SBU CSE 390/590 - Special Topics in Computer Science, Spring 2023:
“C++ Programming for Real Life Challenges”

Note:
This document is in DRAFT mode till Mar-5th. While in draft mode, it may have significant changes without notice and without highlighting them. Even though it is still a draft, you should start working on the exercise.
After the draft period there could still be changes but then they should be highlighted in the document.
You are mostly welcomed to add questions to this document, as comments, during the draft period and/or after it.
Please assign your comments and questions to amir.kirsh@stonybrook.edu so I will get an email when you add them. Then please follow my answer and if it resolves your comment or question please close it.

Submission deadline: Mar-19th, 11:50 pm.
Submission instructions - TBD

Assignment 2 - Requirements and Guidelines

In this assignment, the API and file format would be strictly defined, so your code can work with algorithms implemented by other teams and with house files created by other teams.

House Input File - TEXT file

The format of the house file shall follow the following strict instructions:
Line 1: house name / description - internal for the file, can be ignored by the simulation
Line 2: MaxSteps for the simulation on this house, as: **MaxSteps = <NUM>**
Line 3: MaxBattery (in steps) for the simulation on this house, as: **MaxBattery = <NUM>**
Line 4: Number of Rows in house, as: **Rows = <NUM>**
Line 5: Number of Cols in house, as: **Cols = <NUM>**
Lines 6 and on: the house itself, as described below

Notes:
1. All values in lines 2-5 may or may have spaces around the = sign
2. If the file is invalid you can reject it and print the reason to screen

The actual size of the house is determined by the values in lines 4 and 5. If there are missing lines, they should be filled in with empty lines, if there are missing columns they should be filled in with spaces. After filling all the missing characters, the simulation shall assume that the house is surrounded by walls, on the boundaries beyond the provided rows and cols.
In case the file contains additional rows and/or cols beyond the required number as set in lines 4 and 5, the program shall ignore the redundant lines / cols.

The program should get an input file that represents a house, with walls, corridors, dirt levels and a single docking station.
The following characters shall be used:

Empty Space (“corridors”): space as well as any character which does not have other mapping below
Wall: W
Dirt:
Numbers from 0 to 9 should represent dirt level. Though 0 is valid and shall be supported, it is in fact the same as space.
Docking Station: D
There shall be one and only one docking station otherwise the file is invalid.

Example of valid houses:

```
1-Small house
MaxSteps = 100
MaxBattery=20
Rows = 4
Cols =10
WWWW
W D
W 123456789
```

The house “as should be modeled” (note that the additional rows/cols for the surrounding walls, in yellow, can be considered as walls without actually adding them in memory!!!):

```
WWWWWWWWWWWW
WWWWW      W
WW D       W
WW 12345678W
W          W
WWWWWWWWWWWW
```

```
2-Another Small house
MaxSteps=100
MaxBattery = 20
Rows= 4
Cols=12
WWWWWWWWWWWW 0
W D        W 1
W 123456789W 2
WWWWWWWWWWWW 3
012345678901
```

The house “as should be modeled” (note that the additional rows/cols for the surrounding walls, in yellow, can be considered as walls without actually adding them in memory!!!):

```
WWWWWWWWWWWWWW
WWWWWWWWWWWWWW
WW D        WW
WW 123456789WW
WWWWWWWWWWWWWW
WWWWWWWWWWWWWW
```

API

Given Classes:

You should use the following classes, as provided by the skeleton project, without changing them at all (below you will see the main API of each class, the classes will be given in your skeleton project):

[1]
```
enum class Direction { North, East, South, West };
enum class Step { North, East, South, West, Stay, Finish };
```

[2]
```
class WallsSensor {
public:
      virtual ~WallsSensor() {}
      virtual bool isWall(Direction d) const = 0;
};
```

[3]
```
class DirtSensor {
public:
      virtual ~DirtSensor() {}
      virtual int dirtLevel() const = 0;
};
```

[4]
```
class BatteryMeter {
public:
      virtual ~BatteryMeter() {}
      virtual std::size_t getBatteryState() const = 0;
};
```

[5]
```
class AbstractAlgorithm {
public:
      virtual ~AbstractAlgorithm() {}
      virtual void setMaxSteps(std::size_t maxSteps) = 0;
      virtual void setWallsSensor(const WallsSensor&) = 0;
      virtual void setDirtSensor(const DirtSensor&) = 0;
      virtual void setBatteryMeter(const BatteryMeter&) = 0;
      virtual Step nextStep() = 0;
};
```

Your tasks:

You need to implement the relevant concrete classes for the above abstract classes and to implement a Simulation class that manages the simulation.

Main

Your main should look as the following (note: it doesn't have to be exactly the same, **but the API used below shall be the same**):

```
// getting command line arguments for the house file
int main(int argc, char** argv) {
      MySimulator simulator;
      // TODO: get houseFilePath from command line
      simulator.readHouseFile(houseFilePath);
      MyAlgorithm algo;
      simulator.setAlgorithm(algo);
      simulator.run();
}
```

Note that inside the call to

```
      simulator.setAlgorithm(algo);
```

The following should happen:

```
      algo.setMaxSteps(maxSteps);
      algo.setWallsSensor(wallsSensor);
      algo.setDirtSensor(dirtSensor);
      algo.setBatteryMeter(batteryMeter);
```

Algorithm

In this assignment your algorithm should be smart (read about search algorithms such as BFS or DFS that may help you in mapping the house).
The algorithm goal is to clean the house with a minimal amount of steps and return Finish when back to the docking station, after all accessible locations are clean. Note that an inaccessible location is a location that is separated from the docking station by walls, or a location that is beyond the battery reach (if we try to reach it, we would not be able to reach back to the docking station).

The algorithm should communicate its next requested step as one of: North, East, South, West, Stay, Finish. It can then assume that the requested step was performed. Note that the algorithm should not write anything into the output file, it should communicate back its move decisions based on the dictated API.

Note: your algorithm in this assignment MUST be deterministic, that is no randomness is allowed, to make sure that running your algorithm again on the same house will have the exact same results. (It is not for saying that randomness in algorithms is a bad thing, actually randomness is occasionally used in algorithms to get efficiency, but you should not, as we want to be able to always reproduce the exact same results when running a given algorithm on a given house).

The algorithm should strive to return "Finished" when on dock and the remaining amount of steps (remaining from the given MaxSteps), would not allow cleaning any additional dirt and getting back to the docking on time. Stopping at MaxSteps while not on dock is bad!

Output File - TEXT file format
NumSteps = <NUMBER>
DirtLeft = <NUMBER>
Status = <FINISHED/WORKING/DEAD>
Steps:
<list of characters in one line, no spaces, from: NESWsF – the small s is for STAY, NESW are for North, East, South, West. F is for Finished - only if the algorithm actually returned "Finished">

Note: number of characters in the last line of the file should be equal to the NumSteps value.

No *new* and *delete* in your code
Your code shall not have *new* and *delete*.
You can use std containers.
You can use smart pointers, make_unique, make_shared.
Note:

1. Do not use shared_ptr where unique_ptr can do the job.
2. Do not use allocations managed by smart pointers if stack variables can do the work.

Error Handling
Exact same instructions as in assignment 1.

Running the Program

```
myrobot <house_input_file>
```

Additional Part for Graduates (CSE 590) - OPTIONAL for Undergraduates (CSE 390)
In this assignment there is no additional requirement for Graduates, HOWEVER we expect the algorithm submitted by Graduates to be well thought. Both Grads and UG can add into the readme file a description of the algorithm and special considerations taken into account.

Bonuses
Both Undergraduates and Graduates may be entitled for a bonus for interesting implementation.
The following may be entitled for a bonus:

- adding logging, configuration file or other additions that are not in the requirements.
- having automatic testing, based on GTest for example.
- adding visual simulation (do not make it mandatory to run the program with the visual simulation, you may base the visual simulation on the input and output files as an external utility, it can be written in C++ or in another language).

IMPORTANT NOTE:
1. In order to get a bonus for any addition, you MUST add to your submission, inside the main directory, a *bonus.txt* file that will include a listed description of the additions for which you request for a bonus.
2. If you already asked for a bonus in assignment 1 for a certain addition you shall not ask for the same bonus again. If you added something, focus in your bonus.txt file on the addition.