A volcano erupting at night

Description automatically generated with medium confidence

Simone Campo

SID: 1911840

**Element 010:**

Documentation Report

**Module Leader:**

George Wilson

**Course of Studies:**

Computer Science

**Object** 

**Oriented C++**

MOD003197

Introduction

Report Outline

In the following report, it has been documented a console-based fire simulation in Object Oriented C++ and its characteristic are organised in 3 main sections:

1. **Functionality and Implementation** presents a discussion about the enhancements implemented in the simulation as well as the user interactions.
2. **Simulation Development Evaluation** outlines the approach used to realise the Software and its architecture is presented with the UML Class Diagram and the description of classes and visibility of attributes and methods.
3. **Appendix** gives simple instructions for running the program and the source code of both the header and .cpp files.

Resources used

* MacBook Pro 2020 running macOS 11.2.3 Big Sur.
* HP 2018 running Microsoft Windows 10.
* Apple Xcode, version 2021 for Mac
* Visual Studio, version 2019 for Windows
* Apple Safari web browser.
* Microsoft Word, version 2016 for Mac.
* Draw.io, diagramming website, version 2021.

Table of Contents

[1. Functionality and Implementations 1](#_Toc70093368)

[1.1 Background 1](#_Toc70093369)

[1.2 Functionalities and Enhancements 2](#_Toc70093370)

[1.2.1 Basic Implementation 2](#_Toc70093371)

[1.2.2 Enhanced Implementation 3](#_Toc70093372)

[1.2.3 Statistics 4](#_Toc70093373)

[1.2.4 User Interactions 5](#_Toc70093374)

[2. Simulation Development Evaluation 6](#_Toc70093375)

[2.1 Approach used 6](#_Toc70093376)

[2.1.1 Classes definitions 6](#_Toc70093377)

[2.1.2 UML Class Diagram 7](#_Toc70093378)

[2.2 Program Structure 9](#_Toc70093379)

[Conclusion 10](#_Toc70093380)

[References 11](#_Toc70093381)

[Appendix 12](#_Toc70093382)

[Instructions for running the program 12](#_Toc70093383)

[Source Code 13](#_Toc70093384)

[1. forest.h 13](#_Toc70093385)

[2. listOfTrees.h 15](#_Toc70093386)

[3. program.h 17](#_Toc70093387)

[4. tree.h 18](#_Toc70093388)

[5. wind.h 20](#_Toc70093389)

[6. forest.cpp 21](#_Toc70093390)

[7. listOfTrees.cpp 26](#_Toc70093391)

[8. program.cpp 30](#_Toc70093392)

[9. tree.cpp 33](#_Toc70093393)

[10. wind.cpp 36](#_Toc70093394)

[11. main.cpp 37](#_Toc70093395)

# Functionality and Implementations

## Background

The project aims to develop a console-based fire simulation in Object Oriented C++, which presents a 2D forest of 21x21 grid of cells containing the state of trees (&, @) surrounded by the boundary (0) (Figure 1 and Figure 2).

Text

Description automatically generated Text

Description automatically generated

Figure 1: Initialised Forest (Basic Implementation) Figure 2: Initialised Forest (Enhanced Implementation)

During each time step, some trees burn according to certain condition defined in Section 1.2 and the simulation finishes when either there are no more burning trees in the forest or when the user wants to get out. At the end, the fire simulator statistics are displayed (Figure 5 and Figure 6).

## Functionalities and Enhancements

How introduced in Section 1.1, the user can choose to run either the basic simulation or the enhanced one. While the user interaction and the structure are the same, the enhanced implementation presents different elements that play a crucial role in the trees’ probability of catching the fire.

### Basic Implementation

In the basic implementation, a tree can only have a dry soil (&) and it has a 60% of possibility of catching the fire. Since the probability is evaluated each time that a neighbour tree is burning, the chance of catching the fire increases if more neighbours are burning (Figure 3).

A screenshot of a computer

Description automatically generated with medium confidence

Figure 3: End of the simulation (Basic Implementation)

### Enhanced Implementation

In the enhanced implementation, the trees have either a dry soil (&) or a moist soil (@) (Figure 4). The chance of having one of them is 50%. Another important element is the wind, which direction is either North, South, East and West and the speed (in Km/h) is a random number between 10 and 35.

In this enhanced version, the probability of catching the fire is up to 100:

* Tree with dry soil the probability is set to 60 / 100
* Tree with moist soil the probability is set to 40 / 100
* If the wind blows in the direction the fire comes, the probability increases by the same number as its speed (from 10 to 35).

Therefore, the probability ranges between 40% and 95%.

Text

Description automatically generated with medium confidence

Figure 4: End of the simulation (Enhanced Implementation)

### Statistics

In both implementations, the simulation’s statistics are written on a .txt file called *“statistics”* and then the system asks the user whether they wish the program to read the file and print its content on the screen.

As displayed in Figure 5 and Figure 6, while the structure changes according to the implementation chosen, the information is displayed in a clear and intuitive way proving the user with a complete view of what happened during the simulation.

Text

Description automatically generated

Figure 5: statistics.txt (Basic Implementation)

Text

Description automatically generated

Figure 6: statistics.txt (Enhanced Implementation)

### User Interactions

Even though the user interaction in this program is limited, it is made simple and easy to understand in order to improve the general user experience.

The first user interaction is displayed in Figure 7 and it asks the user whether they want to add wind and moisture to the simulation. To improve the effectiveness, the input is not case sensitive and if a character other than those specified is entered, the system allows the user to try again (Figure 8).

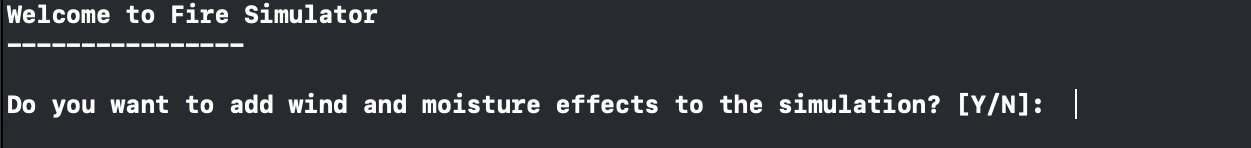


Figure 7: Implementation choice



Figure 8: Implementation choice – wrong input

During the simulation, the end of one time-period and the start of the next is managed by the command displayed in Figure 9. Here, the user can either press enter or ‘x’ to get out from the execution. Any other character other than those defined are simply ignored by the system.

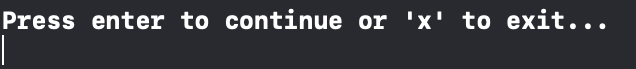


Figure 9: Continue or End the Simulation

The last user interaction is presented in Figure 10 and it prompts the user to read the statistics.txt file. Just like Figure 8, the user can try again if they enter a wrong input (Figure 11).

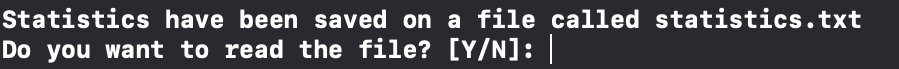


Figure 10: Reading the statistics.txt file



Figure 11: Reading the statistics.txt file – wrong input

# Simulation Development Evaluation

## Approach used

According to Doherty (2020), Object Oriented Programming (OOP) is the most popular programming paradigm used by both developers and programmers and, through concepts like classes and objects, the software application is broken down into simple and reusable piece of codes.

Half (2017) states that OOP programming languages such as C++ have several advantages, some of which are listed below:

1. *Modularity for easier troubleshooting* as each functionality or feature belongs to a class or to an object.
2. *Effective problem solving* by breaking down the software into bite-sized problems that have to be solved.

Being written in C++, the fire simulator has been developed using an Object-Oriented approach and the created classes, along with their properties and methods are explained in Section 2.1.1 and represented in the UML Class Diagram in Section 2.1.2.

### Classes definitions

The classes developed in the Fire Simulator are the following:

1. ***Forest Class*** holds the 2D array of chars as well as parameters used to calculate the statistics. Its methods update the forest object and retrieves the properties.
2. ***ListOfTrees Class*** holds the Linked List data structure, which keeps all the trees object and removes/destroys them as soon as they are burnt. In this way, every time an object tree dies, the memory is deallocated by improving the general performance.
3. ***Tree Class*** holds the properties related to the tree object such as the state and the moisture and methods that can modify or retrieve those properties.
4. ***Wind Class*** holds the direction and speed properties of the wind object and the methods that can retrieve them.
5. ***Program Class*** is the only class which is not supposed to create objects. In this respect, its static methods can perform recurrent tasks in the main function such as getting and evaluating the user choice and write / read from the file.

In general, the classes’ structure is designed following the encapsulation concept, which means that properties are private and, depending on their scope, *getters* and *setters* methods can respectively modify and retrieve their content. In this way, data is hidden and protected against interferences and misuse (tutorialspoint, 2021). Indeed, some attributes such as the wind direction are initialised once in the constructor and do not change.

Additionally, the classes have been related each other in accordance with the files included in each class’s header file and the type of the relationships defined by Visual Paradigm (2020) are the following:

1. **Association:** simple relationship between two independent classes and they link Program class with Forest and Wind classes as well as Wind class and Tree class.
2. **Aggregation:** a *“has a”* relationship between Forest and ListOfTrees classes. Indeed, the Forest object has a ListOfTrees object as attribute. However, the list has an independent life from the Forest.
3. **Composition:** a *“is part of”* relationship between ListOfTrees and Tree classes. The trees are initialised and automatically inserted into the list and their life depends upon the list’s life. In this respect, it does not exist a tree which is not into the list.

Regarding the memory management, C++ allows the programmer to declare pointers of a data type, which hold the address of another variables (Guru99, 2021).

Since they save memory and increase the processing speed (w3schools.in, 2021), they have been used in the Fire Simulator in some cases: The Linked List holds pointers to tree objects while the wind object, being subject to the user choice, it is dynamically declared as “*nullptr”* in main and initialised only if the wind has to be applied to the simulation. In this way, when it is passed as parameter, it can be evaluated on its state, as it remains “*nullptr”* while running in default mode (Figure 12).

Text

Description automatically generated

Figure 12: Wind pointer in Main Function

As presented in the Class Diagram, the classes Forest, ListOfTrees and Wind are supposed to create only one single object. Therefore, it has been applied the Singleton design pattern.

According to GeeksforGeeks (2021), the Singleton consists of a private constructor and a private static \_pointer, which can be initialised and retrieved by a special public and static get function that runs the constructor only if the \_pointer has not been initialised before (= nullptr). In this way, it is not possible to create more than one object within the program. (Figure 13 – Figure 14).

Text

Description automatically generated



Figure 13: Part of the wind.h file

Text

Description automatically generated



Figure 14: Part of the wind.cpp file

### UML Class Diagram

Diagram

Description automatically generated

Figure 15: Fire Simulator UML Class Diagram

## Program Structure

This software program has been structured according to the ‘out-of-line’ method. According to Microsoft (2019), the classes have been declared in a header file (.h) and the implementation of their method in an implementation file (.cpp). Even though the execution speed is slower, and the compiled file size is larger, it promotes hiding of code from 3rd parties through the distribution of the Application Program Interface (API).

The files belonging to this project are listed in the following table:

|  |  |
| --- | --- |
| **Header Files (.h)** | **Implementation File (.cpp)** |
| wind.h | wind.cpp |
| tree.h | tree.cpp |
| listOfTrees.h | listOfTrees.cpp |
| forest.h | forest.cpp |
| program.h | program.cpp |
| - | main.cpp |

# Conclusion

In conclusion, the Author feels to have excellently met the learning outcomes defined in the Object-Oriented C++ module. In this respect, the Author is able to critically understand the difference between procedural and object-oriented style of programming, evaluate the relative merits of static and dynamic memory allocation in a given situation, use a C++ IDE to develop a C++ software solution and analyze a problem that involves writing an object-oriented C++ program that separates declaration from implementation.

# References

* Doherty, E., 2020. *What is Object Oriented Programming? OOP Explained in Depth*. [online] Available at: <<https://www.educative.io/blog/object-oriented-programming>> [Accessed 18 April 2021].
* GeeksforGeeks, 2020. *Singleton Design Pattern | Implementation*. [online] Available at: <<https://www.geeksforgeeks.org/singleton-design-pattern/>> [Accessed 18 April 2021].
* Guru99, 2021. *C++ Pointers with Examples.* [online] Available at: <<https://www.guru99.com/cpp-pointers.html>> [Accessed 18 April 2021].
* Half, R., 2017. *4 Advantages of Object-Oriented Programming*. [online] Available at: <<https://www.roberthalf.com/blog/salaries-and-skills/4-advantages-of-object-oriented-programming>> [Accessed 18 April 2021].
* Microsoft, 2019. *Header Files.* [online] Available at: <<https://docs.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=msvc-160>> [Accessed 18 April 2021].
* tutorialspoint, 2021. *Data Encapsulation in C++.* [online] Available at: <<https://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm>> [Accessed 18 April 2021].
* Visual Paradigm, 2020. *UML Association vs Aggregation vs Composition.* [online] Available at: <<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/>> [Accessed 24 April 2021].
* w3schools.in, 2021. *C++ Pointers*. [online] Available at: <<https://www.w3schools.in/cplusplus-tutorial/pointers/>> [Accessed 18 April 2021].

# Appendix

## Instructions for running the program

As introduced in Section 1, this console-based program has two different versions, and the user has to decide which one to run. Regardless the choice, the forest is then initialised, and the user has to press ‘enter’ to update the forest and to burn other trees. Should the user want to get out from the simulation, the ‘x’ key has to be entered. Otherwise, the simulation ends when there are no more burning trees. Once the simulation finishes, the statistics are written on a text file (.txt) and the user can decide to print the information on the screen. At the end, the user can press any key to close the program.

## Source Code

### forest.h

#ifndef FORESTH

#define FORESTH

#include "listOfTrees.h"

**class** Forest

{

**private**:

**char** forestArray[21][21]; // 2D Array holds the states of the trees

// Counters attributes used for the statistics function and they keep track of the number of trees in the list

**int** totalTrees = 0, totalMoisture = 0, totalDry = 0;

**int** deadTrees = 0, deadMoisture = 0, deadDry = 0;

**int** liveTrees = 0, liveMoisture = 0, liveDry = 0;

**int** burningTrees = 0;

// Pointer to the linked list data structure

ListOfTrees\* list = ListOfTrees::getList();

// Pointer to the static object of the Forest class

**static** Forest\* \_pointer;

/// <summary>

/// Private constructor initialise Forest object with boundary and the burning tree at the center

/// Set the counters attributes of the Forest

/// </summary>

/// <param name="moisture"> true if the user wants to apply the moist soil </param>

Forest(**bool** moisture);

**public**:

/// <summary>

/// Singleton Design Pattern: return the static instance and initialise it only if it's not been already done

/// </summary>

/// <param name="moisture"> true if the user wants to apply the moist soil </param>

/// <returns> pointer to the Forest object </returns>

**static** Forest\* getForest(**bool** moisture);

// Getter

**int** getTotalTrees(**void**);

// Getter

**int** getTotalMoisture(**void**);

// Getter

**int** getTotalDry(**void**);

// Getter

**int** getDeadTrees(**void**);

// Getter

**int** getDeadMoisture(**void**);

// Getter

**int** getDeadDry(**void**);

// Getter

**int** getLiveTrees(**void**);

// Getter

**int** getLiveMoisture(**void**);

// Getter

**int** getLiveDry(**void**);

// Getter

**int** getBurningTrees(**void**);

/// <summary>

/// Update the state of the trees in the forest

/// </summary>

/// <param name="wind">pointer to the wind object</param>

/// <returns> true when there are no more burning trees and therefore the fire finished </returns>

**bool** updateForest(Wind\* wind);

/// <summary>

/// Burn neighborhoods of a tree which location in the forest is passed as parameters

/// </summary>

///<param name="row"> The row index of the tree position in the 2D Array </param>

///<param name="column"> The column index of the tree position in the 2D Array </param>

/// <param name="wind"> Pointer to the wind object </param>

**void** burnNeighbour(**int** row, **int** column, Wind\* wind);

/// <summary>

/// Draw the 2D forest on the console

/// </summary>

**void** drawForest(**void**);

};

#endif

### listOfTrees.h

#ifndef LISTOFTREESH

#define LISTOFTREESH

#include "tree.h"

class ListOfTrees

{

private:

Tree\* start; // First element in the Linked List

Tree\* end; // Last element in the Linked List

static ListOfTrees\* \_pointer; // Private static pointer

/// <summary>

/// Private constructor initialise the linked list data structure

/// </summary>

ListOfTrees(void);

public:

/// <summary>

/// Return the static instance and initialise only if it has not been already defined

/// </summary>

static ListOfTrees\* getList();

/// <summary>

/// Return true when the list is empty

/// </summary>

bool isEmpty(void);

/// <summary>

/// Return the number of elements in the list

/// </summary>

int countTrees(void);

/// <summary>

/// Return the number of trees within the list having the moisture

/// </summary>

int countMoisture(void);

/// <summary>

/// Return the number of burning trees

/// </summary>

int countBurning(void);

/// <summary>

/// append a tree object in the list

/// </summary>

/// <param name="object">object tree to be added</param>

void appendTree(Tree\* object);

/// <summary>

/// Check whether an element having a certain location exists

/// </summary>

///<param name="row">The row index of the tree position in the 2D Array </param>

///<param name="column">The column index of the tree position in the 2D Array </param>

bool elementExist(int row, int column);

/// <summary>

/// Return an element in the list given their location

/// Return NULL if the element does not exist

/// </summary>

///<param name="row">The row index of the tree position in the 2D Array </param>

///<param name="column">The column index of the tree position in the 2D Array </param>

Tree\* getElement(int row, int column);

/// <summary>

/// Remove a tree object from the list

/// </summary>

///<param name="row"> The row index of the tree position in the 2D Array </param>

///<param name="column"> The column index of the tree position in the 2D Array </param>

void removeTree(int row, int column);

/// <summary>

/// Remove the first element from the list

/// </summary>

void removeFirst(void);

/// <summary>

/// Remove the last element from the list

/// </summary>

void removeLast(void);

};

#endif

### program.h

#ifndef PROGRAMH

#define PROGRAMH

#include "wind.h"

#include "forest.h"

#include <string>

**using** **namespace** std;

**class** Program

{

**public**:

/// <summary>

/// Display a question to the user and gets the answer which can be either Yes or No

/// </summary>

/// <param name="question"> question to answer to the user</param>

/// <returns>True if Yes or False if No</returns>

**static** **bool** chooseMode(string question);

/// <summary>

/// Display both the title and the legend to the screen

/// </summary>

/// <param name="wind">pointer to a Wind object</param>

/// <param name="round">round of the simulation</param>

**static** **void** displayMenu(Wind\* wind, **int** round);

/// <summary>

/// Creates / Open a file and write the statistics on it

/// </summary>

/// <param name="wind">pointer to a wind object</param>

/// <param name="forest">forest initialised in Main</param>

/// <param name="rounds">number of rounds played</param>

/// <param name="fileName">name of the file where is intended to create / open</param>

/// <returns>true if statistics have been written, false if the connection have not been opened</returns>

**static** **bool** printStatistics(Wind\* wind, Forest\* forest, **int** rounds, string fileName);

/// <summary>

/// read the file

/// </summary>

/// <param name="fileName">name of the file which is inteded to be read</param>

/// <returns>true if statistics have been read, false if the connection have not been opened</returns>

**static** **bool** readStatistics(string fileName);

};

#endif

### tree.h

#ifndef TREEH

#define TREEH

#include "wind.h"

**class** Tree

{

**private**:

**char** state; // moist soil, dry soil, death, burning

**bool** moisture; // True for moist / False for dry

Tree\* next = **nullptr**; // Next object in the listOfTrees

**int** row, column; // location (x, y) in the 2D forestArray

/// <summary>

/// private method, 50% of probability to allow the moisture (true)

/// </summary>

**bool** setMoisture(**void**);

**public**:

///<summary>

/// Default constructor creates a tree object with default state

///<summary>

///<param name="row"> The row index of the tree position in the 2D Array </param>

///<param name="column"> The column index of the tree position in the 2D Array </param>

///<param name="moist"> True if the user wants to apply the moisture </param>

Tree(**int** row, **int** column, **bool** moist);

///<summary>

/// Overloaded constructor creates a new tree object with no default state

///<summary>

///<param name="row">The row index of the tree position in the 2D Array </param>

///<param name="column"> The column index of the tree position in the 2D Array </param>

///<param name="state"> current state of the object </param>

///<param name="moist"> True if the user wants to apply the moisture </param>

Tree(**int** row, **int** column, **char** state, **bool** moist);

**void** setState(**char** state); // Setter

**char** getState(**void**); // Getter

**void** setNext(Tree\* current); // Setter

Tree\* getNext(**void**); // Getter

**bool** getMoisture(**void**); // Getter

///<summary>

/// Checks whether the object having those indexes exists in the list

///<summary>

///<param name="row">The row index of the tree position in the 2D Array </param>

///<param name="column">The column index of the tree position in the 2D Array </param>

**bool** checkCoordinates(**int** row, **int** column);

/// <summary>

/// Return true if the tree is burning

/// </summary>

**bool** isBurning(**void**);

///<summary>

/// Determine whether the tree object has to be deleted

/// Depending on the case, the probability of catching the fire ranges from 40% to 95%

///<summary>

**bool** ifDeleted(Wind\* wind);

};

#endif

### wind.h

#ifndef WINDH

#define WINDH

#include <string>

**using** **namespace** std;

**class** Wind

{

**private**:

**static** Wind\* \_pointer; // private static pointer

string direction; // North, South, East and West

**int** speed; // Velocity in Km/h

Wind(**void**); // Private constructor

**public**:

/// <summary>

/// return the static instance and initialise only if it has not been already defined

/// </summary>

**static** Wind\* getWind();

string getDirection(**void**); // Getter

**int** getSpeed(**void**); // Getter

};

#endif

### forest.cpp

#include "forest.h"

#include <iostream>

using namespace std;

Forest\* Forest::\_pointer = nullptr; // out of line definition of static pointer

Forest::Forest(bool moisture)

{

for (int i = 0; i < 21; i++)

{

for (int j = 0; j < 21; j++)

{

// Draw the boundaries and do not create the object

if (i == 0 || j == 0 || i == 20 || j == 20)

{

forestArray[i][j] = '0';

}

// If the cell is the center of the forest, creates a burning tree, add into the list

else if ((i == 10) && (j == 10))

{

Tree\* tree = new Tree(i, j, '#', moisture);

forestArray[i][j] = tree->getState();

list->appendTree(tree);

if (tree->getMoisture())

{

totalMoisture++;

}

else

{

totalDry++;

}

}

else

{

Tree\* tree = new Tree(i, j, moisture);

forestArray[i][j] = tree->getState();

list->appendTree(tree);

if (tree->getMoisture())

{

totalMoisture++;

}

else

{

totalDry++;

}

}

}

}

// when the forest has initialised, it sets the counters

totalTrees = totalDry + totalMoisture;

liveTrees = totalTrees;

liveDry = totalDry;

liveMoisture = totalMoisture;

burningTrees = list->countBurning();

}

Forest\* Forest::getForest(bool moisture)

{

// Execute the constructor only if the static pointer has not been initialised

if (\_pointer == nullptr)

{

\_pointer = new Forest(moisture);

}

return \_pointer;

}

int Forest::getTotalTrees(void) { return totalTrees; }

int Forest::getTotalMoisture(void) { return totalMoisture; }

int Forest::getTotalDry(void) { return totalDry; }

int Forest::getDeadTrees(void) { return deadTrees; }

int Forest::getDeadMoisture(void) { return deadMoisture; }

int Forest::getDeadDry(void) { return deadDry; }

int Forest::getLiveTrees(void) { return liveTrees; }

int Forest::getLiveMoisture(void) { return liveMoisture; }

int Forest::getLiveDry(void) { return liveDry; }

int Forest::getBurningTrees(void) { return burningTrees; }

bool Forest::updateForest(Wind\* wind)

{

bool finish = true;

// Iterate through the forest

for (int i = 0; i < 21; i++)

{

for (int j = 0; j < 21; j++)

{

// If tree is burning, remove it from the list, burn its neighbours and set the array's cell as empty

if (forestArray[i][j] == '#')

{

burnNeighbour(i, j, wind);

list->removeTree(i, j);

forestArray[i][j] = ' ';

finish = false; // set to false as more trees ha

}

}

}

// Update counters each time the forest is updated

liveTrees = list->countTrees();

liveMoisture = list->countMoisture();

liveDry = liveTrees - liveMoisture;

deadTrees = totalTrees - liveTrees;

deadMoisture = totalMoisture - liveMoisture;

deadDry = totalDry - liveDry;

burningTrees = list->countBurning();

return finish;

}

void Forest::burnNeighbour(int row, int column, Wind\* wind)

{

// tree on the left

// If it exists and it is not burning

if (list->elementExist(row, column - 1) && list->getElement(row, column - 1)->getState() != '#')

{

// Pass the pointer only if it is initialised and if the direction is West

if (wind != nullptr && wind->getDirection() == "West")

{

if (list->getElement(row, (column - 1))->ifDeleted(wind))

{

list->getElement(row, (column - 1))->setState('#');

forestArray[row][column - 1] = '#';

}

}

else

{

if (list->getElement(row, (column - 1))->ifDeleted(nullptr))

{

list->getElement(row, (column - 1))->setState('#');

forestArray[row][column - 1] = '#';

}

}

}

// tree on the right

// If it exists and it is not burning

if (list->elementExist(row, column + 1) && list->getElement(row, column + 1)->getState() != '#')

{

// Pass the pointer only if it is initialised and if the direction is East

if (wind != nullptr && wind->getDirection() == "East")

{

if (list->getElement(row, column + 1)->ifDeleted(wind))

{

list->getElement(row, (column + 1))->setState('#');

forestArray[row][column + 1] = '#';

}

}

else

{

if (list->getElement(row, column + 1)->ifDeleted(nullptr))

{

list->getElement(row, (column + 1))->setState('#');

forestArray[row][column + 1] = '#';

}

}

}

// Upper tree

// If it exists and it is not burning

if (list->elementExist(row - 1, column) && list->getElement(row - 1, column)->getState() != '#')

{

// Pass the pointer only if it is initialised and if the direction is North

if (wind != nullptr && wind->getDirection() == "North")

{

if (list->getElement(row - 1, column)->ifDeleted(wind))

{

list->getElement((row - 1), column)->setState('#');

forestArray[row - 1][column] = '#';

}

}

else

{

if (list->getElement(row - 1, column)->ifDeleted(nullptr))

{

list->getElement((row - 1), column)->setState('#');

forestArray[row - 1][column] = '#';

}

}

}

// Bottom tree

// If it exists and it is not burning

if (list->elementExist(row + 1, column) && list->getElement(row + 1, column)->getState() != '#')

{

// Pass the pointer only if it is initialised and if the direction is South

if (wind != nullptr && wind->getDirection() == "South")

{

if (list->getElement(row + 1, column)->ifDeleted(wind))

{

list->getElement((row + 1), column)->setState('#');

forestArray[row + 1][column] = '#';

}

}

else

{

if (list->getElement(row + 1, column)->ifDeleted(nullptr))

{

list->getElement((row + 1), column)->setState('#');

forestArray[row + 1][column] = '#';

}

}

}

}

void Forest::drawForest(void)

{

for (int i = 0; i < 21; i++)

{

for (int j = 0; j < 21; j++)

{

cout << forestArray[i][j];

}

cout << endl;

}

}

### listOfTrees.cpp

#include "listOfTrees.h"

ListOfTrees\* ListOfTrees::\_pointer = nullptr; // out of line definition of static pointer

ListOfTrees::ListOfTrees(void)

{

start = NULL;

end = NULL;

}

ListOfTrees\* ListOfTrees::getList()

{

if (\_pointer == nullptr)

{

\_pointer = new ListOfTrees();

}

return \_pointer;

}

bool ListOfTrees::isEmpty(void)

{

if (start == NULL && end == NULL)

{

return true;

}

return false;

}

int ListOfTrees::countTrees(void)

{

int number = 0;

Tree\* current = start;

// Iterate through the linked list

while (current != NULL)

{

number++;

current = current->getNext();

}

return number;

}

int ListOfTrees::countMoisture(void)

{

int number = 0;

Tree\* current = start;

while (current != NULL)

{

if (current->getMoisture())

{

number++;

}

current = current->getNext();

}

return number;

}

int ListOfTrees::countBurning(void)

{

int number = 0;

Tree\* current = start;

while (current != NULL)

{

if (current->isBurning())

{

number++;

}

current = current->getNext();

}

return number;

}

void ListOfTrees::appendTree(Tree\* object)

{

object->setNext(start);

start = object;

}

bool ListOfTrees::elementExist(int row, int column)

{

Tree\* current = start;

while (current != NULL)

{

if (current->checkCoordinates(row, column))

{

return true;

}

else

{

current = current->getNext();

}

}

return false;

}

Tree\* ListOfTrees::getElement(int row, int column)

{

Tree\* current = start;

while (current != NULL)

{

if (current->checkCoordinates(row, column))

{

return current;

}

else

{

current = current->getNext();

}

}

return NULL;

}

void ListOfTrees::removeTree(int row, int column)

{

Tree\* current = start;

Tree\* previous = nullptr;

while (current != NULL)

{

// If the element is not in the required location, it moves further

if (!(current->checkCoordinates(row, column)))

{

previous = current;

current = current->getNext();

}

else

{

// Runs when it is the first element in the linked list

if (previous == nullptr)

{

removeFirst();

break;

}

// Runs if it is the last element in the linked list

else if (current->getNext() == NULL)

{

removeLast();

break;

}

else // it is in the middle of the list

{

previous->setNext(current->getNext());

delete current;

break;

}

}

}

}

void ListOfTrees::removeFirst(void)

{

Tree\* current = start;

start = start->getNext();

delete current;

}

void ListOfTrees::removeLast(void)

{

Tree\* current;

Tree\* previous = nullptr;

current = start;

while (current->getNext() != NULL)

{

previous = current;

current = current->getNext();

}

end = previous;

previous->setNext(NULL);

delete current;

}

### program.cpp

#include "program.h"

#include <iostream>

#include <fstream>

bool Program::chooseMode(string question)

{

char userChoice;

bool AllOk = false;

// keep repeating until the user prompts either Y or F

do

{

cout << question;

cin >> userChoice;

cin.ignore();

// tolower makes the userChoice no case sensitive

switch (tolower(userChoice))

{

case 'y':

AllOk = true;

break;

case 'n':

AllOk = false;

break;

default:

// Assing a default value

userChoice = 'x';

// Clear the console showing the error

system("CLS");

cout << "Not valid option" << endl;

break;

}

} while (userChoice == 'x');

return AllOk;

}

void Program::displayMenu(Wind\* wind, int round)

{

system("CLS");

cout << "FIRE SIMULATOR\n----------------" << endl << endl;

cout << "LEGEND" << endl;

// When the user chose the enhancements

if (wind != nullptr)

{

cout << " - Wind direction: " << wind->getDirection() << '\n'

<< " - Wind speed: " << wind->getSpeed() << "Km/h" << '\n'

<< " - @: Tree with most soil" << endl;

}

// Standard legend information

cout << " - &: Tree with dry soil" << '\n'

<< " - #: Burning Tree" << '\n'

<< " - ' ': Death Tree" << '\n'

<< "----------------" << endl << endl;

cout << "Round: " << round << endl << endl;

}

bool Program::printStatistics(Wind\* wind, Forest\* forest, int rounds, string fileName)

{

ofstream outFile;

outFile.open(fileName); // Open connection

// The connection could not be opened for some reasons, it return false

if (outFile.is\_open() == false)

{

return false;

}

else

{

outFile << "FIRE SIMULATOR STATISTICS\n----------------\n"

<< "Total Rounds: " << rounds << endl;

// Additional information are displayed if the user chose the enhancements

if (wind == nullptr)

{

outFile << " - Total Trees: " << forest->getTotalTrees() << '\n'

<< " - Dead Trees: " << forest->getDeadTrees() << '\n'

<< " - Live Trees: " << forest->getLiveTrees() << '\n'

<< " - Burning Trees: " << forest->getBurningTrees() << '\n';

}

else

{

// Standard statistics information

outFile << "\n1. WIND\n"

<< "\t - Direction: " << wind->getDirection() << '\n'

<< "\t - Speed: " << wind->getSpeed() << " Km/h\n"

<< "\n2. BURNING TREES\n"

<< "\t - Total: " << forest->getBurningTrees() << '\n'

<< "\n3. TREES WITH DRY SOIL\n"

<< "\t - Total: " << forest->getTotalDry() << '\n'

<< "\t - Dead: " << forest->getDeadDry() << '\n'

<< "\t - Live: " << forest->getLiveDry() << '\n'

<< "\n4. TREES WITH MOIST SOIL\n"

<< "\t - Total: " << forest->getTotalMoisture() << '\n'

<< "\t - Dead: " << forest->getDeadMoisture() << '\n'

<< "\t - Live: " << forest->getLiveMoisture() << endl;

outFile.close(); // close connection

}

return true;

}

}

bool Program::readStatistics(string fileName)

{

ifstream inFile;

string line;

system("CLS");

inFile.open(fileName); // Open connection

// The connection could not be opened for some reasons, it return false

if (inFile.is\_open() == false)

{

return false;

}

else

{

// Iterates throught the entire file

while (getline(inFile, line))

{

// Read line by line

cout << line << endl;

}

cout << endl;

inFile.close(); // Close connection

return true;

}

}

### tree.cpp

#include "tree.h"

**bool** Tree::setMoisture(**void**)

{

**int** randNumber = rand() % + 2; // Randomly either 0 or 1

// 50% probability of returning true

**if** (randNumber == 1)

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

Tree::Tree(**int** row, **int** column, **bool** moist)

{

**this**->row = row;

**this**->column = column;

// Runs true when user wants to apply moisture AND when the method allows that

**if** (moist == **true** && setMoisture())

{

**this**->moisture = **true**;

**this**->state = '@'; // @ tree with moist soil

}

**else**

{

**this**->moisture = **false**;

**this**->state = '&'; // & tree with dry soil

}

}

Tree::Tree(**int** row, **int** column, **char** state, **bool** moist)

{

**this**->state = state;

**this**->row = row;

**this**->column = column;

// Runs true when user wants to apply moisture AND when the method allows that

// State is not changed because it is set by the user

**if** (moist == **true** && setMoisture())

{

**this**->moisture = **true**;

}

**else**

{

**this**->moisture = **false**;

}

}

**void** Tree::setState(**char** state) { **this**->state = state; }

**char** Tree::getState() { **return** state; }

**void** Tree::setNext(Tree\* current) { **this**->next = current; }

Tree\* Tree::getNext() { **return** next; }

**bool** Tree::getMoisture() { **return** moisture; }

**bool** Tree::checkCoordinates(**int** row, **int** column)

{

**if** ((**this**->row == row) && (**this**->column == column))

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

**bool** Tree::isBurning()

{

**if** (state == '#')

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

**bool** Tree::ifDeleted(Wind\* wind)

{

**int** probability;

**int** randNumber = rand() % 100 + 1; // Random number from 1 to 100

// When the tree has a moist soil the probability is set to 40 / 100

// When the tree has a dry soil the probability is set to 60 / 100

**if** (moisture == **true**)

{

probability = 40;

}

**else**

{

probability = 60;

}

// If the wind is applied, its velocity is added to the probability of catching the fire [Up to + 35]

**if** (wind != **nullptr**)

{

probability += wind->getSpeed();

}

// Does not catch the fire only if the random number is greater than the number given by the probability

**if** (randNumber <= probability)

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

### wind.cpp

#include "wind.h"

Wind\* Wind::\_pointer = **nullptr**; // out of line definition of static pointer

Wind::Wind()

{

**int** randNumber = rand() % +4; // Random integer from 0 to 3

**this**->speed = rand() % 26 + 10; // Random integer from 10 to 35

// Constructor assigns the direction of Wind according to the number randomly generated

**switch** (randNumber)

{

**case** 0:

direction = "North";

**break**;

**case** 1:

direction = "South";

**break**;

**case** 2:

direction = "East";

**break**;

**case** 3:

direction = "West";

**break**;

**default**:

**break**;

}

}

Wind\* Wind::getWind()

{

**if** (\_pointer == **nullptr**)

{

\_pointer = **new** Wind();

}

**return** \_pointer;

}

string Wind::getDirection(**void**) { **return** direction; }

**int** Wind::getSpeed(**void**) { **return** speed; }

### main.cpp

#include "wind.h"

#include "forest.h"

#include "program.h"

#include <iostream>

#include <ctime>

**using** **namespace** std;

**int** main()

{

**bool** userChoice; // holds the choices made from the user

**char** userValue;

**bool** userEnd = **false**, noMoreBurningTrees = **false**;

**int** round = 1;

Wind\* wind = **nullptr**;

// statement required to generate random numbers

srand(**static\_cast**<**unsigned** **int**>(time(**nullptr**)));

cout << "Welcome to Fire Simulator" << '\n'

<< "----------------" << endl << endl;

userChoice = Program::chooseMode("Do you want to add wind and moisture effects to the simulation? [Y/N]: ");

// Initialise the Wind pointer only if the user wants to apply the wind to the simulation

**if** (userChoice)

{

wind = Wind::getWind();

}

Forest\* forest = Forest::getForest(userChoice);

// As long as the user click 'x' or there are no more burning trees

**while** (userEnd == **false** && noMoreBurningTrees == **false**)

{

Program::displayMenu(wind, round);

forest->drawForest();

cout << endl;

cout << "Press enter to continue or 'x' to exit..." << endl;

userValue = cin.get();

**if** (userValue == '\n')

{

noMoreBurningTrees = forest->updateForest(wind);

round++;

}

**else** **if** (tolower(userValue) != 'x')

{

cin.ignore(); // Ignore any character inserted other than 'x'

}

**else**

{

userEnd = **true**;

}

}

**if** (noMoreBurningTrees == **true**)

{

cout << "Program ended: no more burning trees" << endl << endl;

round--; // Normalise the counter

}

**else**

{

cout << "Program ended: key 'x' entered" << endl << endl;;

}

**if** (Program::printStatistics(wind, forest, round, "statistics.txt"))

{

cout << "Statistics have been saved on a file called statistics.txt" << endl;

userChoice = Program::chooseMode("Do you want to read the file? [Y/N]: ");

cout << endl;

**if** (userChoice)

{

// The file could not be opened

**if** (!Program::readStatistics("statistics.txt"))

{

cout << "Error while opening statistics.txt file." << endl;

}

}

}

**else**

{

cout << "Error while creating statistics.txt file." << endl;

}

cout << endl;

cout << "Press any key to end the program..." << endl;

cin.get();

**return** 0;

}