High Level Software structure:

1. Since the battlefield can be of any size, first I ask the user to input the size of the row and column (RxC) of the battlefield. (Row and Column size can be different)
2. The flattened array representing the battlefield units can be provided in two ways. One approach involves specifying the path to a JSON file (map.json) in the Riskylab JSON format. The code then extracts the relevant data fields from the file, with values such as -1 for ground terrain, 4 for elevated terrain, 6.4 for the starting position, and 0.6 for the target position.
3. Alternatively, the user can manually input the flattened array in the terminal. In this case, the array should only use -1 for ground terrain and 4 for elevated terrain. The user has the option to manually specify the starting and target points in the grid, or the other option is that the code can randomly generate these points. I just provided a small constraint where, in the manually provided grid the starting and target units should contain -1 in the grid.
4. The shortestPath() function is employed to compute the shortest path once the grid details, starting index, and target index are acquired. I've also commented in the code for better clarity and handled most of the exceptions.
5. How to run:

```
g++ -std=c++11 main.cpp
./a.out
```

Algorithm used:

1. Initially, my approach aimed to determine the shortest distance to the target, contemplating the use of DFS or BFS. However, I know both methods presented drawbacks. DFS may not guarantee the fastest route, while BFS, although assured to find the shortest distance, demanded traversing all adjacent units layer by layer, consuming excessive memory by storing irrelevant nodes.
2. In light of these challenges, I considered employing a priority queue integrated with a heuristic to select the most pertinent next unit.
3. The cost calculation involved adding the previous route cost to a heuristic. I deliberated between using Euclidean distance and Manhattan distance. Given the absence of diagonal movement, I concluded that Manhattan distance served as the optimal heuristic to determine the minimum distance between the current state and the target state.
4. The implementation involved adding details to a heap and extracting the unit with the lowest total cost to reach the target.
5. This approach efficiently avoided storing irrelevant units in the fringe, optimizing both search time and memory usage.
6. A visited hashset was implemented to store indexes of visited units, preventing redundant computations. The priority queue ensured that a revisited node wouldn't have a lower cost, as units were only added to the visited set upon reaching them.
7. In the worst case the time complexity of this algorithm with heuristics behaves as a normal BFS O(RxC) but with the provided battlefield problem and the heuristics that I have defined it will be much better than that. The space complexity also includes saving all the visited nodes for each unit and in this algorithm that can be cut down by only storing the most pertinent units.

Note: I have worked on various projects like this in the AI domain and I'm very much interested in working in competitive projects, to provide a most optimized and clean code. There is a similar project that I worked on a few months back but it is in python (Shortest path for robot)
https://github.com/Kowshiks/ProbabilisticRoadMap-In-Simulation