Requirements and Guidelines

In this exercise you should:

1. Implement a Simulator that runs several Algorithms on several Travels and outputs their result in a defined format.
2. Use C++ smart pointers to avoid or bring to minimum direct heap memory allocation and deallocation - using `std::make_unique` and `std::make_shared` appropriately.
3. Implement two Algorithms as .so (shared objects) so that the Algorithms can be loaded dynamically by the Simulator as explained below.
4. Allow your Simulator to run with multiple threads.

Simulator and Algorithm - Two projects

You are required to organize your submission into two folders for two separate projects and submit two separate makefiles:

1. For the Simulator project, include your Simulator as well as other helper classes required by it. The executable name shall be: `simulator`
2. For the algorithms - create two .so files. In order to allow us run your algorithms together with other groups' algorithms, the name of the .so files and the class inside shall be unique, please use the id of one of the submitters, as follows: `_098765432_a.so` (for the first algorithm, id of submitter is 098765432) and `_098765432_b.so` (for the second algorithm). Please use the same person's id for both algorithms (with the suffix _a and _b as presented above).

Error Handling

This chapter is an update on the whole issue of error handling. It is published here above all other chapters so you will not miss it.

[1]
Error handling is divided between the Simulator and the Algorithm:

- The Simulator is required to report errors to error file(s).
- The Algorithm is required to return error code when relevant and also to reject bad containers, but should not write to any error file! Note: your Algorithm is NOT allowed to write to the file system except for the crane instruction files.

[2]
Single Error file or multiple Error files?

You are allowed to work with either a single error file or multiple error files, as long as you follow these rules:

- if you go with **multiple files**, all files shall reside under a folder with the name "errors" that would sit under the "output" directory. You should decide on the name of files under this directory. All files in this directory must not be empty, that is you should NOT have empty error files.
- if you go with **a single file** its name should be "simulation.errors".
- if there are no errors - no error file and neither error directory shall be created. Having an error directory with zero files or empty error file(s) is not allowed.

[3]
Format of the Error file(s): it should be text based - the exact format is *yours* and is free.

[4]
There are several types of errors:

(a) **Fatal errors** that prevent the run of the Simulation: no Algorithm-Travel would run and results file should NOT be created, however errors shall be listed in error file(s). Fatal errors are those errors that cannot allow the run of the simulation, as would be detailed below.

(b) **Travel errors:** bad travel input that cannot be fixed - the travel shall be skipped. The simulation and the algorithm would have the same rules for rejecting an entire travel altogether. If the Simulator revealed the bad input it will document the issues in error file and would skip this travel (i.e. would not pass the travel data to the algorithm). However, in case for some reason the Algorithm did get bad travel input that cannot be fixed, it will return proper error code as detailed below and would skip the travel (if the Simulation is insisting on calling the Algorithm for this bad travel, the Algorithm can create empty instruction files and keep reporting the same error code).

(c) **Bad algorithm behavior:** once an algorithm performs an illegal move, the illegal move shall be documented by the simulation in error file and the result of this Travel-Algorithm pair would be a failure (denoted by the value -1, as detailed below). Your Simulator can decide to stop the

run of this Travel-Algorithm pair immediately at the moment when the bad behavior is revealed, or to "ignore" the bad behavior in some way (of your choice) in order to keep collecting errors that would be documented to error file.

(d) **Bad external input that can be fixed and "ignored":** in such cases, as would be detailed below, the simulation and the algorithm would have the same rules for how to "fix" the error and keep on. **The simulation would always pass to the Algorithm the same input file that it got without fixing it** (the fix would be done by each side for itself, separately and independently, you may use the same code if you wish). The Simulation would report the error to file. The Algorithm would return error code as detailed below.

(e) **Error code returned by the Algorithm:** when the Algorithm returns error code, the Simulator would document this in error file. The simulation may decide based on the error code returned whether it is relevant to continue running the Travel-Algorithm pair or to skip it.

[5]

Algorithm Error Codes

The return value of the Algorithm methods should be 0 in case of success and a bitwise combination of the following error codes otherwise. Note that *one* or *several occurrences* of the same error type is reported the same.

2^0 - ship plan: a position has an equal number of floors, or more, than the number of floors provided in the first line (ignored)
2^1 - ship plan: a given position exceeds the X/Y ship limits (ignored)
2^2 - ship plan: bad line format after first line or duplicate x,y appearance with same data (ignored)
2^3 - ship plan: travel error - bad first line or file cannot be read altogether (cannot run this travel)
2^4 - ship plan: travel error - duplicate x,y appearance with different data (cannot run this travel)
2^5 - travel route: a port appears twice or more consecutively (ignored)
2^6 - travel route: bad port symbol format (ignored)
2^7 - travel route: travel error - empty file or file cannot be read altogether (cannot run this travel)
2^8 - travel route: travel error - file with only a single valid port (cannot run this travel)
2^9 - reserved
2^10 - containers at port: duplicate ID on port (ID rejected)
2^11 - containers at port: ID already on ship (ID rejected)
2^12 - containers at port: bad line format, missing or bad weight (ID rejected)
2^13 - containers at port: bad line format, missing or bad port dest (ID rejected)
2^14 - containers at port: bad line format, ID cannot be read (ignored)
2^15 - containers at port: illegal ID check ISO 6346 (ID rejected)
2^16 - containers at port: file cannot be read altogether (assuming no cargo to be loaded at this port)
2^17 - containers at port: last port has waiting containers (ignored)
2^18 - containers at port: total containers amount exceeds ship capacity (rejecting far containers)

The Simulator

The Simulator is able to run several algorithms (not only 2) on several Travels.
The algorithms are loaded dynamically as .so files, as explained below.
When implementing the Simulator, you may decide whether to run each algorithm separately, one after the other, or traverse "in parallel", over different "copies" of Travels, step by step by all algorithms - this shouldn't affect the algorithms, as each sees its own "copy" of the Travel, that is: the move on one algorithm is isolated and should have no effect on any other algorithm.

Command line parameters

Running the Simulator shall support the following command line options:

```
simulator [-travel_path <path>] [-algorithm_path <algorithm path>]
      [-output <output path>]
```

Details:

1. The order of arguments may change
2. The `-output` sets the location to generate all output files: crane instructions, simulation.results and errors.
3. The `-algorithm_path` sets the location to look for the algorithm .so files.
4. Both `-algorithm_path` and `-output` arguments are optional.
    - In case `-algorithm_path` is missing, look for the algorithm .so files in the run directory / current working directory.
    - In case `-output` is missing:
        (a) generate the simulation.results and errors under the run directory / current working directory.
        (b) Create a crane instructions folder per <algorithm, travel> pair, under the run directory / current working directory, according to crane instruction file location rules. Those folders shall host the crane_instrucitons files associated with the "pair run".
5. The travel folder will contain one or more sub folders, **each for every Travel**.
6. A missing `-travel_path` argument should result in a fatal error, no algorithm is to be run in this case and a proper error should be generated.
7. In each Travel sub folder the following must be found:
    1. One file with ***.ship_plan*** suffix for ship plan. For details see: Ship plan below.
    2. One file with ***.route*** suffix for the route. For details see: Route file below.
    3. Multiple files for containers awaiting along the route - a file per port visit, unless no containers should be loaded at the stop, such as in the final stop. For more details and format see: Containers awaiting at port below.

The program should run and report into the simulation output file the number of crane actions required in each Travel by each algorithm (the .so files).

The Algorithm

To allow common implementation you need to use common .h files for abstract classes and other definitions as listed below.

AbstractAlgorithm - abstract base class

To allow common implementation you need to define an abstract class for AbstractAlgorithm:

```
// AbstractAlgorithm.h

#pragma once

#include <string>

// forward declaration
class WeightBalanceCalculator;

class AbstractAlgorithm {
public:
	virtual ~AbstractAlgorithm(){}

	enum class Action { LOAD = 'L', UNLOAD = 'U', MOVE = 'M', REJECT = 'R'};

	// methods below return int for returning 0 for success
	// and any other number as error code
	virtual int readShipPlan(const std::string& full_path_and_file_name) = 0;
	virtual int readShipRoute(const std::string& full_path_and_file_name) = 0;
	virtual int setWeightBalanceCalculator(WeightBalanceCalculator& calculator) = 0;
	virtual int getInstructionsForCargo(
				const std::string& input_full_path_and_file_name,
				const std::string& output_full_path_and_file_name) = 0;
};
```

Your algorithms must inherit from this exact base class, enabling any simulator to run your algorithms, even if it wasn't created by you.

Note:

- Above header is in use both by the Simulator and Algorithm projects. You can either put it in both as a copy, or point to it, just make sure your **two** makefiles work fine.

The following Algorithm function will be called by the Simulator **per each visit to a port**:

```
int getInstructionsForCargo(const std::string& input_full_path_and_file_name,
                            const std::string&
                           output_full_path_and_file_name)
```

Details about function arguments:

- `Input_full_path_and_file_name`
  The files to be associated with this parameter are the cargo_data files that are part of the Travel data, see the Containers Awaiting at Port section below.

- Crane instructions are to be written into a new file at `Output_full_path_and_file_name`
  The Simulator shall create for each algorithm-travel pair that it runs, a folder named **<algorithm_name>_<travel_name>_crane_instructions** for all crane_instructions files generated by this algorithm-travel pair. The folder should be created under the folder given by the -output argument or under the run directory / current working directory, if the -output option is omitted.
  Inside this folder the algorithm shall generate an output file per port visit, the name of those files shall be:
  <port symbol>_<visit number>.crane_instructions
  were <visit number> is 1 for the first visit to the port, 2 for the second visit and so on.
  A file must be created even if the instruction list is empty. See Cargo output file - Instructions for Crane for exact format.

AlgorithmRegistration

To allow simple algorithms discovery we need to define a common registration process so all loaded algorithms would register themselves automatically.

Note: we will explain in one of the following class meetings (with Amir or with Adam) the way the automatic registration should work. It may seem complicated but it is not so much.

You should use the following header file for AlgorithmRegistration that you need to include in your Algorithm classes:

```
// AlgorithmRegistration.h

#pragma once

#include <functional>
#include <memory>
#include "AbstractAlgorithm.h"

class AlgorithmRegistration {
public:
AlgorithmRegistration(std::function<std::unique_ptr<AbstractAlgorithm>()>);
};

#define REGISTER_ALGORITHM(class_name) \
AlgorithmRegistration register_me_##class_name \
      ([]{return std::make_unique<class_name>();} );
```

You are free to implement the .cpp file for the AlgorithmRegistration as you wish but **you are not allowed to change its header file**! We may implement it differently, however the Algorithm classes are unaware of the actual implementation of the AlgorithmRegistration class. Note that the AlgorithmRegistration class is NOT the registrar itself and you have the freedom to implement the registrar itself as you wish.

Each of your algorithm .cpp files will have the following line in the global scope:

```
REGISTER_ALGORITHM (<class_name>)
```
For example:
```
REGISTER_ALGORITHM (_098765432_a)
```

Note: the algorithm registration header is in use by both the Simulator project and the Algorithm project. You can either put it in both as a copy, or point to it, just make sure your makefiles work fine. However, **`AlgorithmRegistration.cpp` should be part of the Simulator project only** and the implementation is on you.

Travel

ShipPlan File

The mandatory input format to describe the ship plan:
A text file with the following structure:

- First line:
  <num floors>, <num containers in dimension X>, <num containers in dimension Y>
- Zero or more lines with the following data, order of lines is not important:
  <X position>, <Y position>, <actual number of floors in this position>

When a position has a lower number of floors than the number of floors provided in the <num floors> in the first line, it means that this position is only available in the top <actual number of floors in this position> floors (out of <num floors> floors).

Errors:

- In case a position has an equal number of floors, or more, than the number of floors provided in the first line, this input line shall be ignored with a proper error.
- If a given position exceeds the X/Y ship limits (as defined in the first line) it should be ignored and a proper error shall be issued.
- If the file format cannot be read altogether, the file shall be ignored with a proper error. **The simulation should not run any algorithm on the travel associated with this file.**
- See possible other errors in the Error handling section.

Route File

A text file with a line per *port symbol* (5 english letters of seaport code, no space inside, allow both lower or upper case). The same port may appear more than once, but not in two consecutive lines (in case a port does appear twice or more consecutively, this is a non fatal error, you should ignore the excess appearance and report the error from the Simulation / return proper error code from the Algorithm).
Errors:

- If the file format cannot be read altogether, the file shall be ignored and a proper error shall be issued. **Without a Route the simulation should not run any algorithm on the travel associated with this file.**
- See possible other errors in the Error handling section.

Containers Awaiting at Port Files

Each port shall have its own file (or files, if a route has this port more than once).

Name file:

<port symbol>_<number>.cargo_data

Above format is for all ports, even if the port appears only once in the route.
<number> shall be 1 for the first visit to the port (even if there is only a single visit to this port), 2 for the second visit and so on (**Note**: this is a change from *Input-Output file format suggestion for Ex1*).

File format:
The file itself would have a line per container with the following data:
<container ID - as described by ISO 6346>, <weight in kg as integer>, <dest port symbol>

Details:

- "Missing" cargo_data files:
  A missing file for a corresponding <number> visit to a port PPPPP (by the order it appears on the route file) will be treated as an empty Containers awaiting port file. The simulator shall pass to the Algorithm a full path pointing to an empty file (it can be always the same empty file).
  Example: TRIST appears 5 times in the route file, and only TRIST_1.cargo_data and TRIST_3.cargo_data files are available indicates that in the second, fourth and fifth visit to Istanbul seaport (TRIST) no **new** cargo is to be loaded onto ship (but there may be of course cargo to be unloaded). The algorithm shall still get a path to an input file, even if empty.
- Even if no cargo_data file is available, a crane_instructions should still be created for the same port visit, even if empty (i.e. in case no cargo is to be loaded or unloaded).
- The visit number in the input file has no trailing zeros, i.e. TRIST_1.cargo for the 1st visit and TRIST_102.cargo in case there are 102 visits (or more) to TRIST during this Travel.

For details about Algorithm output file see: Cargo output files - Crane instructions per port visit.

Errors:

- Additional cargo_data files that were not in use (not in route or too many files for port) would be ignored but a proper error shall be issued by the Simulator.
- If a cargo_data file does not correspond to a stop in the Route File the file should be ignored and a proper error shall be issued by the Simulator.
- In case the stop is valid but the cargo_data file cannot be read altogether, the file shall be ignored and a proper error shall be issued by the Simulator. The stop associated with the ignored file shall be considered as a stop with no containers to be loaded both by the Simulator and the Algorithm - each one separately (the simulator would still pass the same bad file to the algorithm).
- See possible other errors in the Error handling section.

WeightBalanceCalculator

The calculator shall be initiated by the simulation with the ship plan file:

```
readShipPlan(const std::string& full_path_and_file_name)
```

It shall preserve the state of all loaded cargo (thus, a calculator instance is required per <algorithm, travel> pair, each algorithm shall have its own “new” instance of weight calculator per each travel) -- NOTE that there is no need to actually read the ship plan, but you still have to present this method.
It shall also have the following method:

```
BalanceStatus tryOperation(char loadUnload, int kg, int X, int Y);
```

The method gets:

- char: L or U for Load or Unload
- kg weight
- X and Y position for the load / unload - the calculator knows the floor to load to (lowest possible) or to unload from (upper available container)

The method returns an enum value that can be one of:

- `APPROVED` - operation approved (and registered as done by the balance calculator!)
- `X_IMBALANCED` - operation declined - need a better balance on the X coordinate
- `Y_IMBALANCED` - operation declined - need a better balance on the Y coordinate
- `X_Y_IMBALANCED` - operation declined - need a better balance on both X and Y

Note:

- The balance calculator doesn’t have a “move” operation, as for balance purposes “move” is actually an “unload” followed by “load”. First the “unload” has to be approved, then the “load”.

WeightBalanceCalculator shall have a naive stub implementation approving (with `APPROVED` result) every `tryOperation` call.

WeightBalanceCalculator header file:

```
// WeightBalanceCalculator.h

#pragma once

class WeightBalanceCalculator {
public:

	enum BalanceStatus {
		APPROVED, X_IMBALANCED, Y_IMBALANCED, X_Y_IMBALANCED
	};

	int readShipPlan(const std::string& full_path_and_file_name);

	BalanceStatus tryOperation(char loadUnload, int kg, int x, int y);
};
```

Output files and screen output

Algorithm output files format

Cargo output files - Crane instructions per port visit

The **output** file would have the following format - with a line per operation:
<L / U / M / R>, <container id>, <floor index>, <X index>, <Y index> [, <floor index>, <X index>, <Y index>]

- L / U / M / R - a single char which stands for Load / Unload / Move / Reject
- The data in square brackets would be added only for “Move” operations, as the destination of the container - in the same line. Move operation is more efficient than unload+load, as it is counted as a single operation.
- The order of lines **is important as it dictates the order of operations**.
- All containers awaiting at this port must be at the end of all operations either on the ship or in a Reject line.

Note that the location and name of the **.crane_instructions** files are set by the Simulator, as already described above, at: Algorithm output files location as set by the Simulator.

Multithreading

The simulation would have an additional command line parameter:

`-num_threads <num>`
This parameter, as others, can appear at any location on the command line.
In case not provided - the Simulator would run with a single thread (i.e. would not spawn any new threads, as if **`-num_threads == 1`**).
The **`-num_threads`** parameter dictates the number of parallel threads for running the simulation. Note that the exact number of threads may be lower, in case there is no way to properly utilize the required number of threads. The exact threading model is your decision, try to make the most out of the provided threads.

Simulator output files

simulation.results

The simulator shall create a results file with the name "simulation.results" at the root -output folder, with a CSV table format (no need for tabs, just CSV -- see further rules below):

RESULTS,	<travel name 1>,	<travel name 2>,	…,	Sum,	Num Errors
<Algo name 1>,	total operations,	total operations,	....,	sum,	num errors
<Algo name 2>,	…,	…,	…,	sum,	num errors
...					

The numbers in the cells of the table are the total number of crane instructions per each travel-algorithm pair. The number -1 shall represent a simulation that couldn't be completed due to any kind of an error (including not handling a container that should be handled or any other violations or bad instruction by the algorithm as listed here).
Note that *Travel input violations* are to be revealed by the simulation and the column for the Travel should not appear in the results output file at all in a case of *Travel input violations*.
The Sum column shall be the sum of all operations - ignoring the Travels for which the algorithm had an error, for which it has a -1 cell.
The table shall be sorted, with the best algorithm first. The sorting would be based first on the number of errors from 0 (top of table) and down (more errors appear below) -- then for algorithms with the same number of errors, by the Sum column (lower is better). This means that an algorithm with no errors would appear above an algorithm with one error (-1 Travel cell) and two algorithms with the same number of errors (-1 Travel cells) would be sorted according to their Sum, from a lower Sum (better, thus being above) to higher.

Note:

- Do not assume that there are exactly 4 Algorithms / 3 Travels, there can be any number of Travel / Algorithms.
- We may check this output automatically with awk and grep, your exact formatting may vary (number of spaces, using tabs or not, width of each field) as long as it looks similar to the general format below.

Example:

```
RESULTS,                                                    short_travel,long_travel,
                                                            complex_travel,      Sum,
                                                            Num Errors
_098765432_a,                                               100,  109,    500,    709,
                                                            0
_098765432_b,                                               100,  109,    502,    711,
                                                            0
_331332334_a,                                               210,  -1,     1420,   1630,
                                                            1
_331332334_b,                                               -1,   -1,     1524,   1524,
                                                            2
```

More on Error Handling

You have the freedom to decide regarding your error handling mechanism and exact error messages formats but you must follow these guidelines:

- The Algorithm itself generates nothing except the crane instructions file. Bad cargo id should be just rejected. The algorithm should return error code in case of an error and the simulation should decide whether to issue a proper error, ignore it or do something else. The Algorithm shall not print anything as an error. The printage of errors to file and/or to screen would be handled by the Simulation only and not by the Algorithm.
- If at least one error was found: create simulation.errors file / errors folder to accumulate all general errors and errors associated with the run of the algorithms. **If no error was found - do not create the simulation.errors file / errors folder**.
- No error should ever crash the program.
- You are encouraged to accumulate as many errors as possible and report those before ending your Simulator run. A bonus will be given to those who do it properly and with a good choice of file format and error messages. Note that you are not required to accumulate all violations made by simulated algorithms but you MUST report at least one of those per run of a <algorithm, travel> pair run that resulted with a violation.

Errors output file(s): simulation.errors / errors folder

- Generated iff an error was detected. (it's an intentional *iff* - **no errors? no file!**)
- The file / folder will be created in the location specified by the -output option or in the run directory if the argument was not given.
- If you use a single file, begin with listing all general errors, such as errors about missing -travel_path argument or erroneous Travel data and only after listing all general errors list errors associated with Algorithm runs (<algorithm, travel> pair) such as algorithms that violate any of the rules listed here.

Printing errors to screen

- You may also print errors to the screen for your own convenience using std::cout or std::cerr, the decision is up to you, but there is no such requirement. Note that the updated requirement for output files in case of missing *-output* command line option is to create the output files under the run directory / current working directory.

Additional Requirements

Documented Code

Document your classes with a descent header: author (full name, don't get shy) and proper description. Document complicated code. Do not document what the next line of code does (the code should be self-explanatory as much as possible) - document what you are trying to achieve in the next block, what is the approach, why you do that, etc.
Do not document trivial code. If you believe we can understand the code, don't document it.

Code Quality

Avoid unnecessary copy-paste. Keep methods short and as simple as possible.
Names are crucial: for classes, variables and methods. Be thoughtful of name choices.

General rules for all file formats

- For all files, a line starting with the # symbol should be considered as a comment line and be ignored.
- For all files, leading white spaces and any additional white spaces before or after comma shall be allowed and ignored.
- All files are CSV format / line per data entry
- Bad input shall never crash the program, an error message should be reported.