself seems to need a vector<vector<setSolverSquareSet>>?
    - PupulateBoard takes a vector of strings, which needs to be isolated, converted to a number and populated in the board based on the rules described
- Does the set in SetSolverSquareSet need to remain as a vector?
    - No, it does not. You can use other ways to store each square's possibilities.
- At what exact time is the deadline for the coursework?
    - This is clearly published on the Keats page.