Faculty of Computing Informatics

CCP6124 - Object Oriented Programming and Data Structures

Trimester 2, 2023/2024

Assignment

Lecture Section: TC2L
Tutorial Section: TT5L
Prepared for: Sharaf El-Deen Sami Mohammed Al- Horani

Group Members:

Student ID	Name
1221103666	Sim Yi Jie
1221101183	Tay Zi Yan
1221103143	MUHAMMAD IMRAN BIN MOHD KAMAL

Introduction

The Robot War Simulation programme reads specifications from a file to create a battlefield where various robot types fight each other. The whole simulation process—including initialization, robot deployment, action simulation, and outcome evaluation—is covered in depth in this study.

Class Hierarchy Documentation

This hierarchy outlines the relationships between the main classes and their roles in simulating the battlefield and robot interactions.The program consists of several classes, each representing different components of the simulation:

1. **Battlefield**: Represents the battlefield grid where robots are placed and perform actions.
2. **Circular_linked_list**: Manages a circular list of robots for sequential action simulation.
3. **Queue**: Implements a queue data structure to manage robots awaiting deployment.
4. **Robot (Base Class)**: Represents a generic robot with basic properties and actions like position, name, and battlefield reference.
5. **Derived Robot Classes (MovingRobot, SeeingRobot, ShootingRobot, SteppingRobtot)**: Represent specific types of robots with unique behaviors and attributes.
6. **Special Robot Classes (RoboCop, Terminator, TerminatorRoboCop, BlueThunder, MadBot, RoboTank and UltimateRobot):**

Implementation Details

1. Battlefield Class

```
class Battlefield
{
private:
    int rows, cols;
    string field[MAX_BATTLEFIELD_SIZE][MAX_BATTLEFIELD_SIZE];
    int step;
    vector<Robot *> robots;

public:
    Battlefield(const string &filename, CircularLinklist<Robot *> &lst)
    {
        ifstream file(filename);
        if (!file)
        {
            cerr << "Error opening file: " << filename << endl;
            exit(1);
        }

        string lines;
        string line;

        // Read the M by N : rows cols line
        getline(file, lines);
        istringstream iss1(lines);
        iss1 >> line >> line >> line >> rows >> cols;
        cout << "M by N: " << rows << "  " << cols << endl;

        // Read the steps: steps line
        getline(file, lines);
        istringstream iss2(lines);
        int step;
        iss2 >> line >> step;
        cout << "steps: " << step << endl;

        // Read the robots: count line
        getline(file, lines);
        istringstream iss3(lines);
        int robotCount;
        iss3 >> line >> robotCount;
        cout << "robots: " << robotCount << endl;

        for (int i = 0; i < robotCount; ++i)
        {
            string type, name, x_str, y_str;
            int x, y;
            file >> type >> name >> x_str >> y_str;
            cout << type << " " << name << " " << x_str << " " << y_str
<< "\n";

            // Convert x and y to integers or handle 'random'
            if (x_str == "random")
            {
                do
                {
                    x = rand() % rows;
```

Attributes:

- “`int rows`” : Number of rows in the battlefield.
- “`int cols`” : Number of columns in the battlefield.
- “`string field[MAX_BATTLEFIELD_SIZE][MAX_BATTLEFIELD_SIZE]`” : 2D array representing the battlefield grid.
- “`int step`” : Number of steps the game will run.
- “`vector<Robot *> robots`” : A vector storing pointers to `Robot` objects.

Constructor:

“`Battlefield(const string &filename, CircularLinklist<Robot *> &lst)`”:

- Initializes the battlefield by reading from a file.
- Parses rows, columns, steps, and robots from the file.
- Creates robots and places them on the battlefield.
- Initializes and places robots in the field.

Methods:

Public:

- “`int getRows()`“: Returns the number of rows.
- “`int getCols()`“: Returns the number of columns.
- “`void setStep(int s)` “: Sets the number of steps.
- “`int getStep()`“: Returns the number of steps.
- “`bool isPositionOccupied(int x, int y)`”: Checks if a position (x, y) is occupied by a robot.
- “`Robot *robotGetKilled(int x, int y)` “: Returns the robot at position (x, y) if there is one.
- “`void upgrade(Robot *&robot, CircularLinklist<Robot *> lst)` “: Upgrades a robot if it has enough kills.
- “`bool boundCheck(int x, int y)` “: Checks if the position (x, y) is within the battlefield bounds.
- “`void initializeField()`“: Initializes the battlefield grid.
- “`void placeRobots()`“: Places robots on the battlefield grid.
- “`void updateField()`“: Updates the battlefield grid.
- “`void displayField()` `const`: Displays the battlefield grid.
- “`void updateAndDisplay()`“: Updates the battlefield grid and displays it.
- “`void UpdatedBattlefield(CircularLinklist<Robot *> lst, Battlefield &battlefield, Queue<Robot *> &q, Robot *&robot)` “: Updates the battlefield and displays it by iterating through the list of robots and performing their actions.

Utility Methods:

Private:

- These methods are utility methods since they help in managing the battlefield and robots.
- `bool isPositionOccupied(int x, int y)`
- `Robot *robotGetKilled(int x, int y)`
- `void initializeField()`
- `void placeRobots()`
- `void updateField()`
- `void displayField() const`
- `void updateAndDisplay()`
- `bool boundCheck(int x, int y)`

Usage:

The "`Battlefield`" class is initialized using a filename and a list of robots "(`CircularLinklist<Robot *>`)". It reads the configuration for the battlefield, including rows, columns, steps, and robot details from the provided file. Robots are created and placed on the battlefield grid accordingly. The class provides methods to manage the dimensions and steps of the battlefield "(`getRows()`, `getCols()`, `setStep(int s)`, `getStep()`)". It allows checking if a specific grid position is occupied with "`isPositionOccupied(int x, int y)`" and retrieves a robot at a specified position with "`robotGetKilled(int x, int y)`". Robots can be upgraded based on their kill count using "`upgrade(Robot *&robot, CircularLinklist<Robot *> lst)`". The method "`boundCheck(int x, int y)`" ensures a position is within the battlefield limits. Additionally, methods like "`initializeField()`," "`placeRobots()`," "`updateField()`", and "`displayField() const`" manage the battlefield grid's state and its display.
The "`updateAndDisplay()`" method updates and displays the battlefield, while "`UpdatedBattlefield(CircularLinklist<Robot *> lst`", "`Battlefield &battlefield`", "`Queue<Robot *>`" "`&q`", "`Robot *&robot)`" iterates through the list of robots, performing their actions, and updates and displays the battlefield.

2. Queue Class

```
template <class Q>
class Queue
{
    template <class N>
    struct Node
    {
        Robot *robot;
        Node<N> *next;
        Node(N x)
        {
            robot = x;
            next = nullptr;
        }
    };

    Node<Q> *front;
    Node<Q> *rear;
    int sz;

public:
    Queue()
    {
        sz = 0;
        front = rear = nullptr;
    }

    Queue(const Queue<Q> &other)
    {
        sz = 0;
        front = rear = nullptr;
        Node<Q> *ptr = other.front;
        while (ptr != nullptr)
        {
            enqueue(ptr->robot);
            ptr = ptr->next;
        `
```

Attributes:

Private:

- `Node<Q> *front`: Pointer to the front node of the queue.
- `Node<Q> *rear`: Pointer to the rear node of the queue.
- `int sz`: Size of the queue.

Constructor:

Public:

- `Queue()`: Initializes an empty queue with size zero and `front` and `rear` pointers set to `nullptr`.
- `Queue(const Queue<Q> &other)`: Copy constructor that creates a new queue by copying elements from another queue.

Methods:

Public:

- `Queue()`: Initializes an empty queue with size zero and `front` and `rear` pointers set to `nullptr`.
- `Queue(const Queue<Q> &other)`: Copy constructor that creates a new queue by copying elements from another queue.
-

Utility Method:

Private:

- **Node Structure:**
  - `template <class N> struct Node`
    - `Robot *robot`: Pointer to a `Robot` object.
    - `Node<N> *next`: Pointer to the next node in the queue.
    - `Node(N x)`: Constructor to initialize a node with a `Robot` pointer and `next` set to `nullptr`.

- **Usage**:
  The `Queue` class is a templated data structure that manages a queue of robots. It uses a private `Node` structure to store pointers to `Robot` objects. The queue is initialized as empty with `front` and `rear` pointers set to `nullptr` and size `sz` set to zero. The `Queue(const Queue<Q> &other)` copy constructor allows for creating a new queue by copying another queue's elements. The `isEmpty()` method checks if the queue is empty, and the `dequeue()` method removes and returns the front element, handling the case when the queue is empty by printing an error message. The `enqueue(Q value)` method adds a new element to the rear of the queue, and the `size()` method returns the current size of the queue. These methods collectively facilitate the management of a queue of robots, allowing for enqueueing, dequeueing, and size tracking.

3. CircularLinkList Class

```
mplate <class Q>
ass CircularLinklist

  template <class N>
  struct Node
  {
      Robot *robot;
      Node *next;
  };

  Node<Q> *head; // head of the list
  int size;      // size of the list
```

Attributes:

Private:

- `Node<Q> *head`: Pointer to the head of the list.
- `int size`: Size of the list.

Constructor and Destructor:

Public:

- `CircularLinklist()`: Initializes an empty list with `head` set to `nullptr` and `size` set to 0.
- `~CircularLinklist()`: Destructor that deletes all nodes and their associated robots, deallocating memory.

Methods:

Public:

- `int getSize()`: Returns the size of the list.
- `bool isEmpty()`: Checks if the list is empty.
- `void push_back(Q value)`: Adds a new node with the given value to the end of the list.
- `void print(Battlefield &field)`: Iterates through the list and calls the `action` method of each robot, passing the battlefield as a parameter.
- `void erase(Q value)`: Removes the node with the specified value from the list.
- `Node<Q> *getHead() const`: Returns the head of the list.
- `void checkDead()`: Iterates through the list and removes robots with `getNumLive() == 0`.
- `void queueRevive(Queue<Q> &q)`: Iterates through the list and moves robots that are not alive to the revive queue, then removes them from the list.

Utility Methods:

Private:

- `Node<Q> *getNode(Q value)`: Creates and returns a new node with the given value.
- `Node<Q> *moveLast()`: Moves to the last node in the list and returns it.

Usage:

The `CircularLinklist` class manages a circular linked list of robots. It initializes an empty list and allows adding new robots with `push_back(Q value)`, checking if the list is empty with `isEmpty()`, and getting the size of the list with `getSize()`. Robots can be iterated over with `print(Battlefield &field)`, which calls the `action` method of each robot. The `erase(Q value)` method removes a specific robot from the list, while `checkDead()` removes robots with zero lives, and `queueRevive(Queue<Q> &q)` moves non-alive robots to a revive queue and removes them from the list. The private methods `getNode(Q value)` and `moveLast()` help in node management by creating new nodes and finding the last node in the list, respectively. The destructor ensures that all nodes and their associated robots are properly deallocated when the list is destroyed.

4. Robot Class

```
class Robot
{
private:
    static int counter;

protected:
    int numkills = 0;
    int numlives = 3;
    int PositionX;
```

Attributes:

- **Private:**
  - `static int counter`: A static counter to keep track of the number of `Robot` instances.
- **Protected:**
  - `int numkills = 0`: Number of kills made by the robot.
  - `int numlives = 3`: Number of lives the robot has.
  - `int PositionX`: X-coordinate position of the robot.
  - `int PositionY`: Y-coordinate position of the robot.
  - `string robotName`: Name of the robot.
  - `string robotType`: Type of the robot.
- **Public:**
  - `int totalKill`: Total number of kills made by the robot.
  - `bool isAlive = true`: Status indicating if the robot is alive.

Constructor:

Public:

- `Robot()`:
  - Increments the static `counter` when a `Robot` object is created.

Methods:

Public:

- `virtual void action(Battlefield &field) = 0`: Pure virtual method to define the robot's action on the battlefield.
- `void setPosition(int x, int y)`: Sets the position of the robot.
- `int getPositionX() const`: Returns the X-coordinate of the robot.
- `int getPositionY() const`: Returns the Y-coordinate of the robot.
- `void setRobotType(const string &t)`: Sets the type of the robot.
- `string getRobotType() const`: Returns the type of the robot.
- `void setRobotName(const string &n)`: Sets the name of the robot.
- `string getRobotName() const`: Returns the name of the robot.
- `void setNumLive(const int &l)`: Sets the number of lives of the robot.
- `int getNumLive() const`: Returns the number of lives of the robot.
- `void setNumKill(int k)`: Sets the number of kills of the robot.
- `int getNumKill() const`: Returns the number of kills of the robot.
- `void setTotalNumKill(int k)`: Sets the total number of kills of the robot.
- `int getTotalNumKill() const`: Returns the total number of kills of the robot.
- `void resetKill()`: Resets the number of kills to zero.
- `virtual void kill()`: Increments the number of kills and total kills.
- `virtual void killed()`: Decrements the number of lives.
- `virtual ~Robot()`: Virtual destructor that decrements the static `counter`.

Utility Method:

These methods help in managing robot attributes and actions.

void setPosition(int x, int y)
- Sets the robot's position to the given X and Y coordinates.

int getPositionX() const
- Returns the robot's current X-coordinate.

int getPositionY() const
- Returns the robot's current Y-coordinate.

void setRobotType(const string &t)
- Sets the robot's type to the given string.

string getRobotType() const
- Returns the robot's type.

void setRobotName(const string &n)
- Sets the robot's name to the given string.

string getRobotName() const
- Returns the robot's name.

void setNumLive(const int &l)
- Sets the robot's number of lives to the given integer.

int getNumLive() const
- Returns the robot's number of lives.

void setNumKill(int k)
- Sets the robot's number of kills to the given integer.

int getNumKill() const
- Returns the robot's number of kills.

void setTotalNumKill(int k)
- Sets the robot's total number of kills to the given integer.

int getTotalNumKill() const
- Returns the robot's total number of kills.

void resetKill()
- Resets the robot's kill count to zero.

virtual void kill()
- Increments the robot's kill count and total kill count by one.

virtual void killed()
- Decreases the robot's number of lives by one.

virtual ~Robot()
- Decrements the static counter when a robot object is destroyed.

- **Usage**:
  The `Robot` class serves as a base class for different types of robots in the battlefield. It includes attributes for tracking the robot's position, type, name, number of lives, and kills. The class provides various getter and setter methods to manage these attributes. Additionally, it includes pure virtual method `action(Battlefield &field)` which must be implemented by derived classes to define specific robot actions. The `kill()` and `killed()` methods update the robot's kill count and lives, respectively. The static counter keeps track of the number of `Robot` instances, and the virtual destructor ensures proper cleanup. The `Robot` class provides a foundation for creating diverse robot types with customized behaviors in the battlefield context.

5. MovingRobot Class

```
class MovingRobot : virtual public Robot
{
protected:
    int moveX;
    int moveY;
    int rows; // Battlefield rows
    int cols; // Battlefield cols
    const vector<Robot *> &robots;

public:
    MovingRobot(int r, int c, const vector<Robot *> &robots) : rows(r), cols(c),
robots(robots) {}
    virtual void move(Battlefield &field) = 0;
    virtual ~MovingRobot() {}
```

Attributes:

- **Protected:**

  - `int moveX`: The x-coordinate movement direction.
  - `int moveY`: The y-coordinate movement direction.
  - `int rows`: Number of rows in the battlefield.
  - `int cols`: Number of columns in the battlefield.
  - `const vector<Robot *> &robots`: A reference to a vector of pointers to `Robot` objects.

Constructor:

Public:

- `MovingRobot(int r, int c, const vector<Robot *> &robots)`: Initializes the `rows`, `cols`, and `robots` attributes.

Methods:

Public:

- `virtual void move(Battlefield &field) = 0`: A pure virtual function to be implemented by derived classes for moving the robot on the battlefield.
- `virtual ~MovingRobot() {}`: A virtual destructor.

- `string moving(Battlefield &field)`: Handles the movement of the robot on the battlefield, ensuring the new position is within bounds and not occupied, then updates the position.

Utility Method:

- **Protected:**
  - These are utility methods since they assist in the main functionality of the class but aren't public interfaces.

- **Usage**:

The `Robot` class serves as a base class for different types of robots in the battlefield. It includes attributes for tracking the robot's position, type, name, number of lives, and kills. The class provides various getter and setter methods to manage these attributes. Additionally, it includes pure virtual method `action(Battlefield &field)` which must be implemented by derived classes to define specific robot actions. The `kill()` and `killed()` methods update the robot's kill count and lives, respectively. The static counter keeps track of the number of `Robot` instances, and the virtual destructor ensures proper cleanup. The `Robot` class provides a foundation for creating diverse robot types with customized behaviors in the battlefield context.

6. ShootingRobot Class

```
class MovingRobot : virtual public Robot
{
protected:
    int moveX;
    int moveY;
    int rows; // Battlefield rows
    int cols; // Battlefield cols
    const vector<Robot *> &robots;

public:
    MovingRobot(int r, int c, const vector<Robot *> &robots) : rows(r), cols(c),
robots(robots) {}
    virtual void move(Battlefield &field) = 0;
    virtual ~MovingRobot() {}
```

Attributes:

- **Protected:**
  - `int moveX`: The x-coordinate movement direction.
  - `int moveY`: The y-coordinate movement direction.
  - `int rows`: Number of rows in the battlefield.
  - `int cols`: Number of columns in the battlefield.
  - `const vector<Robot *> &robots`: A reference to a vector of pointers to `Robot` objects.

Constructor:

Public:

- `ShootingRobot(int r, int c, const vector<Robot *> &robots) : rows(r), cols(c), robots(robots) {}`:
  - Initializes a `ShootingRobot` object with specified rows (`r`), columns (`c`), and a reference to a vector of robots (`robots`).

Methods:

Public Virtual Methods:

- `virtual void fire(Battlefield &field) = 0;`:

- Pure virtual method intended to be overridden by derived classes.
- Represents the action of firing or shooting on the battlefield.

Destructor:

- `virtual ~ShootingRobot() {}`:
  - Virtual destructor for proper cleanup in derived classes.

Utility Method:

- Typically, utility methods would include functions that assist in calculations, validations, or other operations related to shooting behavior or robot interaction on the battlefield.

- **Usage**:

Initialization and Configuration:
- Objects of ShootingRobot are initialized with specific dimensions (r, c) of the battlefield and a reference to the vector of robots (robots).

Derived Class Implementation:
- Derived classes from ShootingRobot would override the pure virtual method fire(Battlefield &field) to implement specific shooting behaviors.

Robot Interaction:
- The ShootingRobot class interacts with other robots stored in the robots vector, likely for targeting and shooting actions, leveraging the shared context of the battlefield dimensions and robot positions.

7. SeeingRobot Class

```
class SeeingRobot : virtual public Robot
{
protected:
    int seeX;
    int seeY;
    int rows; // Battlefield rows
    int cols; // Battlefield cols
    Robot *detectedRobot;
    vector<vector<int>> nearby;
    const vector<Robot *> &robots;

public:
    SeeingRobot(int r, int c, const vector<Robot *> &robots)
        : rows(r), cols(c), robots(robots), detectedRobot(nullptr) {}

    void setDetectedRobot(Robot *robot) { detectedRobot = robot; }
    Robot *getDetectedRobot() const { return detectedRobot; }
```

Attributes:

- **Protected:**
  - `int seeX`: X-coordinate for the robot's vision.
  - `int seeY`: Y-coordinate for the robot's vision.
  - `int rows`: Number of rows in the battlefield.
  - `int cols`: Number of columns in the battlefield.
  - `Robot *detectedRobot`: Pointer to the detected enemy robot.
  - `vector<vector<int>> nearby`: Vector of vectors storing nearby positions relative to the robot.
  - `const vector<Robot *> &robots`: Reference to a vector storing pointers to all robots in the battlefield.

Constructor:

Public:

- `SeeingRobot(int r, int c, const vector<Robot *> &robots)`: Initializes the `rows`, `cols`, and `robots` attributes with provided values. Sets `detectedRobot` to `nullptr`.

Methods:

Public:

- `SeeingRobot(int r, int c, const vector<Robot *> &robots)`: Initializes the `rows`, `cols`, and `robots` attributes with provided values. Sets `detectedRobot` to `nullptr`.

Utility Method:

Protected:

- None specified explicitly, but `looking()` serves utility purposes by managing vision and enemy detection.
-

Usage:

- **Initialization and Configuration:**
  - `SeeingRobot` is typically initialized with the number of rows (`r`), columns (`c`), and a reference to a vector (`robots`) containing pointers to all robots in the battlefield.
  - The constructor initializes these attributes and sets `detectedRobot` to `nullptr`.
- **Vision and Enemy Detection:**
  - `looking()` is invoked to scan nearby positions around the robot.
  - It calculates positions within a 3x3 grid centered on the robot's current position.
  - Nearby positions are displayed.
  - It checks each nearby position to see if an enemy robot (`detectedRobot`) is present.
  - If an enemy is detected, a message is printed indicating the detection.

Additional Notes

- `SeeingRobot` inherits virtually from `Robot`, implying it shares a common base class with other types of robots in your system.
- The class is designed to handle vision-related functionalities and enemy detection within a specified battlefield grid.

This class structure facilitates specialized behavior for robots that rely on vision capabilities to interact with their environment, particularly in detecting and responding to nearby enemy robots.

8. SeeingRobot Class

```
class SeeingRobot : virtual public Robot
{
protected:
    int seeX;
    int seeY;
    int rows; // Battlefield rows
    int cols; // Battlefield cols
    Robot *detectedRobot;
    vector<vector<int>> nearby;
    const vector<Robot *> &robots;

public:
    SeeingRobot(int r, int c, const vector<Robot *> &robots)
        : rows(r), cols(c), robots(robots), detectedRobot(nullptr) {}

    void setDetectedRobot(Robot *robot) { detectedRobot = robot; }
    Robot *getDetectedRobot() const { return detectedRobot; }

    virtual bool look() = 0;
    virtual ~SeeingRobot() {}
```

Attributes:

- **Protected:**
  - `int seeX`: X-coordinate for the robot's vision.
  - `int seeY`: Y-coordinate for the robot's vision.
  - `int rows`: Number of rows in the battlefield.
  - `int cols`: Number of columns in the battlefield.
  - `Robot *detectedRobot`: Pointer to the detected enemy robot.
  - `vector<vector<int>> nearby`: Vector of vectors storing nearby positions relative to the robot.
  - `const vector<Robot *> &robots`: Reference to a vector storing pointers to all robots in the battlefield.

Constructor:

Public:

- `SeeingRobot(int r, int c, const vector<Robot *> &robots)`: Initializes the `rows`, `cols`, and `robots` attributes with provided values. Sets `detectedRobot` to `nullptr`.

Methods:

Public:

- `SeeingRobot(int r, int c, const vector<Robot *> &robots)`: Initializes the `rows`, `cols`, and `robots` attributes with provided values. Sets `detectedRobot` to `nullptr`.

Utility Method:

Protected:

- None specified explicitly, but `looking()` serves utility purposes by managing vision and enemy detection.
-

Usage:

- **Initialization and Configuration:**
  - `SeeingRobot` is typically initialized with the number of rows (`r`), columns (`c`), and a reference to a vector (`robots`) containing pointers to all robots in the battlefield.
  - The constructor initializes these attributes and sets `detectedRobot` to `nullptr`.
- **Vision and Enemy Detection:**
  - `looking()` is invoked to scan nearby positions around the robot.
  - It calculates positions within a 3x3 grid centered on the robot's current position.
  - Nearby positions are displayed.
  - It checks each nearby position to see if an enemy robot (`detectedRobot`) is present.
  - If an enemy is detected, a message is printed indicating the detection.

Additional Notes

- `SeeingRobot` inherits virtually from `Robot`, implying it shares a common base class with other types of robots in your system.
- The class is designed to handle vision-related functionalities and enemy detection within a specified battlefield grid.

This class structure facilitates specialized behavior for robots that rely on vision capabilities to interact with their environment, particularly in detecting and responding to nearby enemy robots.

9. RoboCop Class

```
class RoboCop : virtual public MovingRobot, virtual public ShootingRobot, virtual
public SeeingRobot
{
public:
    RoboCop(int r, int c, const vector<Robot *> &robots)
        : MovingRobot(r, c, robots), ShootingRobot(r, c, robots), SeeingRobot(r, c,
robots) {}
    virtual bool look() override { return looking(); }
    virtual void move(Battlefield &field) override { cout << moving(field) << endl; }
    virtual void fire(Battlefield &field) override
    {
        bool inRange = false;
        while (!inRange)
        {
            int randomX = rand() % 10;
            int randomY = rand() % 10;

            int negativeNum = rand() % 4;
            switch (negativeNum)
            {
            case 0:
                randomX = (randomX * -1);
                break;
```

Attributes:

- **Inherited:**
  - Inherits attributes from `MovingRobot`, `ShootingRobot`, and `SeeingRobot` classes.
  - These attributes are typically related to position (x, y), robot type and name, alive status, number of kills, and possibly others depending on the base classes (`Robot` and its derived classes).

Constructor:

Public:

- `RoboCop(int r, int c, const vector<Robot *> &robots)`
  - Initializes a `RoboCop` object with given dimensions (`r`, `c`) and a vector of `Robot` pointers (`robots`).
  - Calls constructors of virtual base classes `MovingRobot`, `ShootingRobot`, and `SeeingRobot` with the same parameters (`r`, `c`, `robots`).

Methods:

Public and Overridden:

- `bool look() override`: Implements vision-related behavior, overriding the `look()` method from `SeeingRobot`.
- `void move(Battlefield &field) override`: Implements movement behavior, overriding the `move()` method from `MovingRobot`.
- `void fire(Battlefield &field) override`: Implements shooting behavior, overriding the `fire()` method from `ShootingRobot`.
- `void action(Battlefield &field) override`: Implements overall action behavior, combining look, move, and fire actions, and handles status updates like kills and lives.

Utility Method:

Private:

- These methods are utility methods specific to `RoboCop` and its actions, aiding in shooting randomly and checking if a position is in range and valid for firing.

Usage:

- **Initialization and Configuration:**
  - A `RoboCop` object is initialized with specific dimensions (`r`, `c`) and a vector of `Robot` pointers (`robots`), typically obtained from the `Battlefield` or other game setup.
  - The constructor initializes the object by invoking constructors of its virtual base classes.
- **Behavior and Actions:**
  - The `look()` method utilizes vision capabilities inherited from `SeeingRobot`.
  - The `move()` method manages movement inherited from `MovingRobot`, interacting with the `Battlefield` to update the robot's position.
  - The `fire()` method simulates shooting behavior inherited from `ShootingRobot`, determining a random target and checking if it hits and kills another robot on the battlefield.
  - The `action()` method orchestrates these behaviors in sequence (`look`, `move`, `fire`, `fire`, `fire`), displaying updates on kills and lives during its execution.

This class encapsulates the behavior of a `RoboCop` robot in a simulated battlefield environment, demonstrating its abilities derived from its multiple inherited robot traits (`MovingRobot`, `ShootingRobot`, `SeeingRobot`).

10. TerminatorBot Class

```
class Terminator : virtual public MovingRobot, virtual public SeeingRobot, virtual
public SteppingRobot
{
protected:
    Robot *nearby;

public:
    Terminator(int r, int c, const vector<Robot *> &robots)
        : MovingRobot(r, c, robots), SeeingRobot(r, c, robots), nearby(nullptr) {}
    virtual bool look() override
    {
        bool detectedRobot = looking();
        nearby = getDetectedRobot();
        return detectedRobot;
    }
    virtual void move(Battlefield &field) override { cout << moving(field) << endl; }
    virtual void step(Battlefield &field) override
    {
        stepX = nearby->getPositionX();
        stepY = nearby->getPositionY();

        cout << getRobotName() << " (" + getRobotType() + ") stepped at (" << stepX <<
", " << stepY << ")" << endl;
        if (field.robotGetKilled(stepX, stepY) != nullptr)
        {
```

Attributes:

- **Protected:**
  - `Robot *nearby`: Pointer to a `Robot` object representing a nearby detected robot.

Constructor

Public:

- `Terminator(int r, int c, const vector<Robot *> &robots)`: Constructor that initializes `Terminator` as a subclass of `MovingRobot`, `SeeingRobot`, and `SteppingRobot`.
  - Calls constructors of base classes `MovingRobot`, `SeeingRobot`, and `SteppingRobot` with `r`, `c`, and `robots`.
  - Initializes `nearby` to `nullptr`.

Methods:

Public and Virtual:

- `bool look() override`: Overrides `SeeingRobot::look()`.
  - Calls `looking()` to detect nearby robots.
  - Updates `nearby` with the detected robot.
  - Returns whether a robot is detected.
- `void move(Battlefield &field) override`: Overrides `MovingRobot::move()`.
  - Moves the Terminator on the battlefield using `moving(field)` and prints the movement result.
- `void step(Battlefield &field) override`: Overrides `SteppingRobot::step()`.
  - Determines the step coordinates (`stepX`, `stepY`) based on the detected `nearby` robot.
  - Checks if a robot is killed at the step coordinates using `field.robotGetKilled(stepX, stepY)`.

- Updates the killed robot's status and counts kills.
- `void action(Battlefield &field) override`: Overrides `Robot::action()`.
  - Checks if the Terminator is alive.
  - Prints current position and performs either `step()` or `move()` based on detection (`look()`).
  - Prints updated kill and live counts after action.

Utility Method:

- **Protected:**
  - These methods are used internally by the class for specific functionalities.
  - `bool look()`
  - `void move(Battlefield &field)`
  - `void step(Battlefield &field)`
  - `void action(Battlefield &field).`

Usage:

- **Initialization and Configuration:**
  - `Terminator` instances are initialized with dimensions `r` and `c` and a vector of existing robots (`robots`).
  - The constructor sets up `Terminator` to inherit behaviors from `MovingRobot`, `SeeingRobot`, and `SteppingRobot`.
- **Robot Actions:**
  - `action(Battlefield &field)` method orchestrates the Terminator's behavior:
    - Checks if the Terminator is alive.
    - Displays current position.
    - Uses `look()` to detect nearby robots and decides to `step()` or `move()` based on detection results.
    - Updates kill and live counts after performing the action.

This class effectively utilizes multiple inheritance to combine behaviors from `MovingRobot`, `SeeingRobot`, and `SteppingRobot` into the specialized behavior of a `Terminator` robot on a battlefield scenario.

11. TerminatorRoboCop Class

```
class TerminatorRobocop : virtual public RoboCop, virtual public Terminator
{
public:
    TerminatorRobocop(int r, int c, const vector<Robot *> &robots)
        : MovingRobot(r, c, robots), ShootingRobot(r, c, robots), SeeingRobot(r, c,
robots),
          RoboCop(r, c, robots), Terminator(r, c, robots) {}
    virtual void move(Battlefield &field) override
    {
        Terminator::move(field);
    }
    virtual bool look() override
    {
        return Terminator::look();
    }
```

Attributes:

- **Inheritance:**
  - Inherits attributes from both `RoboCop` and `Terminator` classes.

Constructor

Public:

- `TerminatorRobocop(int r, int c, const vector<Robot *> &robots)`: Initializes a `TerminatorRobocop` object.
  - Calls constructors of `MovingRobot`, `ShootingRobot`, `SeeingRobot`, `RoboCop`, and `Terminator` classes.
  - Passes dimensions (`r`, `c`) and a vector of robots (`robots`) to these constructors.

Methods:

Public:

- `void move(Battlefield &field) override`: Overrides `move` method from `Terminator` class.
  - Delegates `move` operation to `Terminator` class implementation.
- `bool look() override`: Overrides `look` method from `Terminator` class.
  - Delegates `look` operation to `Terminator` class implementation.

- `void action(Battlefield &field)`: Executes actions specific to `TerminatorRobocop`.
    - Checks if the robot is alive; if not, displays a message and returns.
    - Displays robot's position and performs actions using methods from `Terminator` and `RoboCop` classes.
    - Prints current kill count (`getNumKill()`) and lives (`getNumLive()`).

Usage:

- **Initialization and Configuration:**
    - `TerminatorRobocop` is initialized with dimensions (`r`, `c`) and a vector of robots (`robots`).
    - It inherits functionality from `MovingRobot`, `ShootingRobot`, `SeeingRobot`, `RoboCop`, and `Terminator` classes, ensuring it can move, shoot, and perform other robot-specific actions.
    - The constructor initializes the robot with the specified dimensions and links it to existing robots in the vector.
- **Robot Actions:**
    - `move(Battlefield &field)` and `look()` methods override behaviors defined in the `Terminator` class.
    - `action(Battlefield &field)` method orchestrates actions specific to `TerminatorRobocop`, combining behaviors from `Terminator` and `RoboCop` classes.
    - It checks if the robot is alive, reports its position, and executes actions such as moving and firing.

This class `TerminatorRobocop` effectively combines characteristics from both `RoboCop` and `Terminator` classes, inheriting their functionalities and defining specific behaviors through method overrides and additional actions.

12. BlueThunder Class

```
class BlueThunder : virtual public ShootingRobot
{
protected:
    int turn = 0;

public:
    BlueThunder(int r, int c, const vector<Robot *> &robots) : ShootingRobot(r, c,
robots) {}
    virtual void fire(Battlefield &field) override
    {
        bool inRange = false;
        while (!inRange)
        {
            int clockwise = turn % 8;
            switch (clockwise)
            {
            case 0: // up
                shootX = -1;
                shootY = 0;
                break;
            case 1: // up-right
                shootX = -1;
                shootY = 1;
                break;
            case 2: // right
                shootX = 0;
                shootY = 1;
                break;
            case 3: // down right
                shootX = 1;
                shootY = 1;
                break;
```

Attributes:

- **Protected:**
  - `int turn`: Tracks the current turn or direction for shooting.

Constructor

Public:

- `BlueThunder(int r, int c, const vector<Robot *> &robots) : ShootingRobot(r, c, robots) {}`
  - Initializes a `BlueThunder` object, inheriting from `ShootingRobot`, with specified rows (`r`), columns (`c`), and a vector of `Robot` pointers (`robots`).

Methods:

Public:

- `BlueThunder(int r, int c, const vector<Robot *> &robots) : ShootingRobot(r, c, robots) {}`
  - Initializes a `BlueThunder` object, inheriting from `ShootingRobot`, with specified rows (`r`), columns (`c`), and a vector of `Robot` pointers (`robots`).
  -

Utility Methods

- **Indirectly utilizing:**
  - `bool boundCheck(int x, int y)` from `Battlefield` class to verify shot positions.
  - `Robot *robotGetKilled(int x, int y)` from `Battlefield` class to check if a robot was hit by the shot.

Usage:

- **Initialization and Configuration:**
  - Creates instances of `BlueThunder` with specific rows, columns, and a vector of robots.
- **Battlefield Interaction:**
  - `fire(Battlefield &field)` method is called during the `action(Battlefield &field)` sequence, where `BlueThunder` attempts to shoot at nearby robots within its range.
  - Displays messages about shooting actions and potential kills based on the battlefield state.

This class extends `ShootingRobot` and adds specific behaviors for the `BlueThunder` robot type, focusing on shooting in multiple directions and interacting with the battlefield environment to eliminate enemy robots.

13. Madbot Class

```
class Madbot : virtual public BlueThunder
{
public:
    Madbot(int r, int c, const vector<Robot *> &robots)
        : BlueThunder(r, c, robots), ShootingRobot(r, c, robots) {}
    virtual void fire(Battlefield &field) override
    {
        bool inRange = false;
        while (!inRange)
        {
            int direction = rand() % 8;
            switch (direction)
            {
            case 0: // up
                shootX = -1;
                shootY = 0;
                break;
            case 1: // up-right
                shootX = -1;
                shootY = 1;
                break;
            case 2: // right
                shootX = 0;
                shootY = 1;
                break;
            case 3: // down right
                shootX = 1;
                shootY = 1;
                break;
            case 4: // down
                shootX = 1;
                shootY = 0;
                break;
```

Attributes:

- **Inherited from BlueThunder (and possibly ShootingRobot, assuming common attributes):**
    - These attributes are not explicitly defined in the provided class but would include any inherited attributes from `BlueThunder` and `ShootingRobot`, such as position (`getPositionX()`, `getPositionY()`), robot name (`getRobotName()`), robot type (`getRobotType()`), and methods like `isAlive`, `killed()`, `kill()`, `getNumKill()`, `getNumLive()`, etc.

Constructor

Public:

- `Madbot(int r, int c, const vector<Robot *> &robots)`: Constructor that initializes `Madbot` as a subclass of `BlueThunder` and `ShootingRobot`, passing rows (`r`), columns (`c`), and a vector of robots (`robots`) to its parent constructors.

Methods:

- **Public:**
    - `virtual void fire(Battlefield &field) override`: Implements the `fire` method from the `ShootingRobot` interface.
        - Randomly selects a direction to shoot.
        - Checks if the shot is within the battlefield bounds (`field.boundCheck()`).
        - Prints a message indicating where the shot was fired.
        - Checks if a robot was hit (`field.robotGetKilled()`), updates its status, and records the kill.

• `virtual void action(Battlefield &field) override`: Overrides the `action` method from `Robot`.

- Checks if the `Madbot` is alive (`isAlive`).
- Prints current position and type of the `Madbot`.
- Calls `fire(field)` to perform shooting action.
- Prints current kill and live status of the `Madbot`.
    -

Utility Methods

- However, utility methods inherited from `BlueThunder` and `ShootingRobot` include those related to positioning, state management (`isAlive`, `getNumKill()`, `getNumLive()`), and possibly others.

Usage:

- **Initialization and Configuration:**
  - The `Madbot` class is initialized by passing the number of rows (`r`), columns (`c`), and a vector of robots (`robots`) to its constructor.
  - Inherits behavior from `BlueThunder` and `ShootingRobot`, likely configuring position and robot-specific attributes.
- **Battlefield Operations:**
  - The `fire(Battlefield &field)` method selects a random direction to fire and checks for hits within battlefield bounds.
  - The `action(Battlefield &field)` method checks if the `Madbot` is alive, performs firing action (`fire(field)`), and updates its status.

This breakdown outlines how the `Madbot` class extends functionality from `BlueThunder` and `ShootingRobot`, implementing specific behaviors such as firing and general actions within the context of a battlefield simulation.

14. RoboTank Class

```
class RoboTank : virtual public Madbot
{
public:
    RoboTank(int r, int c, const vector<Robot *> &robots)
        : Madbot(r, c, robots), BlueThunder(r, c, robots), ShootingRobot(r, c, robots)
{}
    virtual void fire(Battlefield &field) override
    {
        bool inRange = false;
        while (!inRange)
        {
            int randomX = rand() % (field.getRows());
            int randomY = rand() % (field.getCols());

            int negativeNum = rand() % 4;
            switch (negativeNum)
            {
            case 0:
                randomX = (randomX * -1);
                break;
            case 1:
                randomY = (randomY * -1);
                break;
            case 2:
                randomX = (randomX * -1);
                randomY = (randomY * -1);
                break;
            }
            shootX = randomX + getPositionX();
            shootY = randomY + getPositionY();
            inRange = ((abs(randomX) + abs(randomY) != 0) && field.boundCheck(shootX,
shootY));
        }
```

Attributes:

- **Inherited Attributes:** Inherits attributes from `Madbot`, `BlueThunder`, and `ShootingRobot`, assuming these classes have their own attributes.

Constructor:

Public:

- `RoboTank(int r, int c, const vector<Robot *> &robots)`:
  - Initializes `RoboTank` with dimensions `r` (rows) and `c` (columns), and a vector of `Robot` pointers (`robots`).
  - Calls constructors of `Madbot`, `BlueThunder`, and `ShootingRobot` classes (assuming they are base classes) to initialize their parts of the object.

Methods:

- **Public:**

- `RoboTank(int r, int c, const vector<Robot *> &robots)`:
  - Initializes `RoboTank` with dimensions `r` (rows) and `c` (columns), and a vector of `Robot` pointers (`robots`).
  - Calls constructors of `Madbot`, `BlueThunder`, and `ShootingRobot` classes (assuming they are base classes) to initialize their parts of the object.

Utility Methods

- However, utility methods inherited from `BlueThunder` and `ShootingRobot` include those related to positioning, state management (`isAlive`, `getNumKill()`, `getNumLive()`), and possibly others.

Usage:

- **Initialization and Configuration:**

  - `RoboTank` is initialized with specific dimensions (`r` and `c`) and a vector of robots (`robots`).
  - It inherits functionality from `Madbot`, `BlueThunder`, and `ShootingRobot`, utilizing their constructors to set up its own state.

- **Action Execution:**

  - `action(Battlefield &field)` method is called when it's the `RoboTank`'s turn to perform actions on the battlefield.

- It checks if the robot is alive, displays its current position, fires at a random target, checks for kills, and updates its kill and live statistics.

15. UltimateRobot Class

```
class UltimateRobot : public TerminatorRobocop, public RoboTank
{
public:
    UltimateRobot(int r, int c, const vector<Robot *> &robots)
        : MovingRobot(r, c, robots), ShootingRobot(r, c, robots), SeeingRobot(r, c,
robots),
          RoboCop(r, c, robots), Terminator(r, c, robots), TerminatorRobocop(r, c,
robots),
          Madbot(r, c, robots), BlueThunder(r, c, robots), RoboTank(r, c, robots) {}

    virtual void fire(Battlefield &field) override
    {
        RoboTank::fire(field);
    }
    void action(Battlefield &field)
    {
        if (!isAlive)
        {
            cout << endl;
            cout << getRobotName() + " (" + getRobotType() + ") is get killed, wait
for revive" << endl
                 << endl;
            cout << getRobotName() << " kills= " << getNumKill() << ", lives= " <<
getNumLive() << endl
                 << endl;
            return;
        }
        cout << endl;
        cout << getRobotName() + " (" + getRobotType() + ") at position (" <<
getPositionX() << "," << getPositionY() << ")" << endl;
        TerminatorRobocop::move(field);
        RoboTank::action(field);
        RoboTank::action(field);
        RoboTank::action(field);
        cout << endl;
        cout << getRobotName() << " kills= " << getNumKill() << ", lives= " <<
```

Attributes:

Inherited:

- Inherits attributes from `TerminatorRobocop` and `RoboTank` classes.

Constructor:

Public:

- `UltimateRobot(int r, int c, const vector<Robot *> &robots)`: Initializes an `UltimateRobot` object with positions `r` and `c` on the battlefield and a vector of `Robot` pointers `robots`.
    - Initializes inherited attributes from `MovingRobot`, `ShootingRobot`, `SeeingRobot`, `RoboCop`, `Terminator`, `TerminatorRobocop`, `Madbot`, `BlueThunder`, and `RoboTank` classes.

Methods:

Public:

- `virtual void fire(Battlefield &field) override`: Overrides the `fire` method from `ShootingRobot` to delegate to `RoboTank`'s `fire` method.
- `void action(Battlefield &field)`: Executes actions specific to the `UltimateRobot`:
    - Checks if the robot is alive.
    - Moves using `TerminatorRobocop::move(field)`.
    - Performs three actions using `RoboTank::action(field)`.
    - Prints the current number of kills and lives.

Usage:

- • **Initialization and Configuration:**
    - An `UltimateRobot` object is initialized with specific positions and a vector of existing robots.
    - Inherits behaviors and attributes from multiple base classes (`TerminatorRobocop` and `RoboTank`).
- **Behavior and Actions:**
    - `fire(Battlefield &field)` method allows the `UltimateRobot` to perform shooting actions via `RoboTank`.
    - `action(Battlefield &field)` method orchestrates a series of movements and actions specific to the `UltimateRobot`, leveraging methods from inherited classes like `TerminatorRobocop` for movement and `RoboTank` for actions.

This class represents an advanced robot type (`UltimateRobot`) inheriting functionalities from `TerminatorRobocop` and `RoboTank`, showcasing specialized behavior in movement and action execution on a battlefield (`Battlefield` object).

MAIN FUNCTION

```
int main()
{
    srand(time(0)); // Seed for random position generation

    Robot *robot;
    Queue<Robot *> q;
    CircularLinklist<Robot *> lst;
    Battlefield battlefield("Robot.txt", lst);
    cout << endl;
    battlefield.displayField();

    for (int i = 0; i < battlefield.getStep(); i++)
    {
        cout << endl;
        cout << "_____Turn " << i + 1 << "_____" <<
endl;
        cout << endl;

        battlefield.UpdatedBattlefield(lst, battlefield, q, robot);

        // check dead and queue revive
        lst.checkDead();
        lst.queueRevive(q);
        // check upgrade
        battlefield.upgrade(robot, lst);
    }

    // lst.print();
    // cout << "Size of list: " << lst.getSize() << endl;
    // lst.delete_front();
    // cout << ":>----------" << endl;
    // lst.print();
    // cout << "Size of list: " << lst.getSize() << endl;
    // cout << ":>----------" << endl;
    // lst.delete_back();
    // lst.print();
    // cout << ":>----------" << endl;
    // cout << "Size of list: " << lst.getSize() << endl;
    // cout << "----------" << endl;

    return 0;
```

Usage:

The `main` function initializes a simulation of a battlefield using robots read from a file named "Robot.txt". Here's a short usage description:

The program starts by seeding the random number generator for position generation. It then initializes a `Battlefield` object named `battlefield` using data from "Robot.txt", which includes setting up the battlefield grid and placing robots. The initial state of the battlefield is displayed using `battlefield.displayField()`.

Next, for each turn up to the number of steps defined by `battlefield.getStep()`, the program executes the following:

- It displays the current turn number.
- The battlefield is updated and displayed using `battlefield.UpdatedBattlefield(lst, battlefield, q, robot)`, where each robot in the circular linked list `lst` performs its action.
- After each turn, the program checks for dead robots and queues them for revival using `lst.checkDead()` and `lst.queueRevive(q)`.
- It also checks if any robots are eligible for upgrades based on their kill count using `battlefield.upgrade(robot, lst)`.

The program concludes after all turns are completed, providing an interactive simulation of robots interacting on a battlefield, managing their actions, deaths, revivals, and upgrades dynamically over multiple turns.

Simulation Process

1. Initialization

FileReader Initialization

The robot.txt file contains the specifications, which are read by the FileReader class to start the simulation. The number of simulation steps, the number of robots, the dimensions of the battlefield, and the specifics of each robot that needs to be deployed are all extracted

```
ifstream file(filename);
       if (!file)
       {
           cerr << "Error opening file: " << filename << endl;
           exit(1);
       }
```

-
-

- **File Handling:** It opens a file (`filename`) to read initialization data for the battlefield configuration and robots.

- **Reading Configuration:**

    - Reads the dimensions (`rows` and `cols`) of the battlefield.
    - Reads the number of steps (`step`) for which the simulation will run.
    - Reads the number of robots (`robotCount`) that will be initialized on the battlefield.

Field Initialization:

Initializes the game field (`field`) with empty strings to represent each cell on the battlefield.

2. Robot Deployment

- Iterates over each robot specified in the file.
- Parses type, name, and initial position (`x` and `y`) of each robot.
- Handles random positions if specified ('random' for `x_str` or `y_str`).
- Checks if the position is occupied; if so, assigns a random position until an unoccupied one is found.
- Creates instances of each robot type (`RoboCop`, `Terminator`, etc.), sets their initial attributes, and adds them to a list (`robots`) and a circular linked list (`lst`).

3. Simulation Execution

- Closes the file after reading all necessary data.
- Calls `initializeField()` to set up the empty game field.

- Calls `placeRobots()` to position each robot on the initialized field.

4. Outcome Evaluation

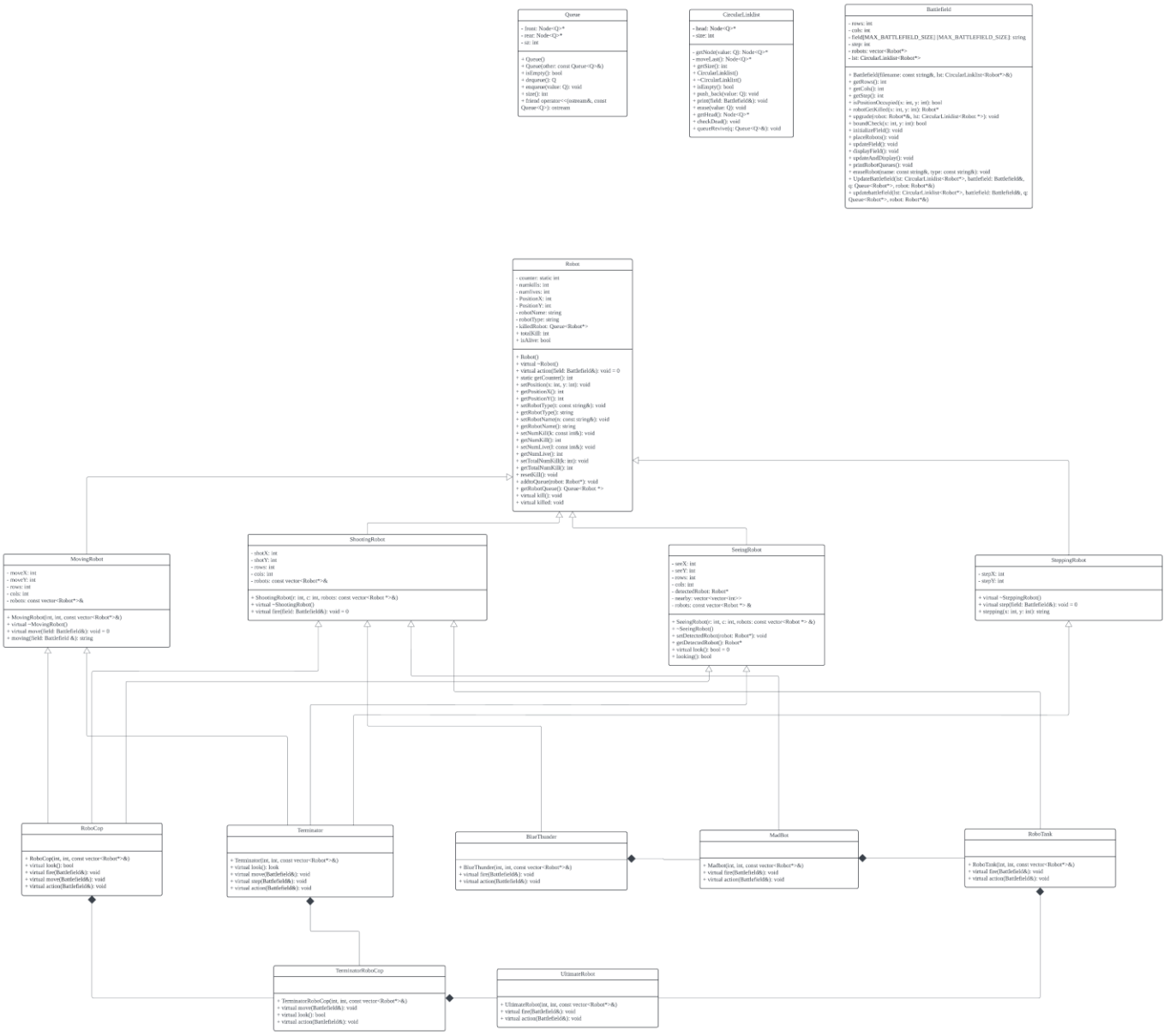
Battlefield Display

After the simulation completes, the state of the battlefield can be displayed to visualize the outcome of robot actions. This helps evaluate the effectiveness and interactions of different robot types during the simulation.

```
void displayField() const
```

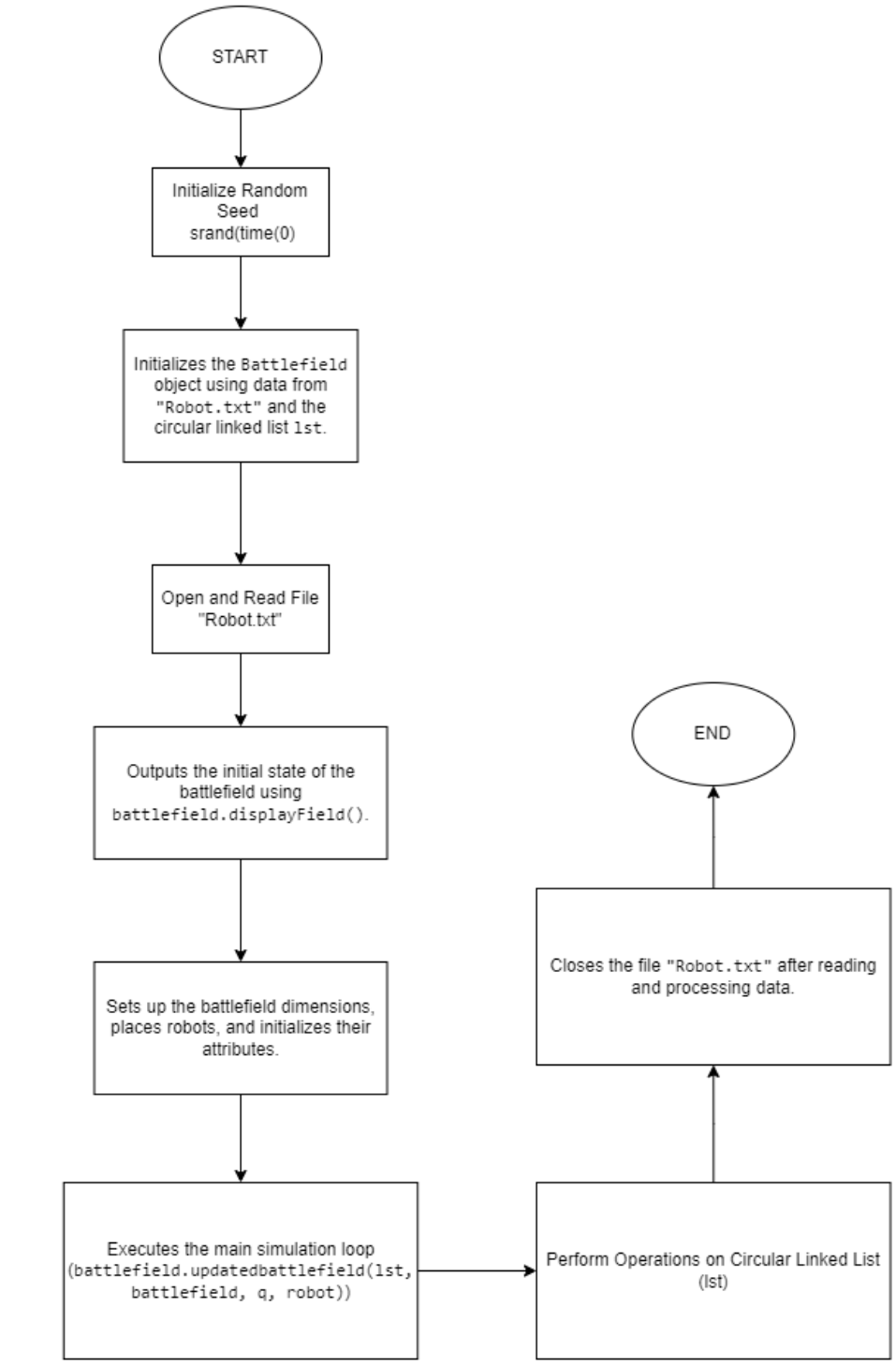
- **Visualization**: The display method of Battlefield can show the current positions and statuses of all robots on the grid, reflecting any changes resulting from their actions.

Class diagram for OOP implementations

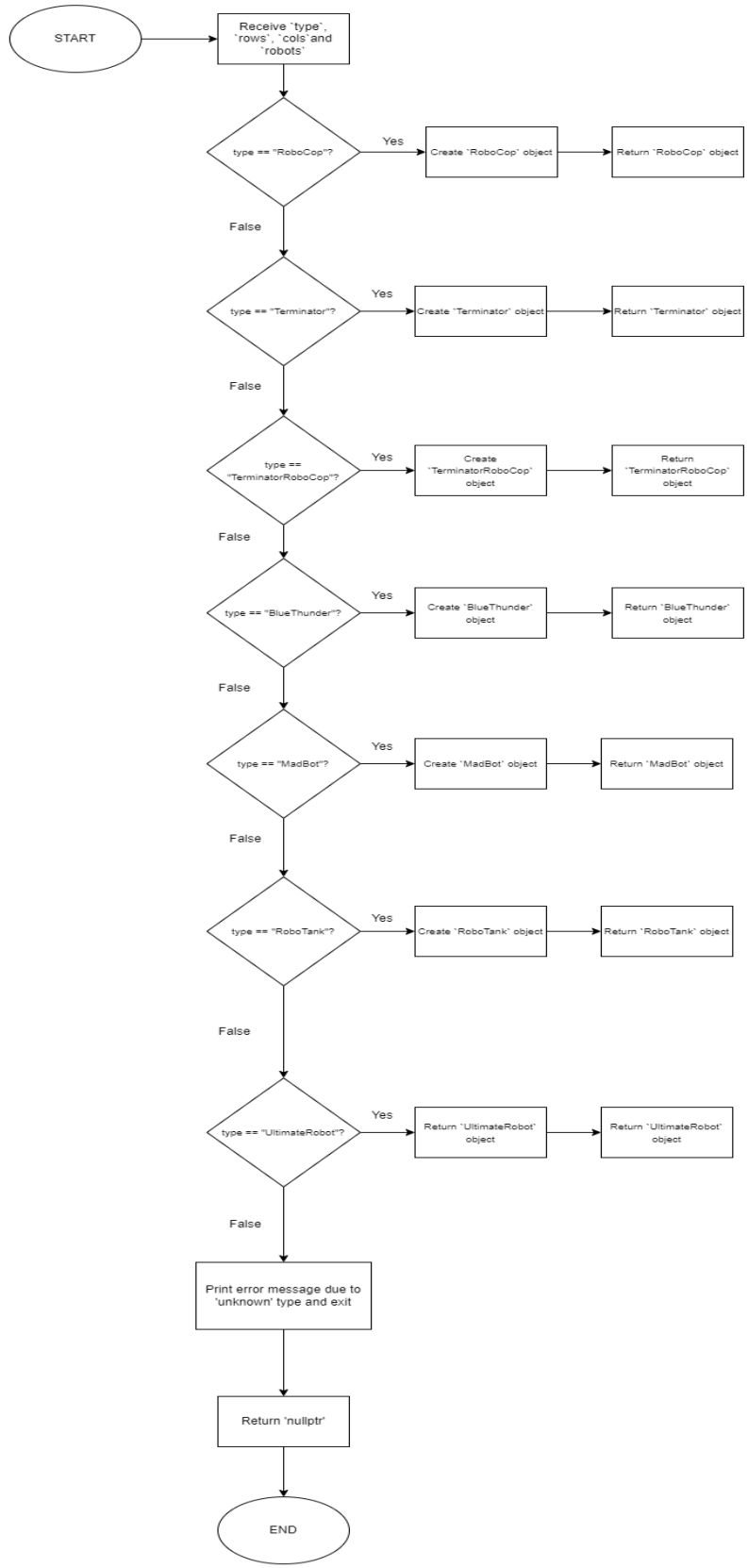

Queue
CircularLinkIst
Battlefield
Robot
MovingRobot
ShootingRobot
SeeingRobot
SteppingRobot
RoboCop
Terminator
BlueThunder
MadBot
RoboTank
TerminatorRoboCop
UltimateRobot

Flowchart

MAIN


START
Initialize Random Seed srand(time(0)
Initializes the Battlefield object using data from "Robot.txt" and the circular linked list lst.
Open and Read File "Robot.txt"
Outputs the initial state of the battlefield using battlefield.displayField().
Sets up the battlefield dimensions, places robots, and initializes their attributes.
Executes the main simulation loop (battlefield.updatedbattlefield(lst, battlefield, q, robot))
Perform Operations on Circular Linked List (lst)
Closes the file "Robot.txt" after reading and processing data.
END

GET ROBOT //ROBOT FACTORY



Conclusion

The Robot War Simulation programme serves as an excellent example of object-oriented programming concepts and appropriate data structure utilisation for dynamic scenario simulation. Through the process of initialization, deployment, simulation, and outcome evaluation, the programme offers a thorough illustration of how OOP can be used to simulate intricate systems such as robot warfare. The organised approach to simulation design and execution is highlighted in this paper, with a focus on real-time interaction inside a controlled environment, modularity, and extensibility.