Faculty of Computing and Informatics (FCI) Multimedia University
Cyberjaya

CCP6124 Object Oriented Programming and Data Structures

Trimester 2410

Assignment

M2

Robot Wars Simulator

Design Documentation

NAME: Venggadanaathan A/L K.Salvam

STUDENT ID: 1231303562

STUDENT ACCOUNT: 1231303562@student.mmu.edu.my

TUTORIAL SECTION: T14L

LECTURE SECTION: TC4L

LECTURER NAME: Ts. Goh Chien Le (clgoh@mmu.edu.my)

GROUP NUMBER: Group 39

INTRODUCTION

This document outlines the design of a robot battlefield simulation program. The program simulates a warfare scenario where various types of robots engage in a strategic battle on a grid-based battlefield. Each robot type has unique capabilities, and they perform actions such as moving, looking around, and firing at opponents based on their predefined strategies. The simulation runs for a specified number of turns, and the robot with the highest score at the end of the simulation is declared the winner.

The simulator is built using standard C++ without the use of standard library containers such as vectors, queues, and linked lists. Instead, custom data structures are implemented to meet the assignment requirements. The simulation demonstrates several Object-Oriented Programming (OOP) concepts including inheritance, polymorphism, encapsulation, operator overloading, and file handling.

Simulation Rules

1. **Battlefield**: The battlefield is represented as a 2-dimensional m x n matrix. Robots and battlefield boundaries are depicted using appropriate characters and symbols.
2. **Turn-Based Simulation**: The simulation is turn-based. In each turn, each robot performs its actions sequentially. A robot may perform a look action (if capable), then move (if capable), and finally fire (if capable).
3. **Position Privacy**: The positions of robots are private. Robots can only reveal their surroundings using the look action.
4. **Movement**: Robots can move to one of their 8 neighboring locations: up, down, left, right, up-left, up-right, down-left, and down-right.
5. **Look Action**: The look action reveals a nine-square area centered on the robot's position. This helps robots decide their next move or target.
6. **Fire Action**: Robots can fire at specified locations based on their fire patterns. The fire action destroys any robot in the targeted location.
7. **Robot Types**: Each robot has a type and a name. Robots can be upgraded based on their performance in the battlefield. The initial robot types include:
    - **RoboCop**: Can look, move, and fire. Upgrades to TerminatorRoboCop.
    - **Terminator**: Can look and move. Upgrades to TerminatorRoboCop.
    - **TerminatorRoboCop**: Can look, move, and fire. Upgrades to UltimateRobot.
    - **BlueThunder**: Can only fire in a predefined sequence. Upgrades to MadBot.
    - **MadBot**: Fires at random neighboring cells. Upgrades to RoboTank.
    - **RoboTank**: Fires at random locations. Upgrades to UltimateRobot.
    - **UltimateRobot**: Can look, move, and fire at multiple locations.
8. **Simulation Initialization**: The initial conditions of the simulation are specified in a text file, which includes the battlefield dimensions, number of turns, number of robots, and details of each robot (type, name, initial position).
9. **Robot Lives and Respawn**: Each robot has three lives. When destroyed, a robot can re-enter the battlefield up to three times. Only one robot can re-enter per turn.
10. **Additional Features**: The simulation may include additional features and capabilities to enhance the experience.

Implementation Requirements

1. **Battlefield Display**: In each turn, the battlefield and the actions of each robot are displayed on the screen and saved to a text file.
2. **Custom Data Structures**: Standard C++ library containers are not used. Custom data structures are implemented to manage the battlefield and robots.
3. **OOP Concepts**: The solution employs OOP concepts such as inheritance, polymorphism, operator overloading, and other C++ features.
4. **Action Tracking**: The simulator tracks the sequence of actions for each robot using an appropriate data structure.
5. **Queue Management**: Queues are used to manage robots that have been destroyed and are waiting to re-enter the battlefield.

OOP CONCEPTS

These are the OOP concepts used in the simulator:

1. Inheritance

- **Base Class:**
  - **Robot Class:** The base class from which all specific robot types inherit.

```
class Robot {
   public:
       // Common attributes for all robots
       int x, y;
       std::string name;
       // Pure virtual function to be implemented by derived
   classes
       virtual void
   performAction(CustomDynamicArray<CustomDynamicArray<std::string
   >>& battlefield, int rows, int cols) = 0;
       virtual std::string getType() const = 0;
       virtual ~Robot() = default;
   };
```

- **Derived Classes:**

```
class RoboCop : public SeeingRobot, public MovingRobot, public
ShootingRobot {
public:
    RoboCop(int x, int y, std::string name, HealthManager&
healthManager)
        : Robot(x, y, name, healthManager), SeeingRobot(x, y, name,
healthManager), MovingRobot(x, y, name, healthManager),
ShootingRobot(x, y, name, healthManager) {}
    // Implementations of abstract methods
};

class BlueThunder : public Robot {
public:
    BlueThunder(int x, int y, std::string name, HealthManager&
healthManager)
        : Robot(x, y, name, healthManager) {}
    // Implementations of abstract methods
};
```

2. Polymorphism

- **Abstract Methods:**
  - The Robot class defines pure virtual functions that are overridden by the derived classes.

```
class Robot {public:

    virtual void
performAction(CustomDynamicArray<CustomDynamicArray<std::string
>>& battlefield, int rows, int cols) = 0;
    virtual std::string getType() const = 0;
};
```

- **Method Overriding:**

```
void
RoboCop::performAction(CustomDynamicArray<CustomDynamicArray<std::string
>>& battlefield, int rows, int cols) override {
       // RoboCop specific action implementation
   }
```

- **Pointer to Base Class:**
  - Using an array of pointers to the base class allows managing different types of robots polymorphically.

```
CustomDynamicArray<Robot*> robots;
```

3. Encapsulation

- **Private Members:**
  - Attributes in the Robot class are private, with public methods providing controlled access.

```
class Robot {
private:
    int x, y;
    std::string name;
public:
    int getX() const { return x; }
    int getY() const { return y; }
    std::string getName() const { return name; }
};
```

4. Constructor and Destructor

- **Constructors:**
  - Initialize object attributes. Each derived class initializes its base class.

```
RoboCop::RoboCop(int x, int y, std::string name, HealthManager&
healthManager)
    : Robot(x, y, name, healthManager) {
    // Initialization
}
```

- **Destructors:**
  - Clean up resources.

```
Robot::~Robot() {
    // Cleanup code if necessary
}
```

5. File Handling

- **Logging:**
  - Using `ofstream` to log actions to a file.

```
std::ofstream logFile("simulation_log.txt");
```

```
if (logFile.is_open()) {
    logFile << "Simulation started.\n";
}
```

6. Copy Constructors and Dynamic Arrays

- **Copy Constructor:**
  - For custom dynamic arrays.

```
CustomDynamicArray(const CustomDynamicArray& other) {
    size = other.size;
    capacity = other.capacity;
    data = new T[capacity];
    for (size_t i = 0; i < size; ++i) {
        data[i] = other.data[i];
    }
}
```

- **Dynamic Arrays:**
  - Managing a dynamic list of elements with custom resizing.

```
template <typename T>
class CustomDynamicArray {
private:
    T* data;
    size_t capacity;
    size_t size;
    void resize(size_t newCapacity) {
        T* newData = new T[newCapacity];
        for (size_t i = 0; i < size; ++i) {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
        capacity = newCapacity;
    }
public:
    CustomDynamicArray(size_t initialCapacity = 10) : data(new
T[initialCapacity]), capacity(initialCapacity), size(0) {}
    ~CustomDynamicArray() {
        delete[] data;
    }
    void push_back(const T& value) {
        if (size == capacity) {
            resize(capacity * 2);
        }
        data[size++] = value;
    }
};
```

Class Diagram

A class diagram is a visual representation of the structure of a software system. It shows the system's classes, their attributes, methods, and the relationships between the classes. For the robot battlefield simulation program, the class diagram will include the following key components:

1. Robot Base Class

- **Attributes**:
  - `int x, y`: Coordinates of the robot on the battlefield.
  - `int kills`: Number of kills the robot has made.
  - `int lives`: Number of lives the robot has left.
  - `int respawns`: Number of times the robot can respawn.
  - `std::string name`: Name of the robot.
  - `std::string targetName`: Name of the target robot.
  - `bool upgraded`: Indicates if the robot has been upgraded.
  - `HealthManager& healthManager`: Reference to the HealthManager for managing health and scores.
- **Methods**:
  - `virtual void performAction(CustomDynamicArray<CustomDynamicArray<std::string >>& battlefield, int rows, int cols) = 0`: Pure virtual function to perform the robot's action.
  - `virtual std::string getType() const = 0`: Pure virtual function to get the type of the robot.
  - `void incrementKills()`: Increments the number of kills and checks for upgrades.
  - `void decrementLives()`: Decrements the number of lives and handles respawn logic.
  - `virtual void upgrade()`: Virtual function to upgrade the robot.

2. Derived Classes from Robot

Each derived class inherits from the `Robot` class and may also inherit from other specialized abstract classes based on their capabilities.

- **RoboCop**
  - Inherits: `Robot`, `SeeingRobot`, `MovingRobot`, `ShootingRobot`
  - Capabilities: Look, move, fire
  - Upgrades to `TerminatorRoboCop`
- **Terminator**
  - Inherits: `Robot`, `SeeingRobot`, `MovingRobot`
  - Capabilities: Look, move
  - Upgrades to `TerminatorRoboCop`

- **TerminatorRoboCop**
  - Inherits: `Robot`, `SeeingRobot`, `MovingRobot`, `ShootingRobot`
  - Capabilities: Look, move, fire
  - Upgrades to `UltimateRobot`
- **BlueThunder**
  - Inherits: `Robot`
  - Capabilities: Fire in a predefined sequence
  - Upgrades to `MadBot`
- **MadBot**
  - Inherits: `Robot`
  - Capabilities: Fire at random neighboring cells
  - Upgrades to `RoboTank`
- **RoboTank**
  - Inherits: `Robot`, `ShootingRobot`
  - Capabilities: Fire at random locations
  - Upgrades to `UltimateRobot`
- **UltimateRobot**
  - Inherits: `Robot`, `SeeingRobot`, `MovingRobot`, `ShootingRobot`
  - Capabilities: Look, move, fire at multiple locations

3. Specialized Abstract Classes

- **SeeingRobot**: For robots that have the capability to look.
  - `virtual void look(CustomDynamicArray<CustomDynamicArray<std::string>>& battlefield, int rows, int cols) = 0`
- **MovingRobot**: For robots that have the capability to move.
  - `virtual void move(CustomDynamicArray<CustomDynamicArray<std::string>>& battlefield, int rows, int cols) = 0`
- **ShootingRobot**: For robots that have the capability to fire.
  - `virtual void fire(CustomDynamicArray<CustomDynamicArray<std::string>>& battlefield, int rows, int cols) = 0`
- **SteppingRobot**: For robots that can step on and kill other robots.
  - `virtual void step(CustomDynamicArray<CustomDynamicArray<std::string>>& battlefield, int rows, int cols) = 0`

4. HealthManager Class

- **Attributes**:
  - `std::unordered_map<std::string, int> health`: Maps robot names to their health.
  - `std::unordered_map<std::string, int> scores`: Maps robot names to their scores.

- **Methods**:
  - `void registerRobot(const std::string& name)`: Registers a new robot.
  - `void decreaseHealth(const std::string& name)`: Decreases the health of a robot.
  - `void increaseScore(const std::string& name, int points)`: Increases the score of a robot.
  - `int getHealth(const std::string& name) const`: Gets the health of a robot.
  - `int getScore(const std::string& name) const`: Gets the score of a robot.
  - `void printHealth() const`: Prints the health of all robots.
  - `void printScores() const`: Prints the scores of all robots.
  - `void logHealth(std::ofstream& logFile) const`: Logs the health of all robots to a file.
  - `void logScores(std::ofstream& logFile) const`: Logs the scores of all robots to a file.
  - `std::string getWinner() const`: Gets the name of the winning robot.

5. CustomDynamicArray Class

- **Attributes**:
  - `T* data`: Pointer to the dynamic array data.
  - `size_t capacity`: Capacity of the dynamic array.
  - `size_t size`: Current size of the dynamic array.
- **Methods**:
  - `void resize(size_t newCapacity)`: Resizes the dynamic array.
  - `void push_back(const T& value)`: Adds a value to the end of the array.
  - `void remove_at(size_t index)`: Removes a value at a specific index.
  - `T& operator[](size_t index)`: Overloads the subscript operator for accessing elements.
  - `const T& operator[](size_t index) const`: Overloads the subscript operator for accessing elements (const version).
  - `size_t getSize() const`: Gets the current size of the array.
  - `void setSize(size_t newSize)`: Sets the size of the array.

6. Utility Functions

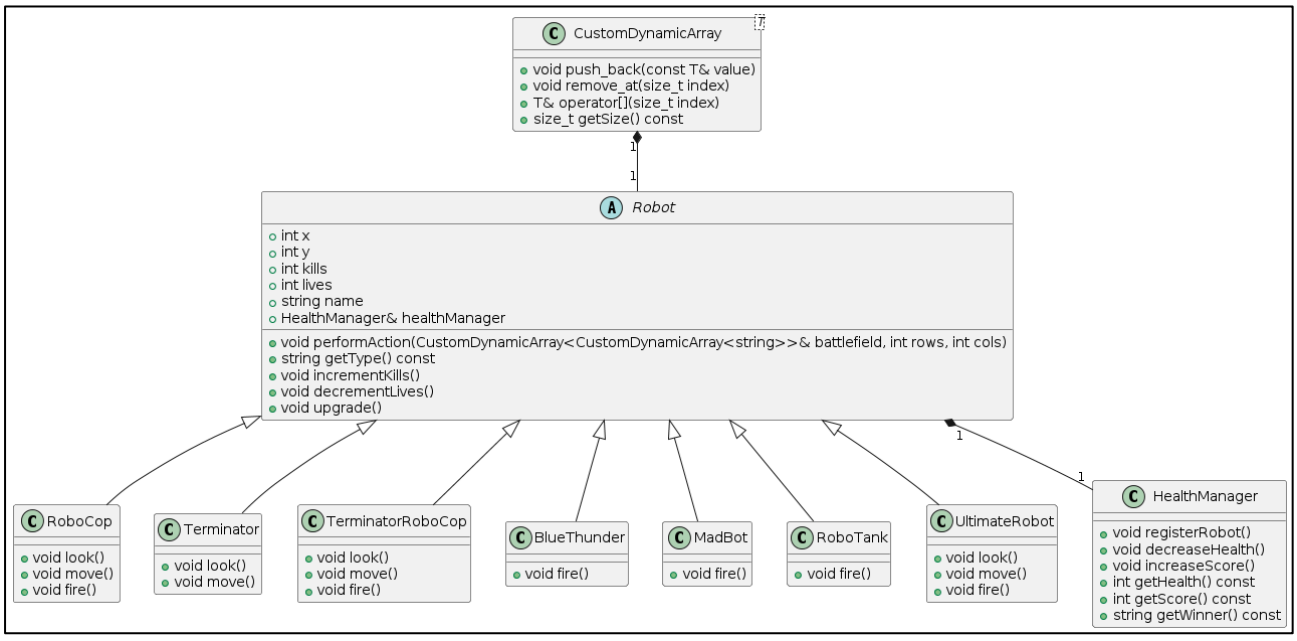
- `void printBattlefield(const CustomDynamicArray<CustomDynamicArray<std::string>>& battlefield)`: Prints the battlefield to the console.
- `void logBattlefield(std::ofstream& logFile, const CustomDynamicArray<CustomDynamicArray<std::string>>& battlefield)`: Logs the battlefield to a file.

Class Diagram Components

1. **Robot Class**: Serves as the base class for all robots, providing common attributes and methods.
2. **Derived Robot Classes**: Specific robot types with unique behaviors and capabilities, inheriting from `Robot` and specialized classes.
3. **Specialized Abstract Classes**: Define specific capabilities like looking, moving, and shooting.
4. **HealthManager**: Manages the health and scores of robots.
5. **CustomDynamicArray**: Implements dynamic arrays for managing collections of robots and battlefield cells.
6. **Utility Functions**: Provide functions for printing and logging the battlefield.

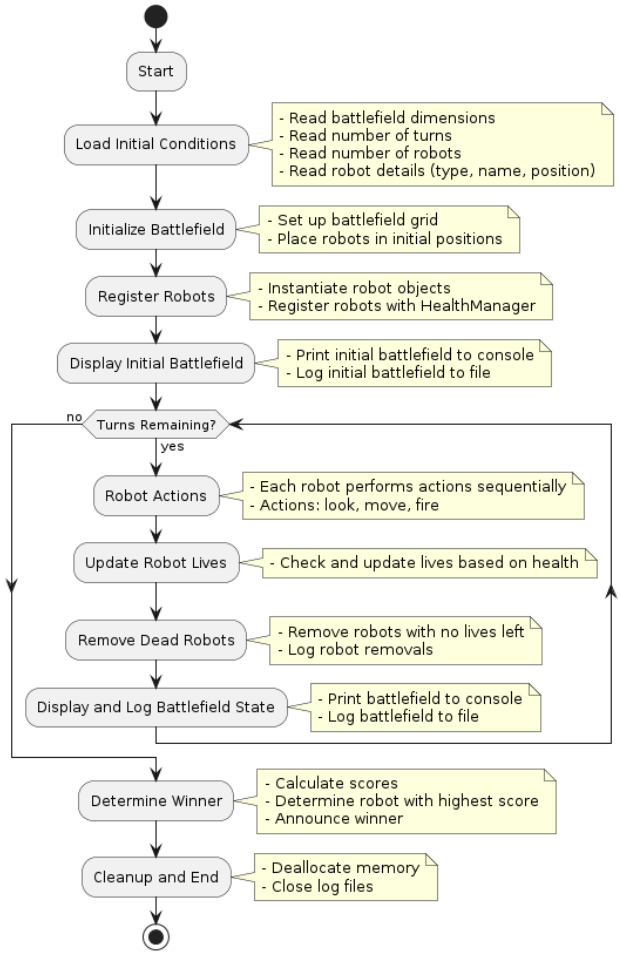
Why These Classes Are Included

- **Robot and Derived Classes**: Essential for defining the behavior and capabilities of different robot types. They use inheritance and polymorphism to allow diverse behaviors while sharing common functionality.
- **Specialized Abstract Classes**: Enable specific capabilities to be added to robots in a modular way, supporting multiple inheritance.
- **HealthManager**: Manages critical game mechanics related to robot health and scoring.
- **CustomDynamicArray**: Provides a dynamic array implementation necessary for managing collections in the absence of standard library containers.
- **Utility Functions**: Facilitate the display and logging of the battlefield, essential for visualizing the simulation's progress.



Flowchart

Basic Flowchart of the simulator:



1. **Start**:
   - The simulation begins.
2. **Load Initial Conditions**:
   - **Purpose**: To load the initial setup for the simulation.
   - **Details**: The program reads the battlefield dimensions (e.g., 40x50), the number of turns the simulation will run, the number of robots, and the initial details of each robot from a specified file. This includes the type of robot (e.g., RoboCop, Terminator), its name, and its starting position on the battlefield (x, y coordinates).
3. **Initialize Battlefield**:
   - **Purpose**: To set up the battlefield based on the loaded initial conditions.
   - **Details**: The battlefield is represented as a 2D grid. The program creates this grid and places each robot at its initial position as specified in the initial conditions.
4. **Register Robots**:
   - **Purpose**: To create robot objects and register them with the health management system.
   - **Details**: The program instantiates robot objects of various types (e.g., RoboCop, BlueThunder). Each robot is registered with the `HealthManager`, which tracks their health and scores throughout the simulation.
5. **Display Initial Battlefield**:
   - **Purpose**: To provide a visual representation of the initial state of the battlefield.
   - **Details**: The initial battlefield is printed to the console and logged to a file for record-keeping. This includes the positions of all robots and the empty spaces.
6. **Turns Loop**:
   a. **Purpose**: To run the simulation for the specified number of turns.
   b. **Details**: The loop continues as long as there are remaining turns in the simulation.
   c. **Robot Actions**:
      i. **Purpose**: To perform the actions of each robot sequentially.
      ii. **Details**: Each robot performs a series of actions based on its capabilities. These actions include looking around to detect other robots, moving to a new position, and firing at enemies. The actions are performed in a specific order, and each robot acts one by one.
   d. **Update Robot Lives**:
      i. **Purpose**: To update the health status of each robot.
      ii. **Details**: The program checks the health of each robot. If a robot's health drops to zero, its lives are decreased. If a robot runs out of lives, it is marked for removal.
   e. **Remove Dead Robots**:
      i. **Purpose**: To remove robots that have no remaining lives from the battlefield.
      ii. **Details**: Robots with zero lives are removed from the battlefield grid. This process is logged to keep track of which robots were removed and when.
   f. **Display and Log Battlefield State**:
      i. **Purpose**: To update the visual representation of the battlefield after each turn.
      ii. **Details**: The updated state of the battlefield is printed to the console and logged to a file. This includes any changes in robot positions, the removal of dead robots, and the actions performed during the turn.
7. **Determine Winner**:
   g. **Purpose**: To identify the robot with the highest score at the end of the simulation.
   h. **Details**: After all turns have been completed, the program calculates the scores of all robots. The robot with the highest score is declared the winner. This information is printed to the console and logged to the file.
8. **Cleanup and End**:
   i. **Purpose**: To properly close the simulation and free up resources.

HEADER FILES

1. **AbstractRobots.h**:
   - **Purpose**: Defines the base abstract class `Robot` and derived abstract classes (`MovingRobot`, `ShootingRobot`, `SeeingRobot`, `SteppingRobot`).
   - **Classes**:
     - `Robot`: Base class for all robots, with common attributes (position, name, health) and pure virtual functions for actions.
     - `MovingRobot`: Inherits from `Robot`, adds pure virtual function `move`.
     - `ShootingRobot`: Inherits from `Robot`, adds pure virtual function `fire`.
     - `SeeingRobot`: Inherits from `Robot`, adds pure virtual function `look`.
     - `SteppingRobot`: Inherits from `Robot`, adds pure virtual function `step`.
2. **RoboCop.h**:
   - **Purpose**: Defines the `RoboCop` class, a type of robot that can look, move, and fire.
   - **Classes**:
     - `RoboCop`: Inherits from `SeeingRobot`, `MovingRobot`, and `ShootingRobot`, implements their pure virtual functions.
3. **BlueThunder.h**:
   - **Purpose**: Defines the `BlueThunder` class, a type of robot that fires at neighboring cells in a sequence.
   - **Classes**:
     - `BlueThunder`: Inherits from `Robot`, implements the `fire` function and its specific behavior.
4. **Madbot.h**:
   - **Purpose**: Defines the `Madbot` class, a type of robot that fires randomly at neighboring cells.
   - **Classes**:
     - `Madbot`: Inherits from `Robot`, implements the `fire` function and its specific behavior.
5. **RoboTank.h**:
   - **Purpose**: Defines the `RoboTank` class, a type of robot that fires at random locations on the battlefield.
   - **Classes**:
     - `RoboTank`: Inherits from `ShootingRobot`, implements the `fire` function and its specific behavior.
6. **Terminator.h**:
   - **Purpose**: Defines the `Terminator` class, a type of robot that moves and steps on other robots.
   - **Classes**:
     - `Terminator`: Inherits from `SeeingRobot`, `MovingRobot`, and `SteppingRobot`, implements their pure virtual functions.
7. **TerminatorRoboCop.h**:
   - **Purpose**: Defines the `TerminatorRoboCop` class, combining capabilities of both `RoboCop` and `Terminator`.
   - **Classes**:
     - `TerminatorRoboCop`: Inherits from `SeeingRobot`, `MovingRobot`, `ShootingRobot`, and `SteppingRobot`, implements their pure virtual functions.
8. **UltimateRobot.h**:
   - **Purpose**: Defines the `UltimateRobot` class, the most advanced robot with combined capabilities.
   - **Classes**:

        - `UltimateRobot`: Inherits from `SeeingRobot`, `MovingRobot`, and `ShootingRobot`, implements their pure virtual functions.
9. **CustomDynamicArray.h**:
    - **Purpose**: Provides a custom implementation of a dynamic array.
    - **Classes**:
        - `CustomDynamicArray<T>`: Template class implementing dynamic array functionality such as resizing, element access, and management.
10. **HealthManager.h**:
    - **Purpose**: Manages the health and scores of the robots.
    - **Classes**:
        - `HealthManager`: Tracks health and scores of each robot, provides methods to modify and access these attributes.
11. **Utility.h**:
    - **Purpose**: Provides utility functions for the simulation.
    - **Functions**:
        - `printBattlefield`: Prints the current state of the battlefield to the console.
        - `logBattlefield`: Logs the current state of the battlefield to a file.

Main File

main.cpp:

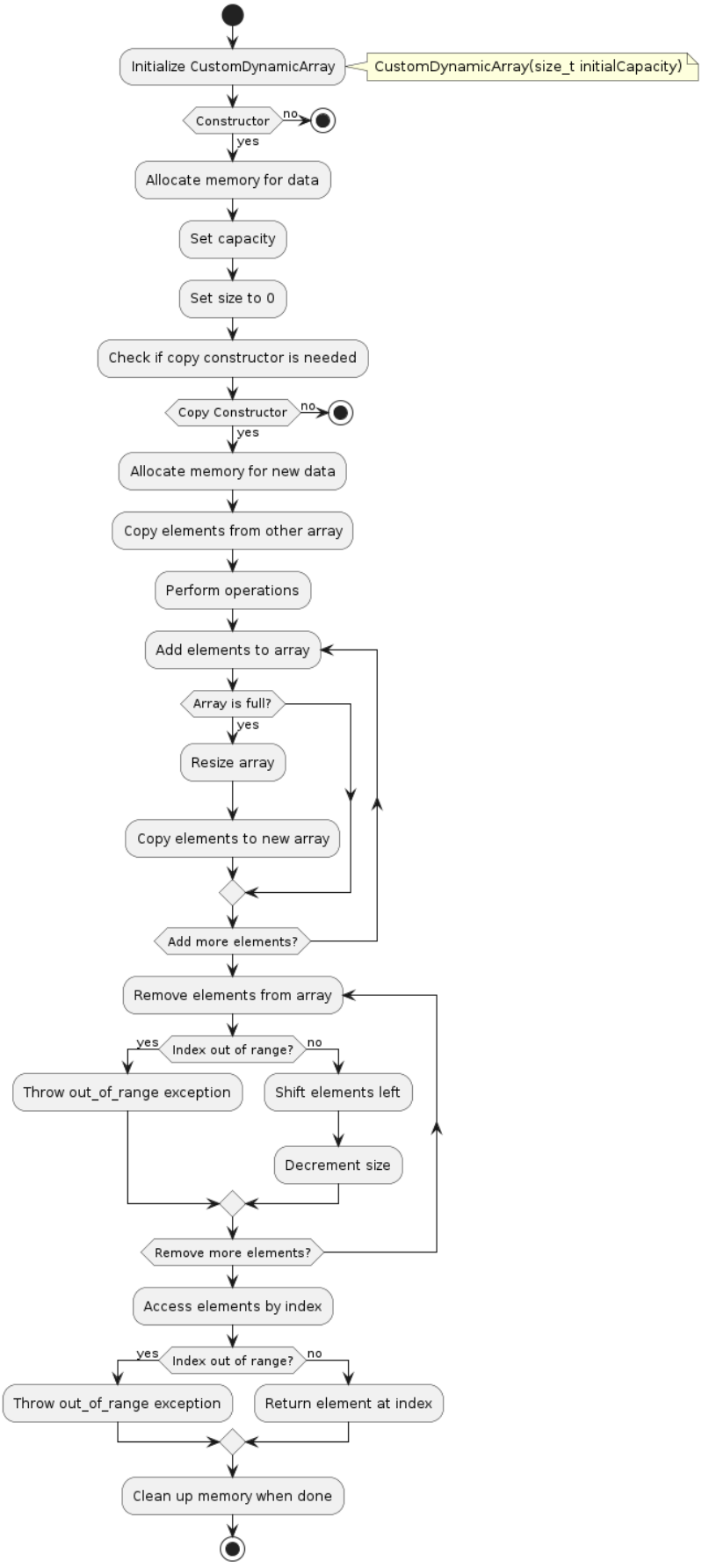
- **Purpose**: The main file that orchestrates the simulation.
- **Functions**:
    - `loadInitialConditions`: Loads initial conditions from a file, sets up the battlefield and robots.
    - `validateCoordinates`: Ensures coordinates are within the battlefield boundaries.
    - `updateRobotLives`: Updates the health status of robots, decreases lives if health is zero.
    - `removeDeadRobots`: Removes robots with no lives left from the battlefield.
    - `performRobotAction`: Executes the actions of each robot (look, move, fire).
- **Main Logic**:
    - Initializes the battlefield and robots based on the input file.
    - Runs the simulation for the specified number of turns.
    - Each turn, each robot performs its actions sequentially.
    - Updates and displays the battlefield, health status, and scores after each turn.
    - Determines and announces the winner at the end of the simulation.

PSEUDOCODE FOR THE CustomDynamicArray.h CLASS

```
Class CustomDynamicArray:
  Private:
    data: Pointer to array of type T
    capacity: Integer, capacity of the array
    size: Integer, current size of the array

    Method resize(newCapacity: Integer):
      Create newData array of size newCapacity
      For i from 0 to size - 1:
        newData[i] = data[i]
      Delete old data array
      data = newData
      capacity = newCapacity

  Public:
    Constructor(initialCapacity: Integer = 10):
      data = new array of size initialCapacity
      capacity = initialCapacity
      size = 0

    Copy Constructor(other: CustomDynamicArray):
      data = new array of size other.capacity
      capacity = other.capacity
      size = other.size
      For i from 0 to size - 1:
        data[i] = other.data[i]

    Destructor:
      Delete data array

    Method push_back(value: T):
      If size == capacity:
        Call resize(capacity * 2)
      data[size] = value
```

```
Method remove_at(index: Integer):
      If index >= size:
        Throw out_of_range exception
      For i from index to size - 2:
        data[i] = data[i + 1]
      size -= 1

    Method operator[](index: Integer) -> T&:
      If index >= size:
        Throw out_of_range exception
      Return data[index]

    Method operator[](index: Integer) const -> const
T&:
      If index >= size:
        Throw out_of_range exception
      Return data[index]

    Method getSize() -> Integer:
      Return size

    Method setSize(newSize: Integer):
      If newSize > capacity:
        Call resize(newSize)
      size = newSize
```

FLOWCHART OF CUSTOMDYNAMICARRAY.H FILE



EXPLANATION

This header file defines a template class `CustomDynamicArray` that implements a dynamic array, similar to the `std::vector` in the C++ Standard Library. This custom dynamic array is used to handle collections of objects (like robots and battlefield cells) in the simulation without relying on standard library containers. Below is an explanation of each part of the file and how it integrates into the simulation:

1. **Private Members**:
    - `T* data`: Pointer to the dynamically allocated array.
    - `size_t capacity`: Maximum number of elements the array can currently hold.
    - `size_t size`: Current number of elements in the array.
    - `void resize(size_t newCapacity)`: Private method to resize the array when it's full.
2. **Public Members**:
    - **Constructors**:
        - `CustomDynamicArray(size_t initialCapacity = 10)`: Initializes the array with a given initial capacity.
        - `CustomDynamicArray(const CustomDynamicArray& other)`: Copy constructor to initialize the array with another array's data.
    - **Destructor**:
        - `~CustomDynamicArray()`: Cleans up the allocated memory when the array is destroyed.
    - **Methods**:
        - `void push_back(const T& value)`: Adds a new element to the end of the array, resizing if necessary.
        - `void remove_at(size_t index)`: Removes an element at a specified index, shifting the subsequent elements.
        - `T& operator[](size_t index)`: Non-const version of the index operator to access elements.
        - `const T& operator[](size_t index) const`: Const version of the index operator to access elements.
        - `size_t getSize() const`: Returns the current size of the array.
        - `void setSize(size_t newSize)`: Sets a new size for the array, resizing if necessary.

UTILITY.H

The `utility.h` file contains functions for printing and logging the battlefield. These utility functions help visualize and record the state of the battlefield at different points in the simulation.

Flowchart Explanation

The flowchart will cover the main functionalities:

1. **Print Battlefield**
2. **Log Battlefield**

Print Battlefield:

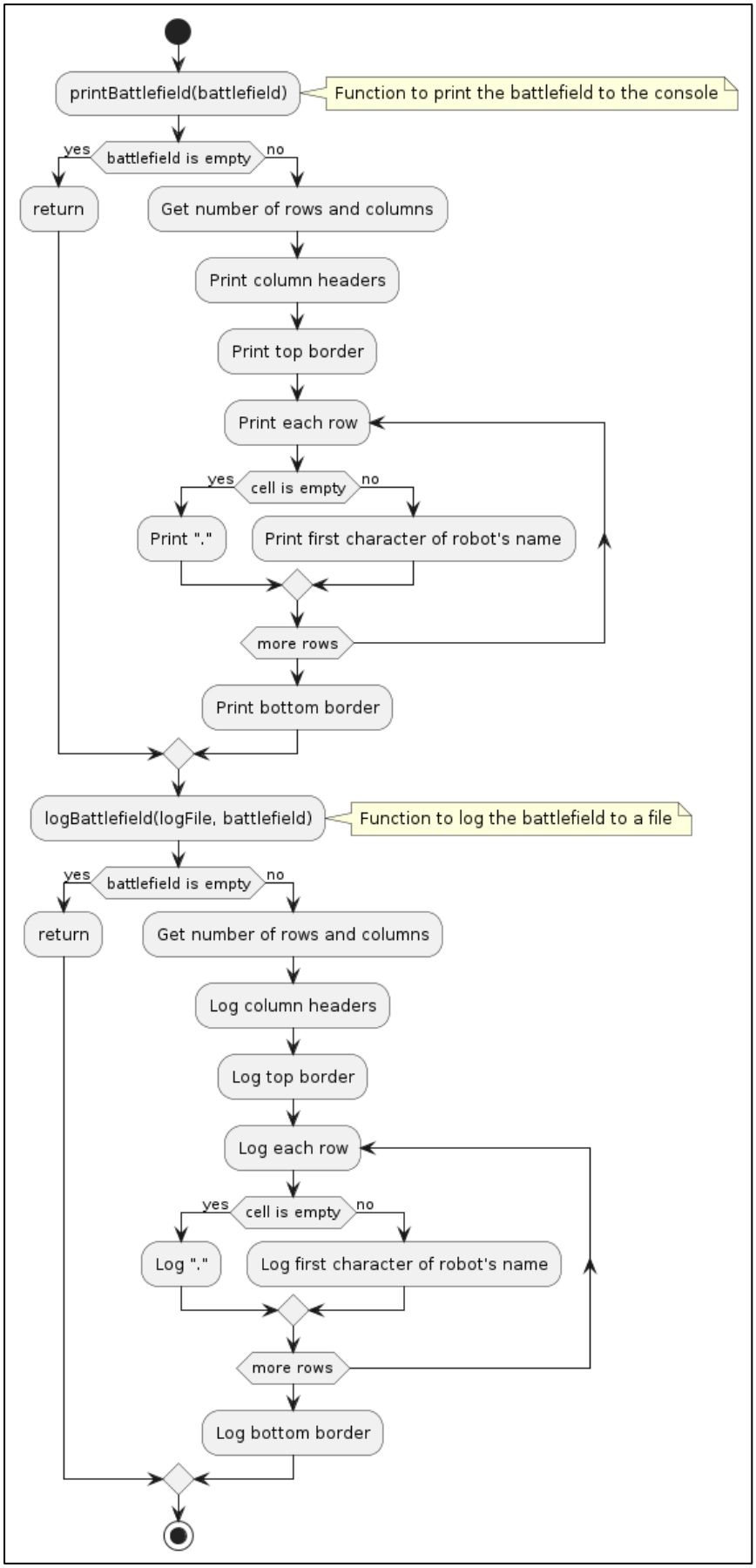
1. The function starts by checking if the battlefield is empty. If it is, the function returns immediately.
2. If the battlefield is not empty, it gets the number of rows and columns.
3. The function then prints the column headers and the top border of the battlefield.
4. It iterates through each row of the battlefield:
    1. For each cell in the row, if the cell is empty, it prints a dot (".").
    2. If the cell is not empty, it prints the first character of the robot's name.
5. After printing all the rows, it prints the bottom border of the battlefield.

Log Battlefield:

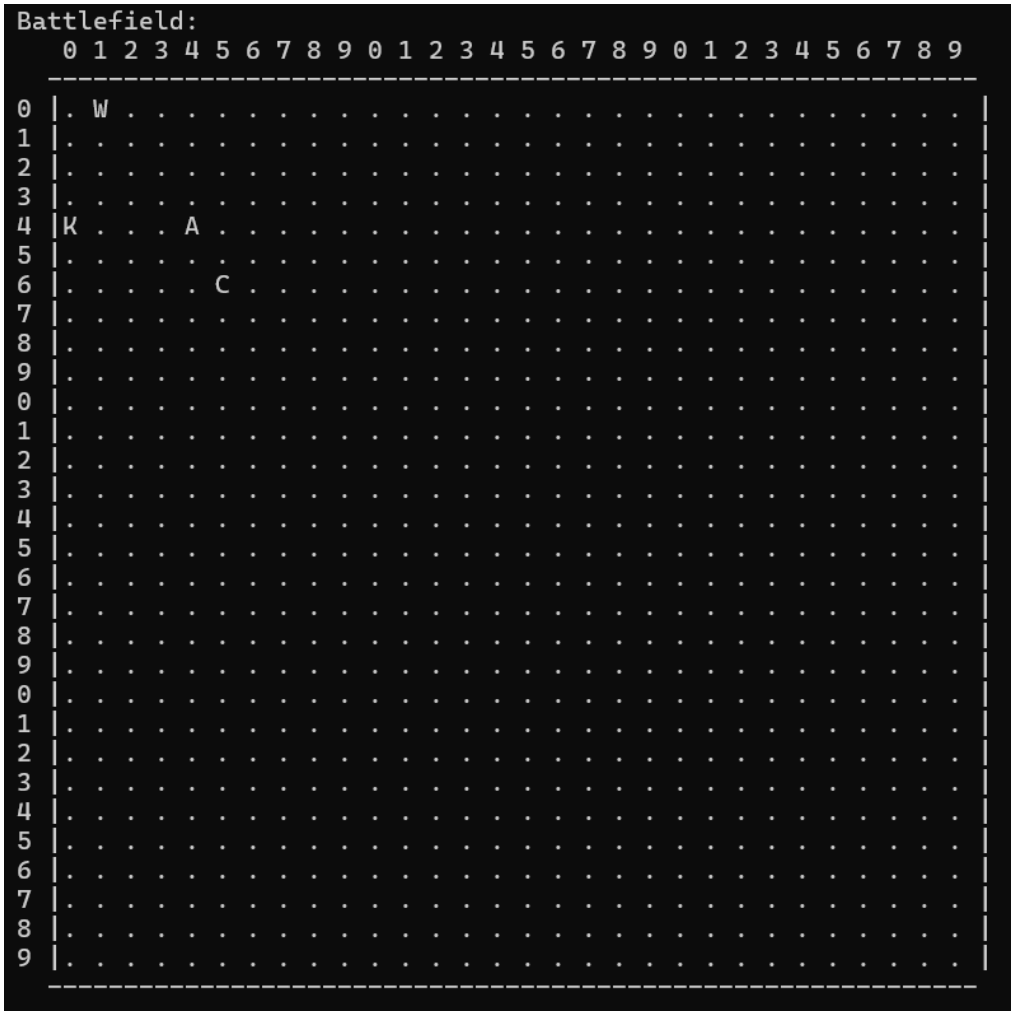
1. The function starts by checking if the battlefield is empty. If it is, the function returns immediately.
2. If the battlefield is not empty, it gets the number of rows and columns.
3. The function then logs the column headers and the top border of the battlefield.
4. It iterates through each row of the battlefield:
    1. For each cell in the row, if the cell is empty, it logs a dot (".").
    2. If the cell is not empty, it logs the first character of the robot's name.
5. After logging all the rows, it logs the bottom border of the battlefield.

These steps ensure that the state of the battlefield is both displayed on the console and recorded in a log file for further analysis or record-keeping.

FLOWCHART FOR UTILITY.H


printBattlefield(battlefield)
Function to print the battlefield to the console
battlefield is empty
yes
no
return
Get number of rows and columns
Print column headers
Print top border
Print each row
cell is empty
yes
no
Print "."
Print first character of robot's name
more rows
Print bottom border
logBattlefield(logFile, battlefield)
Function to log the battlefield to a file
battlefield is empty
yes
no
return
Get number of rows and columns
Log column headers
Log top border
Log each row
cell is empty
yes
no
Log "."
Log first character of robot's name
more rows
Log bottom border

SAMPLE LOOK OF THE BATTLEFIELD



- The grid size is limited to 30x30, which holds up 900 coordinates for the robots to perform their actions.
- The axes are there for the reference, y-axis is given from 0 to 29 but x-axis is only from 0 to 9 and repeated thrice the same order (this is because the digits doesn't align with the dots below precisely, hence given in single digits).
- Borders are given for a sturdy output.

AbstractRobots.h

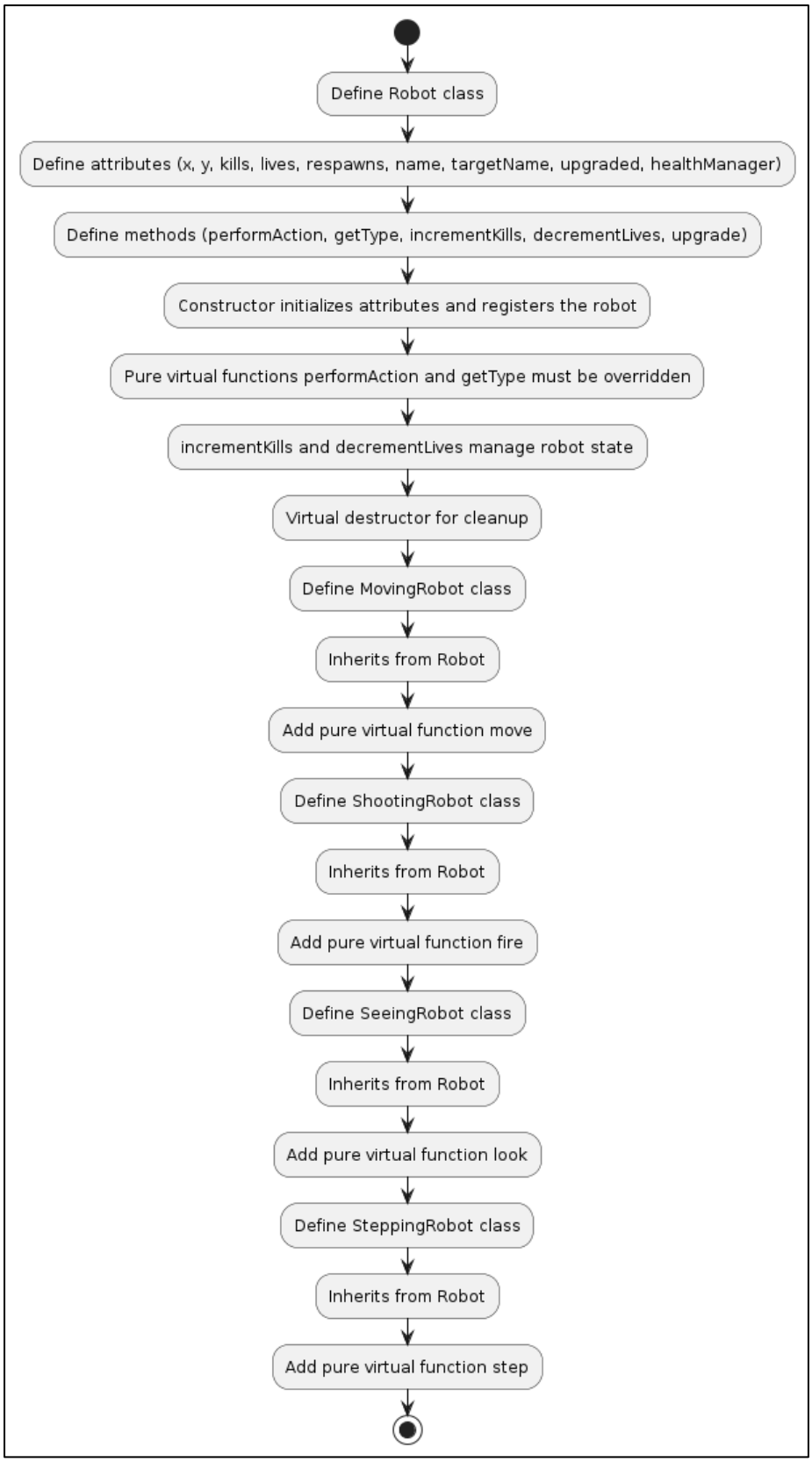
- **Robot Base Class**:
  - The `Robot` class is the base class for all robot types. It defines common attributes such as coordinates (`x`, `y`), `kills`, `lives`, `respawns`, `name`, `targetName`, `upgraded`, and a reference to `HealthManager`.
  - The constructor initializes these attributes and registers the robot with the health manager.
  - It has pure virtual functions `performAction` and `getType` which must be implemented by derived classes.
  - It includes methods `incrementKills`, `decrementLives`, and `upgrade` for managing robot state and behavior.
  - It has a virtual destructor for proper cleanup.
- **MovingRobot Class**:
  - Inherits from `Robot`.
  - Adds a pure virtual function `move` to be implemented by derived classes.
  - The constructor initializes the robot using the base class constructor.
- **ShootingRobot Class**:
  - Inherits from `Robot`.
  - Adds a pure virtual function `fire` to be implemented by derived classes.
  - The constructor initializes the robot using the base class constructor.
- **SeeingRobot Class**:
  - Inherits from `Robot`.
  - Adds a pure virtual function `look` to be implemented by derived classes.
  - The constructor initializes the robot using the base class constructor.
- **SteppingRobot Class**:
  - Inherits from `Robot`.
  - Adds a pure virtual function `step` to be implemented by derived classes.
  - The constructor initializes the robot using the base class constructor.

PSEUDOCODE FOR ABSTRACTROBOTS.H

```
CLASS Robot:
  PUBLIC:
    INT x, y  // Coordinates of the robot
    INT kills  // Number of kills the robot has made
    INT lives  // Number of lives the robot has left
    INT respawns  // Number of respawns the robot has left
    STRING name  // Name of the robot
    STRING targetName  // Name of the target robot
    BOOL upgraded  // Whether the robot is upgraded
    HealthManager& healthManager  // Reference to the health manager

    CONSTRUCTOR Robot(x, y, name, hm):
      INITIALIZE x, y, name, healthManager
      SET kills = 0
      SET lives = 3
      SET respawns = 3
      SET targetName = ""
      SET upgraded = FALSE
      CALL healthManager.registerRobot(name)

    PURE VIRTUAL FUNCTION performAction(battlefield, rows, cols)

    PURE VIRTUAL FUNCTION getType() CONST

    FUNCTION incrementKills():
      INCREMENT kills
      IF kills == 3:
        CALL upgrade()
          SET lives = 3  // Reset lives for respawn
```

```
FUNCTION decrementLives():
        IF lives > 0:
          DECREMENT lives
          IF lives == 0 AND respawns > 0:
            DECREMENT respawns
            SET lives = 3  // Reset lives for respawn

      VIRTUAL FUNCTION upgrade()

      VIRTUAL DESTRUCTOR ~Robot()

CLASS MovingRobot INHERITS Robot:
  PUBLIC:
    CONSTRUCTOR MovingRobot(x, y, name, hm)
    PURE VIRTUAL FUNCTION move(battlefield, rows, cols)

CLASS ShootingRobot INHERITS Robot:
  PUBLIC:
    CONSTRUCTOR ShootingRobot(x, y, name, hm)
    PURE VIRTUAL FUNCTION fire(battlefield, rows, cols)

CLASS SeeingRobot INHERITS Robot:
  PUBLIC:
    CONSTRUCTOR SeeingRobot(x, y, name, hm)
    PURE VIRTUAL FUNCTION look(battlefield, rows, cols)

CLASS SteppingRobot INHERITS Robot:
  PUBLIC:
    CONSTRUCTOR SteppingRobot(x, y, name, hm)
    PURE VIRTUAL FUNCTION step(battlefield, rows, cols)
```

FLOWCHART


Define Robot class
Define attributes (x, y, kills, lives, respawns, name, targetName, upgraded, healthManager)
Define methods (performAction, getType, incrementKills, decrementLives, upgrade)
Constructor initializes attributes and registers the robot
Pure virtual functions performAction and getType must be overridden
incrementKills and decrementLives manage robot state
Virtual destructor for cleanup
Define MovingRobot class
Inherits from Robot
Add pure virtual function move
Define ShootingRobot class
Inherits from Robot
Add pure virtual function fire
Define SeeingRobot class
Inherits from Robot
Add pure virtual function look
Define SteppingRobot class
Inherits from Robot
Add pure virtual function step

HEALTHMANAGER.H

The `HealthManager` class is designed to manage the health and scores of robots in the robot battlefield simulation. It provides various functions to manipulate and retrieve the health and scores of the robots.

Components:

1. **Private Members**:
   - `std::unordered_map<std::string, int> health`: A hash map that stores the health (number of lives) of each robot, keyed by the robot's name.
   - `std::unordered_map<std::string, int> scores`: A hash map that stores the scores (points) of each robot, keyed by the robot's name.
2. **Public Methods**:
   - **Constructor**: Initializes the health and scores of each robot to 3 lives and 0 points, respectively.

     ```
     HealthManager(const std::vector<std::string>& robotNames)
     ```

   - **Copy Constructor**: Creates a copy of another `HealthManager` object.

     ```
     HealthManager(const HealthManager& other)
     ```

   - **registerRobot**: Adds a new robot to the health and scores maps with initial values.

     ```
     void registerRobot(const std::string& name)
     ```

   - **decreaseHealth**: Decreases the health of a specified robot by 1.

     ```
     void decreaseHealth(const std::string& name)
     ```

   - **increaseScore**: Increases the score of a specified robot by a given number of points.

     ```
     void increaseScore(const std::string& name, int points)
     ```

   - **getHealth**: Retrieves the current health of a specified robot.

     ```
     int getHealth(const std::string& name) const
     ```

   - **getScore**: Retrieves the current score of a specified robot.

     ```
     int getScore(const std::string& name) const
     ```

   - **printHealth**: Prints the health of all robots to the console.

     ```
     void printHealth() const
     ```

   - **printScores**: Prints the scores of all robots to the console.

     ```
     void printScores() const
     ```

   - **logHealth**: Logs the health of all robots to a file.

     ```
     void logHealth(std::ofstream& logFile) const
     ```

- **logScores**: Logs the scores of all robots to a file.

```
void logScores(std::ofstream& logFile) const
```

- **getWinner**: Returns the name of the robot with the highest score.

```
std::string getWinner() const
```

Explanation of Key Methods

- **Constructor**: Initializes the health and scores of robots based on the provided list of robot names.

```
HealthManager(const std::vector<std::string>& robotNames) {
    for (const auto& name : robotNames) {
        health[name] = 3;  // Initialize each robot with 3 lives
        scores[name] = 0;  // Initialize each robot with 0 points
    }
}
```

- **Copy Constructor**: Copies the health and scores from another `HealthManager` object.

```
HealthManager(const HealthManager& other)
    : health(other.health), scores(other.scores) {}
```

- **registerRobot**: Adds a new robot to the maps with initial values.

```
void registerRobot(const std::string& name) {
    health[name] = 3;
    scores[name] = 0;
}
```

- **decreaseHealth**: Decreases the health of a robot by 1 if the robot exists and has health remaining.

```
void decreaseHealth(const std::string& name) {
    if (health.find(name) != health.end() && health[name] > 0) {
        health[name]--;
    }
}
```

- **increaseScore**: Increases the score of a robot by the specified number of points.

```
void increaseScore(const std::string& name, int points) {
    if (scores.find(name) != scores.end()) {
        scores[name] += points;
    }
}
```

- **getHealth**: Retrieves the health of a robot. Returns 0 if the robot does not exist.

```
int getHealth(const std::string& name) const {
    auto it = health.find(name);
    return (it != health.end()) ? it->second : 0;
}
```

- **getScore**: Retrieves the score of a robot. Returns 0 if the robot does not exist.

```
int getScore(const std::string& name) const {
    auto it = scores.find(name);
    return (it != scores.end()) ? it->second : 0;
}
```

- **printHealth** and **printScores**: Print the health and scores of all robots to the console, respectively.

```
void printHealth() const {
    for (const auto& entry : health) {
        std::cout << entry.first << " : --- (" << entry.second << " lives
left)\n";
    }
}


void printScores() const {
    for (const auto& entry : scores) {
        std::cout << entry.first << ": " << entry.second << " points\n";
    }
}
```

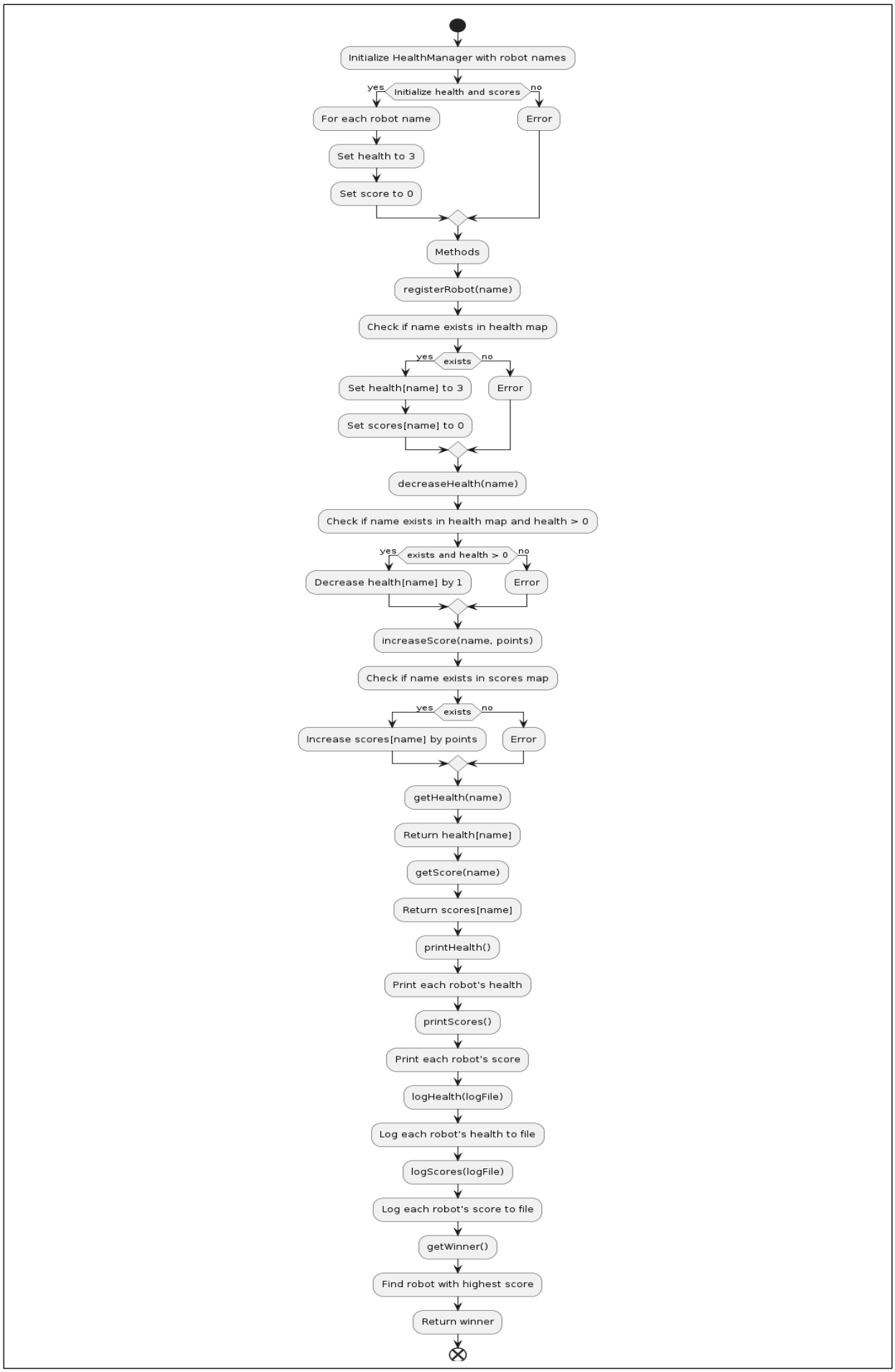
- **logHealth** and **logScores**: Log the health and scores of all robots to a file, respectively.

```
void logHealth(std::ofstream& logFile) const {
    for (const auto& entry : health) {
        logFile << entry.first << " : --- (" << entry.second << " lives
left)\n";
    }
}

void logScores(std::ofstream& logFile) const {
    for (const auto& entry : scores) {
        logFile << entry.first << ": " << entry.second << " points\n";
    }
}
```

- **getWinner**: Returns the name of the robot with the highest score.

```
std::string getWinner() const {
    std::string winner;
    int maxScore = 0;
    for (const auto& entry : scores) {
        if (entry.second > maxScore) {
            maxScore = entry.second;
            winner = entry.first;
        }
    }
    return winner;
}
```

FLOWCHART


Initialize HealthManager with robot names
Initialize health and scores
yes
no
For each robot name
Error
Set health to 3
Set score to 0
Methods
registerRobot(name)
Check if name exists in health map
exists
yes
no
Set health[name] to 3
Error
Set scores[name] to 0
decreaseHealth(name)
Check if name exists in health map and health > 0
exists and health > 0
yes
no
Decrease health[name] by 1
Error
increaseScore(name, points)
Check if name exists in scores map
exists
yes
no
Increase scores[name] by points
Error
getHealth(name)
Return health[name]
getScore(name)
Return scores[name]
printHealth()
Print each robot's health
printScores()
Print each robot's score
logHealth(logFile)
Log each robot's health to file
logScores(logFile)
Log each robot's score to file
getWinner()
Find robot with highest score
Return winner

THE 7 ROBOT TYPE.H FILES

In the robot battlefield simulation, there are several types of robots, each with unique capabilities and behaviors. These robots inherit from a base class and, depending on their type, also from one or more specialized classes that provide specific functionalities like moving, shooting, and seeing.

Robot Types Capabilities Table

Robot Type	Look	Move	Fire	Upgrade Condition	Upgrade To
RoboCop	Yes	Once at random	3 random positions (max city block distance = 10)	3 robots (move to a location, kill robot)	TerminatorRoboCop (TRC)
Terminator	Yes	Random 3x3	Terminate by moving	Allowed to fire, 3 robots	TerminatorRoboCop (TRC)
TerminatorRoboCop (TRC)	Yes	Fire like RoboCop	Steps like Terminator	3 robots	UltimateRobot (UR)
BlueThunder (BT)	No	None	One of 8 surrounding cells (clockwise, not random)	3 robots	Madbot
Madbot	No	None	Random 3x3	3 robots	RoboTank (RT)
RoboTank (RT)	No	None	Random (once) at the entire battlefield	3 robots	UltimateRobot (UR)
UltimateRobot (UR)	Yes	Yes (steps on and kills)	Shoots randomly at 3 locations	None	None

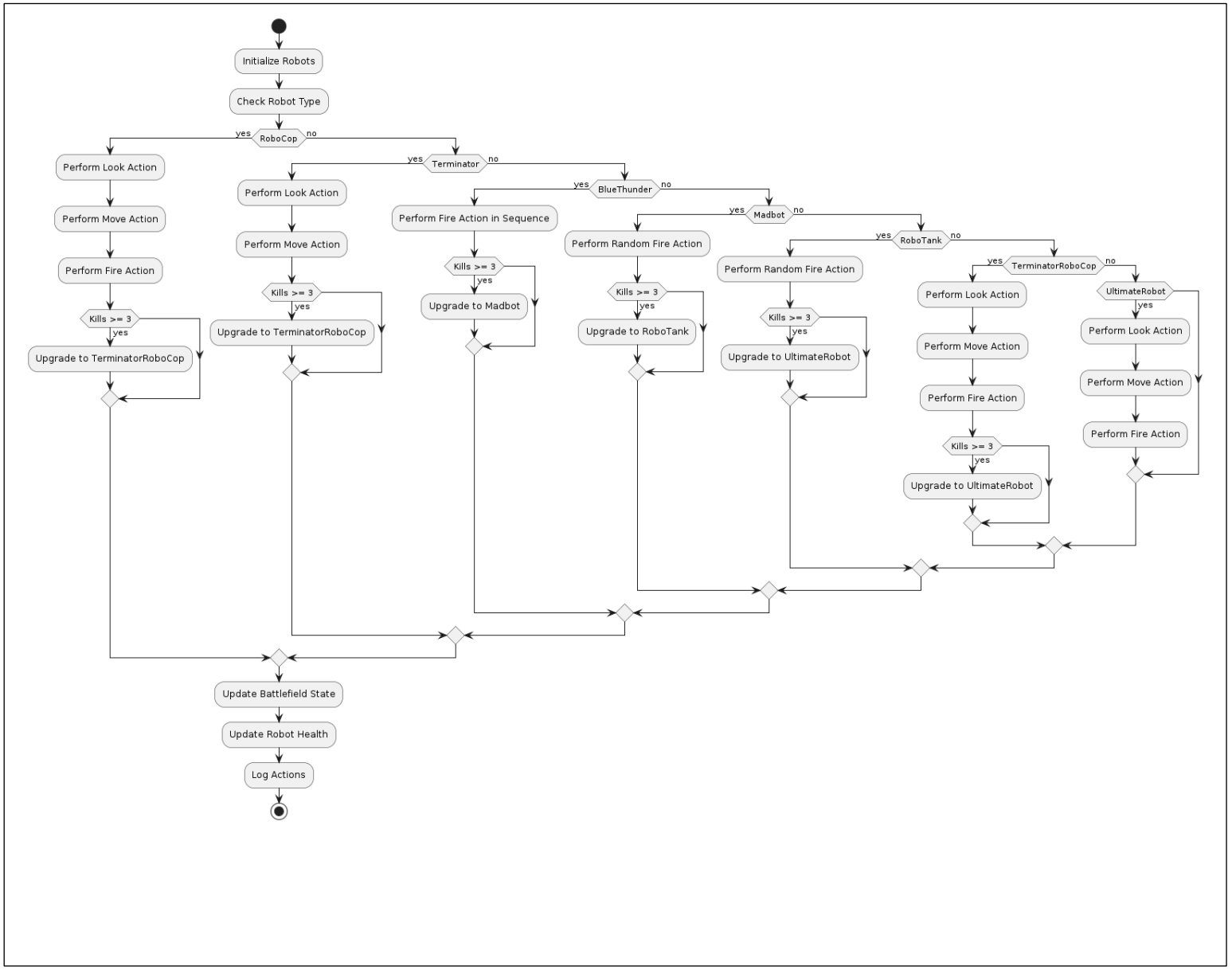
Explanation

This table summarizes the capabilities and upgrade paths for each robot type in the simulation. Each robot has specific actions they can perform during their turn:

1. **RoboCop**:
    - **Look**: Can look around.
    - **Move**: Moves once to a random position.
    - **Fire**: Fires at 3 random positions with a max city block distance of 10.
    - **Upgrade**: Upgrades to TerminatorRoboCop after killing 3 robots.
2. **Terminator**:
    - **Look**: Can look around.
    - **Move**: Moves randomly within a 3x3 grid.
    - **Fire**: Terminates robots by moving into their location.

    - **Upgrade**: Upgrades to TerminatorRoboCop after killing 3 robots, gaining the ability to fire.
3. **TerminatorRoboCop (TRC)**:
    - **Look**: Can look around.
    - **Move**: Moves like a Terminator.
    - **Fire**: Fires like a RoboCop.
    - **Upgrade**: Upgrades to UltimateRobot after killing 3 robots.
4. **BlueThunder (BT)**:
    - **Look**: Cannot look around.
    - **Move**: Does not move.
    - **Fire**: Fires at one of 8 surrounding cells in a clockwise manner.
    - **Upgrade**: Upgrades to Madbot after killing 3 robots.
5. **Madbot**:
    - **Look**: Cannot look around.
    - **Move**: Does not move.
    - **Fire**: Fires randomly within a 3x3 grid.
    - **Upgrade**: Upgrades to RoboTank after killing 3 robots.
6. **RoboTank (RT)**:
    - **Look**: Cannot look around.
    - **Move**: Does not move.
    - **Fire**: Fires randomly at any location on the battlefield.
    - **Upgrade**: Upgrades to UltimateRobot after killing 3 robots.
7. **UltimateRobot (UR)**:
    - **Look**: Can look around.
    - **Move**: Moves and steps on/kills robots.
    - **Fire**: Fires randomly at 3 locations on the battlefield.
    - **Upgrade**: No further upgrades.

FLOWCHART


Initialize Robots
Check Robot Type
RoboCop
yes
no
Perform Look Action
Perform Move Action
Perform Fire Action
Kills >= 3
yes
Upgrade to TerminatorRoboCop
Terminator
Perform Look Action
Perform Move Action
Kills >= 3
yes
Upgrade to TerminatorRoboCop
BlueThunder
Perform Fire Action in Sequence
Kills >= 3
yes
Upgrade to Madbot
Madbot
Perform Random Fire Action
Kills >= 3
yes
Upgrade to RoboTank
RoboTank
Perform Random Fire Action
Kills >= 3
yes
Upgrade to UltimateRobot
TerminatorRoboCop
Perform Look Action
Perform Move Action
Perform Fire Action
Kills >= 3
yes
Upgrade to UltimateRobot
UltimateRobot
Perform Look Action
Perform Move Action
Perform Fire Action
Update Battlefield State
Update Robot Health
Log Actions

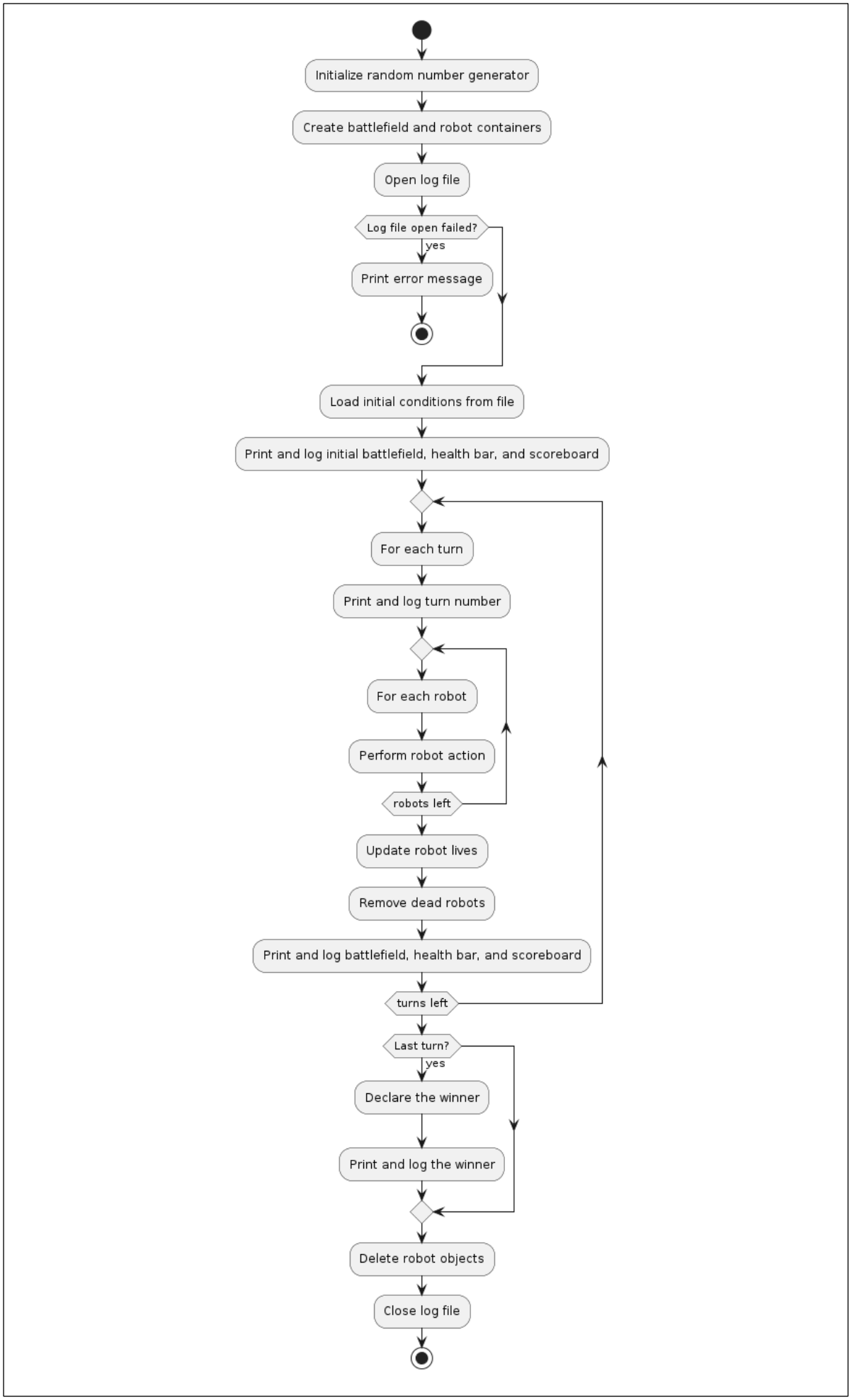
MAIN.CPP

The `main.cpp` file serves as the entry point for the robot battlefield simulation program. The following sections provide an overview of its key components and functionality:

Key Components

1. **Include Statements**
   - The necessary headers are included for input/output, file handling, string stream operations, time functions, and custom classes for robots, utility functions, and the health manager.
2. **Function: loadInitialConditions**
   - Reads the initial conditions from a file to set up the battlefield, the robots, and the number of simulation turns.
   - Initializes the battlefield dimensions and robot details (type, name, coordinates).
   - Validates the battlefield size and coordinates of robots.
   - Registers each robot with the `HealthManager`.
3. **Function: validateCoordinates**
   - Ensures that the coordinates of robots are within the bounds of the battlefield.
4. **Function: updateRobotLives**
   - Updates the lives of robots based on their health status managed by the `HealthManager`.
5. **Function: removeDeadRobots**
   - Removes robots with zero lives from the battlefield and logs the removal.
6. **Function: performRobotAction**
   - Executes the actions for each robot (look, move, fire).
   - Updates the battlefield and the health manager based on the actions performed.
7. **Main Function**
   - Initializes the battlefield and the list of robots.
   - Reads the initial conditions from a file.
   - Sets up the health manager.
   - Runs the simulation for a specified number of turns.
   - Logs the initial and final states of the battlefield, health bar, and scoreboard.
   - Determines and logs the winner based on the scores.

FLOWCHART FOR MAIN.CPP



Detailed Explanation

1. **Initialization**:
    - The random number generator is seeded with the current time.
    - Containers for the battlefield (`CustomDynamicArray<CustomDynamicArray<string>>`) and robots (`CustomDynamicArray<Robot*>`) are created.
    - A log file is opened for writing.
2. **Load Initial Conditions**:
    - The `loadInitialConditions` function reads from a file to set up the battlefield size, number of turns, and robot details (type, name, initial position).
    - Robots are dynamically created based on their type and added to the battlefield and robot list.
    - The `HealthManager` is initialized with the names of the robots.
3. **Initial State Logging**:
    - The initial state of the battlefield, health bar, and scoreboard is printed to the console and logged to the file.
4. **Simulation Loop**:
    - For each turn:
        - The turn number is printed and logged.
        - Each robot performs its action (look, move, fire) using the `performRobotAction` function.
        - The `updateRobotLives` function updates the lives of robots based on their health.
        - The `removeDeadRobots` function removes robots with zero lives from the battlefield and deletes their objects.
        - The state of the battlefield, health bar, and scoreboard is printed and logged.
5. **End of Simulation**:
    - After all turns are completed, the winner is determined based on the highest score.
    - The winner is printed and logged.
    - All dynamically allocated robot objects are deleted.
    - The log file is closed.

INPUT VALIDATIONS

- **main.cpp**

1. **File Opening Validation**:
    1. **Function**: `loadInitialConditions`
    2. **Validation**:

```
if (!inputFile) {
    throw runtime_error("Error opening input
file!");
}
```

    3. **Explanation**: Checks if the input file can be opened. If not, throws a runtime error.
2. **Battlefield Size Validation**:
    1. **Function**: `loadInitialConditions`
    2. **Validation**:

```
if (numRows > 30 || numCols > 30) {
    throw runtime_error("Error: Battlefield size
exceeds 30x30 grid!");
}
```

    3. **Explanation**: Ensures the battlefield size does not exceed 30x30 grid. If it does, throws a runtime error.
3. **Random Position Validation**:
    1. **Function**: `loadInitialConditions`
    2. **Validation**:

```
if (name == "random") {
    x = rand() % numRows;
    y = rand() % numCols;
}
```

    3. **Explanation**: Generates random coordinates for robots if their position is specified as "random".
4. **Coordinate Validation**:
    1. **Function**: `validateCoordinates`
    2. **Validation**:

```
if (x < 0) x = 0;
if (x >= maxX) x = maxX - 1;
if (y < 0) y = 0;
if (y >= maxY) y = maxY - 1;
```

    3. **Explanation**: Ensures robot coordinates are within the battlefield boundaries. Adjusts coordinates if they are out of bounds.
5. **Health Validation**:
    1. **Function**: `updateRobotLives`

2. **Validation**:

```
if (healthManager.getHealth(robots[i]->name) <= 0) {
    robots[i]->decrementLives();
}
```

3. **Explanation**: Checks if a robot's health is zero or less, and decrements its lives if true.

❖ **CustomDynamicArray.h**

1. **Array Access Validation**:
   - **Function**: `operator[]` and `operator[] const`
   - **Validation**:

```
if (index >= size) {
    throw std::out_of_range("Index out of range in
operator[]");
}
```

   - **Explanation**: Ensures the accessed index is within the valid range of the array. Throws an out_of_range exception if it is not.
2. **Array Removal Validation**:
   - **Function**: `remove_at`
   - **Validation**:

```
if (index >= size) {
    throw std::out_of_range("Index out of range in
remove_at");
}
```

   - **Explanation**: Ensures the index to be removed is within the valid range of the array. Throws an out_of_range exception if it is not.

❖ **HealthManager.h**

1. **Health Decrease Validation**:
   - **Function**: `decreaseHealth`
   - **Validation**:

```
if (health.find(name) != health.end() &&
health[name] > 0) {
    health[name]--;
}
```

   - **Explanation**: Checks if the robot exists and its health is greater than zero before decreasing its health.
2. **Score Increase Validation**:
   - **Function**: `increaseScore`
   - **Validation**:

```
if (scores.find(name) != scores.end()) {
    scores[name] += points;
}
```

- **Explanation**: Checks if the robot exists in the scores map before increasing its score.

❖ **AbstractRobots.h**

1. **Lives Decrement and Respawn Validation**:
   - **Function**: `decrementLives`
   - **Validation**:

```
if (lives > 0) {
    lives--;
    if (lives == 0 && respawns > 0) {
        respawns--;
        lives = 3; // Reset lives for respawn
    }
}
```

   - **Explanation**: Ensures lives are only decremented if they are greater than zero. Handles respawn by resetting lives if respawns are available.

INPUT VALIDATION: THE ROBOT CLASSES

There are 3 main input validations for the robot files, here is 2 examples:

❖ TerminatorRoboCop.h

1. **Move Validation**:
   - **Function**: `move`
   - **Validation**:

```
if (newX >= 0 && newY >= 0 && newX < rows && newY < cols &&
battlefield[newX][newY] != "") {
    healthManager.decreaseHealth(battlefield[newX][newY]);
}
```

   - **Explanation**: Ensures the target coordinates are within the battlefield boundaries and the target cell is not empty before moving and terminating the target.
2. **Upgrade Validation**:
   - **Function**: `upgrade`
   - **Validation**:

```
if (kills >= 3 && !upgraded) {
    upgraded = true;
}
```

- **Explanation**: Ensures the robot is upgraded only if it has achieved 3 kills and has not already been upgraded.

❖ UltimateRobot.h

1. **Move Validation**:
   - **Function**: `move`
   - **Validation**:

     ```
     if (newX >= 0 && newY >= 0 && newX < rows && newY < cols &&
     battlefield[newX][newY] != "") {
         healthManager.decreaseHealth(battlefield[newX][newY]);
     }
     ```

   - **Explanation**: Ensures the target coordinates are within the battlefield boundaries and the target cell is not empty before moving and terminating the target.
2. **Fire Validation**:
   - **Function**: `fire`
   - **Validation**:

     ```
     if (!battlefield[targetX][targetY].empty()) {

     healthManager.decreaseHealth(battlefield[targetX][targetY]);
     }
     ```

   - **Explanation**: Ensures the target cell is not empty before firing.

Summary of Input Validations in Robot Types

- **Upgrade Validation**: Ensures robots are upgraded only after achieving the required number of kills.
- **Move Validation**: Ensures new coordinates are within battlefield boundaries and target cells are valid.
- **Fire Validation**: Ensures target coordinates are within battlefield boundaries and target cells are not empty.

EXAMPLE PROGRAM OUTPUT

1. In the initial_conditions text file, this is how the input looks like,

```
10 10
10
4
UltimateRobot Widd 3 6
RoboTank Jet 3 1
Terminator Alpha 7 8
RoboCop Beta random random
```

-Grid size (fixed)
-No of turns
-No of robots
-Robot type, Name and Coordinates (anywhere from (0,0) to (30,30)).

2. If we run the main.cpp file, this is the sample output looks like,

```
Initial Battlefield:
   0 1 2 3 4 5 6 7 8 9
  ---------------------
0 |. . . . . . . . B . |
1 |. . . . . . . . . . |
2 |. . . . . . . . . . |
3 |. J . . . . W . . . |
4 |. . . . . . . . . . |
5 |. . . . . . . . . . |
6 |. . . . . . . . . . |
7 |. . . . . . . . A . |
8 |. . . . . . . . . . |
9 |. . . . . . . . . . |
  ---------------------

HEALTHBAR:
Beta : --- (3 lives left)
Alpha : --- (3 lives left)
Widd : --- (3 lives left)
Jet : --- (3 lives left)

SCOREBOARD:
Beta: 0 points
Alpha: 0 points
Widd: 0 points
Jet: 0 points
```

```
Turn 1
Widd looks around at (3, 6).
Widd moves to (2, 5).
Widd fires at (7, 8).
Widd has fired at Alpha.
Widd fires at (3, 8).
Widd fires at (6, 4).
Jet fires at (0, 3).
Alpha looks around at (7, 8).
Alpha moves to (7, 9).
Alpha fires at (1, 6).
Alpha fires at (8, 1).
Alpha fires at (8, 7).
Beta looks around at (0, 8).
Beta targets Widd at (2, 5).
Beta moves to (0, 7).
Beta fires at (7, 4).
Beta fires at (1, 8).
Beta fires at (6, 8).

Battlefield:
   0 1 2 3 4 5 6 7 8 9
  ---------------------
0 |. . . . . . . B . . |
1 |. . . . . . . . . . |
2 |. . . . . W . . . . |
3 |. J . . . . . . . . |
4 |. . . . . . . . . . |
5 |. . . . . . . . . . |
6 |. . . . . . . . . . |
7 |. . . . . . . . . A |
8 |. . . . . . . . . . |
9 |. . . . . . . . . . |
  ---------------------

HEALTHBAR:
Beta : --- (3 lives left)
Alpha : --- (2 lives left)
Widd : --- (2 lives left)
Jet : --- (3 lives left)
```

```
SCOREBOARD:
Beta: 5 points
Alpha: 0 points
Widd: 0 points
Jet: 0 points
```

...(SKIPPED TURN 2 TO TURN 9 DUE TO MANY PAGES)

```
Turn 10
Widd looks around at (0, 1).
Widd moves to (0, 2).
Widd fires at (8, 0).
Widd fires at (2, 3).
Widd fires at (1, 9).
Jet has fired at Beta.
Jet fires at (1, 8).
Alpha looks around at (6, 5).
Alpha moves to (5, 6).
Alpha fires at (8, 3).
Alpha fires at (6, 9).
Alpha fires at (6, 7).
Beta looks around at (1, 8).
Beta targets Widd at (0, 2).
Beta moves to (0, 8).
Beta fires at (0, 2).
Beta has fired at Widd.
Beta fires at (3, 2).
Beta fires at (6, 8).

Battlefield:
   0 1 2 3 4 5 6 7 8 9
  ---------------------
0 |. . . . . . . . B . |
1 |. . . . . . . . . . |
2 |. . . . . . . . . . |
3 |. J . . . . . . . . |
4 |. . . . . . . . . . |
5 |. . . . . . A . . . |
6 |. . . . . . . . . . |
7 |. . . . . . . . . . |
8 |. . . . . . . . . . |
9 |. . . . . . . . . . |
  ---------------------
```

```
HEALTHBAR:
Beta : --- (3 lives left)
Alpha : --- (2 lives left)
Widd : --- (0 lives left)
Jet : --- (2 lives left)

SCOREBOARD:
Beta: 55 points
Alpha: 0 points
Widd: 0 points
Jet: 5 points

The winner is: Beta with 55 points!
```

3. The expected output will be also stored in the simulation_log.txt file.

```
Turn 1
   0 1 2 3 4 5 6 7 8 9
  ---------------------
0 |. . . . . . . B . . |
1 |. . . . . . . . . . |
2 |. . . . . W . . . . |
3 |. J . . . . . . . . |
4 |. . . . . . . . . . |
5 |. . . . . . . . . . |
6 |. . . . . . . . . . |
7 |. . . . . . . . . A |
8 |. . . . . . . . . . |
9 |. . . . . . . . . . |
  ---------------------

HEALTHBAR:
Beta : --- (3 lives left)
Alpha : --- (2 lives left)
Widd : --- (2 lives left)
Jet : --- (3 lives left)

SCOREBOARD:
Beta: 5 points
Alpha: 0 points
Widd: 0 points
Jet: 0 points
```

- Only the battlefield, healthbar and scoreboard are printed in the simulation_log.txt file, the actions are not.

INSTRUCTIONS TO COMPILE AND RUN THE ROBOT WARS SIMULATOR

1. Prerequisites:

C++ Compiler: Ensure you have a C++ compiler installed on your system. Popular choices include GCC (GNU Compiler Collection) for Unix-like systems or MinGW for Windows.

C++ Standard Library: The code utilizes standard C++ libraries, which are typically included with the compiler.

2. Upload the Code:

Download and upload the C++ code for the robot war simulator. Save the code to a directory of your choice.

3. Compilation:

Open a terminal or command prompt and navigate to the directory where you saved the code.

Unix-like Systems (Using GCC):

```
g++ -o robot_war_simulator main.cpp CustomDynamicArray.cpp
HealthManager.cpp RoboCop.cpp BlueThunder.cpp Madbot.cpp RoboTank.cpp
Terminator.cpp TerminatorRoboCop.cpp UltimateRobot.cpp Utility.cpp
```

Windows (Using MinGW):

```
g++ -o robot_war_simulator.exe main.cpp CustomDynamicArray.cpp
HealthManager.cpp RoboCop.cpp BlueThunder.cpp Madbot.cpp RoboTank.cpp
Terminator.cpp TerminatorRoboCop.cpp UltimateRobot.cpp Utility.cpp
```

This command compiles the code and generates an executable named `robot_war_simulator` (or `robot_war_simulator.exe` on Windows).

4. Running the Robot War Simulator:

Unix-like Systems:

```
./robot_war_simulator
```

Windows:

```
robot_war_simulator.exe
```

5. Writing the Data to Output File:

Since the C++ program prints information to the console using `cout`, you can use the `>` operator to redirect the output to a file.

Unix-like Systems:

```
./robot_war_simulator > output.txt
```

Windows:

```
robot_war_simulator.exe > output.txt
```

This command will execute your robot war simulator and redirect the output to `output.txt`. The `>` operator is used for output redirection. After running the program with this modified command, the execution log, including the state of the simulation after each turn, will be written to the `output.txt` file in the same directory. You can then open and review this file to see the program's detailed execution log.

6. Interpreting the Simulation Output:

Once the robot war simulator is running, it will execute the simulation based on the initial conditions specified in the input file. The simulator will display the state of the battlefield, the actions of each robot, and the scores after each turn.

7. Output Log:

The simulator generates an output log file named `output.txt` in the same directory. This file contains the simulation's execution log, including the state of the battlefield and the health and scores of each robot after each turn. You can refer to this log for detailed information about the simulation's execution.

8. Troubleshooting:

- Ensure the input file `initial_conditions.txt` is in the correct format and contains valid initial conditions for the simulation.
- Check for any error messages displayed during compilation and execution.
- Make sure the input file path provided in the command-line argument is correct.
- Verify that all necessary files (header and implementation files) are present in the directory when compiling.

CONCLUSION

In conclusion, this assignment provided an in-depth exploration into the design and implementation of a turn-based robot war simulator using standard C++. The simulator's architecture, composed of a variety of robot types each with unique abilities, simulates a battlefield where strategy and randomness play crucial roles. Through this project, several key object-oriented programming (OOP) concepts such as inheritance, polymorphism, encapsulation, and operator overloading were effectively utilized.

The implementation required developing robust C++ classes for different robot types, managing their actions on a dynamic battlefield, and tracking their health and scores. The project showcased the importance of designing modular, reusable, and maintainable code. Extensive use of custom data structures, like dynamic arrays, highlighted the flexibility and power of C++ in managing complex data.

By tackling this project, significant insights were gained into simulation design, dynamic memory management, and the practical application of OOP principles. The task also emphasized the importance of thorough input validation, error handling, and comprehensive testing to ensure the reliability of the simulation.

Overall, this assignment was a valuable exercise in applying theoretical knowledge to a practical and engaging problem, enhancing understanding of advanced C++ programming, simulation development, and OOP methodologies. It underscored the importance of careful design, meticulous implementation, and the ability to adapt theoretical concepts to solve real-world challenges.

REFERENCES

Website links:
https://www.google.com/search?q=robot+wars+c%2B%2B&source=lmns&bih=608&biw=1372&rlz=1C1ONGR_enMY1020MY1020&hl=en&sa=X&ved=2ahUKEwiihaDEwICHAxVMkmMGHQn2DPkQ0pQJKAB6BAgBEAI

- MinGW original page.
https://www.mingw.org/
or
https://sourceforge.net/projects/mingw/
or https://mingw.osdn.io/

- Codeblocks homepage.
https://www.codeblocks.org/

Individuals:

- Ts. Goh Chien Le
Assistant Professor
FACULTY OF COMPUTING AND INFORMATICS (FCI)
BR4015 , Multimedia University, Persiaran Multimedia
63100, Cyberjaya, Selangor
03-83125251
clgoh@mmu.edu.my

- Mr. Sharaf El-Deen Sami Mohammed Al- Horani
Specialist 2
FACULTY OF COMPUTING AND INFORMATICS (FCI)
BR4007 , Multimedia University, Persiaran Multimedia
63100, Cyberjaya, Selangor
03-83125232
sharaf.horani@mmu.edu.my

-THE END-