Zhuojian Chen (James)
COMP IV Sec 203: Project Portfolio
Spring 2024
Time to complete: Apr 23, 2024

Contents

1 PS0 - Hello World with SFML

1.1 Discussion

This project creates a simple game scenario using SFML. Players can control a battle plane using **W**(Up), **A**(Left), **S**(Down), and **D**(Right). The battle plane may fly out of the screen, as boundary collision is not detected. Players can control the plane to move along two axes at the same time. For instance, the plane can move up and right at the same time when the player press ‘W‘ and ‘D‘. The plane stops moving when players release the keys.

To implement these features, I began by establishing key constants and constructing a robust **Direction** class for future utilization.

This is my first foray into learning SFML and game libraries. I faced some challenges during PS0 due to my lack of experience in downloading C++ libraries and linking them during compilation. Specifically, on my MacBook, I had to add two flags to inform the compiler about the path to the SFML library:

```
-I /opt/homebrew/include
-L /opt/homebrew/lib
```

However, I discovered that this step was unnecessary later on, and I'll explain the solution in more detail in PS4.

In my initial attempt into C++ with the SFML library, I didn't adhere to best practices, resulting in imperfect code. Additionally, I neglected to carefully consider the naming of classes, variables, and functions, leading to code that was difficult to read. However, I recognize these shortcomings and am committed to overcoming them as I continue to refine my skills in the future.

1.2 Codebase

```makefile
CC = g++
#CFLAGS = --std=c++17 -Wall -Werror -pedantic -g
#LIB = -lsfml-graphics -lsfml-audio -lsfml-window -lsfml-system -lboost_unit_test_framework
CFLAGS = --std=c++17 -Wall -Werror -pedantic -g -I /opt/homebrew/include
LIB = -L /opt/homebrew/lib -lsfml-graphics -lsfml-audio -lsfml-window -lsfml-system
# Your .hpp files
DEPS =
# Your compiled .o files
OBJECTS =
# The name of your program
PROGRAM = sfml-app

.PHONY: all clean lint

all: $(PROGRAM)

# Wildcard recipe to make .o files from corresponding .cpp file
%.o: %.cpp $(DEPS)
	$(CC) $(CFLAGS) -c $<

$(PROGRAM): main.o $(OBJECTS)
	$(CC) $(CFLAGS) -o $@ $^ $(LIB)

clean:
	rm *.o $(PROGRAM)

```

```
lint:
	cpplint *.cpp *.hpp
```

```cpp
// <main.cpp>
#include <cmath>
#include <filesystem>
#include <iostream>
#include <SFML/Graphics.hpp>

// Global constants
constexpr int WINDOW_WIDTH = 1080;
constexpr int WINDOW_HEIGHT = 720;
constexpr int WINDOW_FPS = 60;
constexpr char WINDOW_TITLE[] = "First Experience with SFML";
constexpr char SPRITE_FILEPATH[] = "sprite.png";

class Direction {
    enum class Orientation { UP, DOWN, LEFT, RIGHT };

 public:
    static const Direction UP;
    static const Direction DOWN;
    static const Direction LEFT;
    static const Direction RIGHT;
    constexpr static int ORIENTAION_COUNT = 4;

    /**
     * @brief Creates a direction.
     * @param initOrientation The initial orientation.
     */
    explicit Direction(const Orientation initOrientation) {
        this->orientation = initOrientation;
    }

    /**
     * @brief Returns the corresponding integer.
     */
    [[nodiscard]] int toInt() const {
        return static_cast<int>(this->orientation);
    }

    /**
     * @brief Returns the opposite direction.
     */
    [[nodiscard]] Direction opposite() const {
        const auto orientationIndex = toInt();
        return Direction(static_cast<Orientation>(orientationIndex ^ 1));
    }

 private:
    Orientation orientation;
};
```

Define four static **Direction** objects with corresponding orientations.

```
const Direction Direction::UP{ Orientation::UP };
const Direction Direction::DOWN{ Orientation::DOWN };
const Direction Direction::LEFT{ Orientation::LEFT };
const Direction Direction::RIGHT{ Orientation::RIGHT };
```

Next, I introduced a **ControllableSprite** class, building upon the **sf::Sprite** from SFML library. In its constructor, a file path to the sprite's image is expected, enabling the creation of a texture from it. This class incorporates two essential arrays: **keyState**, responsible for tracking the selection status of each orientation, and **movementState**, which signifies the sprite's movement direction. For instance, if **movementState** equals $[1, 0, 0, 1]$, the sprite advances towards the upper-left direction.

```
class ControllableSprite final : public sf::Sprite {
    /**
     * @brief To simplify the logic, the sprite moves at a constant speed.
     */
    constexpr static float SPEED{ 0.5 };

 public:
    /**
     * @brief Creates a sprite.
     * @param filepath The filepath of the texture file.
     */
    explicit ControllableSprite(const std::string& filepath) {
        const std::filesystem::path absolutePath =
            std::filesystem::absolute(filepath);
        std::cout << "Loading texture file: " << absolutePath << std::endl;

        if (!texture.loadFromFile(filepath)) {
            throw std::invalid_argument("Failed to load the texture.");
        }

        this->setTexture(texture);
        this->setPosition(sf::Vector2f(0.0F, 0.0F));
    }

    /**
     * @brief Enables a specific dierction
     * @param direction The direction to enable.
     */
    void enableDirection(const Direction& direction) {
        // Set the key state to true
        this->keyState[direction.toInt()] = true;

        // Set the opposite movement state to true if its key state is true
        const auto opposite = direction.opposite();
        if (!this->keyState[opposite.toInt()]) {
            this->movementState[direction.toInt()] = true;
        }
    }

    /**
     * @brief Disables a specific direction.
     * @param direction The direction to disable.
     */
    void disableDirection(const Direction& direction) {
```

```cpp
        // Set the key state and movement state to false
        this->keyState[direction.toInt()] = false;
        this->movementState[direction.toInt()] = false;

        // Set the opposite movement state to true if its key state is true
        const auto opposite = direction.opposite();
        if (this->keyState[opposite.toInt()]) {
            this->movementState[opposite.toInt()] = true;
        }
    }

    /**
     * @brief Updates the position of this sprite.
     * @param dt Delta time in milliseconds.
     */
    void update(const int& dt) {
        int x = 0;
        int y = 0;

        if (movementState[Direction::UP.toInt()]) {
            y = -1;
        }
        if (movementState[Direction::DOWN.toInt()]) {
            y = 1;
        }
        if (movementState[Direction::LEFT.toInt()]) {
            x = -1;
        }
        if (movementState[Direction::RIGHT.toInt()]) {
            x = 1;
        }

        const float ds = 1.0F / static_cast<float>(sqrt(x * x + y * y));
        auto position = this->getPosition();
        position.x += static_cast<float>(x) * ds * static_cast<float>(dt);
        position.y += static_cast<float>(y) * ds * static_cast<float>(dt);
        this->setPosition(position);
    }

 private:
    sf::Texture texture;
    std::array<bool, Direction::ORIENTAION_COUNT> keyState{};
    std::array<bool, Direction::ORIENTAION_COUNT> movementState{};
};
```

With these preparations complete, I swiftly constructed the main function. Firstly, I instantiated a window and positioned a circle at its center. Next, a clock was initialized to facilitate the calculation of delta time between each frame and the previous one. Subsequently, a game loop was established, continuously rendering the window while it remained open. Within this loop, an inner loop was introduced to handle game events. If an **sf::Event::Closed** event occurred, the window would close. In the case of an **sf::Event::KeyPressed** event, the corresponding direction for the sprite was activated (enabled) based on the pressed key. Similarly, upon detecting an **sf::Event::KeyReleased** event, the direction for the sprite associated with the released key was disactived (disabled).

```cpp
int main() {
    // Create a window
    sf::RenderWindow window(
        sf::VideoMode(WINDOW_WIDTH, WINDOW_HEIGHT), WINDOW_TITLE);
    window.setFramerateLimit(WINDOW_FPS);

    // Create a circle and a sprite
    constexpr int CIRCLE_RADIUS = 100.F;
    constexpr float HALF = 0.5F;
    sf::CircleShape circleShape(CIRCLE_RADIUS);
    circleShape.setFillColor(sf::Color::Green);
    circleShape.setPosition(sf::Vector2f(
        static_cast<float>(WINDOW_WIDTH) * HALF - CIRCLE_RADIUS,
        static_cast<float>(WINDOW_HEIGHT) * HALF - CIRCLE_RADIUS));
    ControllableSprite sprite(SPRITE_FILEPATH);

    // Game loop
    sf::Clock clock;
    while (window.isOpen()) {
        const int dt = clock.restart().asMilliseconds();

        sf::Event event{};
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
                break;
            }

            if (event.type == sf::Event::KeyPressed) {
                if (event.key.code == sf::Keyboard::Key::W) {
                    sprite.enableDirection(Direction::UP);
                } else if (event.key.code == sf::Keyboard::Key::A) {
                    sprite.enableDirection(Direction::LEFT);
                } else if (event.key.code == sf::Keyboard::Key::S) {
                    sprite.enableDirection(Direction::DOWN);
                } else if (event.key.code == sf::Keyboard::Key::D) {
                    sprite.enableDirection(Direction::RIGHT);
                }
            }

            if (event.type == sf::Event::KeyReleased) {
                if (event.key.code == sf::Keyboard::Key::W) {
                    sprite.disableDirection(Direction::UP);
                } else if (event.key.code == sf::Keyboard::Key::A) {
                    sprite.disableDirection(Direction::LEFT);
                } else if (event.key.code == sf::Keyboard::Key::S) {
                    sprite.disableDirection(Direction::DOWN);
                } else if (event.key.code == sf::Keyboard::Key::D) {
                    sprite.disableDirection(Direction::RIGHT);
                }
            }
        }

        // Clear the window before rendering
        window.clear();

        // Draw the circle
        window.draw(circleShape);

        // Update sprite and draw the sprite onto the window
        sprite.update(dt);
        window.draw(sprite);

        // display the new frame
        window.display();
    }
}
```

2 PS1 - Linear Feedback Shift Register

2.1 PS1a

2.1.1 Discussion

For this assignment, I crafted a program to generate pseudo-random bits through the simulation of a linear feedback shift register (LFSR). An LFSR is a register that computes a linear function of its previous state as an input. In this implementation, I opted for the ”XOR” function.

In PS1a, I delved into the keyword ”constexpr,” which allowed me to create constant static member variables for classes, significantly enhancing readability. Additionally, I explored the **std::bitset** class for manipulating bitstrings.

2.1.2 Codebase

```
# C++ Compiler
COMPILER = g++

# C++ Flags
# CFLAGS = --std=c++17 -Wall -Werror -pedantic -g -I /opt/homebrew/include
CFLAGS = --std=c++17 -Wall -Werror -pedantic -g

# Libraries
# LIB = -L /opt/homebrew/lib -lboost_unit_test_framework
LIB = -lboost_unit_test_framework

# Hpp files (dependencies)
DEPS = FibLFSR.hpp

# Cpp files that should be compiled into object files
OBJECTS = FibLFSR.o PhotoMagic.o
OBJECTS_PHOTO_MAGIC = main.o
OBJECTS_TEST = test.o

# Programs
PROGRAM_PHOTO_MAGIC = PhotoMagic
PROGRAM_TEST = test

# Static library
STATIC_LIB = PhotoMagic.a

# Generate `PhotoMagic`, `test`, and `PhotoMagic.a` (static library)
.PHONY: all clean lint

all: $(PROGRAM_TEST) $(PROGRAM_PHOTO_MAGIC)

# Wildcard recipe to make .o files from corresponding .cpp file
# Note: this command matches .cpp file one by one, and "$<" here refers to the
# first prerequisite, which is the .cpp file matched.
%.o: %.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

# Program `PhotoMagic`
$(PROGRAM_PHOTO_MAGIC): $(STATIC_LIB) $(OBJECTS_PHOTO_MAGIC)
        $(COMPILER) $(CFLAGS) -o $@ $(OBJECTS_PHOTO_MAGIC) $(STATIC_LIB) $(LIB)

```

```
# Program `test`
$(PROGRAM_TEST): $(STATIC_LIB) $(OBJECTS_TEST)
        $(COMPILER) $(CFLAGS) -o $@ $(OBJECTS_TEST) $(STATIC_LIB) $(LIB)

# Create a PhotoMagi.a static library containing FibLFSR.o and PhotoMagic.o
$(STATIC_LIB): $(OBJECTS)
        ar rcs $(STATIC_LIB) $(OBJECTS)

# Run the PhotoMagic program and clean after running it
runPhotoMagic: $(PROGRAM_PHOTO_MAGIC)
        ./$(PROGRAM_PHOTO_MAGIC) && make clean

# Run all tests with Boost and clean after running it
runTest: $(PROGRAM_TEST)
        ./$(PROGRAM_TEST) && make clean

# Clean all object files and program files
# "-f" flag refers to "force", which suppresses the "No such file or directory"
# warning
clean:
        rm -f *.o $(STATIC_LIB) $(PROGRAM_PHOTO_MAGIC) $(PROGRAM_TEST)

# Use cpplint
lint:
        cpplint *.cpp *.hpp
```

```
// Copyright 2024 James Chan
#include <iostream>

int main() {
    std::cout << "This is the main function in main.cpp" << std::endl;

    return 0;
}
```

To begin, I established a **FibLFSR** class within the **PhotoMagic** namespace. It's worth noting that all components within PS1 are encapsulated within this namespace.

```
// <FibLFSR.hpp>
#include <array>
#include <bitset>
#include <iostream>
#include <string>

namespace PhotoMagic {
/**
 * @brief This class implemented Fibonacci LFSR (Linear Feedback Shift Register)
 * algorithm.
 */
class FibLFSR {
    /**
     * @brief The length of seeds.
     */
    constexpr static int SEED_LENGTH = 16;
```

```cpp

    /**
     * @brief The indexes of tabs. In each step, the bits at the tab indexes
     * will be used to perform XOR operations with the leftmost bit. In this
     * homework, tap indexes are 10, 12, and 13.
     */
    constexpr static std::array<int, 3> tabIndexes = { 10, 12, 13 };

 public:
    /**
     * @brief Creates an instance with the given seed.
     * @param seed A binary string (ascii) of length 16. Each character should
     * either be '0' or '1'.
     */
    explicit FibLFSR(const std::string& seed);

    /**
     * @brief Simulates one step and return the new (rightmost) bit.
     */
    int step();

    /**
     * @brief Simulates k steps and return a k-bit integer.
     * @param k The number of steps to perform.
     */
    int generate(int k);

    /**
     * @brief Returns the binary string form of the LFSR integer.
     */
    [[nodiscard]] std::string getLfsrBinaryString() const;

 private:
    std::bitset<SEED_LENGTH> lfsr;
};

/**
 * @brief Output a LFSR instance. A binary string form of the LFSR of the
 * instance will be output by the given ostream.
 * @param lfsr The LFSR instance to output.
 */
std::ostream& operator<<(std::ostream&, const FibLFSR& lfsr);

}  // namespace PhotoMagic
```

The member functions' implementation is straightforward. The constructor begins by verifying whether the provided seed string matches the expected length, which is defined as **SEED_LENGTH** and set to 16. Subsequently, the seed string undergoes validation to ensure that all characters are either '0' or '1'. Upon successful validation, the string is converted into a bitset, which initializes the LFSR.

```cpp
// <FibLFSR.cpp>
#include "FibLFSR.hpp"
#include <bitset>
#include <sstream>
#include <string>
```

```cpp

namespace PhotoMagic {

FibLFSR::FibLFSR(const std::string& seed) {
    // Check if the seed is legal
    if (seed.length() != SEED_LENGTH) {
        const std::string message =
            "The length of seed should be " + std::to_string(SEED_LENGTH);
        throw std::invalid_argument(message);
    }
    for (const char& bit : seed) {
        if (bit != '0' && bit != '1') {
            const std::string message =
                "Each character in the seed should either be '0' or '1'";
            throw std::invalid_argument(message);
        }
    }

    // Convert the seed string into the initial LFSR
    lfsr = std::bitset<SEED_LENGTH>{ seed };
}
```

The implementation of the **generate** function iteratively invokes the **step** function, which we will discuss soon, and inserts the returned bit to the right side of the return value.

```cpp
int FibLFSR::generate(const int k) {
    int ans = 0;
    for (int i = 0; i < k; ++i) {
        ans = (ans << 1) | step();
    }

    return ans;
}
```

The **step** function is the core of this algorithm. This function advances the linear feedback shift register (LFSR) by one step, generating a pseudo-random bit. It follows these steps:

1. Retrieve the current most significant bit (leftmost bit) from the LFSR.
2. **XOR**: Perform XOR operations between the retrieved bit and predetermined bits specified by tabIndexes. These indexes represent the positions in the register where XOR operations are applied to generate the next bit.
3. **Update LFSR**: Shift the entire register to the left by one position, effectively discarding the most significant bit. Set the least significant bit to the result of the XOR operations, thereby updating the LFSR with the newly generated bit.
4. Return the generated bit, which also serves as the output of the function.

```cpp
int FibLFSR::step() {
    // Get the current most significant bit (leftmost bit)
    int ans = lfsr[SEED_LENGTH - 1];

    // Let the bit perform XOR operations with tabs
    for (const int& tabIndex : tabIndexes) {
        ans ^= lfsr[tabIndex];
    }

    // Update lfsr
    lfsr <<= 1;
    lfsr.set(0, ans);

    return ans;
}
```

Finally, I overloaded the ”¡¡” operator to facilitate output using the output stream. However, I overlooked a crucial aspect: I didn’t declare the operator overload function as a ”friend” of the class. Consequently, I had to implement a separate function named **getLfsrBinaryString** to retrieve the LFSR’s binary string for output purposes.

```
std::string FibLFSR::getLfsrBinaryString() const { return lfsr.to_string(); }

std::ostream& operator<<(std::ostream& os, const FibLFSR& lfsr) {
    os << lfsr.getLfsrBinaryString();

    return os;
}

}  // namespace PhotoMagic
```

I wrote two unit tests to verify the functionality of my code:

- **testStepInstr**: The first unit test assesses the effectiveness of the overridden ”¡¡” operator. Once overridden, this operator is tasked with displaying a string representation of the Linear Feedback Shift Register (LFSR) in a readable binary format to the specified output stream.
- **testStepInstr**: In the second unit test, I further examine the overridden ”¡¡” operator. Here, the operator is utilized to generate an output string following nine iterations of the LFSR. The anticipated state of the LFSR for comparison purposes is determined through manual calculation.

```
// <test.cpp>
#include <iostream>
#include <sstream>
#include <string>
#include "FibLFSR.hpp"

#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main

#include <boost/test/unit_test.hpp>

using PhotoMagic::FibLFSR;

BOOST_AUTO_TEST_CASE(testStepInstr) {
    FibLFSR l("1011011000110110");
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 1);
    BOOST_REQUIRE_EQUAL(l.step(), 1);
    BOOST_REQUIRE_EQUAL(l.step(), 0);
```

```cpp
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 1);
}

BOOST_AUTO_TEST_CASE(testGenerateInstr) {
    FibLFSR l("1011011000110110");
    BOOST_REQUIRE_EQUAL(l.generate(9), 51);
}

// Test the output operator. The output string should be equal to the seed
// string if the LFSR object is not modified
BOOST_AUTO_TEST_CASE(testOutputOperator) {
    const std::string initialLFSR = "0110110001101100";
    const FibLFSR l(initialLFSR);
    std::stringstream ss;
    ss << l;
    BOOST_REQUIRE_EQUAL(ss.str(), initialLFSR);
}

// The the output operator. The output string should change after calling the
// generate method (k >= 1). The expected LFSR value is
BOOST_AUTO_TEST_CASE(testGenerateAndOutput) {
    const std::string initialLFSR = "0110110001101100";
    const std::string expectedLFSRAfterGenerate = "1101100001100110";
    FibLFSR l(initialLFSR);
    l.generate(9);

    std::stringstream ss;
    ss << l;
    BOOST_REQUIRE_EQUAL(ss.str(), expectedLFSRAfterGenerate);
}
```

2.2 PS1b

2.2.1 Discussion

In my PS2 project, I developed a program that can encrypt and decrypt images using a password string. The program displays both the original and the processed image in the same window.

This assignment enlightened me to the fact that a computer image essentially comprises a matrix of pixels, with each pixel containing three bytes for Red, Green, and Blue respectively. Furthermore, I grasped the concept of ”pseudo-randomness” through the implementation of an LFSR. Lastly, I gained insight into the fundamental principles of encrypting an image.

2.2.2 Achievements

The program's final output is indeed impressive. Initially, I encrypted an image of a cat, resulting in an encrypted image comprising seemingly random pixels, rendering the original content unrecognizable (**Figure 1.1**). Upon decrypting this encrypted image using the program once more, I achieved an exact replica of the original cat image, showcasing the program's robust encryption and decryption capabilities (**Figure 1.2**).

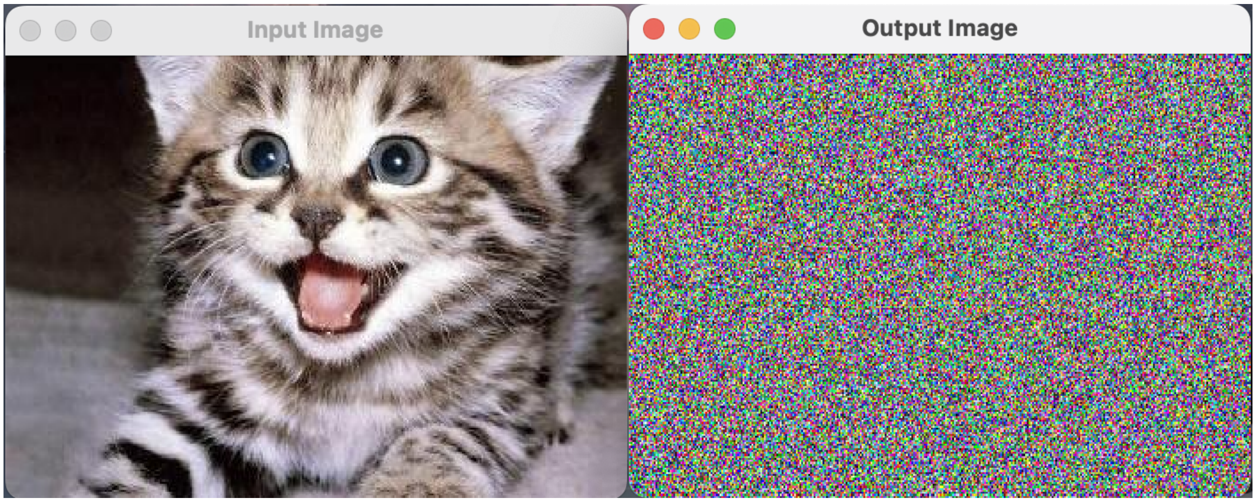

Figure 1.1 Encrypting an image of a cat.

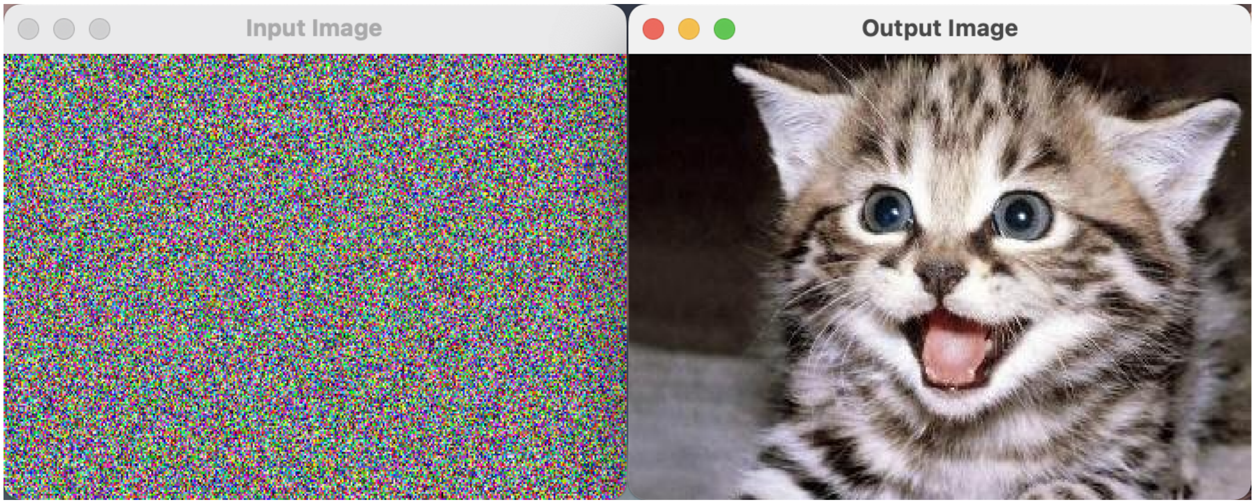

Figure 1.2 Decrypting the encrypted image and getting the replica of the original image.

2.2.3 Codebase

```makefile
# C++ Compiler
COMPILER = g++

# C++ Flags
CFLAGS = --std=c++17 -Wall -Werror -pedantic -g -I /opt/homebrew/include
#CFLAGS = --std=c++17 -Wall -Werror -pedantic -g

# Libraries
LIB = -L /opt/homebrew/lib -lsfml-graphics -lsfml-window -lsfml-system -lboost_unit_test_framework
#LIB = -lsfml-graphics -lsfml-window -lsfml-system -lboost_unit_test_framework

# Hpp files (dependencies)
DEPS = FibLFSR.hpp PhotoMagic.hpp

# Cpp files that should be compiled into object files
OBJECTS = FibLFSR.o PhotoMagic.o
OBJECTS_PHOTO_MAGIC = main.o
OBJECTS_TEST = test.o

# Programs
PROGRAM_PHOTO_MAGIC = PhotoMagic
PROGRAM_TEST = test

# Static library
STATIC_LIB = PhotoMagic.a

.PHONY: all clean lint

# Generate `PhotoMagic`, `test`, and `PhotoMagic.a` (static library)
all: $(PROGRAM_TEST) $(PROGRAM_PHOTO_MAGIC)

# Wildcard recipe to make .o files from corresponding .cpp file
# Note: this command matches .cpp file one by one, and "$<" here refers to the
# first prerequisite, which is the .cpp file matched.
%.o: %.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

# Program `PhotoMagic`
$(PROGRAM_PHOTO_MAGIC): $(STATIC_LIB) $(OBJECTS_PHOTO_MAGIC)
        $(COMPILER) $(CFLAGS) -o $@ $(OBJECTS_PHOTO_MAGIC) $(STATIC_LIB) $(LIB)

# Program `test`
$(PROGRAM_TEST): $(STATIC_LIB) $(OBJECTS_TEST)
        $(COMPILER) $(CFLAGS) -o $@ $(OBJECTS_TEST) $(STATIC_LIB) $(LIB)

# Create a PhotoMagi.a static library containing FibLFSR.o and PhotoMagic.o
$(STATIC_LIB): $(OBJECTS)
        ar rcs $(STATIC_LIB) $(OBJECTS)

# Run the PhotoMagic program and clean after running it
# Here, $(filter-out $@,$(MAKECMDGOALS)) passes all the arguments given to this
# target. It filters out the target name from the list of goals.
runPhotoMagic: $(PROGRAM_PHOTO_MAGIC)
        ./$(PROGRAM_PHOTO_MAGIC) $(filter-out $@,$(MAKECMDGOALS)) && make clean
```

```

# Run all tests with Boost and clean after running it
runTest: $(PROGRAM_TEST)
        ./$(PROGRAM_TEST) && make clean

# Clean all object files
cleanObjects:
        rm -f *.o

# Clean all object files and program files
# "-f" flag refers to "force", which suppresses the "No such file or directory"
# warning
clean:
        rm -f *.o $(STATIC_LIB) $(PROGRAM_PHOTO_MAGIC) $(PROGRAM_TEST)

# Use cpplint
lint:
        cpplint *.cpp *.hpp
```

The *main.cpp* is pretty straightforward:

```
// <main.cpp>
#include <string>
#include "PhotoMagic.hpp"

/**
 * @brief Takes three arguments: an input picture filename, an output picture
 * filename, and a binary password (the initial LFSR seed). It should display
 * the transformed picture on the screen. Note that since I implement a function
 * that can convert a alphanumeric string into a seed string, the third arugment
 * can be a alphanumeric password.
 * @param size The size of the argument list.
 * @param arguments The command line arguments.
 */
int main(const int size, const char* arguments[]) {
    if (size < 4) {
        std::cout << "Not enough arguments!" << std::endl;

        return -1;
    }

    const std::string inputFilename{ arguments[1] };
    const std::string outputFilename{ arguments[2] };
    const std::string seed{ PhotoMagic::convertPasswordToSeed(arguments[3]) };

    // Create images
    sf::Image originalImage;
    sf::Image processedImage;
    if (!originalImage.loadFromFile(inputFilename)) {
        return -1;
    }
    if (!processedImage.loadFromFile(inputFilename)) {
        return -1;
    }

    // Create a FibLFSR instance with the initial seed
    PhotoMagic::FibLFSR fibLfsr{ seed };

    // Transform the image using the FibLFSR object
    transform(processedImage, &fibLfsr);

    // Output the photo to the hard disk (output filename)
    if (processedImage.saveToFile(outputFilename)) {
        std::cout << "Successfully output file to: " << outputFilename
                  << std::endl;
    } else {
        std::cout << "Fail to output file to: " << outputFilename << std::endl;
    }

    // Display the original image and the processed image using SFML
    PhotoMagic::displayImages(originalImage, processedImage);

    return 0;
}
```

```
// <PhotoMagic.hpp>
#ifndef PHOTOMAGIC_H
#define PHOTOMAGIC_H

#include <algorithm>
#include <memory>
#include <string>
#include <SFML/Graphics.hpp>
#include "FibLFSR.hpp"

namespace PhotoMagic {

class FibLFSR;

/**
 * @brief Transforms image using FibLFSR. For each pixel (x, y) in row-major
 * order, extra the red, green, and blue components of the color (each component
 * is an integer between 0 and 255). Then XOR the red component with a newly
 * generated 8-bit integer. Do the same for the green (using another new 8-bit
 * integer), and finally the blue. Create a new color using the result of the
 * XOR operations, and set the pixel in the new picture to that color.
 * @param image The image to transform.
 * @param fibLfsr The FibLFSR object to use.
 */
void transform(sf::Image& image, FibLFSR* fibLfsr);

/**
 * @brief Converts an alphanumeric password to a LFSR initial seed.
 * @param password The alphanumeric password to convert.
 */
std::string convertPasswordToSeed(const std::string& password);

/**
 * @brief A struct containing two shared pointers: texture and sprite.
 */
```

```cpp
struct SpriteTexture {
    std::shared_ptr<sf::Texture> texture;
    std::shared_ptr<sf::Sprite> sprite;

    /**
     * @brief Creates a SpriteTexture struct with an image.
     * @param image The image to use.
     */
    explicit SpriteTexture(const sf::Image& image);
};

/**
 * @brief Displays two images with SFML.
 * @param inputImage The input (original) image to display.
 * @param outputImage The output (processed) image to display.
 */
void displayImages(sf::Image& inputImage, sf::Image& outputImage);

}  // namespace PhotoMagic

#endif
```

The **transform** function serves as the cornerstone of this project. Its primary role is to manipulate an image using the Linear Feedback Shift Register (LFSR) established in PS1a. For each pixel within the image, the function conducts an ”XOR” operation between each color component and a randomly generated number from the LFSR. This process ensures the encryption or decryption of the image data, depending on the context.

```cpp
// <PhotoMagic.cpp>
#include "PhotoMagic.hpp"
#include <climits>
#include <string>

namespace PhotoMagic {

void transform(sf::Image& image, FibLFSR* fibLfsr) {
    const sf::Vector2u size = image.getSize();
    for (unsigned int row = 0; row < size.y; ++row) {
        for (unsigned int col = 0; col < size.x; ++col) {
            sf::Color pixel = image.getPixel(col, row);
            pixel.r ^= fibLfsr->generate(8);
            pixel.g ^= fibLfsr->generate(8);
            pixel.b ^= fibLfsr->generate(8);

            image.setPixel(col, row, pixel);
        }
    }
}
```

The **convertPasswordToSeed** function transforms a given password, represented as a string, into a seed string of precisely 16 characters. The function iterates over each character of the password, converting it into an integer, left-shifting it by the remainder of its position divided by 16, and then XORing it with the accumulating result. This process generates a 32-bit integer. Subsequently, this integer is compressed into a 16-bit seed string, where each bit represents either ’1’ or ’0’ based on the parity of corresponding bits in the integer.

```cpp
std::string convertPasswordToSeed(const std::string& password) {
    constexpr size_t SEED_LENGTH = 16;
    unsigned int ans = 0;

    for (size_t i = 0; i < password.length(); ++i) {
        unsigned int t = static_cast<unsigned char>(password[i]);
        t <<= i % SEED_LENGTH;
        ans ^= t;
    }

    std::string seed;
    seed.reserve(SEED_LENGTH);

    for (size_t i = 0; i < SEED_LENGTH; ++i) {
        const unsigned int lowerBit = ans & (1u << i);
        const unsigned int upperBit = ans & (1u << (i + SEED_LENGTH));
        const bool isBitSet = (lowerBit ^ upperBit) == 0;
        seed.push_back(isBitSet ? '1' : '0');
    }

    return seed;
}
```

The **SpriteTexture** constructor initializes both a texture and a sprite by loading the provided image. Following this, the **displayImages** function arranges the input image to the left of the window and the output image to the right, with a small gap separating the two.

```cpp
SpriteTexture::SpriteTexture(const sf::Image& image) {
    texture = std::make_shared<sf::Texture>();
    texture->loadFromImage(image);
    sprite = std::make_shared<sf::Sprite>(*texture);
}

void displayImages(sf::Image& inputImage, sf::Image& outputImage) {
    static constexpr unsigned WINDOW_FPS = 60;

    // Get the size of two images
    const auto inputImageSize = inputImage.getSize();
    const auto outputImageSize = outputImage.getSize();

    // Create two windows for the two images respectively
    sf::RenderWindow inputImageWindow(
        sf::VideoMode(inputImageSize.x, inputImageSize.y), "Input Image");
    sf::RenderWindow outputImageWindow(
        sf::VideoMode(outputImageSize.x, outputImageSize.y), "Output Image");
    inputImageWindow.setFramerateLimit(WINDOW_FPS);
    outputImageWindow.setFramerateLimit(WINDOW_FPS);

    const SpriteTexture inputSpriteTexture(inputImage);
    const SpriteTexture outputSpriteTexture(outputImage);

    // Render images and display windows
    inputImageWindow.clear();
    inputImageWindow.draw(*inputSpriteTexture.sprite);
    inputImageWindow.display();
    outputImageWindow.clear();
    outputImageWindow.draw(*outputSpriteTexture.sprite);
    outputImageWindow.display();

    while (inputImageWindow.isOpen() && outputImageWindow.isOpen()) {
        sf::Event event{};
        while (inputImageWindow.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                inputImageWindow.close();
            }
        }

        while (outputImageWindow.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                outputImageWindow.close();
            }
        }
    }
}

}  // namespace PhotoMagic
```

To validate the code's accuracy, I devised the following test cases:

- **testOutputOperator**: This test assesses the functionality of the output operator << within the **FibLFSR** class. It aims to confirm that the output string matches the seed string if the object remains unaltered.
- **testConvertPasswordToSeed**: This test evaluates the **convertPasswordToSeed** function's return value, ensuring it meets two criteria: a length of 16 and comprising only '0' or '1' characters.
- **testTwoTranformation**: This test examines whether an image remains unchanged after undergoing two consecutive transformations using the same seed. Given that two FibLFSR objects with identical initial seeds always generate identical pseudo-number sequences, and an integer (or bit stream) retains its original value when XORed with the same number twice, this test confirms that an image remains unaltered after undergoing two transformations with the same seed.

```
// <test.cpp>
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main

#include <iostream>
#include <sstream>
#include <string>
#include <boost/test/unit_test.hpp>
#include "FibLFSR.hpp"
#include "PhotoMagic.hpp"

using PhotoMagic::FibLFSR;

BOOST_AUTO_TEST_CASE(testStepInstr) {
    FibLFSR l("1011011000110110");
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 1);
    BOOST_REQUIRE_EQUAL(l.step(), 1);
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 0);
    BOOST_REQUIRE_EQUAL(l.step(), 1);
}

BOOST_AUTO_TEST_CASE(testGenerateInstr) {
    FibLFSR l("1011011000110110");
    BOOST_REQUIRE_EQUAL(l.generate(9), 51);
}

// Test the output operator. The output string should be equal to the seed
// string if the LFSR object is not modified
BOOST_AUTO_TEST_CASE(testOutputOperator) {
    const std::string initialLFSR = "0110110001101100";
    const FibLFSR l(initialLFSR);
    std::stringstream ss;
    ss << l;
    BOOST_REQUIRE_EQUAL(ss.str(), initialLFSR);
}

// The the output operator. The output string should change after calling the
// generate method (k >= 1). The expected LFSR value is
BOOST_AUTO_TEST_CASE(testGenerateAndOutput) {
    const std::string initialLFSR = "0110110001101100";
    const std::string expectedLFSRAfterGenerate = "1101100001100110";
    FibLFSR l(initialLFSR);
    l.generate(9);

    std::stringstream ss;
    ss << l;
    BOOST_REQUIRE_EQUAL(ss.str(), expectedLFSRAfterGenerate);
}

// Test the PhotoMagic::convertPasswordToSeed() method
BOOST_AUTO_TEST_CASE(testConvertPasswordToSeed) {
    const std::string password = "fd79a712hdsa9";
    const std::string seed = PhotoMagic::convertPasswordToSeed(password);

    // Expect the length of the seed to be 16
    BOOST_REQUIRE_EQUAL(seed.length(), PhotoMagic::FibLFSR::SEED_LENGTH);

    // Expect all characters to be either '0' or '1'
    bool isAllBit = true;
    for (const char& c : seed) {
        if (c != '0' && c != '1') {
            isAllBit = false;
            break;
        }
    }
    BOOST_REQUIRE(isAllBit);
}

// An image should be completely the same as the original image after being
// transformed twice by the same seed.
```

```cpp
BOOST_AUTO_TEST_CASE(testTwoTranformation) {
    static const auto* const SEED = "0000111100001111";

    sf::Image image;
    image.loadFromFile("assets/cat.jpg");

    // Transform the image
    FibLFSR fibLfsr{ SEED };
    transform(image, &fibLfsr);

    // Tranform the image back using the same seed (restoring)
    fibLfsr = FibLFSR{ SEED };
    transform(image, &fibLfsr);

    const sf::Image originalImage;
    image.loadFromFile("assets/cat.jpg");

    // Traverse all pixels and check if the restored image is completely equals
    // to the original image
    bool isTwoImagesSame = true;
    const sf::Vector2u size = originalImage.getSize();
    for (unsigned int row = 0; row < size.y; ++row) {
        for (unsigned int col = 0; col < size.x; ++col) {
            const sf::Color originalPixel = originalImage.getPixel(col, row);
            const sf::Color restoredPixel = image.getPixel(col, row);

            if (originalPixel.r != restoredPixel.r ||
                originalPixel.g != restoredPixel.g ||
                originalPixel.b != restoredPixel.b) {
                isTwoImagesSame = false;
                break;
            }
        }
    }
    BOOST_REQUIRE(isTwoImagesSame);
}
```

3 PS2 - Recursive Graphics (Pythagoras Tree)

3.1 Discussion

In this project, I've developed a versatile program enabling users to input up to three arguments: L (the length of the base square), N (the depth of recursion), and optionally A (the angle theta). With these inputs, the program generates a visually captivating Pythagorean tree.

To encapsulate the project's functionalities neatly, I've organized them within a **PTree** namespace. Inside this namespace resides a struct called **Square**, alongside several essential functions: **pTree**, **drawSquare**, **getNextSquares**, and various auxiliary helper functions. These functions have been meticulously documented with comments to elaborate on their purposes, ensuring clarity and ease of understanding for future developers.

From this project, I acquired a deeper understanding of recursion and how to translate mathematical formulas into programming languages. In terms of recursion, I familiarized myself with the typical structure of a recursive function: checking for the base case, performing a task, and recursively invoking itself for the next step.

3.2 Achievements

By running the following command:

```
./PTree 160 9 30
```

We can get the window displayed as shown in **Figure 2.1**.

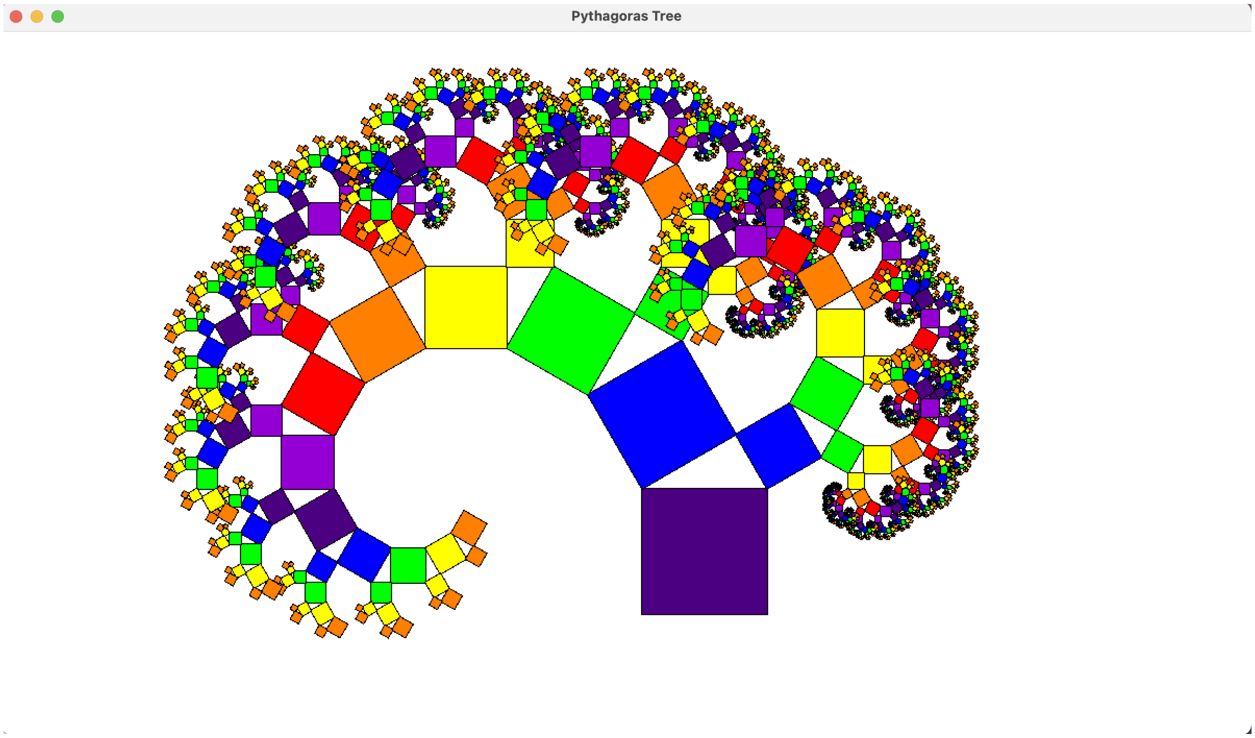

Figure 2.1 The Pythagorean tree displayed by the program.

3.3 Codebase

```
# C++ Compiler
COMPILER = g++

# C++ Flags
#CFLAGS = --std=c++17 -Wall -Werror -pedantic -g -I /opt/homebrew/include
CFLAGS = --std=c++17 -Wall -Werror -pedantic -g

# Libraries
#LIB = -L /opt/homebrew/lib -lsfml-graphics -lsfml-window -lsfml-system
LIB = -lsfml-graphics -lsfml-window -lsfml-system

# Hpp files (dependencies)
DEPS = PTree.hpp

# Cpp files that should be compiled into object files
OBJECTS = PTree.o main.o

# Program
PROGRAM = PTree

.PHONY: all clean lint run

# Generate `PTree`
all: $(PROGRAM)

# Wildcard recipe to make .o files from corresponding .cpp file
%.o: %.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

# Run the PTree program and clean after running it
# Here, $(filter-out $@,$(MAKECMDGOALS)) passes all the arguments given to this
# target. It filters out the target name from the list of goals.
$(PROGRAM): $(OBJECTS)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

run: $(PROGRAM)
        ./$(PROGRAM) 160 9 30 && make clean

# Clean all object files and program files
# "-f" flag refers to "force", which suppresses the "No such file or directory"
# warning
clean:
        rm -f *.o $(PROGRAM)

lint:
        cpplint *.hpp *.cpp
```

The **main** function validates command-line arguments and parses them into variables for the base square's dimensions and recursion parameters. It then creates a window, determines the coordinates of the base square, calls the pTree function to draw the Pythagorean tree, displays the window, and enters the game loop until the window is closed.

```cpp
// <main.cpp>
#include <cerrno>
#include <iostream>
#include <string>
#include <utility>
#include <SFML/Graphics.hpp>
#include "PTree.hpp"

// Global constants
auto WINDOW_TITLE = "Pythagoras Tree";
auto ICON_FILENAME = "./assets/icon.png";
constexpr unsigned WINDOW_FPS = 60;
constexpr int EXIT_CODE = 1;
constexpr int DEFAULT_ANGLE = 45;

/**
 * @brief Parses the two or three arguments.
 * @param LStr The L string to parse.
 * @param NStr The N string to parse.
 * @param AStr (optional) The A string to parse.
 * @return A typle containing parsed L and N.
 */
std::tuple<float, int, float>
parseArguments(const std::string& LStr, const std::string& NStr, const std::string& AStr);

/**
 * @brief
 * @param size The size of the argument list.
 * @param arguments The command line arguments. The first argument is L, and the
 * second one is N. I added
 * @note L: The length of one side of the base square (double).
 * @note N: The depth of the recursion (int).
 * @note A: The angle alpha. (double).
 */
int main(const int size, const char* arguments[]) {
    // Checks the arguments
    if (size < 3 || size > 4) {
        std::cout << "Invalid number of arguments!" << std::endl;
        std::cout << "[argument list]" << std::endl
                  << "(1) L: The length of one side of the base square. (double)" << std::endl
                  << "(2) N: The depth of the recursion. (int)" << std::endl
                  << "(3) A: The angle alpha. (double)" << std::endl;

        return EXIT_CODE;
    }

    // Get L, N, and A
    const auto tuple{ parseArguments(arguments[1], arguments[2], size == 4 ? arguments[3] : "0") };
    const auto L{ std::get<0>(tuple) };
    const auto N{ std::get<1>(tuple) };
    const auto A{ std::get<2>(tuple) };

    // Create a window
    const unsigned windowWidth{ static_cast<unsigned>(L) * 4 + std::min(N, 9) * 80 };
    const unsigned windowHeight{ windowWidth * 10 / 16 };
    sf::RenderWindow window(sf::VideoMode(windowWidth, windowHeight), WINDOW_TITLE);
    window.setFramerateLimit(WINDOW_FPS);
    window.clear(sf::Color::White);

    // Load the icon (extra feature)
    sf::Image icon;
    if (icon.loadFromFile(ICON_FILENAME)) {
        window.setIcon(icon.getSize().x, icon.getSize().y, icon.getPixelsPtr());
    }

    // Determine the coordinates of top-left corner and right-left corner
    const float leftX{ (static_cast<float>(windowWidth) - L) / 2 - (A - 45) * 5 };
    const float rightX{ (static_cast<float>(windowWidth) + L) / 2 - (A - 45) * 5 };
    const float y{ static_cast<float>(windowHeight) * 0.7F };
    const sf::Vector2f baseSquareTlVertex{ leftX, y };
    const sf::Vector2f baseSquareTrVertex{ rightX, y };

    // Call pTree()
    const PTree::Square baseSquare{ baseSquareTlVertex, baseSquareTrVertex, L, 0 };
    pTree(&window, baseSquare, A, N);

    // Display the window
    window.display();

    // Game loop
    while (window.isOpen()) {
        sf::Event event{};
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
            }
        }
    }

    return 0;
}

std::tuple<float, int, float>
parseArguments(const std::string& LStr, const std::string& NStr, const std::string& AStr) {
    float L;
    float A;
    int N;

    try {
        L = static_cast<float>(std::stod(LStr));
    } catch (const std::exception& _) {
        std::cerr << "L should be a double, but " << LStr << " is given";
        exit(EXIT_CODE);
    }

    try {
        N = std::stoi(NStr);
    } catch (const std::exception& _) {
        std::cerr << "N should be an integer, but " << NStr << " is given";
        exit(EXIT_CODE);
    }

    try {
        A = static_cast<float>(std::stod(AStr));
        if (A == 0) {
            A = DEFAULT_ANGLE;
        }
    } catch (const std::exception& _) {
        std::cerr << "A should be a double, but " << LStr << " is given";
        exit(EXIT_CODE);
    }

    return std::make_tuple(L, N, A);
}
```

The **Square** struct stands as the cornerstone of this program, embodying the essence of a square through essential attributes: the coordinates of its top-left and top-right vertices, its side length, and the angle in degrees between its significant side and the horizontal axis. Here, the significant side refers to the line segment connecting the top-left and top-right vertices, encapsulating the square's defining characteristics comprehensively.

The term ”next two squares” denotes the pair of squares derived from the original square within the context of the Pythagorean tree construction. The math derivation of its implementation will be introduced later.

```cpp
// <PTree.hpp>
#ifndef PTREE_H
#define PTREE_H

#include <SFML/Graphics.hpp>

namespace PTree {

/**
 * @brief A square can be unambiguously determined by one vertext, the length of
 * four sides, and the angle between the significant side and the horizontal
 * axis (X axis). For the sake of symmetry, top-left and top-right are stored.
 */
struct Square {
    /**
     * @brief The top-left vertex.
     */
    sf::Vector2f tlVertex;

    /**
     * @brief The top-right vertex.
     */
    sf::Vector2f trVertex;

    /**
     * @brief The length of the side.
     */
    float sideLength{ 0 };

    /**
     * @brief The angle in degrees between the significant side and the
     * horizontal axis.
     */
    float alpha{ 0 };
};

/**
 * @brief Draws a Pythagoras tree recursively.
 * @param window The window to draw onto.
 * @param square The initial square to draw.
 * @param deltaAlpha The difference in alpha.
 * @param N A recursive variable. The function stops recurse when N equals to 1.
 * Every recursion takes 1 from N.
 */
void pTree(sf::RenderWindow* window, const Square& square, const float& deltaAlpha, int N);

/**
 * @brief Draws a square.
 * @param window The window to draw onto.
 * @param square THe square to draw.
 * @param color The fill color of the square.
 */
void drawSquare(sf::RenderWindow* window, const Square& square, const sf::Color& color);

/**
 * @brief Returns the next two squares.
 * @param square The current square.
 * @param deltaAlpha The difference in alpha.
 */
std::array<Square, 2> getNextSquares(const Square& square, const float& deltaAlpha);

/**
 * @brief Converts a degree into radian.
 * @param degree The degree to convert.
 */
inline float degreeToRadian(const float& degree);

/**
 * @brief Finds the sine value of a degree.
 */
inline float sinDeg(const float& degree);

/**
 * @brief Finds the consine value of a degree.
 */
inline float cosDeg(const float& degree);

}  // namespace PTree

#endif
```

The **pTree** function orchestrates the recursive drawing of squares within the Pythagorean tree. Initially, it draws the base square specified by the parameter. Subsequently, it retrieves the next two squares and recursively invokes **pTree** to draw them and their respective offspring.

```
// <PTree.cpp>
#include "PTree.hpp"
#include <cmath>
```

```cpp
#include <iostream>
#include <SFML/Graphics.hpp>

namespace PTree {

void pTree(sf::RenderWindow* window, const Square& square, const float& deltaAlpha, int N) {
    // Fill colors pool
    static constexpr unsigned NUM_FILL_COLOR = 7;
    static const std::array<sf::Color, NUM_FILL_COLOR> FILL_COLORS{
        sf::Color(255, 0, 0), sf::Color(255, 127, 0), sf::Color(255, 255, 0), sf::Color(0, 255, 0),
        sf::Color(0, 0, 255), sf::Color(75, 0, 130),  sf::Color(148, 0, 211),
    };

    // Terminate the recursion when N <= 0
    if (N <= 0) {
        return;
    }

    // Draw the square
    drawSquare(window, square, FILL_COLORS[N % NUM_FILL_COLOR]);

    // Recursion
    if (N > 1) {
        --N;
        const auto nextSquares = getNextSquares(square, deltaAlpha);
        pTree(window, nextSquares[0], deltaAlpha, N);
        pTree(window, nextSquares[1], deltaAlpha, N);
    }
}
```

The **drawSquare** straightforwardly draws a square onto a window using SFML components.

```cpp
void drawSquare(sf::RenderWindow* window, const Square& square, const sf::Color& color) {
    // Create a square shape
    sf::RectangleShape squareShape;
    squareShape.setPosition(square.tlVertex.x, square.tlVertex.y);
    squareShape.setSize(sf::Vector2f(square.sideLength, square.sideLength));
    squareShape.setRotation(-square.alpha);
    squareShape.setFillColor(color);
    squareShape.setOutlineThickness(1);
    squareShape.setOutlineColor(sf::Color::Black);

    // Draw the square shape onto the screen
    window->draw(squareShape);
}
```

The **getNextSquares** is the core algorithm oin this project. The math derivation is as follows:

Assume that α is the angle of the current square, and $\Delta\alpha$ is the increment of α in each recursion. The sides of the following two squares are:

$$S_l = S \cdot \cos(\Delta\alpha) S_r = S \cdot \sin(\Delta\alpha)$$

Where S_l is the side of the relatively left square, and S_r is the side of the relatively right square. Let

$\beta = \alpha + \Delta\alpha$. The vertices of the next two squares are:

$$\begin{aligned}
V_{ll} =& V_l + S_l \cdot (-\sin\beta, -\cos\beta) \\
V_{lr} =& V_l + S_l \cdot (\cos\beta - \sin\beta, -\sin\beta - \cos\beta) \\
V_{rl} =& V_r + S_r \cdot (\cos\beta - \sin\beta, -\sin\beta - \cos\beta) \\
V_{rr} =& V_r + S_r \cdot (\cos\beta, -\sin\beta)
\end{aligned}$$

Where V_l and V_r are two vertices of the current square, V_{ll} and V_{lr} are two vertices of the next left square, and V_{rl} and V_{rr} are two vertices of the next right square.

```cpp
std::array<Square, 2> getNextSquares(const Square& square, const float& deltaAlpha) {
    // Beta in degrees
    const auto beta = square.alpha + deltaAlpha;
    const auto sinBeta = sinDeg(beta);
    const auto cosBeta = cosDeg(beta);

    // Find the next two sides
    const auto leftSideLength = square.sideLength * cosDeg(deltaAlpha);
    const auto rightSideLength = square.sideLength * sinDeg(deltaAlpha);

    // Tranformers
    const sf::Vector2f tlLeftVec{ -sinBeta, -cosBeta };
    const sf::Vector2f trVec{ cosBeta - sinBeta, -sinBeta - cosBeta };
    const sf::Vector2f trRightVec{ +cosBeta, -sinBeta };

    // Vertices
    const sf::Vector2f leftTlVertex = square.tlVertex + leftSideLength * tlLeftVec;
    const sf::Vector2f leftTrVertex = square.tlVertex + leftSideLength * trVec;
    const sf::Vector2f rightTlVertex = square.trVertex + rightSideLength * trVec;
    const sf::Vector2f rightTrVertex = square.trVertex + rightSideLength * trRightVec;

    // Create the next two squares
    const Square leftSquare{ leftTlVertex, leftTrVertex, leftSideLength, beta };
    const Square rightSquare{ rightTlVertex, rightTrVertex, rightSideLength, beta - 90 };

    return { leftSquare, rightSquare };
}
```

The helper functions are straightforward in their implementation, and for efficiency enhancement, the ”inline” modifier has been applied to all three functions.

```cpp
inline float degreeToRadian(const float& degree) {
    static constexpr float C = M_PI / 180;
    return C * degree;
}

inline float sinDeg(const float& degree) { return std::sin(degreeToRadian(degree)); }

inline float cosDeg(const float& degree) { return std::cos(degreeToRadian(degree)); }

}  // namespace PTree
```

4 PS3 - Sokoban

4.1 PS3a

4.1.1 Discussion

In this project, I created a small Sokoban game by leveraging SFML. In ps3a, I have just implemented the functionalities that read contents from a level file and initialize the UI. More functionalities would be implemented in PS3b.

First I defined some constants, including tileset, tile filenames, tile characters, and so on. I also established a **Direction** enumeration to increase the readability and maintainability of the code. Finally, I created a class called **Sokoban**, representing the Sokoban game. It's worth noting that all components related to the Sokoban game are encapsulated within the **SB** namespace.

From this assignment, I gained valuable insights into organizing game resources, commonly referred to as assets, and efficiently loading them into C++ using SFML. Additionally, I learned to utilize these resources effectively, including displaying images and playing sounds within the game environment. In terms of game development, I acquired foundational knowledge about essential concepts such as the game loop, handling game events, and understanding the principles behind tiles and sprites.

4.1.2 Achievements

In PS3a, the game window remains static without any interactive elements, as shown in **Figure 3.1**. Additional functionalities, including movement and gameplay features, are introduced in PS3b.

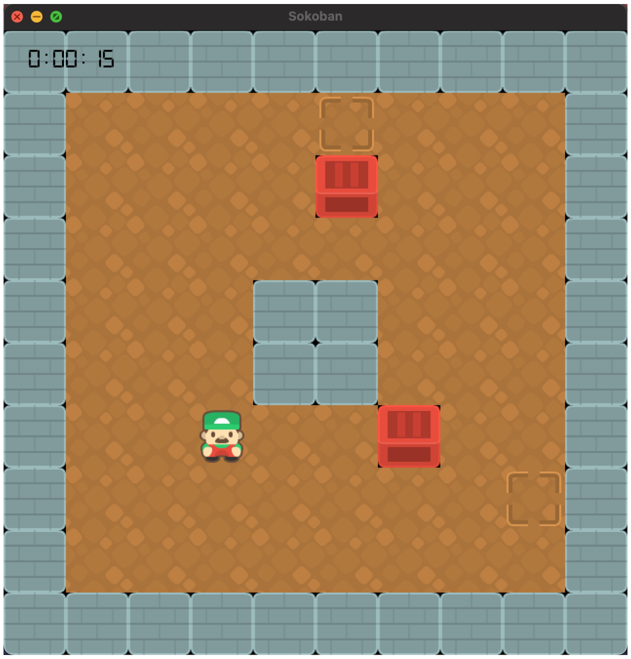

Figure 3.1 The Sokoban game window.

4.1.3 Codebase

```
# C++ Compiler
COMPILER = g++

# C++ Flags
#CFLAGS = --std=c++17 -Wall -Werror -pedantic -g -I /opt/homebrew/include
CFLAGS = --std=c++17 -Wall -Werror -pedantic -g
```

```makefile

# Libraries
#LIB = -L /opt/homebrew/lib -lsfml-graphics -lsfml-window -lsfml-system
LIB = -lsfml-graphics -lsfml-window -lsfml-system

# Code source directory
SRC = ./

# Hpp files (dependencies)
DEPS = ${SRC}Sokoban.hpp

# Object files
OBJECTS = ${SRC}main.o

# The object files that the static library includes
STATIC_LIB_OBJECTS = ${SRC}Sokoban.o

# Static library
STATIC_LIB = Sokoban.a

# Program
PROGRAM = Sokoban

.PHONY: all clean lint run

${SRC}%.o: ${SRC}%.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

$(PROGRAM): $(OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

$(STATIC_LIB): $(STATIC_LIB_OBJECTS)
        ar rcs $@ $^

all: $(PROGRAM)

run: $(PROGRAM)
        ./$(PROGRAM) assets/level/level4.lvl && make clean

clean:
        rm -f ${SRC}*.o $(PROGRAM) $(STATIC_LIB)

lint:
        cpplint *.hpp *.cpp
```

The program accepts a level file as a parameter, instantiates a **Sokoban** object, reads the contents of the level file into the instance, and finally displays it on the window.

```cpp
// <main.cpp>
#include <iostream>
#include "Sokoban.hpp"

int main(const int size, const char* arguments[]) {
    if (size < 2) {
        std::cout << "Too few arguments!" << std::endl;
        return 1;
    }

    const std::string level = arguments[1];
    SB::Sokoban sokoban;
    loadLevel(sokoban, level);

    // Create a window
    const auto windowWidth = sokoban.width() * SB::TILE_WIDTH;
    const auto windowHeight = sokoban.height() * SB::TILE_HEIGHT;
    const auto videoMode = sf::VideoMode(windowWidth, windowHeight);
    sf::RenderWindow window(videoMode, SB::GAME_NAME);
    window.setFramerateLimit(60);

    // Game loop
    sf::Clock clock;
    while (window.isOpen()) {
        sf::Event event{};
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
                break;
            }
        }

        sokoban.update(clock.restart().asMilliseconds());

        if (window.isOpen()) {
            window.clear();
            window.draw(sokoban);
            window.display();
        }
    }
}
```

```
// <Sokoban.hpp>
#ifndef SOKOBAN_H
#define SOKOBAN_H

#include <string>
#include <unordered_map>
#include <vector>
#include <SFML/Graphics.hpp>

/**
 * @brief Sokoban namespace.
 */
namespace SB {

// The name of the game
inline const char* GAME_NAME = "Sokoban";

// The size of each tile
inline constexpr int TILE_HEIGHT = 64;
inline constexpr int TILE_WIDTH = 64;
```

```

// Tileset directory
inline const std::string TILESET_DIR = "assets/tileset/";

// Tiles filename
inline const std::string TILE_BLOCK_06_FILENAME = TILESET_DIR + "block_06.png";
inline const std::string TILE_CRATE_03_FILENAME = TILESET_DIR + "crate_03.png";
inline const std::string TILE_ENVIRONMENT_03_FILENAME = TILESET_DIR + "environment_03.png";
inline const std::string TILE_GROUND_01_FILENAME = TILESET_DIR + "ground_01.png";
inline const std::string TILE_GROUND_04_FILENAME = TILESET_DIR + "ground_04.png";
inline const std::string TILE_PLAYER_05_FILENAME = TILESET_DIR + "player_05.png";
inline const std::string TILE_PLAYER_08_FILENAME = TILESET_DIR + "player_08.png";
inline const std::string TILE_PLAYER_17_FILENAME = TILESET_DIR + "player_17.png";
inline const std::string TILE_PLAYER_20_FILENAME = TILESET_DIR + "player_20.png";

// Tile characters
inline constexpr char TILE_CHAR_PLYAER = '@';
inline constexpr char TILE_CHAR_EMPTY = '.';
inline constexpr char TILE_CHAR_WALL = '#';
inline constexpr char TILE_CHAR_BOX = 'A';
inline constexpr char TILE_CHAR_STORAGE = 'a';
inline constexpr char TILE_CHAR_BOX_STORAGE = '1';

/**
 * @brief A direction enumeration including four directions. The naming convention keeps with
 * SFML's.
 */
enum class Direction { Up, Down, Left, Right };

class Sokoban final : public sf::Drawable {
 public:
    /**
     * @brief Creates a Sokoban object.
     */
    Sokoban();

    /**
     * @brief Deletes textures and tiles.
     */
    ~Sokoban() override;

    /**
     * @brief Returns the width of the game board.
     */
    [[nodiscard]] int width() const;

    /**
     * @brief Returns the height of the game board.
     */
    [[nodiscard]] int height() const;

    /**
     * @brief Returns the players' current position; (0, 0) represents the upper-left cell in the
     * upper-left corner.
     */
    [[nodiscard]] sf::Vector2i playerLoc() const;
```

```

   /**
    * @brief Changes the player's location based on a specified direction.
    * @param direction The direction for the player to move.
    */
   void movePlayer(const Direction& direction);

   /**
    * @brief Returns whether the player has won the game.
    */
   bool isWon();

   /**
    * @brief Updates the game.
    */
   void update(const int& dt);

   /**
    * @brief Reads a map from a level file (.lvl) and loads the content to the sokoban object.
    */
   friend std::ifstream& operator>>(std::ifstream& ifstream, Sokoban& sokoban);

   /**
    * @brief Outputs (or saves) a sokoban game to a level file (.lvl).
    */
   friend std::ofstream& operator<<(std::ofstream& ofstream, const Sokoban& sokoban);

protected:
   void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

private:
   int m_width;
   int m_height;
   sf::Vector2i m_playerLoc;
   std::unordered_map<char, sf::Texture*> tileMap;
   std::unordered_map<Direction, sf::Texture*> playerTextureMap;
   sf::Sprite* player;
   std::vector<sf::Sprite*> tiles;
   unsigned timeElapsedInMs;
   sf::Font m_font;

   /**
    * @brief Returns the tile of a specific coordinate.
    * @param coordinate
    */
   [[nodiscard]] sf::Sprite* getTile(const sf::Vector2i& coordinate) const;

   /**
    * @brief Converts a character to the corresponding sprite.
    */
   [[nodiscard]] sf::Sprite* charToTile(const char& c) const;

   /**
    * @brief Initializes the tile map.
    */
   void initTileMap();
```

```cpp

    /**
     * @brief Initializes the player texture map.
     */
    void initPlayerTextureMap();

    /**
     * @brief Sets the player's orientation.
     * @param direction The direction the player orientente.
     */
    void setPlayerOrientation(const Direction& direction);
};

/**
 * @brief Loads a level from a level file.
 * @param sokoban
 * @param levelFilename
 */
void loadLevel(Sokoban& sokoban, const std::string& levelFilename);

}  // namespace SB

#endif
```

The implementation is quite straightforward overall. However, two functions warrant additional explanation. The **charToTile** function plays a pivotal role. Initially, it translates a character into a tile by utilizing the **tileMap**. Subsequently, it dynamically generates a sprite based on this tile and returns it. The **operator¿¿** overloading function is responsible for reading from a text file. The first line of the file contains the width and height of the Sokoban game, while subsequent lines contain characters representing the types of tiles at specific positions within the game grid. For each character, **charToTile** is called to create a corresponding sprite.

```cpp
// <Sokoban.cpp>
#include "Sokoban.hpp"
#include <fstream>
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

namespace SB {

Sokoban::Sokoban() :
    m_width(0), m_height(0), m_playerLoc({ 0, 0 }), player(nullptr), timeElapsedInMs(0) {
    initTileMap();
    initPlayerTextureMap();

    // The player orientates downward at the beginning
    setPlayerOrientation(Direction::Down);

    // Initailizes font
    m_font.loadFromFile("assets/font/digital-7.mono.ttf");
}

Sokoban::˜Sokoban() {
    // Delete tiles in tiles vector
    for (const auto& tile : tiles) {
        delete tile;
    }
    tiles.clear();

    // Delete player textures
    for (auto& [name, playerTexture] : playerTextureMap) {
        delete playerTexture;
    }
    playerTextureMap.clear();

    // Delete player sprite
    delete player;
}

int Sokoban::width() const { return m_width; }

int Sokoban::height() const { return m_height; }

sf::Vector2i Sokoban::playerLoc() const { return m_playerLoc; }

// ReSharper disable once CppMemberFunctionMayBeStatic
void Sokoban::movePlayer(const Direction& direction) {}

// ReSharper disable once CppMemberFunctionMayBeStatic
bool Sokoban::isWon() { return false; }

void Sokoban::update(const int& dt) { timeElapsedInMs += dt; }

sf::Sprite* Sokoban::getTile(const sf::Vector2i& coordinate) const {
    const int index = coordinate.x + coordinate.y * m_width;
    return tiles.at(index);
}

void Sokoban::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    // Draw tiles
    for (int row = 0; row < m_height; ++row) {
        for (int col = 0; col < m_width; ++col) {
            auto* const tile = getTile({ col, row });
            if (tile == nullptr) {
                continue;
            }

            auto position = sf::Vector2f{ static_cast<float>(col * TILE_WIDTH),
                                          static_cast<float>(row * TILE_HEIGHT) };
            tile->setPosition(position);
            target.draw(*tile);
        }
    }

    // Draw the player
    player->setPosition({
        static_cast<float>(m_playerLoc.x * TILE_WIDTH),
        static_cast<float>(m_playerLoc.y * TILE_HEIGHT),
    });
    target.draw(*player);

    // Draw the elapsed time for extra credit
    const unsigned seconds = timeElapsedInMs / 1000U;
    const unsigned minutes = seconds / 60U;
    const unsigned hours = minutes / 60U;
    const unsigned second = seconds % 60U;
    const unsigned minute = minutes % 60U;
    const std::string secondStr = (second < 10 ? "0" : "") + std::to_string(second);
    const std::string minuteStr = (minute < 10 ? "0" : "") + std::to_string(minute);
    const std::string hourStr = std::to_string(hours);
    const std::string stringToPrint = hourStr + ":" + minuteStr + ":" + secondStr;

    // Create a text and draw it on the target window
    sf::Text text;
    text.setFont(m_font);
    text.setString(stringToPrint);
    text.setCharacterSize(28);
    text.setFillColor(sf::Color::Black);
    text.setPosition(25, 10);
    target.draw(text);
}

std::ifstream& operator>>(std::ifstream& ifstream, Sokoban& sokoban) {
    // Clear all tiles
    for (auto& tile : sokoban.tiles) {
        delete tile;
        tile = nullptr;
    }
    sokoban.tiles.clear();

    // The first line consists of height and width
    ifstream >> sokoban.m_height >> sokoban.m_width;

    // Continue the read the following lines
    std::string line;
    getline(ifstream, line);
    for (int row = 0; row < sokoban.m_height; ++row) {
        getline(ifstream, line);
        for (int col = 0; col < sokoban.m_width; ++col) {
            const char c = line.at(col);
            auto* const tile = sokoban.charToTile(c);
            if (tile == nullptr) {
                continue;
            }

            sokoban.tiles.push_back(tile);
            if (c == TILE_CHAR_PLYAER) {
                sokoban.m_playerLoc = { col, row };
            }
        }
    }

    return ifstream;
}

```

```
std::ofstream& operator<<(std::ofstream& ofstream, const Sokoban& sokoban) { return ofstream; }

void loadLevel(Sokoban& sokoban, const std::string& levelFilename) {
    std::ifstream ifstream{ levelFilename };
    if (!ifstream.is_open()) {
        throw std::invalid_argument("File not found: " + levelFilename);
    }

    ifstream >> sokoban;
}

sf::Sprite* Sokoban::charToTile(const char& c) const {
    const auto it = tileMap.find(c);
    if (it == tileMap.end()) {
        return nullptr;
    }

    auto* const texture{ it->second };
    auto* const sprite{ new sf::Sprite };
    sprite->setTexture(*texture);

    return sprite;
}

void Sokoban::initTileMap() {
    auto* const groundTexture{ new sf::Texture };
    auto* const groundStorageTexture{ new sf::Texture };
    auto* const crateTexture{ new sf::Texture };
    auto* const wallTexture{ new sf::Texture };

    groundTexture->loadFromFile(TILE_GROUND_01_FILENAME);
    groundStorageTexture->loadFromFile(TILE_GROUND_04_FILENAME);
    crateTexture->loadFromFile(TILE_CRATE_03_FILENAME);
    wallTexture->loadFromFile(TILE_BLOCK_06_FILENAME);

    tileMap[TILE_CHAR_PLYAER] = groundTexture;
    tileMap[TILE_CHAR_EMPTY] = groundTexture;
    tileMap[TILE_CHAR_WALL] = wallTexture;
    tileMap[TILE_CHAR_BOX] = crateTexture;
    tileMap[TILE_CHAR_STORAGE] = groundStorageTexture;
    tileMap[TILE_CHAR_BOX_STORAGE] = crateTexture;
}

void Sokoban::initPlayerTextureMap() {
    auto* const playerUpTexture{ new sf::Texture };
    auto* const playerRightTexture{ new sf::Texture };
    auto* const playerDownTexture{ new sf::Texture };
    auto* const playerLeftTexture{ new sf::Texture };

    playerUpTexture->loadFromFile(TILE_PLAYER_08_FILENAME);
    playerRightTexture->loadFromFile(TILE_PLAYER_17_FILENAME);
    playerDownTexture->loadFromFile(TILE_PLAYER_05_FILENAME);
    playerLeftTexture->loadFromFile(TILE_PLAYER_20_FILENAME);

    playerTextureMap[Direction::Up] = playerUpTexture;
    playerTextureMap[Direction::Right] = playerRightTexture;
    playerTextureMap[Direction::Down] = playerDownTexture;
    playerTextureMap[Direction::Left] = playerLeftTexture;
}

void Sokoban::setPlayerOrientation(const Direction& direction) {
    delete player;

    auto* const texture{ playerTextureMap.at(direction) };
    auto* const sprite{ new sf::Sprite };
    sprite->setTexture(*texture);
    player = sprite;
}

}  // namespace SB
```

4.2 PS3b

4.2.1 Discussion

In PS3b, I've expanded upon the functionalities and gameplay features introduced in PS3a. To enhance readability and maintainability, I've devised a modular architecture that organizes the codebase into four distinct modules:

- **SokobanElapsedTime**: This module oversees the elapsed time system within Sokoban. It is responsible for rendering the elapsed time on the screen.
- **SokobanPlayer**: This module stores essential player information such as sprite, location, and orientation, and is responsible for rendering the player on the screen.
- **SokobanScore**: This module handles the scoring system within Sokoban, where the score corresponds to the number of boxes correctly stowed. It renders the score and maximum score on the screen.
- **SokobanTileGrid**: Responsible for managing the tile grid system within Sokoban, this module is tasked with rendering the tile grid on the screen.

The four modules correspond to four distinct classes, each derived from **sf::Drawable**. All of these classes are extended by the overarching **Sokoban** class.

The Sokoban game has following features:

- **Classic Sokoban Gameplay**: Players can navigate the game grid using either the "WASD" keys or the arrow keys. Players cannot move through walls obstructing their path. They can push boxes onto designated targeted locations, encountering limitations such as being unable to move if facing a wall block, unable to push a box if it abuts a wall or another box blocks its path from behind.
- **Elapsed Time**: In the upper-left corner of the game screen, players will find a stopwatch recording the elapsed time. Strategically, the less time a player utilizes to complete a level, the higher their final score will soar upon victory.
- **Box and Storage Numbers**: In the upper-right corner of the game screen, players will notice a pair of numbers separated by a slash. The first number denotes the count of boxes successfully placed in storage units. In contrast, the second number indicates the maximum capacity of the storage units, calculated using the formula 'min(numberOfBoxes, numberOfStorages)'. As the game progresses, aligning these numbers signals the player's path to victory: when the counts are equal, indicating all boxes are securely stowed, the player achieves victory.
- **Undo and Restart**: Players can undo their moves during gameplay without halting the progress of the elapsed time. Additionally, they can restart the entire game, reverting the map to its initial state and resetting the elapsed time in the process.
- **Result Screen**: After achieving victory, players encounter a result screen centered with the message 'You win!' prominently displayed. Below, a final score awaits, determined by the player's efficiency in time and moves: the quicker and fewer moves taken, the higher the score attained. The player cannot move after winning, but they can restart the game and start a new attempt.
- **Music and Sound Effect**: At the beginning of the game, background music sets the tone, persisting until the player secures victory. Upon winning, players are greeted with a celebratory sound effect, marking their triumphant accomplishment.

In PS3b, I expanded my practical understanding of game development significantly. Firstly, I delved into the techniques for enabling character movement based on player input, understanding the mechanics behind key presses. Secondly, I gained insight into collision detection, crucial for preventing players from traversing through solid wall blocks. Most notably, I honed my skills in constructing class hierarchies, with a particular focus on leveraging the "virtual" keyword to implement polymorphism effectively.

4.2.2 Achievements

Run the Sokoban game with the level file containing the following content:

```
9 8
..#####.
###...#.
#a@A..#.
###.Aa#.
#a##A.#.
#.#.a.##
#A..AAa#
#...a..#
########

See: https://en.wikipedia.org/wiki/Sokoban
```

Upon initiating the game, a window akin to that depicted in **Figure 3.2** emerges. Upon successfully stowing away all the boxes, the game unveils a result screen akin to the one portrayed in **Figure 3.3**.

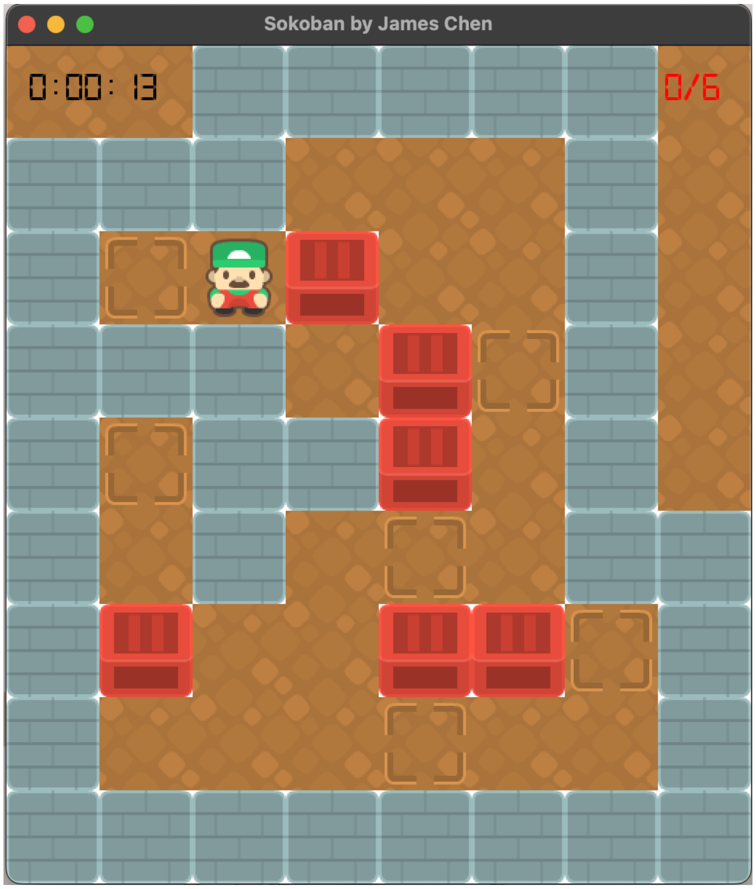

Figure 3.3 A snapshot showcasing the initial stage of the Sokoban game.

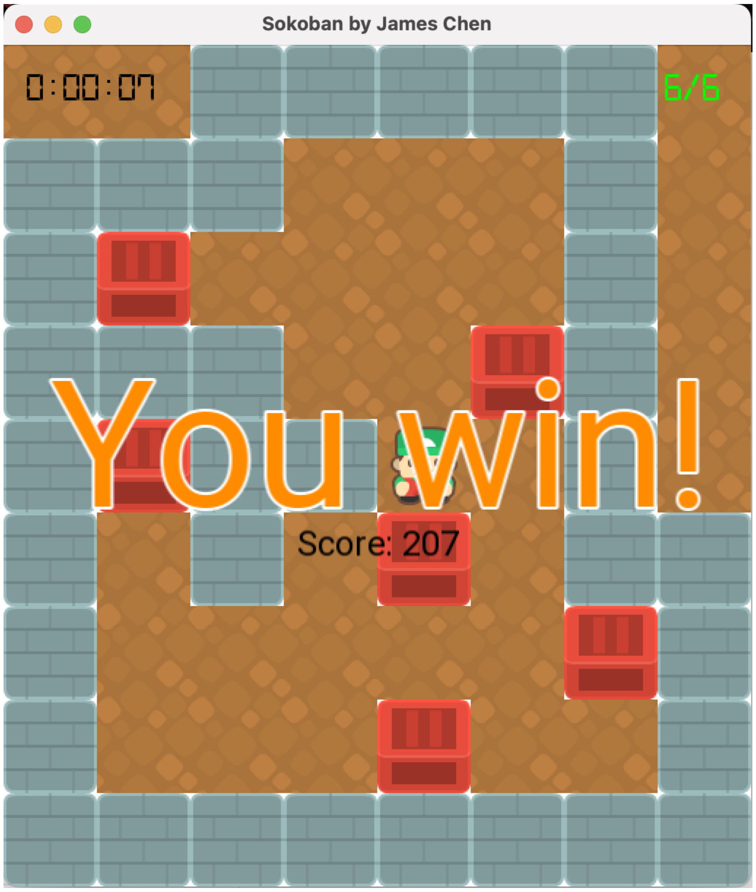

Figure 3.3 A snapshot showcasing the result screen of the Sokoban game.

4.2.3 Codebase

```
# C++ Compiler
COMPILER = g++

# C++ Flags
CFLAGS = --std=c++20 -Wall -Werror -pedantic -g

# Libraries
LIB = -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -lboost_unit_test_framework

# Code source directory
SRC = ./

# Hpp files (dependencies)
DEPS = $(SRC)Sokoban.hpp \
       $(SRC)SokobanConstants.hpp \
       $(SRC)SokobanTileGrid.hpp \
       $(SRC)SokobanPlayer.hpp \
       $(SRC)SokobanScore.hpp \
       $(SRC)SokobanElapsedTime.hpp \
       $(SRC)InvalidCoordinateException.hpp \

# Object files that are not in the static library
OBJECTS = $(SRC)main.o

# The object files that the static library includes
STATIC_LIB_OBJECTS = $(SRC)Sokoban.o \
                                         $(SRC)SokobanTileGrid.o \
                                         $(SRC)SokobanPlayer.o \
                                         $(SRC)SokobanScore.o \
                                          $(SRC)SokobanElapsedTime.o \
                                          $(SRC)InvalidCoordinateException.o \

# Static library
STATIC_LIB = Sokoban.a

# Program
PROGRAM = Sokoban

# The test object files
TEST_OBJECTS = $(SRC)test.o

# The test program
TEST_PROGRAM = test

.PHONY: all clean lint

all: $(TEST_PROGRAM) $(PROGRAM)

$(SRC)%.o: $(SRC)%.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

$(PROGRAM): $(OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

```

```
$(STATIC_LIB): $(STATIC_LIB_OBJECTS)
        ar rcs $@ $^

$(TEST_PROGRAM): $(TEST_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

# Run unit test
boost: $(TEST_PROGRAM)
        ./$<

run: $(PROGRAM)
        ./$< assets/level/level7.lvl

clean:
        rm -f $(SRC)*.o $(PROGRAM) $(STATIC_LIB) $(TEST_PROGRAM)

lint:
        cpplint *.hpp *.cpp
```

```cpp
// <main.cpp>
#include <fstream>
#include <iostream>
#include "Sokoban.hpp"

/**
 * @brief Starts a Sokoban game.
 * @param size The size of the argument list.
 * @param arguments The command line arguments. This game requires one argument, which is the
 * filename of the level file to load.
 */
int main(const int size, const char* arguments[]) {
    // Check arguments
    if (size < 2) {
        std::cout << "Too few arguments! Require the filename of the level file." << std::endl;
        return 1;
    }

    // Create a Sokoban game object and load the level file
    const std::string levelFilename{ arguments[1] };
    SB::Sokoban sokoban{ levelFilename };

    // Create a window based on the Sokoban game width and height
    const auto windowWidth{ sokoban.width() * SB::TILE_WIDTH };
    const auto windowHeight{ sokoban.height() * SB::TILE_HEIGHT };
    const auto windowVideoMode{ sf::VideoMode(windowWidth, windowHeight) };
    const auto windowTitle = SB::GAME_NAME + " by " + SB::AUTHOR_NAME;
    sf::RenderWindow window(windowVideoMode, windowTitle);
    window.setFramerateLimit(60);

    // Create a map that binds keyboard keys to directions for the player to move
    // Initializer list syntax
    const std::unordered_map<const sf::Keyboard::Key, SB::Direction> movePlayerKeyMap{
        { sf::Keyboard::Key::W, SB::Direction::Up },
        { sf::Keyboard::Key::A, SB::Direction::Left },
        { sf::Keyboard::Key::S, SB::Direction::Down },
        { sf::Keyboard::Key::D, SB::Direction::Right },
        { sf::Keyboard::Key::Up, SB::Direction::Up },
        { sf::Keyboard::Key::Left, SB::Direction::Left },
        { sf::Keyboard::Key::Down, SB::Direction::Down },
        { sf::Keyboard::Key::Right, SB::Direction::Right }
    };

    // Game loop
    sf::Clock clock;
    while (window.isOpen()) {
        sf::Event event{};
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
                break;
            }

            // Listen to keypress event
            if (event.type == sf::Event::KeyPressed) {
                // Move player
                const auto itDirection = movePlayerKeyMap.find(event.key.code);
                if (itDirection != movePlayerKeyMap.end()) {
                    sokoban.movePlayer(itDirection->second);
                }

                // Reset the game
                if (event.key.code == sf::Keyboard::R) {
                    sokoban.reset();
                }

                // Undo a move
                if (event.key.code == sf::Keyboard::U) {
                    sokoban.undo();
                }
            }
        }

        sokoban.update(clock.restart().asMicroseconds());

        if (window.isOpen()) {
            window.clear(sf::Color::White);
            window.draw(sokoban);
            window.display();
        }
    }
}
```

Here are the four module classes. Within **SokobanElapsedTime**, there is a function called **update**, which is invoked every frame to update the elapsed time. This class draws the elapsed time in seconds onto the window's upper-left corner.

```
// <SokobanElapsedTime.hpp>
#ifndef SOKOBANELAPSEDTIME_H
#define SOKOBANELAPSEDTIME_H

```

```cpp
#include <SFML/Graphics.hpp>

namespace SB {

/**
 * @brief This class manages the elapsed time system in Sokoban.
 */
class SokobanElapsedTime : virtual public sf::Drawable {
 public:
    /**
     * @brief Creates a SokobanElapsedTime instance: initializes the font.
     */
    SokobanElapsedTime();

    /**
     * @brief Updates the game in a game frame. This will update the elapsed time.
     * @param dt The delta time in microseconds between this frame and the previous frame.
     */
    virtual void update(const int64_t& dt);

 protected:
    /**
     * @brief Draws the elapsed time in the format of "H:MM:SS" in the upper-left corner.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

    /**
     * @brief The elasped time in microseconds.
     */
    int64_t m_elapsedTimeInMicroseconds = 0;

    /**
     * @brief The font for the diplayed text.
     */
    sf::Font m_font;
};

}  // namespace SB

#endif
```

```cpp
// <SokobanElapsedTime.cpp>
#include "SokobanElapsedTime.hpp"
#include <string>
#include "SokobanConstants.hpp"

namespace SB {

SokobanElapsedTime::SokobanElapsedTime() { m_font.loadFromFile(FONT_DIGITAL7_FILENAME); }

void SokobanElapsedTime::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    const unsigned seconds = m_elapsedTimeInMicroseconds / 1000000u;
    const unsigned minutes = seconds / 60u;
    const unsigned hours = minutes / 60u;
    const unsigned second = seconds % 60u;
    const unsigned minute = minutes % 60u;
    const std::string secondStr = (second < 10 ? "0" : "") + std::to_string(second);
    const std::string minuteStr = (minute < 10 ? "0" : "") + std::to_string(minute);
    const std::string hourStr = std::to_string(hours);
    const std::string stringToPrint = hourStr + ":" + minuteStr + ":" + secondStr;

    sf::Text text;
    text.setFont(m_font);
    text.setString(stringToPrint);
    text.setCharacterSize(28);
    text.setFillColor(sf::Color::Black);
    text.setPosition(15, 10);
    target.draw(text);
}

void SokobanElapsedTime::update(const int64_t& dt) { m_elapsedTimeInMicroseconds += dt; }

}  // namespace SB
```

In the **SokobanPlayer** class, the **playLoc** method retrieves the player's current location. Derived classes are responsible for implementing the logic to update the player's location based on game events. Furthermore, I've introduced a **m_playerTextureMap** to facilitate mapping directions to player textures. This enables the display of different textures corresponding to the player's orientation. This class draws the player onto the window.

```cpp
// <SokobanPlayer.hpp>
#ifndef SOKOBANPLAYER_HPP
#define SOKOBANPLAYER_HPP

#include <memory>
#include <unordered_map>
#include <SFML/Graphics.hpp>
#include "SokobanConstants.hpp"

namespace SB {

/**
 * @brief This class manages the player system in Sokoban.
 */
class SokobanPlayer : public virtual sf::Drawable {
 public:
    /**
     * @brief Returns the players' current position; (0, 0) represents the upper-left cell in the
     * upper-left corner.
     */
    [[nodiscard]] sf::Vector2u playerLoc() const;

 protected:
    /**
     * @brief Creates a SokobanPlayer instance; initalizes the player texture map and the player
     * sprite map.
     */
    SokobanPlayer();

    /**
```

```
     * @brief Draws the player onto the target.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;


    /**
     * @brief The player's default orientation.
     */
    inline static Direction DEFAULT_ORIENTATION = Direction::Down;

    /**
     * @brief Player location. Note the unit of this coordinate is tile instead of pixel.
     */
    sf::Vector2i m_playerLoc = { 0, 0 };

    /**
     * @brief Associates each direction with the corresponding player texture.
     */
    std::unordered_map<Direction, std::shared_ptr<sf::Texture>> m_playerTextureMap;

    /**
     * @brief Associates each direction with the corresponding player sprite. Player sprites vary
     * depending on the orientation.
     */
    std::unordered_map<Direction, std::shared_ptr<sf::Sprite>> m_playerSpriteMap;

    /**
     * @brief Player's current orientation. The default orientation is down.
     */
    Direction m_playerOrientation = DEFAULT_ORIENTATION;
};

}  // namespace SB

#endif
```

```
// <SokobanPlayer.cpp>
#include "SokobanPlayer.hpp"
#include <memory>

namespace SB {

SokobanPlayer::SokobanPlayer() {
    const auto playerUpTexture{ std::make_shared<sf::Texture>() };
    const auto playerRightTexture{ std::make_shared<sf::Texture>() };
    const auto playerDownTexture{ std::make_shared<sf::Texture>() };
    const auto playerLeftTexture{ std::make_shared<sf::Texture>() };
    playerUpTexture->loadFromFile(TILE_PLAYER_08_FILENAME);
    playerRightTexture->loadFromFile(TILE_PLAYER_17_FILENAME);
    playerDownTexture->loadFromFile(TILE_PLAYER_05_FILENAME);
    playerLeftTexture->loadFromFile(TILE_PLAYER_20_FILENAME);

    m_playerTextureMap[Direction::Up] = { playerUpTexture };
    m_playerTextureMap[Direction::Right] = { playerRightTexture };
    m_playerTextureMap[Direction::Down] = { playerDownTexture };
    m_playerTextureMap[Direction::Left] = { playerLeftTexture };

    const auto playerUpSprite{ std::make_shared<sf::Sprite>() };
    const auto playerRightSprite{ std::make_shared<sf::Sprite>() };
    const auto playerDownSprite{ std::make_shared<sf::Sprite>() };
    const auto playerLeftSprite{ std::make_shared<sf::Sprite>() };
    playerUpSprite->setTexture(*playerUpTexture);
    playerRightSprite->setTexture(*playerRightTexture);
    playerDownSprite->setTexture(*playerDownTexture);
    playerLeftSprite->setTexture(*playerLeftTexture);

    m_playerSpriteMap[Direction::Up] = playerUpSprite;
    m_playerSpriteMap[Direction::Right] = playerRightSprite;
    m_playerSpriteMap[Direction::Down] = playerDownSprite;
    m_playerSpriteMap[Direction::Left] = playerLeftSprite;
}

void SokobanPlayer::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    const auto player = m_playerSpriteMap.at(m_playerOrientation);
    player->setPosition({
        static_cast<float>(m_playerLoc.x * TILE_WIDTH),
        static_cast<float>(m_playerLoc.y * TILE_HEIGHT),
    });
    target.draw(*player);
}

sf::Vector2u SokobanPlayer::playerLoc() const {
    return { static_cast<unsigned>(m_playerLoc.x), static_cast<unsigned>(m_playerLoc.y) };
}

}  // namespace SB
```

Within the **SokobanScore** class, the **isWon** method determines whether the player has won the game. This is done by comparing the current score (**m_score**) with the maximum score (**m_maxScore**). The current score (**m_score**) represents the number of boxes successfully stashed in storage locations. The maximum score (**m_maxScore**), on the other hand, is calculated as the minimum of the storage count and box count, added to the number of boxes already stashed:

$$\text{MAX\_SCORE} = min\{\#\text{Storage}, \#\text{Box}\} + \#\text{BoxStorage}$$

This class draws the score and max score on the window's upper-right corner.

```
// <SokobanScore.hpp>
#ifndef SOKOBANSCORE_HPP
#define SOKOBANSCORE_HPP

#include <SFML/Graphics.hpp>

namespace SB {

/**
 * @brief This class manages the score system in Sokoban.
 */
class SokobanScore : public virtual sf::Drawable {
 public:
```

```cpp
    /**
     * @brief Creates a SokobanScore instance; initializes the font.
     */
    SokobanScore();

    /**
     * @brief Checks if the player has won the game.
     * @return True if the player has won the game; false otherwise.
     */
    bool isWon() const;

 protected:
    /**
     * @brief Draws the score and the max score onto the target.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

    /**
     * @brief The player's current score. Players get one score when they successfully put a box to
     * a storage. In a word, the score is equals to the number of "StorageBox" block in the map.
     */
    int m_score = 0;

    /**
     * @brief The player's max score in the current level. The player wins the game when the score
     * equals the max score.
     */
    int m_maxScore = 1;

    /**
     * @brief The font for the diplayed text.
     */
    sf::Font m_font;
};

}  // namespace SB

#endif
```

```cpp
// <SokobanScore.cpp>
#include "SokobanScore.hpp"
#include <string>
#include "SokobanConstants.hpp"

namespace SB {

SokobanScore::SokobanScore() { m_font.loadFromFile(FONT_DIGITAL7_FILENAME); }

bool SokobanScore::isWon() const { return m_score == m_maxScore; }

void SokobanScore::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    const std::string stringToPrint = std::to_string(m_score) + "/" + std::to_string(m_maxScore);

    sf::Text text;
    text.setFont(m_font);
    text.setString(stringToPrint);
    text.setCharacterSize(28);
    text.setFillColor(isWon() ? sf::Color::Green : sf::Color::Red);
    text.setPosition(target.getSize().x - 60, 10);
    target.draw(text);
}

}  // namespace SB
```

Before delving into the **SokobanTileGrid** class, it's essential to clarify two terms:

- **Tile character**: A character that can be mapped to a corresponding tile.
- **Tile**: A sprite representing a cell in the Sokoban game.

The **m_initialTileCharGrid** stores the initial grid of tile characters when reading the level file. The **m_tileCharGrid** is subsequently updated as the player moves. Additionally, the class includes a **getTileChar** function, which retrieves the tile character at a specified coordinate.

```
// <SokobanTileGrid.hpp>
#ifndef SOKOBANTILEGRID_HPP
#define SOKOBANTILEGRID_HPP

#include <memory>
#include <unordered_map>
#include <vector>
#include <SFML/Graphics.hpp>
#include "SokobanConstants.hpp"

namespace SB {

/**
 * @brief This class manages the tile grid system in Sokoban. Tiles includes the unmovable things in
 * the game, inlcuding wall blocks, ground blocks, box blocks, and so on. Note the player is not
 * included in tiles.
 */
class SokobanTileGrid : public virtual sf::Drawable {
 public:
    /**
     * @brief Returns the width of the game board, which is the number of tile columns.
     */
    [[nodiscard]] int width() const;

    /**
     * @brief Returns the height of the game board, which is the number of the tile rows.
     */
    [[nodiscard]] int height() const;

    /**
     * @brief Returns the tile character at a specified coordinate.
     * @param coordinate The coordinate of the tile character to get.
     */
    [[nodiscard]] TileChar getTileChar(const sf::Vector2i& coordinate) const;

 protected:
    /**
     * @brief Creates a SokobanTileGrid instance; initalizes the tile texture map.
```

```cpp
     */
    SokobanTileGrid();

    /**
     * @brief draws the tile grid onto the target.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

    /**
     * @brief Returns the corresponding index of a specified coordiante.
     * @param coordinate Coordinate to analyze.
     */
    [[nodiscard]] int getIndex(const sf::Vector2i& coordinate) const;

    /**
     * @brief Sets the tile character for a specified coordiante. If the tile character changes, the
     * corresponding tile in the tile grid changes synchronously.
     * @param coordinate The coordiante of the tile character to set.
     * @param tileChar The tile character to set.
     */
    void setTileChar(const sf::Vector2i& coordinate, TileChar tileChar);

    /**
     * @brief Iterates over each tile character in the grid and invokes the specified callback
     * function for each tile, providing the tile's coordinate and its associated tile character.
     * The callback function should return true to continue the traversal or false to stop it.
     * @param callback The callback function is to be invoked for each tile.
     */
    void traverseTileCharGrid(const std::function<bool(sf::Vector2i, TileChar)>& callback) const;

    /**
     * @brief Converts a character into the corresponding tile sprite.
     * @return The corresponding tile sprite; nullptr if the tile char is not supported.
     */
    [[nodiscard]] std::shared_ptr<sf::Sprite> getTile(const TileChar& tileChar) const;

    /**
     * @brief The number of tile columns.
     */
    int m_width = 0;

    /**
     * @brief The number of tile rows.
     */
    int m_height = 0;

    /**
     * @brief Associates characters with their respective tile texture. Refer to
     * `SokobanConstants.h` for additional details. This mapping is crucial for constructing the
     * sprite grid.
     */
    std::unordered_map<TileChar, std::shared_ptr<sf::Texture>> m_tileTextureMap;

    /**
     * @brief The initial tile char grid. It is unchanged until the level changes.
     */
    std::vector<TileChar> m_initialTileCharGrid;

    /**
     * @brief Represents the tile character grid, which is mapping into an one-dimentional array in
     * row-major order.
     */
    std::vector<TileChar> m_tileCharGrid;

    /**
     * @brief Represents the tile grid, which is mapping into an one-dimentional array in row-major
     * order.
     */
    std::vector<std::shared_ptr<sf::Sprite>> m_tileGrid;

 private:
    /**
     * @brief Checks if a specified coordinate is valid. A valid coordiante should be able to be
     * located in the tile char grid.
     * @param coordinate The coordinate to check.
     * @return A index corresponding to the coordinate.
     * @throws InvalidCoordinateException if the coordinate is invalid.
     */
    [[nodiscard]] int checkCoordinate(const sf::Vector2i& coordinate) const;
};

}  // namespace SB

#endif
```

```
// <SokobanTileGrid.cpp>
#include "SokobanTileGrid.hpp"
#include <memory>
#include <SFML/Graphics.hpp>
#include "InvalidCoordinateException.hpp"

namespace SB {

SokobanTileGrid::SokobanTileGrid() {
    const auto groundTexture{ std::make_shared<sf::Texture>() };
    const auto groundStorageTexture{ std::make_shared<sf::Texture>() };
    const auto boxTexture{ std::make_shared<sf::Texture>() };
    const auto wallTexture{ std::make_shared<sf::Texture>() };
    groundTexture->loadFromFile(TILE_GROUND_01_FILENAME);
    groundStorageTexture->loadFromFile(TILE_GROUND_04_FILENAME);
    boxTexture->loadFromFile(TILE_CRATE_03_FILENAME);
    wallTexture->loadFromFile(TILE_BLOCK_06_FILENAME);

    m_tileTextureMap[TileChar::Player] = { groundTexture };
    m_tileTextureMap[TileChar::Empty] = { groundTexture };
    m_tileTextureMap[TileChar::Wall] = { wallTexture };
    m_tileTextureMap[TileChar::Box] = { boxTexture };
    m_tileTextureMap[TileChar::Storage] = { groundStorageTexture };
    m_tileTextureMap[TileChar::BoxStorage] = { boxTexture };
}

```

```
void SokobanTileGrid::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    traverseTileCharGrid([&](auto coordinate, auto tileChar) {
        const sf::Vector2f position({ static_cast<float>(coordinate.x * TILE_WIDTH),
                                      static_cast<float>(coordinate.y * TILE_HEIGHT) });
        const auto tile = getTile(tileChar);
        tile->setPosition(position);
        target.draw(*tile);

        return false;
    });
}

int SokobanTileGrid::getIndex(const sf::Vector2i& coordinate) const {
    return coordinate.x + coordinate.y * m_width;
}

int SokobanTileGrid::height() const { return m_height; }

int SokobanTileGrid::width() const { return m_width; }

TileChar SokobanTileGrid::getTileChar(const sf::Vector2i& coordinate) const {
    return m_tileCharGrid.at(checkCoordinate(coordinate));
}

void SokobanTileGrid::setTileChar(const sf::Vector2i& coordinate, const TileChar tileChar) {
    const auto index = checkCoordinate(coordinate);
    const auto originalTileChar = m_tileCharGrid[index];

    if (originalTileChar != tileChar) {
        m_tileCharGrid[index] = tileChar;
        const auto newTile{ std::make_shared<sf::Sprite>() };
        newTile->setTexture(*m_tileTextureMap.at(tileChar));
    }
}

void SokobanTileGrid::traverseTileCharGrid(
    const std::function<bool(sf::Vector2i, TileChar)>& callback) const {
    auto stopIteration = false;
    for (int row = 0; !stopIteration && row < m_height; ++row) {
        for (int col = 0; !stopIteration && col < m_width; ++col) {
            const auto tileChar = getTileChar({ col, row });
            stopIteration = callback({ col, row }, tileChar);
        }
    }
}

std::shared_ptr<sf::Sprite> SokobanTileGrid::getTile(const TileChar& tileChar) const {
    const auto it = m_tileTextureMap.find(tileChar);
    if (it == m_tileTextureMap.end()) {
        return nullptr;
    }

    const auto sprite{ std::make_shared<sf::Sprite>() };
    sprite->setTexture(*it->second);

    return sprite;
```

```cpp
}

int SokobanTileGrid::checkCoordinate(const sf::Vector2i& coordinate) const {
    const auto index = getIndex(coordinate);
    const auto gridSize = static_cast<int>(m_initialTileCharGrid.size());
    if (index < 0 || index >= gridSize) {
        throw InvalidCoordinateException(coordinate);
    }

    return index;
}

}  // namespace SB
```

Since the **Sokoban** class definition is thoroughly commented, we'll primarily concentrate on its implementation details.

```cpp
// <Sokoban.hpp>
#ifndef SOKOBAN_H
#define SOKOBAN_H

#include <functional>
#include <memory>
#include <stack>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>
#include <SFML/Audio.hpp>
#include <SFML/Graphics.hpp>
#include "SokobanConstants.hpp"
#include "SokobanElapsedTime.hpp"
#include "SokobanPlayer.hpp"
#include "SokobanScore.hpp"
#include "SokobanTileGrid.hpp"

namespace SB {

/**
 * @brief Game state.
 */
struct State {
    Direction playerOrientation;
    sf::Vector2i playerLoc;
    std::vector<TileChar> tileCharGrid;
    int score;
};

/**
 * @brief This class implements all gameplay.
 */
class Sokoban final : public SokobanTileGrid,
                      public SokobanPlayer,
                      public SokobanElapsedTime,
                      public SokobanScore {
```

```
public:
   /**
    * @brief Creates a Sokoban instance; initializes sound.
    */
   Sokoban();

   /**
    * @brief A convenient constructor that initializes with a specified filename of a a level file.
    * @param filename The filename of a level file.
    */
   explicit Sokoban(const std::string& filename);

   /**
    * @brief Changes the player's location for one tile with the given direction.
    * @param direction The direction for the player to move.
    */
   void movePlayer(const Direction& direction);

   /**
    * @brief Resets the game. The game will return back to the initial form.
    */
   void reset();

   /**
    * @brief Undoes one move. If no moves are available to undo, do nothing.
    */
   void undo();

   /**
    * @brief Updates the game in a game frame.
    * @param dt The delta time in microseconds between this frame and the previous frame.
    */
   void update(const int64_t& dt) override;

   /**
    * @brief Reads a map from a level file (.lvl) and loads the content to the sokoban object.
    */
   friend std::ifstream& operator>>(std::ifstream& ifstream, Sokoban& sokoban);

   /**
    * @brief Outputs a sokoban game to a level file (.lvl).
    */
   friend std::ofstream& operator<<(std::ofstream& ofstream, const Sokoban& sokoban);

protected:
   /**
    * @brief Draws everything onto the target.
    */
   void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

private:
   /**
    * @brief Returns the next location based on the current location and the orientation.
    * @param currentLoc The current location.
    * @param orientation The orientation.
    */
    [[nodiscard]] static sf::Vector2i
    getNextLoc(const sf::Vector2i& currentLoc, const Direction& orientation);

    /**
     * @brief Moves a box towards a specified direction. Note that the block at the from coordinate
     * must be a box. The box that has already been stowed properly can be moved, and when it is
     * moved out from the storage, the score decrement.
     * @param fromCoordinate The initial coordinate.
     * @param direction The direction to move the box.
     * @return true if the box can be moved; false otherwise.
     */
    bool moveBox(const sf::Vector2i& fromCoordinate, const Direction& direction);

    /**
     * @brief Loads a sound file.
     * @param soundFilename The name of the sound file.
     */
    void loadSound(const std::string& soundFilename);

    /**
     * @brief Draws the result screen: triump message and final score.
     */
    void drawResultScreen(sf::RenderTarget& target, sf::RenderStates states) const;

    /**
     * @brief If the player has won the game.
     */
    bool m_hasWon = false;

    /**
     * @brief The game sound effects, including background music. The keys of this map are sound
     * filenames.
     */
    std::unordered_map<
        std::string,
        std::pair<std::shared_ptr<sf::Sound>, std::shared_ptr<sf::SoundBuffer>>>
        m_soundMap;

    /**
     * @brief The font for the triumph message.
     */
    sf::Font m_font;

    /**
     * @brief The stack of states.
     */
    std::stack<State> m_stateStack;
};

}  // namespace SB

#endif
```

The **Sokoban** constructor initializes sound and font resources. Additionally, a helper constructor is provided, accepting a filename and reading its content into the object.

```cpp
// <Sokoban.cpp>
#include "Sokoban.hpp"
#include <algorithm>
#include <cmath>
#include <fstream>
#include <iostream>
#include <limits>
#include <memory>
#include <string>
#include <utility>
#include "SokobanConstants.hpp"

namespace SB {

Sokoban::Sokoban() {
    loadSound(SOUND_BACKGROUND);
    loadSound(SOUND_WIN);

    m_font.loadFromFile(FONT_ROBOTO_FILENAME);
}

Sokoban::Sokoban(const std::string& filename) : Sokoban() {
    std::ifstream ifstream{ filename };
    if (!ifstream.is_open()) {
        throw std::invalid_argument("File not found: " + filename);
    }

    ifstream >> *this;
}
```

The **movePlayer** function is a bit complex. It begins by checking if the player has already won the game, and if so, it immediately returns. Next, it creates a **State** object containing the player's orientation, location, the tile character grid, and the current score.

Afterwards, it changes the player's orientation and examines whether the block in front of the player is movable. The player can move if: (1) the block is not a wall, or (2) the block is a box and can be pushed forward. If the player successfully moves, the previously created state is pushed onto the **m_stateStack**. Finally, the user's location is updated.

```cpp
void Sokoban::movePlayer(const Direction& direction) {
    // If the player has won the game, it can't move anymore
    if (isWon()) {
        return;
    }

    const State state = { m_playerOrientation, m_playerLoc, m_tileCharGrid, m_score };

    // Change the player's orientation
    m_playerOrientation = direction;

    // Find the coordinate of the block to move to
    const auto nextLoc{ getNextLoc(m_playerLoc, direction) };

    // If the next location is out of the map, stay on the spot
    const auto nextLocIndex = getIndex(nextLoc);
    if (nextLoc.x < 0 || nextLoc.x >= m_width || nextLocIndex < 0 ||
        nextLocIndex >= m_width * m_height) {
        return;
    }

    // Get the texture of the next block
    const auto nextBlock = getTileChar(nextLoc);

    // If the coordinate corresponds to a wall block or a box storage, stay on the spot
    if (nextBlock == TileChar::Wall) {
        return;
    }

    // If the coordinate corresponds to an box block, try to push the box to the other side
    if (nextBlock == TileChar::Box || nextBlock == TileChar::BoxStorage) {
        const auto canMoveBox = moveBox(nextLoc, direction);
        if (!canMoveBox) {
            return;
        }
    }

    // Save the current move
    m_stateStack.push(state);

    // Update player location
    m_playerLoc = nextLoc;
}
```

The **reset** function first clears the **m_hasWon** flag and the state stack. Then, it performs a shallow copy of the **m_initialTileCharGrid** and assigns it to **m_tileCharGrid**. After that, it iterates through the tile grid to initialize the map tiles. Subsequently, the score and max score are set, and the player's orientation and elapsed time are reset. Finally, the background music is reset to play from the beginning.

```
void Sokoban::reset() {
    // Reset m_hasWon and m_stateStack
    m_hasWon = false;
    while (!m_stateStack.empty()) {
        m_stateStack.pop();
    }

    // Perform a shallow copy for the tile char grid
    m_tileCharGrid = m_initialTileCharGrid;

    // Traverse the tile grid
    auto boxCount{ 0 };
    auto storageCount{ 0 };
    auto boxStorageCount{ 0 };
    traverseTileCharGrid([&](auto coordinate, auto tileChar) {
        m_tileGrid.push_back(getTile(tileChar));
        if (tileChar == TileChar::Player) {
            m_playerLoc = coordinate;
            setTileChar(coordinate, TileChar::Empty);
        } else if (tileChar == TileChar::Box) {
            ++boxCount;
        } else if (tileChar == TileChar::Storage) {
            ++storageCount;
        } else if (tileChar == TileChar::BoxStorage) {
            ++boxStorageCount;
        }

        return false;
    });

    // Set the score and max score
    m_score = boxStorageCount;
    m_maxScore = std::min(storageCount, boxCount) + boxStorageCount;

    // Reset the player's orientation
    m_playerOrientation = DEFAULT_ORIENTATION;

    // Reset the time
    m_elapsedTimeInMicroseconds = 0;

    // Reset the background music
    if (m_soundMap.find(SOUND_BACKGROUND) != m_soundMap.end()) {
        m_soundMap.at(SOUND_BACKGROUND).first->play();
    }
}
```

The **undo** function pops the state stack to retrieve the previous state, allowing the game to roll back to its previous state.

```
void Sokoban::undo() {
    if (isWon() || m_stateStack.size() == 0) {
        return;
    }

    const auto [playerOrientation, playerLoc, tileCharGrid, score] = m_stateStack.top();
    m_stateStack.pop();

    m_playerOrientation = playerOrientation;
    m_playerLoc = playerLoc;
    m_tileCharGrid = tileCharGrid;

    m_tileGrid.clear();
    traverseTileCharGrid([&](auto coordinate, auto tileChar) {
        m_tileGrid.push_back(getTile(tileChar));
        if (tileChar == TileChar::Player) {
            m_playerLoc = coordinate;
            setTileChar(coordinate, TileChar::Empty);
        }

        return false;
    });

    m_score = score;
}
```

The **update** function is called in every frame of the game. It first checks if the player has won. If so, the final score is displayed and a victory sound effect is played. The **m_hasWon** flag is then set to true to prevent further updates.

```cpp
void Sokoban::update(const int64_t& dt) {
    if (!isWon()) {
        // If the player has won, don't update the elapsed time
        SokobanElapsedTime::update(dt);
    }

    // Check if the player wins the game
    if (!m_hasWon && isWon()) {
        m_hasWon = true;

        // Stop the background music
        if (m_soundMap.find(SOUND_BACKGROUND) != m_soundMap.end()) {
            m_soundMap.at(SOUND_BACKGROUND).first->stop();
        }

        // Reset the player's orientation
        m_playerOrientation = DEFAULT_ORIENTATION;

        // Play the win sound effect
        if (m_soundMap.find(SOUND_WIN) != m_soundMap.end()) {
            m_soundMap.at(SOUND_WIN).first->play();
        }
    }
}

std::ifstream& operator>>(std::ifstream& ifstream, Sokoban& sokoban) {
    sokoban.m_initialTileCharGrid.clear();

    // The first line consists of height and width; ignore the rest of the line
    ifstream >> sokoban.m_height >> sokoban.m_width;
    ifstream.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    // Continue the read the following lines
    for (int row{ 0 }; row < sokoban.m_height; ++row) {
        std::string line;
        getline(ifstream, line);
        for (int col{ 0 }; col < sokoban.m_width; ++col) {
            sokoban.m_initialTileCharGrid.push_back(static_cast<TileChar>(line.at(col)));
        }
    }

    sokoban.reset();

    return ifstream;
}

std::ofstream& operator<<(std::ofstream& ofstream, const Sokoban& sokoban) {
    ofstream << sokoban.height() << sokoban.width();

    const auto player_loc = sokoban.m_playerLoc;
    sokoban.traverseTileCharGrid([&](auto coordinate, auto tileChar) {
        if (coordinate.x == 0) {
            ofstream << std::endl;
        }

        if (coordinate == player_loc) {
            ofstream << static_cast<char>(TileChar::Player);
        } else {
            ofstream << static_cast<char>(tileChar);
        }

        return false;
    });

    return ofstream;
}
```

The **draw** function sequentially renders the following child classes (modules): **SokobanTileGrid**, **SokobanPlayer**, **SokobanElapsedTime**, and **SokobanScore**.

```
void Sokoban::draw(sf::RenderTarget& target, const sf::RenderStates states) const {
    SokobanTileGrid::draw(target, states);
    SokobanPlayer::draw(target, states);
    SokobanElapsedTime::draw(target, states);
    SokobanScore::draw(target, states);

    // Display the victory notice if the player has won the game
    if (m_hasWon) {
        drawResultScreen(target, states);
    }
}

sf::Vector2i Sokoban::getNextLoc(const sf::Vector2i& currentLoc, const Direction& orientation) {
    sf::Vector2i nextLoc(currentLoc);
    switch (orientation) {
        case Direction::Up:
            --nextLoc.y;
            break;
        case Direction::Down:
            ++nextLoc.y;
            break;
        case Direction::Left:
            --nextLoc.x;
            break;
        case Direction::Right:
            ++nextLoc.x;
            break;
    }

    return nextLoc;
}
```

The **moveBox** function's logic is similarly intricate, but it mirrors the structure of the **movePlayer** function. Therefore, I won't elaborate on it redundantly.

```
bool Sokoban::moveBox(const sf::Vector2i& fromCoordinate, const Direction& direction) {
    const auto toCoordinate{ getNextLoc(fromCoordinate, direction) };

    // If the destination coordinate is out of the map, return false
    const auto toCoordinateIndex = getIndex(toCoordinate);
    if (toCoordinate.x < 0 || toCoordinate.x >= m_width || toCoordinateIndex < 0 ||
        toCoordinateIndex >= m_width * m_height) {
        return false;
    }

    const auto currentBlock{ getTileChar(fromCoordinate) };
    const auto nextBlock{ getTileChar(toCoordinate) };
    const auto isCurrentBlockBoxStorage = currentBlock == TileChar::BoxStorage;

    if (nextBlock == TileChar::Empty) {
        // Swap the blocks at the initial coordiante and the destination coordinate
        setTileChar(fromCoordinate, isCurrentBlockBoxStorage ? TileChar::Storage : TileChar::Empty);
        setTileChar(toCoordinate, TileChar::Box);

        if (isCurrentBlockBoxStorage)
            --m_score;

        return true;
    }

    if (nextBlock == TileChar::Storage) {
        // The block at the initial coordiante should become an empty block (or a storage block if
        // the current block is a box-storage block); the block at the destination coordinate should
        // become a box-storage block
        setTileChar(fromCoordinate, isCurrentBlockBoxStorage ? TileChar::Storage : TileChar::Empty);
        setTileChar(toCoordinate, TileChar::BoxStorage);

        // Score increments by 1
        if (!isCurrentBlockBoxStorage)
            ++m_score;

        return true;
    }

    return false;
}

void Sokoban::loadSound(const std::string& soundFilename) {
    const auto soundBuffer{ std::make_shared<sf::SoundBuffer>() };
    const auto sound{ std::make_shared<sf::Sound>() };
    if (soundBuffer->loadFromFile(soundFilename)) {
        sound->setBuffer(*soundBuffer);
        m_soundMap[soundFilename] = std::make_pair(sound, soundBuffer);
    }
}
```

As for the final score of this game, I came up with a formula according to the game data:

$$F = e^{1-T/e^2} \cdot S \cdot (WH - M)$$

Where F is the final score, T is the consumed time in seconds, S is the number of boxes that are stowed in the storages, W is the width of the map, H is the height of the map, and M is the number of moves the player used.

```
void Sokoban::drawResultScreen(sf::RenderTarget& target, sf::RenderStates states) const {
    // Draw "You win!" in the center of the screen
    sf::Text winText;
    winText.setString("You win!");
    winText.setFont(m_font);
    winText.setFillColor(sf::Color(255, 140, 0));
    winText.setCharacterSize(15 * m_width);
    winText.setOutlineColor(sf::Color::White);
    winText.setOutlineThickness(2);

    // Compute the position of winText
    const auto winTextRect = winText.getLocalBounds();
    const float targetWidth = static_cast<float>(target.getSize().x);
    const float targetHeight = static_cast<float>(target.getSize().y);
    winText.setOrigin({ (static_cast<float>(winTextRect.width)) / 2.0f,
                        (static_cast<float>(winTextRect.height)) });
    winText.setPosition({ targetWidth / 2.0f, targetHeight / 2.0f });
    target.draw(winText);

    // Final score
    const auto moveScore = m_width * m_height - m_stateStack.size();
    const auto timeInSeconds = static_cast<double>(m_elapsedTimeInMicroseconds) / 1000000.0;
    const auto timeScore = std::exp(1 - timeInSeconds / std::exp(2));
    const auto finalScore = static_cast<int>(std::floor(moveScore * timeScore * m_score));

    // Draw the score down below the "You win!"
    sf::Text scoreText;
    scoreText.setString("Score: " + std::to_string(finalScore));
    scoreText.setFont(m_font);
    scoreText.setFillColor(sf::Color::Black);
    scoreText.setCharacterSize(3 * m_width);
    scoreText.setOutlineColor(sf::Color::White);
    winText.setOutlineThickness(2);

    // Compute the position of scoreText
    const auto scoreTextRect = scoreText.getLocalBounds();
    scoreText.setOrigin({ static_cast<float>(scoreTextRect.width) / 2.0f,
                          (scoreTextRect.height - static_cast<float>(winTextRect.height)) / 2.0f });
    scoreText.setPosition({ targetWidth / 2.0f, targetHeight / 2.0f });
    target.draw(scoreText);
}

}  // namespace SB
```

There is a **InvalidCoordinateException** that is thrown when an invalid coordinate is used.

```
// <InvalidCoordinateException.hpp>
#ifndef INVALIDCOORDINATEEXCEPTION_H
#define INVALIDCOORDINATEEXCEPTION_H

#include <string>
#include <SFML/Graphics.hpp>

namespace SB {

```

```
/**
 * @brief This exception is thrown when an invalid coordinate is used.
 */
class InvalidCoordinateException final : public std::exception {
 public:
    /**
     * @brief Creates an InvalidCoordinateException instance.
     * @param coordinate The invalid coordinate.
     */
    explicit InvalidCoordinateException(const sf::Vector2i& coordinate) noexcept;

    /**
     * @brief Gets the excception message.
     */
    [[nodiscard]] const char* what() const noexcept override;

 private:
    /**
     * @brief The exception message to display.
     */
    std::string message;
};

}  // namespace SB

#endif

// <InvalidCoordinateException.cpp>
#include "InvalidCoordinateException.hpp"
#include <sstream>

namespace SB {

InvalidCoordinateException::InvalidCoordinateException(const sf::Vector2i& coordinate) noexcept {
    std::ostringstream oss;
    oss << "Invalid coordinate: (" << coordinate.x << ", " << coordinate.y << ")";
    message = oss.str();
}

const char* InvalidCoordinateException::what() const noexcept { return message.c_str(); }

}  // namespace SB
```

There is a **SokobanConstant.hpp** file that stores all sorts of game constants.

```
// <SokobanConstant.hpp>
#ifndef SOKOBANCONSTANTS_H
#define SOKOBANCONSTANTS_H

#include <string>

/**
 * @brief Sokoban game constants. Include but not limit to the following:
 * 1. The name of the game and the author
 * 2. The size of each tiles
 * 3. Tilesets' filenames
 * 4. Enumeration classes.
 */
namespace SB {

// The name of the game
inline const std::string GAME_NAME = "Sokoban";

// The author name
inline const std::string AUTHOR_NAME = "James Chen";

// The height and width in pixel of each tile
inline constexpr int TILE_HEIGHT = 64;
inline constexpr int TILE_WIDTH = 64;

// Tile characters. In the level (.lvl) files, each character corresponds to a specific texture of
// has particular meaning to the corresponding position.
// '@' - The initial position of the player.
// '.' - An empty space, which the player can move through.
// '#' - A wall, which blocks movement.
// 'A' - A box, which can be paused by the player.
// 'a' - A storage location, where the player is trying to push a box.
// '1' - A box that is already in a storage location.
inline constexpr char TILE_CHAR_PLYAER = '@';
inline constexpr char TILE_CHAR_EMPTY = '.';
inline constexpr char TILE_CHAR_WALL = '#';
inline constexpr char TILE_CHAR_BOX = 'A';
inline constexpr char TILE_CHAR_STORAGE = 'a';
inline constexpr char TILE_CHAR_BOX_STORAGE = '1';

// Assets directory
inline const std::string ASSETS_DIR = "./assets/";

// Tileset directory
inline const std::string TILESET_DIR = ASSETS_DIR + "tileset/";

// Level directory
inline const std::string LEVEL_DIR = ASSETS_DIR + "level/";

// Font directory
inline const std::string FONT_DIR = ASSETS_DIR + "font/";

// Sound directory
inline const std::string SOUND_DIR = ASSETS_DIR + "sound/";

// Tiles filenames
inline const std::string TILE_ENVIRONMENT_03_FILENAME = TILESET_DIR + "environment_03.png";
inline const std::string TILE_BLOCK_06_FILENAME = TILESET_DIR + "block_06.png";
inline const std::string TILE_CRATE_03_FILENAME = TILESET_DIR + "crate_03.png";
inline const std::string TILE_GROUND_01_FILENAME = TILESET_DIR + "ground_01.png";
inline const std::string TILE_GROUND_04_FILENAME = TILESET_DIR + "ground_04.png";
inline const std::string TILE_PLAYER_05_FILENAME = TILESET_DIR + "player_05.png";
inline const std::string TILE_PLAYER_08_FILENAME = TILESET_DIR + "player_08.png";
inline const std::string TILE_PLAYER_17_FILENAME = TILESET_DIR + "player_17.png";
inline const std::string TILE_PLAYER_20_FILENAME = TILESET_DIR + "player_20.png";

```

```cpp
// Fonts filenames
inline const std::string FONT_DIGITAL7_FILENAME = FONT_DIR + "digital-7.mono.ttf";
inline const std::string FONT_ROBOTO_FILENAME = FONT_DIR + "roboto-regular.ttf";

// Sound filenames
inline const std::string SOUND_BACKGROUND = SOUND_DIR + "background.wav";
inline const std::string SOUND_WIN = SOUND_DIR + "win.wav";

/**
 * @brief Enumerates four cardinal directions: Up, Down, Left, and Right. This enumeration follows
 * the naming convention used in SFML.
 */
enum class Direction { Up, Down, Left, Right };

/**
 * @brief Enumerates tile characters.
 */
enum class TileChar : char {
    Player = TILE_CHAR_PLYAER,
    Empty = TILE_CHAR_EMPTY,
    Wall = TILE_CHAR_WALL,
    Box = TILE_CHAR_BOX,
    Storage = TILE_CHAR_STORAGE,
    BoxStorage = TILE_CHAR_BOX_STORAGE,
};

}  // namespace SB

#endif
```

Several test cases are established to verify the correctness of the program, and all of them are well commented on.

```cpp
// <test.cpp>
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main

#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <boost/test/unit_test.hpp>
#include "Sokoban.hpp"

/**
 * @brief Checks if two coordinates are the same.
 * @param first The first coordinate.
 * @param second The second coordiante.
 * @return True if the two components of the two coordinates are equal respectively; false
 * otherwise.
 */
bool isCoordinateEqual(const sf::Vector2u& first, const sf::Vector2u& second) noexcept {
    return first.x == second.x && first.y == second.y;
}

```

```
// Tests if `height()` and `width()` returns the height and width of a map correctly.
BOOST_AUTO_TEST_CASE(testHeightWidth) {
    const SB::Sokoban sokoban{ "assets/level/level2.lvl" };

    BOOST_REQUIRE_EQUAL(sokoban.height(), 10);
    BOOST_REQUIRE_EQUAL(sokoban.width(), 12);
}

// Tests if `playLoc()` returns the correct player location in the beginning of the game.
BOOST_AUTO_TEST_CASE(testPlayerPosition) {
    const SB::Sokoban sokoban{ "assets/level/level2.lvl" };

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 8, 5 }));
}

// Tests if `movePlayer(SB::Direction)` works correctly: a player should be able to push a box if
// the box is not blocked by a wall or another box.
BOOST_AUTO_TEST_CASE(testMovePayer) {
    SB::Sokoban sokoban{ "assets/level/level2.lvl" };
    sokoban.movePlayer(SB::Direction::Right);

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 9, 5 }));
}

// Tests if `movePlayer(SB::Direction)` works correctly: a player should not push a box that is
// blocked by another box.
BOOST_AUTO_TEST_CASE(testMovePayerBlockedByBox) {
    SB::Sokoban sokoban{ "assets/level/level2.lvl" };

    // Since there are two boxes in a row in the up direction, the player is not able to move no
    // matter how many times they try to move upwards.
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Up);

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 8, 5 }));
}

// Tests if `movePlayer(SB::Direction)` works correctly: a player should not push a box that is
// blocked by a wall block.
BOOST_AUTO_TEST_CASE(testMovePayerBlockedByWall) {
    SB::Sokoban sokoban{ "assets/level/level2.lvl" };

    // Move rightward twice. For the first move, the player pushes the box rightawrds;
    // For the second move, the player saty on the pot, as the box cannot be pushed
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 9, 5 }));
}

// Tests if `movePlayer(SB::Direction)` works correctly: a player should not move out of the map
// from the upper border.
BOOST_AUTO_TEST_CASE(testMovePayerUpBorder) {
    SB::Sokoban sokoban{ "assets/level/swapoff.lvl" };
    sokoban.movePlayer(SB::Direction::Left);
    sokoban.movePlayer(SB::Direction::Up);
```

```cpp
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Up);

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 1, 0 }));
}

// Tests if `movePlayer(SB::Direction)` works correctly: a player should not move out of the map
// from the right border.
BOOST_AUTO_TEST_CASE(testMovePayerRightBorder) {
    SB::Sokoban sokoban{ "assets/level/swapoff.lvl" };
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 4, 2 }));
}

// Tests if `movePlayer(SB::Direction)` works correctly: a player should not move out of the map
// from the down border.
BOOST_AUTO_TEST_CASE(testMovePayerDownBorder) {
    SB::Sokoban sokoban{ "assets/level/swapoff.lvl" };
    sokoban.movePlayer(SB::Direction::Down);
    sokoban.movePlayer(SB::Direction::Down);
    sokoban.movePlayer(SB::Direction::Down);

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 2, 4 }));
}

// Tests if `movePlayer(SB::Direction)` works correctly: a player should not move out of the map
// from the left border.
BOOST_AUTO_TEST_CASE(testMovePayerLeftBorder) {
    SB::Sokoban sokoban{ "assets/level/swapoff.lvl" };
    sokoban.movePlayer(SB::Direction::Left);
    sokoban.movePlayer(SB::Direction::Left);
    sokoban.movePlayer(SB::Direction::Left);

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 0, 2 }));
}

// Tests if `movePlayer(SB::Direction)` works correctly: a player should not push a box out of the
// map.
BOOST_AUTO_TEST_CASE(testMovePayerPushBoxOffScreen) {
    SB::Sokoban sokoban{ "assets/level/swapoff.lvl" };

    // Try to push a box out of the map, the player should stay on the spot in the second move
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Up);

    BOOST_REQUIRE(isCoordinateEqual(sokoban.playerLoc(), { 2, 1 }));
}

// Tests if `isWon()` works correctly: If all boxes are already in the storages, it should return
// true.
BOOST_AUTO_TEST_CASE(testIsWon) {
    const SB::Sokoban sokoban{ "assets/level/autowin2.lvl" };

    BOOST_REQUIRE(sokoban.isWon());
}

// Tests if `isWon()` works correctly: If there are two boxes and only one storage, players win
// as long as they push one box to the storage.
BOOST_AUTO_TEST_CASE(testIsWonTooManyBoxes) {
    SB::Sokoban sokoban{ "assets/level/level5.lvl" };
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Down);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Up);

    BOOST_REQUIRE(sokoban.isWon());
}

// Tests if `isWon()` works correctly: If there are three storages but only two boxes, players win
// when they stow the boxes properly.
BOOST_AUTO_TEST_CASE(testIsWonTooManyStorages) {
    SB::Sokoban sokoban{ "assets/level/level6.lvl" };
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Down);
    sokoban.movePlayer(SB::Direction::Down);
    sokoban.movePlayer(SB::Direction::Left);
    sokoban.movePlayer(SB::Direction::Left);
    sokoban.movePlayer(SB::Direction::Left);
    sokoban.movePlayer(SB::Direction::Left);
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Right);
    sokoban.movePlayer(SB::Direction::Up);
    sokoban.movePlayer(SB::Direction::Left);

    BOOST_REQUIRE(sokoban.isWon());
}

```

5 PS4 - N-Body Simulation

5.1 PS4a

5.1.1 Discussion

In PS4a, I made a program that loads and displays a static universe. It reads input data, including the number of planets, the radius of the universe, and the data of each planet, which consists of initial position, velocity, mass, and the filename of the image. After reading all the data, the program shows a window displaying the planets with a captivating background.

Through this assignment, I acquired valuable understanding of shared pointers and memory management. Exploring their advantages and limitations provided me with a nuanced perspective, yet overall, employing shared pointers aligns well with the contemporary best practices of C++ development. Subsequently, I've consistently incorporated smart pointers into my coding practices, leveraging their benefits across various projects.

5.1.2 Achievements

By running the program with a file containing the following content:

```
5
2.50e+11
 1.4960e+11  0.0000e+00  0.0000e+00  2.9800e+04  5.9740e+24    earth.gif
 2.2790e+11  0.0000e+00  0.0000e+00  2.4100e+04  6.4190e+23     mars.gif
 5.7900e+10  0.0000e+00  0.0000e+00  4.7900e+04  3.3020e+23  mercury.gif
 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  1.9890e+30      sun.gif
 1.0820e+11  0.0000e+00  0.0000e+00  3.5000e+04  4.8690e+24    venus.gif

This file contains the sun and the inner 4 planets of our Solar System.
```

A window will display a static image as shown in **Figure 4.1**.

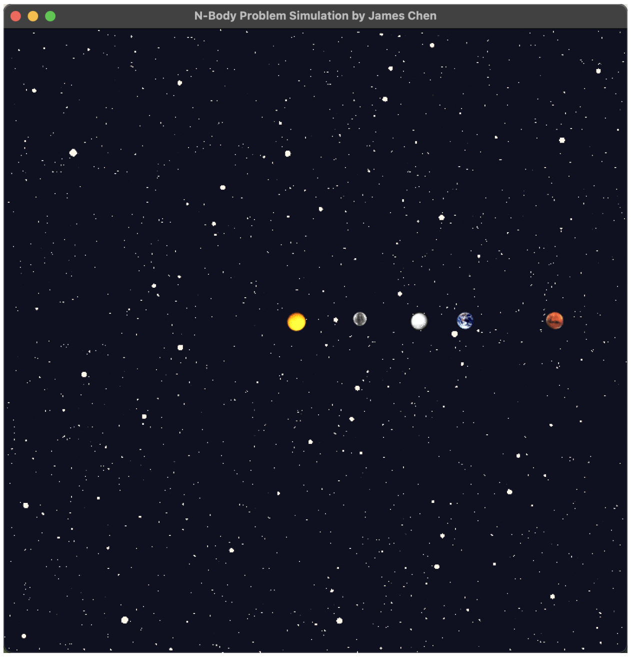

Figure 4.1 A snapshot showcasing the NBody program.

5.1.3 Codebase

```makefile
# C++ Compiler
COMPILER = g++

# C++ Flags
CFLAGS = --std=c++20 -Wall -Werror -pedantic -g

# Libraries
LIB = -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -lboost_unit_test_framework

# Code source directory
SRC = ./

# Hpp files (dependencies)
DEPS = $(SRC)Universe.hpp \
       $(SRC)CelestialBody.hpp \
       $(SRC)NBodyConstant.hpp

# Static library
STATIC_LIB = NBody.a

# The object files that the static library includes
STATIC_LIB_OBJECTS = $(SRC)Universe.o \
                                     $(SRC)CelestialBody.o

# Program
PROGRAM = NBody

# Program object files
MAIN_OBJECTS = $(SRC)main.o

# Test program
TEST_PROGRAM = test

# Test object files
TEST_OBJECTS = $(SRC)test.o

all: $(PROGRAM) $(TEST_PROGRAM)

$(SRC)%.o: $(SRC)%.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

$(PROGRAM): $(MAIN_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

$(STATIC_LIB): $(STATIC_LIB_OBJECTS)
        ar rcs $@ $^

$(TEST_PROGRAM): $(TEST_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

clean:
        rm -f $(SRC)*.o $(PROGRAM) $(STATIC_LIB) $(TEST_PROGRAM)

lint:
        cpplint *.hpp *.cpp

boost: $(TEST_PROGRAM)
        ./$(TEST_PROGRAM)

run: $(PROGRAM)
        ./$(PROGRAM) < assets/planets.txt
```

```cpp
// <main.cpp>
#include <iostream>
#include "NBodyConstant.hpp"
#include "Universe.hpp"

/**
 * @brief Starts the universe simulation.
 */
int main() {
    NB::Universe universe;
    std::cin >> universe;
    universe.loadResources();

    const sf::VideoMode windowVideoMode{ NB::WINDOW_WIDTH, NB::WINDOW_HEIGHT };
    sf::RenderWindow window(windowVideoMode, std::string(NB::WINDOW_TITLE));
    while (window.isOpen()) {
        sf::Event event{};
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
                break;
            }
        }

        if (window.isOpen()) {
            window.clear(sf::Color::White);
            window.draw(universe);
            window.display();
        }
    }

    return 0;
}
```

This project encompasses the creation and implementation of two classes: **Universe** and **CelestialBody**. Universe encapsulates a vector of celestial bodies, alongside attributes such as radius and scale. The scale represents the ratio of the universe's diameter to the width of the displayed window.

```cpp
// <Universe.hpp>
#ifndef UNIVERSE_HPP
#define UNIVERSE_HPP

#include <memory>
#include <string>
#include <utility>
#include <vector>
```

```cpp
#include <SFML/Audio.hpp>
#include <SFML/Graphics.hpp>
#include "CelestialBody.hpp"

namespace NB {

class CelestialBody;

class Universe final : public sf::Drawable {
 public:
    /**
     * @brief Creates a universe.
     */
    Universe();

    /**
     * @brief Creates a universe from a file.
     * @param filename The name of the file.
     */
    explicit Universe(const std::string& filename);

    /**
     * @brief Loads resources (background image and music).
     */
    void loadResources();

    /**
     * @brief Returns the numer of planets in this universe.
     */
    [[nodiscard]] int numPlanets() const;

    /**
     * @brief Returns the radius of this universe.
     */
    [[nodiscard]] double radius() const;

    /**
     * @brief Gets the scale of the universe.
     */
    [[nodiscard]] double scale() const;

    /**
     * Prints the number of planets and the radius of this universe.
     */
    friend std::istream& operator>>(std::istream& istream, Universe& universe);

    /**
     * Prints the number of planets and the radius of this universe.
     */
    friend std::ostream& operator<<(std::ostream& ostream, const Universe& universe);

    /**
     * Returns the celestial body at specified index.
     * @param index The index of the celestial body to retrieve.
     */
    CelestialBody& operator[](const std::size_t& index) const;

 protected:
    /**
     * Draws the universe onto the target.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

 private:
    /**
     * @brief The number of planets.
     */
    int m_numPlanets = 0;

    /**
     * @brief The radius of this universe.
     */
    double m_radius = 0.0;

    /**
     * @brief The ratio of the universe diameter to the width of the window.
     */
    double m_scale = 1.0;

    /**
     * @brief A vector of celestial bodies in this universe.
     */
    std::vector<std::shared_ptr<CelestialBody>> m_celestialBodyVector;

    /**
     * @brief The background image.
     */
    std::pair<std::shared_ptr<sf::Texture>, std::shared_ptr<sf::Sprite>> m_backgroundImage;

    /**
     * @brief The background music.
     */
    std::pair<std::shared_ptr<sf::SoundBuffer>, std::shared_ptr<sf::Sound>> m_backgroundMusic;
};

}  // namespace NB

#endif
```

```cpp
// <Universe.cpp>
#include "Universe.hpp"
#include <fstream>
#include <iostream>
#include <limits>
#include "CelestialBody.hpp"
#include "NBodyConstant.hpp"

namespace NB {

Universe::Universe() = default;

```

```
Universe::Universe(const std::string& filename) : Universe() {
    std::fstream fstream{ filename };

    // Check if the file is opened successfully
    if (!fstream.is_open()) {
        throw std::invalid_argument("Cannot open: " + filename);
    }

    fstream >> *this;
}

void Universe::loadResources() {
    // Load the background image
    m_backgroundImage.first = { std::make_shared<sf::Texture>() };
    m_backgroundImage.second = { std::make_shared<sf::Sprite>() };
    m_backgroundImage.first->loadFromFile(IMAGE_BACKGRROUND);
    m_backgroundImage.second->setTexture(*m_backgroundImage.first);

    // Rescale the background image so that the image fits the window
    const auto backgroundTexture = m_backgroundImage.first;
    const auto backgroundSprite = m_backgroundImage.second;
    const auto textureSize = backgroundTexture->getSize();
    backgroundSprite->setScale(
        static_cast<float>(WINDOW_WIDTH) / textureSize.x,
        static_cast<float>(WINDOW_HEIGHT) / textureSize.y);

    // Load and play the background music
    const auto soundBuffer{ std::make_shared<sf::SoundBuffer>() };
    const auto sound{ std::make_shared<sf::Sound>() };
    if (soundBuffer->loadFromFile(SOUND_BACKGROUND_MUSIC)) {
        sound->setBuffer(*soundBuffer);
        m_backgroundMusic.first = soundBuffer;
        m_backgroundMusic.second = sound;

        sound->play();
    }

    // Load celestial bodies' images
    for (auto const& celestialBody : m_celestialBodyVector) {
        celestialBody->loadResource();
    }
}

int Universe::numPlanets() const { return m_numPlanets; }

double Universe::radius() const { return m_radius; }

double Universe::scale() const { return m_scale; }

std::istream& operator>>(std::istream& istream, Universe& universe) {
    istream >> universe.m_numPlanets >> universe.m_radius;

    // Set the scale (1.1x larger, as some planets' trajectories are ecllipses)
    universe.m_scale = universe.m_radius * 2.0 / WINDOW_WIDTH * 1.1;

    istream.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
```

```cpp

    // Celestial bodies
    for (int i = 0; i < universe.m_numPlanets; ++i) {
        auto celestialBody = std::make_shared<CelestialBody>(&universe);
        universe.m_celestialBodyVector.push_back(celestialBody);
        istream >> *celestialBody;
    }

    return istream;
}

std::ostream& operator<<(std::ostream& ostream, const Universe& universe) {
    ostream << universe.m_numPlanets << std::endl << universe.m_radius << std::endl;

    // Output celestial bodies
    for (const auto& celestialBody : universe.m_celestialBodyVector) {
        ostream << *celestialBody << std::endl;
    }

    return ostream;
}

CelestialBody& Universe::operator[](const std::size_t& index) const {
    return *m_celestialBodyVector.at(index);
}

void Universe::draw(sf::RenderTarget& target, const sf::RenderStates states) const {
    // Set the background for the target
    target.draw(*m_backgroundImage.second);

    auto drawCelestialBody = [&](const std::shared_ptr<CelestialBody>& celestialBody) {
        celestialBody->draw(target, states);
    };

    std::for_each(m_celestialBodyVector.cbegin(), m_celestialBodyVector.cend(), drawCelestialBody);
}

}  // namespace NB
```

On the other hand, **CelestialBody** comprises attributes such as position, velocity, mass, and more. Notably, CelestialBody features a distinctive constructor that accepts a Universe object as a parameter, facilitating its instantiation within the universe context.

```cpp
// <CelestialBody.hpp>
// Copyright 2024 James Chen

#ifndef CELESTIALBODY_HPP
#define CELESTIALBODY_HPP

#include <memory>
#include <string>
#include <utility>
#include <SFML/Graphics.hpp>
#include "Universe.hpp"

```

```
namespace NB {

class Universe;

class CelestialBody final : public sf::Drawable {
 public:
    /**
     * @brief Creates a CelestialBody instance.
     * @param universePtr The universe this celesitial body is in.
     */
    explicit CelestialBody(Universe* universePtr);

    /**
     * @brief Returns the universe this celesitial body is in.
     */
    [[nodiscard]] Universe* universe() const;

    /**
     * @brief Returns the position of this celestial body.
     */
    [[nodiscard]] sf::Vector2f position() const;

    /**
     * @brief Returns the velocity of this celestial body.
     */
    [[nodiscard]] sf::Vector2f velocity() const;

    /**
     * @brief Returns the mass of this celestial body.
     */
    [[nodiscard]] float mass() const;

    /**
     * @brief Loads resource (image).
     */
    void loadResource();

    /**
     * Draws this celestial body onto the target.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

 private:
    /**
     * @brief The universe this celesitial body is in.
     */
    Universe* m_universePtr;

    /**
     * @brief The center coordiante of this celestial body.
     */
    sf::Vector2<double> m_position;

    /**
     * @brief The velocity of this celestial body.
     */
```

```cpp
    sf::Vector2<double> m_velocity;

    /**
     * @brief The mass of this celestial body.
     */
    double m_mass = 0.0;

    /**
     * @brief The filename of the image.
     */
    std::string m_image_filename;

    /**
     * @brief The image (Sprite and corresponding texture) of this celestial body.
     */
    std::pair<std::shared_ptr<sf::Texture>, std::shared_ptr<sf::Sprite>> m_image;

    /**
     * @brief Reads data from the std::istream to the celestial body.
     */
    friend std::istream& operator>>(std::istream& istream, CelestialBody& celestialBody);

    /**
     * @brief Writes data from the celesitial body to the std::ostream.
     */
    friend std::ostream& operator<<(std::ostream& ostream, const CelestialBody& celestialBody);
};

}  // namespace NB

#endif
```

```cpp
// <CelestialBody.cpp>
#include "CelestialBody.hpp"
#include <iostream>
#include <sstream>
#include <string>
#include <boost/algorithm/string.hpp>
#include "NBodyConstant.hpp"

namespace NB {

/**
 * Converts a double number into a string in the scientific form.
 * @param number The number to convert.
 * @return a string in the form of "x.yye?zz", where x is the integer part, yy is the fraction part,
 * ? is negative sign, and zz is the exponent.
 */
std::string to_standard_scientific_string(const double& number) {
    std::ostringstream stream;
    stream << std::scientific << std::setprecision(2) << number;
    std::string result = stream.str();

    return result;
}
```

```cpp


CelestialBody::CelestialBody(Universe* universePtr) : m_universePtr(universePtr) {}

sf::Vector2f CelestialBody::position() const {
    return { static_cast<float>(m_position.x), static_cast<float>(m_position.y) };
}

sf::Vector2f CelestialBody::velocity() const {
    return { static_cast<float>(m_velocity.x), static_cast<float>(m_velocity.y) };
}

float CelestialBody::mass() const { return static_cast<float>(m_mass); }

void CelestialBody::loadResource() {
    // Load the image file
    m_image.first = std::make_shared<sf::Texture>();
    m_image.second = std::make_shared<sf::Sprite>();
    m_image.first->loadFromFile(ASSETS_IMAGE_DIR / m_image_filename);
    m_image.second->setTexture(*m_image.first);
}

Universe* CelestialBody::universe() const { return m_universePtr; }

void CelestialBody::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    const auto universeRadius = m_universePtr->radius();
    const auto sprite = m_image.second;
    const auto universeScale = m_universePtr->scale();
    const sf::Vector2f realPosition{
        static_cast<float>((universeRadius + this->m_position.x) / universeScale),
        static_cast<float>((universeRadius - this->m_position.y) / universeScale),
    };
    sprite->setPosition(realPosition);

    target.draw(*sprite);
}

std::istream& operator>>(std::istream& istream, CelestialBody& celestialBody) {
    std::string line;

    // Skip blank lines
    while (line.empty() && !istream.eof()) {
        getline(istream, line);
        boost::trim(line);
    }

    std::stringstream stringstream(line);
    stringstream >> celestialBody.m_position.x >> celestialBody.m_position.y >>
        celestialBody.m_velocity.x >> celestialBody.m_velocity.y >> celestialBody.m_mass >>
        celestialBody.m_image_filename;

    if (stringstream.fail()) {
        throw std::runtime_error("Invalid input: " + line);
    }

    return istream;
}

std::ostream& operator<<(std::ostream& ostream, const CelestialBody& celestialBody) {
    const auto position = celestialBody.position();
    const auto velocity = celestialBody.velocity();
    ostream << position.x << " " << position.y << " " << velocity.x << " " << velocity.y << " "
            << celestialBody.mass() << " " << celestialBody.m_image_filename;

    return ostream;
}

}  // namespace NB
```

```cpp
// <NBodyConstant.hpp>
#ifndef NBODYCONSTANT_HPP
#define NBODYCONSTANT_HPP

#include <filesystem>
#include <string>

namespace NB {

// Use string literals directly for constexpr variables:
constexpr std::string_view WINDOW_TITLE = "N-Body Problem Simulation by James Chen";

// Window fixed height
constexpr unsigned WINDOW_WIDTH = 720;
constexpr unsigned WINDOW_HEIGHT = WINDOW_WIDTH;

// Use std::filesystem for path handling:
const std::filesystem::path ASSETS_DIR = "assets/";

// Images
const std::filesystem::path ASSETS_IMAGE_DIR = ASSETS_DIR;
const std::filesystem::path IMAGE_BACKGRROUND = ASSETS_IMAGE_DIR / "background.jpg";

// Sounds
const std::filesystem::path ASSETS_SOUND_DIR = ASSETS_DIR;
const std::filesystem::path SOUND_BACKGROUND_MUSIC = ASSETS_SOUND_DIR / "2001.wav";

}  // namespace NB

#endif
```

Various tests have been set up to validate the accuracy of the class **Universe** and **CelestialBody**:

```cpp
// <test.cpp>
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main

#include <fstream>
#include <iostream>
#include <sstream>
#include <boost/test/unit_test.hpp>
```

```
#include "Universe.hpp"

// Tests if `Universe::getNumPlanets()` and `Universe::getRadius` work correcly.
BOOST_AUTO_TEST_CASE(testUniverseBasic) {
    const NB::Universe universe{ "assets/1body.txt" };

    constexpr auto EXPECTED_NUM_PLANETS = 1;
    constexpr auto EXPECTED_RADIUS = 100.0;
    BOOST_REQUIRE_EQUAL(universe.numPlanets(), EXPECTED_NUM_PLANETS);
    BOOST_REQUIRE_EQUAL(universe.radius(), EXPECTED_RADIUS);
}

// Tests if `Universe::getNumPlanets()` and `Universe::getRadius` work correcly.
BOOST_AUTO_TEST_CASE(testUniverseBasic2) {
    const NB::Universe universe{ "assets/binary.txt" };

    constexpr auto EXPECTED_NUM_PLANETS = 2;
    constexpr auto EXPECTED_RADIUS = 5.0e10;
    BOOST_REQUIRE_EQUAL(universe.numPlanets(), EXPECTED_NUM_PLANETS);
    BOOST_REQUIRE_EQUAL(universe.radius(), EXPECTED_RADIUS);
}

// Tests if `CelestialBody::getNumPlanets()`, `CelestialBody::getRadius()` and
// `CelestialBody::getMass()` work correcly.
BOOST_AUTO_TEST_CASE(testCelestialBodyBasic) {
    const NB::Universe universe{ "assets/planets.txt" };
    const auto celestialBody = universe[0];

    constexpr float EXPECTED_POSITION_X = 1.4960e+11;
    constexpr float EXPECTED_POSITION_Y = 0.0000e+00;
    constexpr float EXPECTED_VELOCITY_X = 0.0000e+00;
    constexpr float EXPECTED_VELOCITY_Y = 2.9800e+04;
    constexpr float EXPECTED_MASS = 5.9740e+24;
    BOOST_REQUIRE_EQUAL(celestialBody.position().x, EXPECTED_POSITION_X);
    BOOST_REQUIRE_EQUAL(celestialBody.position().y, EXPECTED_POSITION_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().x, EXPECTED_VELOCITY_X);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().y, EXPECTED_VELOCITY_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.mass(), EXPECTED_MASS);
}

// Tests if `Universe::operator[]` works for the non-first elements.
BOOST_AUTO_TEST_CASE(testUniverseBracketOperator1) {
    const NB::Universe universe{ "assets/3body.txt" };
    const auto celestialBody = universe[1];

    constexpr float EXPECTED_POSITION_X = 0.0;
    constexpr float EXPECTED_POSITION_Y = 4.50e10;
    constexpr float EXPECTED_VELOCITY_X = 3.00e04;
    constexpr float EXPECTED_VELOCITY_Y = 0.0e00;
    constexpr float EXPECTED_MASS = 1.989e30;
    BOOST_REQUIRE_EQUAL(celestialBody.position().x, EXPECTED_POSITION_X);
    BOOST_REQUIRE_EQUAL(celestialBody.position().y, EXPECTED_POSITION_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().x, EXPECTED_VELOCITY_X);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().y, EXPECTED_VELOCITY_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.mass(), EXPECTED_MASS);
}
```

```

// Tests if `Universe::operator[]` works for the last elements.
BOOST_AUTO_TEST_CASE(testUniverseBracketOperator2) {
    const NB::Universe universe{ "assets/uniform8.txt" };
    const auto celestialBody = universe[universe.numPlanets() - 1];

    constexpr float EXPECTED_POSITION_X = 3.535534e+08;
    constexpr float EXPECTED_POSITION_Y = -3.535534e+08;
    constexpr float EXPECTED_VELOCITY_X = -1.934345e+02;
    constexpr float EXPECTED_VELOCITY_Y = -1.934345e+02;
    constexpr float EXPECTED_MASS = 2.00e+23;
    BOOST_REQUIRE_EQUAL(celestialBody.position().x, EXPECTED_POSITION_X);
    BOOST_REQUIRE_EQUAL(celestialBody.position().y, EXPECTED_POSITION_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().x, EXPECTED_VELOCITY_X);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().y, EXPECTED_VELOCITY_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.mass(), EXPECTED_MASS);
}

// Tests if `Universe::operator[]` works for the last elements.
BOOST_AUTO_TEST_CASE(testUniverseBracketOperator3) {
    const NB::Universe universe{ "assets/8star-rotation.txt" };
    const auto celestialBody = universe[universe.numPlanets() - 1];

    constexpr float EXPECTED_POSITION_X = -13.125e10;
    constexpr float EXPECTED_POSITION_Y = 0;
    constexpr float EXPECTED_VELOCITY_X = 0;
    constexpr float EXPECTED_VELOCITY_Y = 81e3;
    constexpr float EXPECTED_MASS = 5e29;
    BOOST_REQUIRE_EQUAL(celestialBody.position().x, EXPECTED_POSITION_X);
    BOOST_REQUIRE_EQUAL(celestialBody.position().y, EXPECTED_POSITION_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().x, EXPECTED_VELOCITY_X);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().y, EXPECTED_VELOCITY_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.mass(), EXPECTED_MASS);
}

// Tests if "CelestialBody::operator>>" and "CelestialBody::operator<<" works correctly.
BOOST_AUTO_TEST_CASE(testCelestialBodyOutput) {
    std::ifstream fileStream{ "assets/uniform8.txt" };
    std::string line;
    getline(fileStream, line);
    getline(fileStream, line);
    getline(fileStream, line);
    getline(fileStream, line);

    const std::string EXPECT_LINE =
        " 3.535534e+08  3.535534e+08  1.934345e+02 -1.934345e+02 2.00e+23 earth.gif";
    BOOST_REQUIRE_EQUAL(line, EXPECT_LINE);
}

// Tests if "Universe::operator>>" and "Universe::operator<<" works correctly.
BOOST_AUTO_TEST_CASE(testUniverseOutput) {
    const NB::Universe universe{ "assets/3body.txt" };
    std::stringstream stringstream;
    stringstream << universe;
    std::string output = stringstream.str();

    const std::string EXPECT_STRING = "3\n1.25e+11\n"
                                      "0 0 500 0 5.974e+24 earth.gif\n"
                                      "0 4.5e+10 30000 0 1.989e+30 sun.gif\n"
                                      "0 -4.5e+10 -30000 0 1.989e+30 sun.gif\n";
    BOOST_REQUIRE_EQUAL(output, EXPECT_STRING);
}
```

5.2 PS4b

5.2.1 Discussion

Building on the foundation laid in PS4a, this program is designed to load, visualize, and simulate the motion of celestial bodies within a universe. It operates by taking two crucial arguments: the total simulation time T and the time step Δt. Upon execution, it ingests input data, which includes information such as the number of planets, the universe's radius, and specific details for each celestial body like initial position, velocity, mass, and the corresponding image filename. Once all data is gathered, the program presents a visually engaging window displaying the celestial bodies against a captivating background image. Subsequently, it executes a simulation of the celestial bodies' movements over the time span T, employing the leapfrog finite difference approximation scheme with the given time step Δt. Upon the accumulated step time is over total time T, the simulation stops and the final state of the universe will be printed on the console.

In PS4b, the **Universe** and **CelestialBody** classes are enhanced to facilitate the simulation of N-Body motion. Notably, a **step** function is introduced in the **Universe** class, serving as the core of the program.

5.2.2 Achievements

Running the program with the same file as in PS4a and using the following command:

```
./Nobdoy 157788000.0 25000.0 < assets/planets.txt
```

allows us to observe the simulation of the solar system, which typically concludes after approximately 2 minutes. The final state image is depicted in **Figure 4.2**.

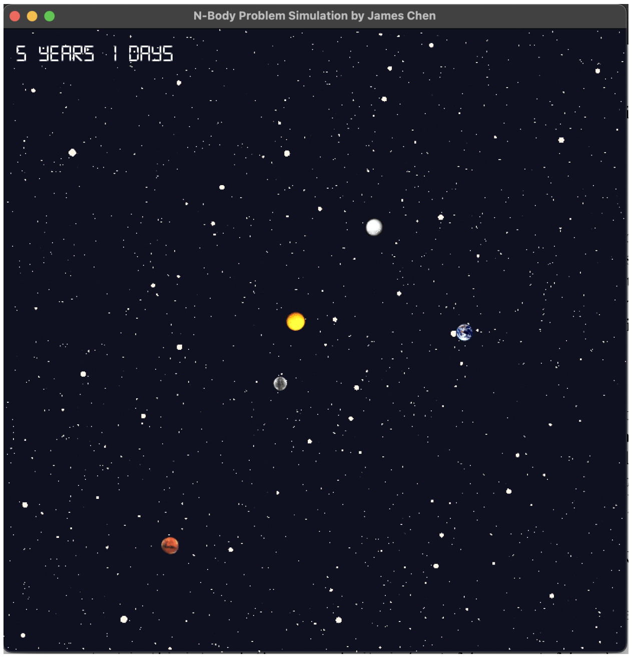

Figure 4.2 A snapshot showcasing the final state image of the NBody program.

5.2.3 Codebase

From this project, I gained proficiency in simulating the movement of celestial bodies through computer programming. I discovered that the continuous motion of celestial bodies can be effectively represented by discrete changes. By updating the position, velocity, and acceleration of each celestial body within defined time intervals (delta time), I learned to achieve accurate results. It became evident that the accuracy of the simulation is directly influenced by the size of the delta time. However, I also realized the importance of striking a balance between simulation time and delta time to ensure computational efficiency without compromising precision.

```makefile
# C++ Compiler
COMPILER = g++

# C++ Flags
CFLAGS = --std=c++20 -Wall -Werror -pedantic -g

# Libraries
LIB = -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -lboost_unit_test_framework

# Code source directory
SRC = ./

# Hpp files (dependencies)
DEPS = $(SRC)Universe.hpp \
       $(SRC)CelestialBody.hpp \
       $(SRC)NBodyConstant.hpp \
       $(SRC)UniverseElapsedTime.hpp

# Static library
STATIC_LIB = NBody.a

# The object files that the static library includes
STATIC_LIB_OBJECTS = $(SRC)Universe.o \
                                     $(SRC)CelestialBody.o \
                                     $(SRC)UniverseElapsedTime.o

# Program
PROGRAM = NBody

# Program object files
MAIN_OBJECTS = $(SRC)main.o

# Test program
TEST_PROGRAM = test

# Test object files
TEST_OBJECTS = $(SRC)test.o

all: $(PROGRAM) $(TEST_PROGRAM)

$(SRC)%.o: $(SRC)%.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

$(PROGRAM): $(MAIN_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

$(STATIC_LIB): $(STATIC_LIB_OBJECTS)
        ar rcs $@ $^

$(TEST_PROGRAM): $(TEST_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

```

```
clean:
        rm -f $(SRC)*.o $(PROGRAM) $(STATIC_LIB) $(TEST_PROGRAM)

lint:
        cpplint *.hpp *.cpp

boost: $(TEST_PROGRAM)
        ./$(TEST_PROGRAM)

run: $(PROGRAM)
        ./$(PROGRAM) 157788000.0 25000.0 < assets/gravity-assist.txt
```

```
// <main.cpp>
#include <iostream>
#include <SFML/Graphics.hpp>
#include "NBodyConstant.hpp"
#include "Universe.hpp"

/**
 * @brief Starts the universe simulation.
 * @param argc The number of arguments.
 * @param argv The arguments vector. This program requires two arguments:
 * 1. T (double): Total time.
 * 2. Delta t (double): time quantum measured in seconds.
 */
int main(const int argc, const char* argv[]) {
    if (argc != 3) {
        std::cerr << "Not enough arguments!";
        return 1;
    }

    const double totalTime = std::stod(argv[1]);
    const double deltaTime = std::stod(argv[2]);
    double elapsedTime = 0.0;

    // Create a universe and load data and resources
    NB::Universe universe;
    std::cin >> universe;
    universe.loadResources();

    // Create a window
    const sf::VideoMode windowVideoMode{ NB::WINDOW_WIDTH, NB::WINDOW_HEIGHT };
    sf::RenderWindow window(windowVideoMode, std::string(NB::WINDOW_TITLE));
    window.setFramerateLimit(NB::WINDOW_FPS);

    bool hasPrintedFinalState = false;
    while (window.isOpen()) {
        sf::Event event{};
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
                break;
            }
        }

        if (window.isOpen()) {
            // The simulation stops when T >= t
            if (elapsedTime < totalTime) {
                universe.step(deltaTime);
            } else if (!hasPrintedFinalState) {
                std::cout << universe;
                hasPrintedFinalState = true;
            }

            window.clear(sf::Color::White);
            window.draw(universe);
            window.display();
        }

        elapsedTime += deltaTime;
    }

    return 0;
}
```

```
// <Universe.hpp>
#ifndef UNIVERSE_HPP
#define UNIVERSE_HPP

#include <memory>
#include <string>
#include <utility>
#include <vector>
#include <SFML/Audio.hpp>
#include <SFML/Graphics.hpp>
#include "CelestialBody.hpp"
#include "UniverseElapsedTime.hpp"

namespace NB {

class CelestialBody;

class Universe final : public UniverseElapsedTime {
 public:
    /**
     * Constructs a Universe.
     */
    Universe();

    /**
     * @brief Constructs a Universe from a file.
     * @param filename The filename to load Universe data from.
     */
    explicit Universe(const std::string& filename);

    /**
     * @brief Loads resources necessary for the Universe (background image and music).
     */
    void loadResources();

    /**
     * @brief Returns the number of planets in this Universe.
     * @return Number of planets.
     */
    [[nodiscard]] int numPlanets() const;

    /**
     * @brief Returns the radius of this Universe.
     * @return Radius of the Universe.
     */
    [[nodiscard]] double radius() const;

    /**
     * @brief Returns the scale of the Universe.
     * @return Scale of the Universe.
     */
    [[nodiscard]] double scale() const;

    /**
     * Simulates one step.
     * @param deltaTime Delta time in seconds.
     */
    void step(double deltaTime) override;

    /**
     * @brief Reads the number of planets and the radius of this Universe from input stream.
     * @param istream Input stream to read from.
     * @param universe Universe object to store the data.
     * @return Reference to the input stream.
     */
    friend std::istream& operator>>(std::istream& istream, Universe& universe);

    /**
     * @brief Writes the number of planets and the radius of this Universe to output stream.
     * @param ostream Output stream to write to.
     * @param universe Universe object to retrieve the data from.
     * @return Reference to the output stream.
     */
    friend std::ostream& operator<<(std::ostream& ostream, const Universe& universe);

    /**
     * @brief Accesses the CelestialBody at the specified index.
     * @param index Index of the CelestialBody to access.
     * @return Reference to the CelestialBody.
     */
    CelestialBody& operator[](const std::size_t& index) const;

protected:
    /**
     * @brief Draws the Universe onto the target.
     * @param target Render target to draw onto.
     * @param states Render states to use for drawing.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

private:
    /**
     * @brief Creates a matrix, in which each element is a double vector.
     */
    std::vector<std::vector<sf::Vector2<double>>> createMatrix() const;

    /**
     * @brief Number of planets in this Universe.
     */
    int m_numPlanets = 0;

    /**
     * @brief Radius of this Universe.
     */
    double m_radius = 0.0;

    /**
     * @brief Ratio of the Universe diameter to the width of the window.
     */
    double m_scale = 1.0;

    /**
     * @brief Vector of CelestialBodies in this Universe.
     */
    std::vector<std::shared_ptr<CelestialBody>> m_celestialBodyVector;

    /**
     * @brief Background image of this Universe.
     */
    std::pair<std::shared_ptr<sf::Texture>, std::shared_ptr<sf::Sprite>> m_backgroundImage;

    /**
     * @brief Background music of this Universe.
     */
    std::pair<std::shared_ptr<sf::SoundBuffer>, std::shared_ptr<sf::Sound>> m_backgroundMusic;
};

}  // namespace NB

#endif
```

The overall implementation remains akin to PS4a. New functions, **magnitude_vector2** and **normalize_vector2**, have been introduced. These functions will play a crucial role in the forthcoming discussion of the **step** function.

```
// <Universe.cpp>
#include "Universe.hpp"
#include <cmath>
#include <fstream>
#include <iostream>
#include <limits>
#include <numeric>
#include <sstream>
#include <SFML/Audio.hpp>
#include "NBodyConstant.hpp"

```

```cpp
/**
 * @brief Converts a double number into a string in the scientific form.
 * @param number The number to convert.
 * @return a string in the form of "x.yye?zz", where x is the integer part, yy is the fraction part,
 * ? is negative sign (optional), and zz is the exponent.
 */
std::string to_standard_scientific_string(const double& number) {
    std::ostringstream stream;
    stream << std::scientific << std::setprecision(2) << number;
    std::string result = stream.str();

    // Remove the '+' sign from the exponent part if present
    const size_t pos = result.find('e');
    if (pos != std::string::npos && result[pos + 1] == '+') {
        result.erase(pos + 1, 1);
    }

    return result;
}

/**
 * @brief Returns the magnitude of a 2-dimensional vector.
 * @tparam T The type of the vector.
 * @param vector2 The 2-dimensional vector to find the magnitude.
 */
template <typename T>
T magnitude_vector2(const sf::Vector2<T>& vector2) {
    return static_cast<T>(std::sqrt(vector2.x * vector2.x + vector2.y * vector2.y));
}

/**
 * @brief Returns the normalized form of a 2-dimensional vector.
 * @tparam T The type of the vector.
 * @param vector2 The 2-dimensional vector to normalize.
 */
template <typename T>
sf::Vector2<T> normalize_vector2(const sf::Vector2<T>& vector2) {
    T magnitude = magnitude_vector2(vector2);
    if (magnitude == 0) {
        throw std::invalid_argument("Cannot normalize a zero vector!");
    }

    return { vector2.x / magnitude, vector2.y / magnitude };
}

namespace NB {

Universe::Universe() = default;

Universe::Universe(const std::string& filename) {
    std::ifstream ifstream{ filename };

    // Check if the file is opened successfully
    if (!ifstream.is_open()) {
        throw std::invalid_argument("Cannot open: " + filename);
    }

    ifstream >> *this;
}

int Universe::numPlanets() const { return m_numPlanets; }

double Universe::radius() const { return m_radius; }

double Universe::scale() const { return m_scale; }
```

The following function, **Universe::step**, is crucial for simulating the motion of celestial bodies within the universe. Here's a breakdown of its functionality:

- **Unit Vector Calculation**: It calculates the unit vector pointing from one planet to another, storing the results in the **unitVectorMatrix**, which stores the unit vector from planet i to planet j.
- **Net Force Calculation**: It computes the net forces between celestial bodies, storing the gravitational forces in the **netForceMatrix**. The **netForceMatrix**$[i][j]$ entry represents the gravitational force exerted on planet i by planet j.
- **Acceleration Calculation**: It calculates the acceleration vector for each planet based on the net forces acting on it. The resultant acceleration vector is determined by summing up the gravitational forces exerted by all other planets.
- **Velocity Update**: It updates the velocity of each planet based on its current velocity and calculated acceleration. The new velocity is determined by adding the product of acceleration and time delta to the current velocity.
- **Position Update**: It updates the position of each planet based on its current position and velocity. The new position is determined by adding the product of velocity and time delta to the current position.
- **Elapsed Time Update**: It updates the elapsed time within the universe.

```
void Universe::step(const double deltaTime) {
    // Calculate the unit vector pointing from one planet to another
    // unitVector[i][j] stores the unit vector from planet i to planet j
    auto unitVectorMatrix = createMatrix();
    for (int i = 0; i < m_numPlanets; ++i) {
        const auto firstPlanet = m_celestialBodyVector[i];
        for (int j = i + 1; j < m_numPlanets; ++j) {
            const auto secondPlanet = m_celestialBodyVector[j];
            const auto firstToSecondUnitVector =
                normalize_vector2((secondPlanet->positionDouble() - firstPlanet->positionDouble()));
            unitVectorMatrix[i][j] = firstToSecondUnitVector;
            unitVectorMatrix[j][i] = { -firstToSecondUnitVector.x, -firstToSecondUnitVector.y };
        }
    }

    // Cacluate the net forces between celestial bodies
    // netForceMatrix[i][j] stores the gravity from planet i to planet j
    // summation of netForceMatrix[i] is the gravity exerted on planet i
    auto netForceMatrix = createMatrix();
    for (int i = 0; i < m_numPlanets; ++i) {
        const auto firstPlanet = m_celestialBodyVector[i];
        for (int j = i + 1; j < m_numPlanets; ++j) {
            const auto secondPlanet = m_celestialBodyVector[j];
            const auto distance =
                magnitude_vector2(firstPlanet->positionDouble() - secondPlanet->positionDouble());
            const auto gravityMagnitude = GravitationalConstant * firstPlanet->massDouble() *
                                          secondPlanet->massDouble() / (distance * distance);

            netForceMatrix[i][j] = unitVectorMatrix[i][j] * gravityMagnitude;
            netForceMatrix[j][i] = unitVectorMatrix[j][i] * gravityMagnitude;
        }
    }

    // Calculate the acceleration
    // accelerationVector[i] stores the acceleration of planet i
    std::vector<sf::Vector2<double>> accelerationVector;
    accelerationVector.reserve(m_numPlanets);
    for (int i = 0; i < m_numPlanets; ++i) {
        // Find the resultant of planet i
        const auto& netForces = netForceMatrix[i];
        sf::Vector2 resultant{ 0.0, 0.0 };
        for (int j = 0; j < m_numPlanets; ++j) {
            resultant = resultant + netForces[j];
        }

        const auto planet = m_celestialBodyVector[i];
        const auto accelerationX = resultant.x / planet->massDouble();
        const auto accelerationY = resultant.y / planet->massDouble();

        accelerationVector.emplace_back(
            std::isnan(accelerationX) ? 0 : accelerationX,
            std::isnan(accelerationY) ? 0 : accelerationY);
    }

    // Calculate the new velocity of each planet
    for (int i = 0; i < m_numPlanets; ++i) {
        const auto planet = m_celestialBodyVector[i];
        const auto velocity = planet->velocityDouble();
        const auto acceleration = accelerationVector[i];

        planet->velocity(
            { velocity.x + acceleration.x * deltaTime, velocity.y + acceleration.y * deltaTime });
    }

    // Calculate the new position of each planet
    for (int i = 0; i < m_numPlanets; ++i) {
        const auto planet = m_celestialBodyVector[i];
        const auto position = planet->positionDouble();
        const auto velocity = planet->velocityDouble();

        planet->position(
            { position.x + velocity.x * deltaTime, position.y + velocity.y * deltaTime });
    }

    // Update the elapsed time
    UniverseElapsedTime::step(deltaTime);
}

void Universe::loadResources() {
    // Load the background image
    m_backgroundImage.first = { std::make_shared<sf::Texture>() };
```

```cpp
    m_backgroundImage.second = { std::make_shared<sf::Sprite>() };
    m_backgroundImage.first->loadFromFile(IMAGE_BACKGRROUND);
    m_backgroundImage.second->setTexture(*m_backgroundImage.first);

    // Rescale the background image so that the image fits the window
    const auto backgroundTexture = m_backgroundImage.first;
    const auto backgroundSprite = m_backgroundImage.second;
    const auto textureSize = backgroundTexture->getSize();
    backgroundSprite->setScale(
        static_cast<float>(WINDOW_WIDTH) / static_cast<float>(textureSize.x),
        static_cast<float>(WINDOW_HEIGHT) / static_cast<float>(textureSize.y));

    // Load and play the background music
    const auto soundBuffer{ std::make_shared<sf::SoundBuffer>() };
    const auto sound{ std::make_shared<sf::Sound>() };
    if (soundBuffer->loadFromFile(SOUND_BACKGROUND_MUSIC)) {
        sound->setBuffer(*soundBuffer);
        m_backgroundMusic.first = soundBuffer;
        m_backgroundMusic.second = sound;
        sound->setLoop(true);
        sound->play();
    }

    // Load celestial bodies' resources
    for (auto const& celestialBody : m_celestialBodyVector) {
        celestialBody->loadResources();
    }
}

void Universe::draw(sf::RenderTarget& target, const sf::RenderStates states) const {
    target.draw(*m_backgroundImage.second, states);

    auto drawCelestialBody = [&](const std::shared_ptr<CelestialBody>& celestialBody) {
        target.draw(*celestialBody, states);
    };

    std::for_each(m_celestialBodyVector.cbegin(), m_celestialBodyVector.cend(), drawCelestialBody);

    // Draw the elapsed time
    UniverseElapsedTime::draw(target, states);
}

std::vector<std::vector<sf::Vector2<double>>> Universe::createMatrix() const {
    std::vector<std::vector<sf::Vector2<double>>> matrix;
    matrix.reserve(m_numPlanets);
    for (int i = 0; i < m_numPlanets; ++i) {
        matrix.push_back(std::vector<sf::Vector2<double>>(m_numPlanets, { 0.0, 0.0 }));
    }

    return matrix;
}

std::istream& operator>>(std::istream& istream, Universe& universe) {
    istream >> universe.m_numPlanets >> universe.m_radius;

    // Set the scale; scale factor enlarges the universe to accommodate all planets' trajectories
    // while they are moving
    universe.m_scale = universe.m_radius / DOUBLE_HALF / WINDOW_WIDTH * SCALE_FACTOR;

    istream.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    // Celestial bodies
    for (int i = 0; i < universe.m_numPlanets; ++i) {
        auto celestialBody = std::make_shared<CelestialBody>(&universe);
        universe.m_celestialBodyVector.push_back(celestialBody);
        istream >> *celestialBody;
    }

    return istream;
}

std::ostream& operator<<(std::ostream& ostream, const Universe& universe) {
    ostream << universe.m_numPlanets << std::endl << universe.m_radius << std::endl;

    // Output celestial bodies
    for (size_t i = 0; i < universe.m_celestialBodyVector.size(); ++i) {
        const auto celestialBody = universe.m_celestialBodyVector[i];
        ostream << *universe.m_celestialBodyVector[i] << std::endl;
    }

    return ostream;
}

CelestialBody& Universe::operator[](const std::size_t& index) const {
    return *m_celestialBodyVector.at(index);
}

}  // namespace NB
```

Nearly nothing is changed in the class **CelestialBody**.

```
// <CelestialBody.hpp>
#ifndef CELESTIALBODY_HPP
#define CELESTIALBODY_HPP

#include <memory>
#include <string>
#include <utility>
#include <SFML/Graphics.hpp>
#include "Universe.hpp"

namespace NB {

class Universe;

class CelestialBody final : public sf::Drawable {
 public:
    /**
     * @brief Constructs a CelestialBody instance.
     */
    CelestialBody();
```

```cpp

    /**
     * @brief Constructs a CelestialBody instance within a specified Universe.
     * @param universePtr Pointer to the Universe this CelestialBody belongs to.
     */
    explicit CelestialBody(Universe* universePtr);

    /**
     * @brief Retrieves the Universe this CelestialBody belongs to.
     * @return Pointer to the Universe.
     */
    [[nodiscard]] Universe* universe() const;

    /**
     * @brief Retrieves the position vector of this CelestialBody.
     * @return Position vector as sf::Vector2f.
     */
    [[nodiscard]] sf::Vector2f position() const;

    /**
     * @brief Retrieves the velocity vector of this CelestialBody.
     * @return Velocity vector as sf::Vector2f.
     */
    [[nodiscard]] sf::Vector2f velocity() const;

    /**
     * @brief Retrieves the mass of this CelestialBody.
     * @return Mass of the CelestialBody.
     */
    [[nodiscard]] float mass() const;

    /**
     * @brief Retrieves the position vector of this CelestialBody.
     * @return Position vector as sf::Vector2<double>.
     */
    [[nodiscard]] sf::Vector2<double> positionDouble() const;

    /**
     * @brief Retrieves the velocity vector of this CelestialBody.
     * @return Velocity vector as sf::Vector2<double>.
     */
    [[nodiscard]] sf::Vector2<double> velocityDouble() const;

    /**
     * @brief Retrieves the mass of this CelestialBody.
     * @return Mass of the CelestialBody.
     */
    [[nodiscard]] double massDouble() const;

    /**
     * @brief Sets the new position for this CelestialBody.
     */
    void position(sf::Vector2<double> newPosition);

    /**
     * @brief Sets the new velocity for this CelestialBody.
     */
    void velocity(sf::Vector2<double> newVelocity);

    /**
     * @brief Loads the resource (image) associated with this CelestialBody.
     */
    void loadResources();

protected:
    /**
     * @brief Draws this CelestialBody onto a given render target.
     * @param target The render target to draw onto.
     * @param states The render states to use for drawing.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

private:
    /**
     * @brief Pointer to the Universe this CelestialBody belongs to.
     */
    Universe* m_universePtr;

    /**
     * @brief The center coordinate of this CelestialBody.
     */
    sf::Vector2<double> m_position;

    /**
     * @brief The velocity vector of this CelestialBody.
     */
    sf::Vector2<double> m_velocity;

    /**
     * @brief The mass of this CelestialBody.
     */
    double m_mass = 0.0;

    /**
     * @brief The filename of the associated image.
     */
    std::string m_image_filename;

    /**
     * @brief Pair containing shared pointers to the texture and sprite of this CelestialBody.
     */
    std::pair<std::shared_ptr<sf::Texture>, std::shared_ptr<sf::Sprite>> m_image;

    /**
     * @brief Reads data from a std::istream into the CelestialBody.
     */
    friend std::istream& operator>>(std::istream& istream, CelestialBody& celestialBody);

    /**
     * @brief Writes data from the CelestialBody to a std::ostream.
     */
    friend std::ostream& operator<<(std::ostream& ostream, const CelestialBody& celestialBody);
```

```
};

}  // namespace NB

#endif
```

```
#include "CelestialBody.hpp"
#include <iostream>
#include <sstream>
#include <boost/algorithm/string.hpp>
#include "NBodyConstant.hpp"

namespace NB {

CelestialBody::CelestialBody() : m_universePtr(nullptr) {}

CelestialBody::CelestialBody(Universe* universePtr) :
    m_universePtr(universePtr) {}

sf::Vector2f CelestialBody::position() const {
    return {
        static_cast<float>(m_position.x),
        static_cast<float>(m_position.y),
    };
}

sf::Vector2f CelestialBody::velocity() const {
    return {
        static_cast<float>(m_velocity.x),
        static_cast<float>(m_velocity.y),
    };
}

float CelestialBody::mass() const { return static_cast<float>(m_mass); }

sf::Vector2<double> CelestialBody::positionDouble() const { return m_position; }

sf::Vector2<double> CelestialBody::velocityDouble() const { return m_velocity; }

double CelestialBody::massDouble() const { return m_mass; }

void CelestialBody::position(const sf::Vector2<double> newPosition) {
    m_position = newPosition;
}

void CelestialBody::velocity(const sf::Vector2<double> newVelocity) {
    m_velocity = newVelocity;
}

void CelestialBody::loadResources() {
    // Load the image file
    m_image.first = std::make_shared<sf::Texture>();
    m_image.second = std::make_shared<sf::Sprite>();
    m_image.first->loadFromFile(ASSETS_IMAGE_DIR / m_image_filename);
    m_image.second->setTexture(*m_image.first);
```

```cpp
}

void CelestialBody::draw(
    sf::RenderTarget& target, const sf::RenderStates states) const {
    if (m_universePtr == nullptr) {
        return;
    }

    const auto universeRadius = m_universePtr->radius();
    const auto universeScale = m_universePtr->scale();
    const auto imageSprite = m_image.second;

    const sf::Vector2f realPosition{
        static_cast<float>(
            (universeRadius + this->m_position.x) / universeScale),
        static_cast<float>(
            (universeRadius - this->m_position.y) / universeScale),
    };
    imageSprite->setPosition(realPosition);

    target.draw(*imageSprite, states);
}

std::istream& operator>>(std::istream& istream, CelestialBody& celestialBody) {
    std::string line;

    // Skip the empty lines
    while (line.empty() && !istream.eof()) {
        getline(istream, line);
    }

    // Read data from the line string
    std::stringstream stringstream(line);
    stringstream >> celestialBody.m_position.x >> celestialBody.m_position.y >>
        celestialBody.m_velocity.x >> celestialBody.m_velocity.y >>
        celestialBody.m_mass >> celestialBody.m_image_filename;

    return istream;
}

std::ostream&
operator<<(std::ostream& ostream, const CelestialBody& celestialBody) {
    const auto position = celestialBody.position();
    const auto velocity = celestialBody.velocityDouble();
    ostream << position.x << " " << position.y << " " << velocity.x << " "
            << velocity.y << " " << celestialBody.mass() << " "
            << celestialBody.m_image_filename;

    return ostream;
}

}  // namespace NB
```

```cpp
// <NBodyConstant.hpp>
#ifndef NBODYCONSTANT_HPP
```

```
#define NBODYCONSTANT_HPP

#include <filesystem>
#include <string>

namespace NB {

// Window title
constexpr std::string_view WINDOW_TITLE = "N-Body Problem Simulation by James Chen";

// Window FPS
constexpr int WINDOW_FPS = 120;

// Window has a fixed height/width; the window is a square
constexpr unsigned WINDOW_WIDTH = 720;
constexpr unsigned WINDOW_HEIGHT = WINDOW_WIDTH;

// Assets directory and subdirectories
const std::filesystem::path ASSETS_DIR = "assets";
const std::filesystem::path ASSETS_IMAGE_DIR = ASSETS_DIR;
const std::filesystem::path ASSETS_SOUND_DIR = ASSETS_DIR;
const std::filesystem::path ASSETS_FONT_DIR = ASSETS_DIR / "font";

// Images
const std::filesystem::path IMAGE_BACKGRROUND = ASSETS_IMAGE_DIR / "background.jpg";

// Sounds
const std::filesystem::path SOUND_BACKGROUND_MUSIC = ASSETS_SOUND_DIR / "2001.wav";

// Fonts
const std::filesystem::path FONT_DIGITAL7 = ASSETS_FONT_DIR / "digital-7.mono.ttf";

// Double constants
constexpr double DOUBLE_HALF = 0.5;
constexpr double SCALE_FACTOR = 1.1;

// Gravitational constant
constexpr double GravitationalConstant = 6.67e-11;

// Seconds in day/year
constexpr int SEONCDS_IN_DAY = 86400;
constexpr int SEONCDS_IN_YEAR = 31536000;

}  // namespace NB

#endif

```

An additional class, **UniverseElapsedTime**, has been introduced to display the simulation time on the upper-right corner of the window. The elapsed time dynamically adjusts its units based on the duration. When the elapsed time is less than one day, it's represented in seconds. If it exceeds one day but is less than a year, it switches to days. Beyond a year, it shows the duration in years and days for comprehensive tracking.

```cpp
// <UniverseElapsedTime.hpp>
#ifndef UNIVERSEELAPSEDTIME_HPP
#define UNIVERSEELAPSEDTIME_HPP
#include <SFML/Graphics.hpp>

namespace NB {

class UniverseElapsedTime : virtual public sf::Drawable {
 protected:
    /**
     * @brief Constructs a UniverseElapsedTime instance.
     */
    UniverseElapsedTime();

    /**
     * @brief Applies one step; adds delta time to the elapsed time.
     * @param deltaTime The delta time in seconds.
     */
    virtual void step(double deltaTime);

    /**
     * @brief Draws the elapsed time onto the target.
     * @param target Render target to draw onto.
     * @param states Render states to use for drawing.
     */
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

 private:
    /**
     * @brief The elapsed time text font size.
     */
    static constexpr unsigned FONT_SIZE = 28;

    /**
     * @brief The x component of the  elapsed time text position.
     */
    static constexpr unsigned FONT_POSITION_X = 15;

    /**
     * @brief The y component of the  elapsed time text position.
     */
    static constexpr unsigned FONT_POSITION_Y = 10;

    /**
     * @brief The elapsed time.
     */
    double m_elapsedTime = 0.0;

    /**
     * @brief The font for the diplayed text.
     */
    sf::Font m_font;
};

}  // namespace NB

```

```
#endif
```

```
#include "UniverseElapsedTime.hpp"
#include <string>
#include "NBodyConstant.hpp"

namespace NB {

UniverseElapsedTime::UniverseElapsedTime() { m_font.loadFromFile(FONT_DIGITAL7); }

void UniverseElapsedTime::step(const double deltaTime) { m_elapsedTime += deltaTime; }

void UniverseElapsedTime::draw(sf::RenderTarget& target, const sf::RenderStates states) const {
    const auto seconds = static_cast<unsigned>(m_elapsedTime);
    const auto days = (seconds / SEONCDS_IN_DAY) % 365;
    const auto years = seconds / SEONCDS_IN_YEAR;
    const auto secondsStr = std::to_string(seconds);
    const auto daysStr = std::to_string(days);
    const auto yearsStr = std::to_string(years);
    std::string stringToPrint;

    if (seconds >= SEONCDS_IN_YEAR) {
        stringToPrint = yearsStr + " years " + daysStr + " days";
    } else if (seconds >= SEONCDS_IN_DAY) {
        stringToPrint = daysStr + " days";
    } else {
        stringToPrint = secondsStr + " seconds";
    }

    sf::Text text;
    text.setFont(m_font);
    text.setString(stringToPrint);
    text.setCharacterSize(FONT_SIZE);
    text.setFillColor(sf::Color::White);
    text.setPosition(FONT_POSITION_X, FONT_POSITION_Y);
    target.draw(text);
}

}  // namespace NB
```

A bunch of tests are facilitated to verify the correctness of functionalities of the classes.

```
// <test.cpp>
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main

#include <cmath>
#include <fstream>
#include <iostream>
#include <sstream>
#include <boost/test/unit_test.hpp>
#include "Universe.hpp"

constexpr float MASS_MAX_TOLERANCE = 0.1;
constexpr float STEP_TOLERANCE = 0.1;

// Tests if `Universe::getNumPlanets()` and `Universe::getRadius` work correcly.
BOOST_AUTO_TEST_CASE(testUniverseBasic) {
    const NB::Universe universe{ "assets/1body.txt" };

    constexpr auto EXPECTED_NUM_PLANETS = 1;
    constexpr auto EXPECTED_RADIUS = 100.0;
    BOOST_REQUIRE_EQUAL(universe.numPlanets(), EXPECTED_NUM_PLANETS);
    BOOST_REQUIRE_EQUAL(universe.radius(), EXPECTED_RADIUS);
}

// Tests if `Universe::getNumPlanets()` and `Universe::getRadius` work correcly.
BOOST_AUTO_TEST_CASE(testUniverseBasic2) {
    const NB::Universe universe{ "assets/binary.txt" };

    constexpr auto EXPECTED_NUM_PLANETS = 2;
    constexpr auto EXPECTED_RADIUS = 5.0e10;
    BOOST_REQUIRE_EQUAL(universe.numPlanets(), EXPECTED_NUM_PLANETS);
    BOOST_REQUIRE_EQUAL(universe.radius(), EXPECTED_RADIUS);
}

// Tests if `CelestialBody::getNumPlanets()`, `CelestialBody::getRadius()` and
// `CelestialBody::getMass()` work correcly.
BOOST_AUTO_TEST_CASE(testCelestialBodyBasic) {
    const NB::Universe universe{ "assets/planets.txt" };
    const auto celestialBody = universe[0];

    constexpr float EXPECTED_POSITION_X = 1.4960e+11;
    constexpr float EXPECTED_POSITION_Y = 0.0000e+00;
    constexpr float EXPECTED_VELOCITY_X = 0.0000e+00;
    constexpr float EXPECTED_VELOCITY_Y = 2.9800e+04;
    constexpr float EXPECTED_MASS = 5.9740e+24;
    BOOST_REQUIRE_EQUAL(celestialBody.position().x, EXPECTED_POSITION_X);
    BOOST_REQUIRE_EQUAL(celestialBody.position().y, EXPECTED_POSITION_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().x, EXPECTED_VELOCITY_X);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().y, EXPECTED_VELOCITY_Y);
    BOOST_REQUIRE_CLOSE(celestialBody.mass(), EXPECTED_MASS, MASS_MAX_TOLERANCE);
}

// Tests if `Universe::operator[]` works for the non-first elements.
BOOST_AUTO_TEST_CASE(testUniverseBracketOperator1) {
    const NB::Universe universe{ "assets/3body.txt" };
    const auto celestialBody = universe[1];

    constexpr float EXPECTED_POSITION_X = 0.0;
    constexpr float EXPECTED_POSITION_Y = 4.50e10;
    constexpr float EXPECTED_VELOCITY_X = 3.00e04;
    constexpr float EXPECTED_VELOCITY_Y = 0.0e00;
    constexpr float EXPECTED_MASS = 1.989e30;
    BOOST_REQUIRE_EQUAL(celestialBody.position().x, EXPECTED_POSITION_X);
    BOOST_REQUIRE_EQUAL(celestialBody.position().y, EXPECTED_POSITION_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().x, EXPECTED_VELOCITY_X);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().y, EXPECTED_VELOCITY_Y);
    BOOST_REQUIRE_CLOSE(celestialBody.mass(), EXPECTED_MASS, MASS_MAX_TOLERANCE);
}
```

```cpp

// Tests if `Universe::operator[]` works for the last elements.
BOOST_AUTO_TEST_CASE(testUniverseBracketOperator3) {
    const NB::Universe universe{ "assets/8star-rotation.txt" };
    const auto celestialBody = universe[universe.numPlanets() - 1];

    constexpr float EXPECTED_POSITION_X = -13.125e10;
    constexpr float EXPECTED_POSITION_Y = 0;
    constexpr float EXPECTED_VELOCITY_X = 0;
    constexpr float EXPECTED_VELOCITY_Y = 81e3;
    constexpr float EXPECTED_MASS = 5e29;
    BOOST_REQUIRE_EQUAL(celestialBody.position().x, EXPECTED_POSITION_X);
    BOOST_REQUIRE_EQUAL(celestialBody.position().y, EXPECTED_POSITION_Y);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().x, EXPECTED_VELOCITY_X);
    BOOST_REQUIRE_EQUAL(celestialBody.velocity().y, EXPECTED_VELOCITY_Y);
    BOOST_REQUIRE_CLOSE(celestialBody.mass(), EXPECTED_MASS, MASS_MAX_TOLERANCE);
}

// Tests if "CelestialBody::operator>>" and "CelestialBody::operator<<" works correctly.
BOOST_AUTO_TEST_CASE(testCelestialBodyOutput) {
    std::ifstream fileStream{ "assets/uniform8.txt" };
    std::string line;
    getline(fileStream, line);
    getline(fileStream, line);
    getline(fileStream, line);
    getline(fileStream, line);
    std::istringstream istringstream{ line };
    NB::CelestialBody celestialBody;
    istringstream >> celestialBody;

    std::ostringstream ostringstream;
    ostringstream << celestialBody;

    // original line:
    // 3.535534e+08  3.535534e+08  1.934345e+02 -1.934345e+02 2.00e+23 earth.gif
    const std::string EXPECT_LINE = "3.53553e+08 3.53553e+08 193.435 -193.435 2e+23 earth.gif";
    BOOST_REQUIRE_EQUAL(ostringstream.str(), EXPECT_LINE);
}

// Tests if `Universe::step()` works correctly by performing one step.
BOOST_AUTO_TEST_CASE(testUniverseStep1) {
    NB::Universe universe{ "assets/planets.txt" };
    constexpr double DELTA_TIME = 25000.0;
    universe.step(DELTA_TIME);

    constexpr float EXPECTED_POSITION_X = 1.4960e+11F;
    constexpr float EXPECTED_POSITION_Y = 7.4500e+08F;
    constexpr float EXPECTED_VELOCITY_X = -1.4820e+02F;
    constexpr float EXPECTED_VELOCITY_Y = 2.9800e+04F;
    const auto celestialBody = universe[0];
    BOOST_REQUIRE_CLOSE(celestialBody.position().x, EXPECTED_POSITION_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.position().y, EXPECTED_POSITION_Y, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().x, EXPECTED_VELOCITY_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().y, EXPECTED_VELOCITY_Y, STEP_TOLERANCE);
}

```

```cpp
// Tests if `Universe::step()` works correctly by performing two steps.
BOOST_AUTO_TEST_CASE(testUniverseStep2) {
    NB::Universe universe{ "assets/planets.txt" };
    constexpr double DELTA_TIME = 25000.0;
    universe.step(DELTA_TIME);
    universe.step(DELTA_TIME);

    constexpr float EXPECTED_POSITION_X = 2.2790e+11F;
    constexpr float EXPECTED_POSITION_Y = 1.2050e+09F;
    constexpr float EXPECTED_VELOCITY_X = -1.2772e+02F;
    constexpr float EXPECTED_VELOCITY_Y = 2.4100e+04F;
    const auto celestialBody = universe[1];
    BOOST_REQUIRE_CLOSE(celestialBody.position().x, EXPECTED_POSITION_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.position().y, EXPECTED_POSITION_Y, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().x, EXPECTED_VELOCITY_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().y, EXPECTED_VELOCITY_Y, STEP_TOLERANCE);
}

// Tests if `Universe::step()` works correctly by performing three steps.
BOOST_AUTO_TEST_CASE(testUniverseStep3) {
    NB::Universe universe{ "assets/planets.txt" };
    constexpr double DELTA_TIME = 25000.0;
    universe.step(DELTA_TIME);
    universe.step(DELTA_TIME);
    universe.step(DELTA_TIME);

    constexpr float EXPECTED_POSITION_X = 2.2789e+11F;
    constexpr float EXPECTED_POSITION_Y = 1.8075e+09F;
    constexpr float EXPECTED_VELOCITY_X = -1.9158e+02F;
    constexpr float EXPECTED_VELOCITY_Y = 2.4099e+04F;
    const auto celestialBody = universe[1];
    BOOST_REQUIRE_CLOSE(celestialBody.position().x, EXPECTED_POSITION_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.position().y, EXPECTED_POSITION_Y, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().x, EXPECTED_VELOCITY_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().y, EXPECTED_VELOCITY_Y, STEP_TOLERANCE);
}

// Tests if `Universe::step()` works correctly by performing multiple steps until the time reach one
// year.
BOOST_AUTO_TEST_CASE(testUniverseStepOneYear) {
    NB::Universe universe{ "assets/planets.txt" };
    constexpr double TOTAL_TIME = 31557600.0;
    double accumulatedTime = 0.0;
    while (accumulatedTime < TOTAL_TIME) {
        constexpr double DELTA_TIME = 25000.0;
        universe.step(DELTA_TIME);
        accumulatedTime += DELTA_TIME;
    }

    constexpr float EXPECTED_POSITION_X = -7.3731e+10F;
    constexpr float EXPECTED_POSITION_Y = -7.9391e+10F;
    constexpr float EXPECTED_VELOCITY_X = 2.5433e+04F;
    constexpr float EXPECTED_VELOCITY_Y = -2.3973e+04F;
    const auto celestialBody = universe[4];
    BOOST_REQUIRE_CLOSE(celestialBody.position().x, EXPECTED_POSITION_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.position().y, EXPECTED_POSITION_Y, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().x, EXPECTED_VELOCITY_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().y, EXPECTED_VELOCITY_Y, STEP_TOLERANCE);
}

// Tests if `Universe::step()` works correctly for negative delta time.
BOOST_AUTO_TEST_CASE(testUniverseStepNegativeDeltaTime) {
    NB::Universe universe{ "assets/planets.txt" };
    constexpr double DELTA_TIME = -25000.0;
    universe.step(DELTA_TIME);

    constexpr float EXPECTED_POSITION_X = 2.2789e11F;
    constexpr float EXPECTED_POSITION_Y = -602499968.F;
    constexpr float EXPECTED_VELOCITY_X = 63.8597F;
    constexpr float EXPECTED_VELOCITY_Y = 24100.F;
    const auto celestialBody = universe[1];
    BOOST_REQUIRE_CLOSE(celestialBody.position().x, EXPECTED_POSITION_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.position().y, EXPECTED_POSITION_Y, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().x, EXPECTED_VELOCITY_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().y, EXPECTED_VELOCITY_Y, STEP_TOLERANCE);
}

// Tests if `Universe::step()` works correctly by passing to different delta time consequently.
BOOST_AUTO_TEST_CASE(testUniverseStepDifferentDeltaTime) {
    NB::Universe universe{ "assets/planets.txt" };
    constexpr double DELTA_TIME_1 = 25000.0;
    constexpr double DELTA_TIME_2 = 45000.0;
    universe.step(DELTA_TIME_1);
    universe.step(DELTA_TIME_2);

    constexpr float EXPECTED_POSITION_X = 2.2789e11F;
    constexpr float EXPECTED_POSITION_Y = 1.68699e09F;
    constexpr float EXPECTED_VELOCITY_X = -178.808F;
    constexpr float EXPECTED_VELOCITY_Y = 24099.7F;
    const auto celestialBody = universe[1];
    BOOST_REQUIRE_CLOSE(celestialBody.position().x, EXPECTED_POSITION_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.position().y, EXPECTED_POSITION_Y, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().x, EXPECTED_VELOCITY_X, STEP_TOLERANCE);
    BOOST_REQUIRE_CLOSE(celestialBody.velocity().y, EXPECTED_VELOCITY_Y, STEP_TOLERANCE);
}
```

6 PS5 - DNA Sequence Alignment

6.1 Discussion

PS5 is the project that I am most proud of among all. This innovative program tackles a core challenge in computational biology: DNA sequence alignment. Leveraging the timeless Needleman-Wunsch method and Hirschberg's algorithm, it offers robust solutions to this crucial task. What sets this program apart is its versatile approach, employing various classes to implement functionalities in distinct yet harmoniously integrated ways, ensuring a seamless and efficient alignment process.

Through this project, I delved into a renowned algorithm within the realm of Bioinformatics and its optimization techniques. This journey provided me with a comprehensive understanding of dynamic programming, divide and conquer strategies, and the art of algorithm optimization.

6.2 Achievements

We can run the program to find the edit distances of two different strings. Suppose our input is as follows:

```
abcdefghizzzzjklmnop
azzbcdefghijklmnop
```

Where the first line is gene ”X” and the second line is gene ”Y”. We can obtain a dynamic programming matrix using Needleman-Wunsch algorithm.

```
12  11   9  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38  40
13  12  10   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38
11  12  12  10   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36
12  10  11  12  10   8  10  12  14  16  18  20  22  24  26  28  30  32  34
13  11   9  10  11  10   8  10  12  14  16  18  20  22  24  26  28  30  32
14  12  10   8   9  10  10   8  10  12  14  16  18  20  22  24  26  28  30
15  13  11   9   7   8   9  10   8  10  12  14  16  18  20  22  24  26  28
15  14  12  10   8   6   7   8   9   8  10  12  14  16  18  20  22  24  26
15  14  13  11   9   7   5   6   7   8   8  10  12  14  16  18  20  22  24
16  14  13  12  10   8   6   4   5   6   7   8  10  12  14  16  18  20  22
17  15  14  13  11   9   7   5   3   4   5   6   8  10  12  14  16  18  20
18  16  15  14  12  10   8   6   4   2   3   4   6   8  10  12  14  16  18
20  18  16  15  13  11   9   7   5   3   1   2   4   6   8  10  12  14  16
22  20  18  16  14  12  10   8   6   4   2   0   2   4   6   8  10  12  14
24  22  20  18  16  14  12  10   8   6   4   2   0   2   4   6   8  10  12
26  24  22  20  18  16  14  12  10   8   6   4   2   0   2   4   6   8  10
28  26  24  22  20  18  16  14  12  10   8   6   4   2   0   2   4   6   8
30  28  26  24  22  20  18  16  14  12  10   8   6   4   2   0   2   4   6
32  30  28  26  24  22  20  18  16  14  12  10   8   6   4   2   0   2   4
34  32  30  28  26  24  22  20  18  16  14  12  10   8   6   4   2   0   2
36  34  32  30  28  26  24  22  20  18  16  14  12  10   8   6   4   2   0
```

And the output of the program is:

```
Edit distance: 12
a a 0
- z 2
- z 2
b b 0
c c 0
d d 0
```

```
e e 0
f f 0
g g 0
h h 0
i i 0
z - 2
z - 2
z - 2
z - 2
j j 0
k k 0
l l 0
m m 0
n n 0
o o 0
p p 0

Execution time is: 0.000315 seconds
```

Table 5.1 The result and performance on different data files.

Data File	Size(N)	Distance	Memory(MB)	Time(Seconds)
ecoli2500.txt	2500	118	1.3	0.089
ecoli5000.txt	5000	160	1.7	0.337
ecoli10000.txt	10000	223	2.5	1.130
ecoli20000.txt	20000	3135	4.5	4.410
ecoli50000.txt	50000	19523	7.0	26.294
ecoli100000.txt	100000	24189	11.1	101.811
ecoli500000.txt	500000	187916	33.2	2542.79

I have conducted tests using various file sizes on my MacBook Pro with 8GB of memory and an M1 chip. The results are summarized in **Table 5.1**. It's worth noting that the optimization flag ”-O3” is applied during the program compilation process.

From the data presented in **Table 5.1**, we observe a relationship between the time in seconds and the input size, approximately given by:

$$2.9092 \times N^{1.9772}$$

Where N is the size of the input. If computation time is restricted to one day, the largest input size that my program can handle is approximately 2033689.

6.3 Codebase

```
# C++ Compiler
COMPILER = g++

# C++ Flags (C++ version: 20)
CFLAGS = --std=c++20 -Wall -Werror -pedantic -g -O3

# Libraries
LIB = -lsfml-system -lboost_unit_test_framework
```

```

# Code source directory
SRC = ./

# Hpp files (dependencies)
DEPS = $(SRC)EDistance.hpp \
           $(SRC)AbstractEDistance.hpp \
           $(SRC)NeedlemanWunschEDistance.hpp \
           $(SRC)OptimizedEDistance.hpp \
           $(SRC)HirshbergEDistance.hpp

# Static library
STATIC_LIB = EDistance.a

# The object files that the static library includes
STATIC_LIB_OBJECTS = $(SRC)EDistance.o \
                                         $(SRC)AbstractEDistance.o \
                                         $(SRC)NeedlemanWunschEDistance.o \
                                         $(SRC)OptimizedEDistance.o \
                                         $(SRC)HirshbergEDistance.o

# Program
PROGRAM = EDistance

# Program object files
MAIN_OBJECTS = $(SRC)main.o

# Test program
TEST_PROGRAM = test

# Test object files
TEST_OBJECTS = $(SRC)test.o

all: $(PROGRAM) $(TEST_PROGRAM)

$(SRC)%.o: $(SRC)%.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

$(PROGRAM): $(MAIN_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

$(STATIC_LIB): $(STATIC_LIB_OBJECTS)
        ar rcs $@ $^

$(TEST_PROGRAM): $(TEST_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

clean:
        rm -f $(SRC)*.o $(PROGRAM) $(STATIC_LIB) $(TEST_PROGRAM)

lint:
        cpplint *.hpp *.cpp

boost: $(TEST_PROGRAM)
        ./$(TEST_PROGRAM)

```

```
run: $(PROGRAM)
        ./$(PROGRAM) < sequence/example10.txt

run-20: $(PROGRAM)
        ./$(PROGRAM) < sequence/bothgaps20.txt
```

In the ‘main‘ function, we print both the elapsed time taken to find the edit distance and the alignment.

```
// <main.cpp>
#include <iostream>
#include <SFML/System.hpp>
#include "EDistance.hpp"
#include "NeedlemanWunschEDistance.hpp"
#include "OptimizedEDistance.hpp"

/**
 * @brief Finds the edit distance of two genes.
 */
int main() {
    const sf::Clock clock;

    // Read the two genes and find the edit distance
    std::string geneX;
    std::string geneY;
    std::cin >> geneX >> geneY;
    EDistance eDistance{ geneX, geneY };
    const auto editDistance = eDistance.optDistance();

    // Print the edit distance
    std::cout << "Edit distance: " << editDistance << std::endl;

    // Print the alignment table
    std::cout << eDistance.alignment() << std::endl;

    // Print the elapsed time in seconds
    const auto elapsedTime = clock.getElapsedTime();
    std::cout << "Execution time is: " << elapsedTime.asSeconds() << " seconds" << std::endl;

    return 0;
}
```

The **EDistance** class simply extends **HirshbergEDistance**.

```
// <EDistance.hpp>
#ifndef EDISTANCE_HPP
#define EDISTANCE_HPP

#include <string>
#include "HirshbergEDistance.hpp"

/**
 * @brief An implementation of AbstractEDistance using Hirschberg’s Algorithm.
 */
class EDistance final : public HirshbergEDistance {
```

```cpp
 public:
    /**
     * @brief Calculates the penalty for aligning characters 'a' and 'b'.
     * @param num1 The first character.
     * @param num2 The second character.
     * @return The penalty value, which is 0 if the characters are identical, and 1 otherwise.
     */
    static int penalty(const char& num1, const char& num2);

    /**
     * @brief Returns the minimum of three integer values.
     * @param a The first integer.
     * @param b The second integer.
     * @param c The third integer.
     * @return The minimum value among the three integers.
     */
    static int min3(const int& a, const int& b, const int& c);

    /**
     * Constructs an EDistance instance with two input genes.
     * @param geneX The first gene string.
     * @param geneY The second gene string.
     */
    EDistance(const std::string& geneX, const std::string& geneY);
};

#endif

// <EDistance.cpp>
#include "EDistance.hpp"
#include <string>
#include "AbstractEDistance.hpp"

int EDistance::penalty(const char& num1, const char& num2) {
    return AbstractEDistance::penalty(num1, num2);
}

int EDistance::min3(const int& a, const int& b, const int& c) {
    return AbstractEDistance::min3(a, b, c);
}

EDistance::EDistance(const std::string& geneX, const std::string& geneY) :
    HirshbergEDistance(geneX, geneY) {}
```

Firstly, I’ve devised an abstract class named ‘**AbstractEDistance**‘, comprising static helper functions, a constructor that takes two strings for comparison, and two abstract methods: ‘**optDistance**‘ and ‘**alignment**‘. The ‘**optDistance**‘ method determines the optimal edit distance between the two strings, while the ‘**alignment**‘ method displays the optimal alignment of the gene strings. We’ll delve into three distinct approaches to implementing this abstract class shortly.

```cpp
// <AbstractEDistance.hpp>
#ifndef ABSTRACTEDISTANCE_HPP
#define ABSTRACTEDISTANCE_HPP

#include <string>
```

```cpp

/**
 * @brief Abstract base class for calculating edit distance between two gene strings. Subclasses
 * should populate an internal matrix to compute the edit distance efficiently. The gene strings to
 * be compared are provided during construction.
 */
class AbstractEDistance {
 public:
    /**
     * @brief Calculates the penalty for aligning characters 'a' and 'b'.
     * @param num1 The first character.
     * @param num2 The second character.
     * @return The penalty value, which is 0 if the characters are identical, and 1 otherwise.
     */
    static int penalty(const char& num1, const char& num2);

    /**
     * @brief Returns the minimum of three integer values.
     * @param a The first integer.
     * @param b The second integer.
     * @param c The third integer.
     * @return The minimum value among the three integers.
     */
    static int min3(const int& a, const int& b, const int& c);

    /**
     * @brief Calculates the optimal edit distance between two gene strings.
     * Populates the internal matrix and returns the optimal distance (from the [0][0] cell
     * of the matrix when done).
     * @return The optimal edit distance between the two gene strings.
     */
    virtual int optDistance() = 0;

    /**
     * @brief Traces the populated matrix and returns a string representing the optimal
     * alignment of the two gene strings.
     * @return A string representing the optimal alignment.
     */
    [[nodiscard]] virtual std::string alignment() const = 0;

    /**
     * @brief Destructor for AbstractEDistance.
     */
    virtual ~AbstractEDistance();

 protected:
    /**
     * @brief Constructs an AbstractEDistance instance with two input genes.
     * @param geneX The first gene string.
     * @param geneY The second gene string.
     */
    AbstractEDistance(std::string geneX, std::string geneY);

    /**
     * @brief The first gene string.
     */
    const std::string m_geneX;

    /**
     * @brief The second gene string.
     */
    const std::string m_geneY;
};

#endif

// <AbstractEDistance.cpp>
#include "AbstractEDistance.hpp"
#include <algorithm>
#include <utility>

int AbstractEDistance::penalty(const char& num1, const char& num2) { return num1 == num2 ? 0 : 1; }

int AbstractEDistance::min3(const int& a, const int& b, const int& c) {
    return std::min({ a, b, c });
}

AbstractEDistance::AbstractEDistance(std::string geneX, std::string geneY) :
    m_geneX(std::move(geneX)), m_geneY(std::move(geneY)) {}

AbstractEDistance::~AbstractEDistance() = default;
```

In our initial implementation, we employ the Needleman-Wunsch algorithm. The **NeedlemanWunschEDistance** class extends **AbstractEDistance** and utilizes a two-dimensional array to encapsulate the entire matrix. Assuming the lengths of the two input strings are m and n, both the time and space complexities of this algorithm amount to $O(mn)$. The time complexity of the **alignment** method is $O(m+n)$.

```
// <NeedlemanWunschEDistance.hpp>
#ifndef NEEDLEMANWUNSCHEDISTANCE_HPP
#define NEEDLEMANWUNSCHEDISTANCE_HPP

#include <string>
#include "AbstractEDistance.hpp"

/**
 * @brief An implementation of AbstractEDistance using Needleman & Wunsch's method.
 */
class NeedlemanWunschEDistance final : AbstractEDistance {
 public:
    /**
     * Constructs an NeedlemanWunschEDistance instance with two input genes.
     * @param geneX The first gene string.
     * @param geneY The second gene string.
     */
    NeedlemanWunschEDistance(const std::string& geneX, const std::string& geneY);

    /**
     * @brief Destructor for AbstractEDistance.
     */
    ~NeedlemanWunschEDistance() override;

    /**
     * @brief Calculates the optimal edit distance between the two gene strings. Populates the
     * internal matrix and returns the optimal distance (from the [0][0] cell of the matrix when
     * done).
     * @return The optimal edit distance between the two gene strings.
     */
    int optDistance() override;

    /**
     * @brief Traces the populated matrix and returns a string representing the optimal
     * alignment of the two gene strings.
     * @return A string representing the optimal alignment.
     */
    [[nodiscard]] std::string alignment() const override;

    /**
     * @brief Return the matrix used to trace the edit distances.
     */
    [[nodiscard]] int** matrix() const;

 private:
    /**
     * @brief Prints the matrix. This function is used for debugging.
     */
    void printMatrix() const;

    /**
     * @brief The matrix used to trace the edit distances.
     */
    int** m_matrix;
};

#endif

// <NeedlemanWunschEDistance.cp>
#include "NeedlemanWunschEDistance.hpp"
#include <cmath>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
#include <utility>

NeedlemanWunschEDistance::NeedlemanWunschEDistance(
    const std::string& geneX, const std::string& geneY) :
    AbstractEDistance(geneX, geneY), m_matrix(nullptr) {}

int NeedlemanWunschEDistance::optDistance() {
    // Initialize the matrix
    const size_t rowCount = m_geneX.length() + 1;
    const size_t colCount = m_geneY.length() + 1;
    m_matrix = new int*[rowCount];
    for (size_t i = 0; i < rowCount; ++i) {
        m_matrix[i] = new int[colCount]{};
    }

    // Filling the last row and last column
    auto* const lastRow = m_matrix[rowCount - 1];
    for (size_t i = colCount - 1; i < colCount; --i) {
        lastRow[i] = static_cast<int>((colCount - 1 - i)) << 1;
    }
    for (size_t i = rowCount - 1; i < rowCount; --i) {
        m_matrix[i][colCount - 1] = static_cast<int>((rowCount - 1 - i)) << 1;
    }

    // Populate the matrix column by column
    for (size_t i = colCount - 2; i < colCount; --i) {
        const char xChar = m_geneY.at(i);
        for (size_t j = rowCount - 2; j < rowCount; --j) {
            const char yChar = m_geneX.at(j);
            const auto fromRight = m_matrix[j][i + 1] + 2;
            const auto fromDown = m_matrix[j + 1][i] + 2;
            const auto fromDiagonal = m_matrix[j + 1][i + 1] + penalty(xChar, yChar);
            m_matrix[j][i] = min3(fromRight, fromDown, fromDiagonal);
        }
    }

    return m_matrix[0][0];
}

std::string NeedlemanWunschEDistance::alignment() const {
    static constexpr auto CHAR_GAP = '-';
    static constexpr auto CHAR_SPACE = ' ';

    std::ostringstream ostringstream;
    const size_t maxRowIndex = m_geneX.length();
    const size_t maxColIndex = m_geneY.length();
    const size_t maxIndexSum = maxRowIndex + maxColIndex;

    size_t i = 0;
    size_t j = 0;
    while (i + j < maxIndexSum) {
        const auto val = m_matrix[i][j];
        const auto isFromBottom = i < maxRowIndex && m_matrix[i + 1][j] == val - 2;
        const auto isFromRight = j < maxColIndex && m_matrix[i][j + 1] == val - 2;

        // ===================================
        // | isFromBottom | isFromRight | i++ | j++ | update xChar | update yChar |
        // |            0 |           0 |   1 |   1 |            0 |            0 |
        // |            0 |           1 |   0 |   1 |            1 |            0 |
        // |            1 |           0 |   1 |   0 |            0 |            1 |
        // |            1 |           1 |   1 |   0 |            0 |            1 |
        // ===================================
        //
        // i++ = p || !(p || q)
        // j++ = !p
        // update xChar = !isFromBottom && isFromRight
        // update yChar = isFromBottom
        // !(update xChar) = isFromBottom || !isFromRight
        // !(update yChar) = !isFromBottom

        const auto xChar =
```

```cpp
            i < maxRowIndex && (isFromBottom || !isFromRight) ? m_geneX.at(i) : CHAR_GAP;
        const auto yChar = j < maxColIndex && !isFromBottom ? m_geneY.at(j) : CHAR_GAP;
        const auto cost = !isFromBottom && !isFromRight ? penalty(xChar, yChar) : 2;

        // Increment
        i += static_cast<int>(isFromBottom || !(isFromRight || isFromBottom));
        j += static_cast<int>(!isFromBottom);

        // Output: "<xChar> <yChar> <cost>\n"
        ostringstream << xChar << CHAR_SPACE << yChar << CHAR_SPACE << cost << std::endl;
    }

    return ostringstream.str();
}

NeedlemanWunschEDistance::~NeedlemanWunschEDistance() {
    if (m_matrix != nullptr) {
        for (size_t i = 0; i <= m_geneX.length(); ++i) {
            delete m_matrix[i];
        }

        delete m_matrix;
    }
}

int** NeedlemanWunschEDistance::matrix() const { return m_matrix; }

void NeedlemanWunschEDistance::printMatrix() const {
    for (size_t row = 0; row < m_geneX.length() + 1; ++row) {
        for (size_t col = 0; col < m_geneY.length() + 1; ++col) {
            std::cout << std::setw(4) << m_matrix[row][col] << " ";
        }
        std::cout << std::endl;
    }
}
```

Given the typical assumption that the lengths of the compared strings are similar, let's denote their length as n. Consequently, both the time and space complexities of **NeedlemanWunschEDistance** are bounded by $O(n^2)$.

An important observation lies in the way we populate the matrix within the **optDistance** method in **NeedlemanWunschEDistance**: it's done column by column. Consequently, we can maintain just a single column at any given moment until we reach the final column, thereby determining the edit distance. To address this optimization, I devised **OptimizedEDistance**, which implements the **NeedlemanWunschEDistance** algorithm using a one-dimensional array, effectively slashing the space complexity down to $O(n)$.

Nevertheless, due to the fact that the complete matrix isn't retained during the population process, we're unable to backtrack the arrow path. Consequently, this impedes the implementation of the ‘**alignment**‘ method.

```cpp
// <OptimizedEDistance.hpp>
#ifndef OPTIMIZEDEDISTANCE_H
#define OPTIMIZEDEDISTANCE_H

#include <string>
```

```cpp
#include <vector>
#include "AbstractEDistance.hpp"

/**
 * @brief An implementation of AbstractEDistance using Needleman & Wunsch's method with O(n)
 * space complexity, where n is the length of the gene X.
 */
class OptimizedEDistance final : AbstractEDistance {
 public:
    /**
     * Constructs an NeedlemanWunschEDistance instance with two input genes.
     * @param geneX The first gene string.
     * @param geneY The second gene string.
     */
    OptimizedEDistance(const std::string& geneX, const std::string& geneY);

    /**
     * @brief Calculates the optimal edit distance between the two gene strings.
     * @return The optimal edit distance between the two gene strings.
     */
    int optDistance() override;

    /**
     * @brief Since no matrix is maintained, backtracking the alignment path is not feasible,
     *        hence this method is not implemented.
     * @throws std::exception Always throws an exception.
     */
    [[nodiscard]] std::string alignment() const override;

    /**
     * @brief Returns the row array containing the first row's elements during the population process.
     */
    [[nodiscard]] std::vector<int> row() const;

 private:
    /**
     * @brief The column vector used for populating the virtual matrix.
     */
    std::vector<int> m_column;

    /**
     * @brief The row vector containing the first row's elements during the population process.
     */
    std::vector<int> m_row;
};

#endif

// <OptimizedEDistance.cpp>
#include "OptimizedEDistance.hpp"

OptimizedEDistance::OptimizedEDistance(const std::string& geneX, const std::string& geneY) :
    AbstractEDistance(geneX, geneY) {
    m_column.resize(geneX.length() + 1, 0);
    m_row.resize(geneY.length() + 1, 0);
}
```

```cpp

int OptimizedEDistance::optDistance() {
    // Initialize column
    const size_t rowCount = m_geneX.length() + 1;
    const size_t colCount = m_geneY.length() + 1;
    for (size_t i = rowCount - 1; i < rowCount; --i) {
        m_column[i] = static_cast<int>((rowCount - 1 - i)) << 1;
    }

    m_row[colCount - 1] = m_column[0];

    // Update the column as if populating the matrix using Needleman-Wunsch method
    for (size_t i = colCount - 2; i < colCount; --i) {
        const char xChar = m_geneY.at(i);
        int reservedDiagnoal = m_column[rowCount - 1];
        m_column[rowCount - 1] += 2;
        for (size_t j = rowCount - 2; j < rowCount; --j) {
            const char yChar = m_geneX.at(j);
            const auto fromRight = m_column[j] + 2;
            const auto fromDown = m_column[j + 1] + 2;
            const auto fromDiagonal = reservedDiagnoal + penalty(xChar, yChar);
            reservedDiagnoal = m_column[j];
            m_column[j] = min3(fromRight, fromDown, fromDiagonal);
        }

        // Record the last row element
        m_row[i] = m_column[0];
    }

    return m_column[0];
}

std::string OptimizedEDistance::alignment() const { throw std::exception(); }

std::vector<int> OptimizedEDistance::row() const { return m_row; }
```

An important point to highlight is the existence of a ‘**row**‘ method, which returns the row array containing the elements of the first row during the population process. This proves invaluable in the implementation of Hirschberg’s algorithm.

To address the issue of losing the arrow path post-population, we introduce Hirschberg’s algorithm. This method harnesses the divide-and-conquer technique along with recursive methods to compute alignments efficiently.

```cpp
// <HirshbergEDistance.hpp>
#ifndef HIRSHBERGEDISTANCE_H
#define HIRSHBERGEDISTANCE_H

#include <string>
#include <utility>
#include <vector>
#include "AbstractEDistance.hpp"

/**
 * @brief An implementation of AbstractEDistance using Hirschberg’s Algorithm.
 */
```

```cpp
class HirshbergEDistance : public AbstractEDistance {
 public:
    /**
     * Constructs a HirshbergEDistance instance with two input genes.
     * @param geneX The first gene string.
     * @param geneY The second gene string.
     */
    HirshbergEDistance(const std::string& geneX, const std::string& geneY);

    /**
     * @brief Calculates the optimal edit distance between the two gene strings.
     * Populates the internal matrix and returns the optimal distance.
     * @return The optimal edit distance between the two gene strings.
     */
    int optDistance() override;

    /**
     * @brief Traces the populated matrix and returns a string representing the optimal alignment
     * of the two gene strings.
     * @return A string representing the optimal alignment.
     */
    [[nodiscard]] std::string alignment() const override;

 private:
    /**
     * @brief Returns an array containing prefix costs.
     * @param geneX The first gene string.
     * @param geneY The second gene string.
     * @return An array containing prefix costs.
     */
    static std::vector<int> allYPrefixCosts(const std::string& geneX, const std::string& geneY);

    /**
     * @brief Returns an array containing suffix costs.
     * @param geneX The first gene string.
     * @param geneY The second gene string.
     * @return An array containing suffix costs.
     */
    static std::vector<int> allYSuffixCosts(const std::string& geneX, const std::string& geneY);

    /**
     * @brief Recursively aligns the gene strings using Hirschberg's algorithm.
     * @param geneX The first gene string.
     * @param geneY The second gene string.
     * @param offset The offset of the alignment.
     */
    void align(
        const std::string& geneX,
        const std::string& geneY,
        const std::pair<size_t, size_t>& offset);

    /**
     * @brief Inserts a coordinate into the arrow path.
     * @param coordinate The coordinate to insert.
     */
    void insertArrowPathCoordinate(const std::pair<size_t, size_t>& coordinate);

    /**
     * @brief Represents the arrow path.
     */
    std::vector<std::pair<size_t, size_t>> arrowPath;
};

#endif
```

We will focus on the implementation of this class. The **allYPrefixCosts** method gathers all prefix costs (edit distances) for gene Y (or the second string). It achieves this by reversing the provided X and Y strings and applying the optimized Needleman-Wunsch algorithm to obtain the reversed first row.

```cpp
// <HirshbergEDistance.cpp>
#include "HirshbergEDistance.hpp"
#include <algorithm>
#include <iostream>
#include <sstream>
#include <string>
#include <utility>
#include "NeedlemanWunschEDistance.hpp"
#include "OptimizedEDistance.hpp"

HirshbergEDistance::HirshbergEDistance(const std::string& geneX, const std::string& geneY) :
    AbstractEDistance(geneX, geneY) {}

std::vector<int>
HirshbergEDistance::allYPrefixCosts(const std::string& geneX, const std::string& geneY) {
    auto reversedGeneX = geneX;
    auto reversedGeneY = geneY;
    std::reverse(reversedGeneX.begin(), reversedGeneX.end());
    std::reverse(reversedGeneY.begin(), reversedGeneY.end());
    OptimizedEDistance optimizedEDistance{ reversedGeneX, reversedGeneY };
    optimizedEDistance.optDistance();
    auto row = optimizedEDistance.row();

    // Also reverse the row
    std::reverse(row.begin(), row.end());

    return row;
}
```

Likewise, we can get all suffix costs by applying the optimized Needleman-Wunsch algorithm on the two given strings and obtain the first row.

```cpp
std::vector<int>
HirshbergEDistance::allYSuffixCosts(const std::string& geneX, const std::string& geneY) {
    OptimizedEDistance optimizedEDistance{ geneX, geneY };
    optimizedEDistance.optDistance();

    return optimizedEDistance.row();
}
```

The **align** method is the core of the Hirschberg's algorithm. Let's break down its key components:

- **Initialization**: Get the length of the gene X and gene Y(**xLength** and **yLength**), and the midpoint of X (**xHalfLength**).
- **Base Case**: If either **xLength** or **yLength** is less than 2, then we apply the standard Needleman-Wunsch alignment, and get the arrow path based on the offset.
- **Divide and Conquer**: When both **xLength** and **yLength** exceed 2, we divide X into two halves: **xHalf1** and **xHalf2**. Following this, we calculate the prefix costs by aligning **xHalf1** with Y, and the suffix costs by aligning **xHalf2** with Y.
- **Find the Best Cost**: The optimal cost is determined by aggregating each prefix cost with its corresponding suffix cost. Then, we insert this position into the arrow path.
- **Recurssion**: We recursively invoke the align function with **xHalf1**, **xHalf2**, the substring of geneY, and the updated offset values, respectively. This allows us to further refine the alignment process and collect all coordinates on the arrow path.

It is worth noting that the arrow path in HirshbergEDistance is represented by the data structure

```
std::vector<std::pair<size_t, size_t>>
```

In this structure, the index corresponds to the summation of the pair, where each pair represents a coordinate on the dynamic programming matrix. Notably, the summations of coordinates along the path are distinct.

Pairs denoted as $(0, 0)$ within the arrow path are referred to as ”skipped coordinates”. They arise when some arrows transition diagonally. Given an index t, if arrowPath$[t] = (u, v)$ is not a skipped coordinate, then the coordinate (u, v) lies on the arrow path. Additionally, if arrowPath$[t + 1] = (u, v)$ is also not a skipped coordinate, it implies that (u, v) derives from either the cell on the right or the one at the bottom. Conversely, if arrowPath$[t + 1] = (u, v)$ is a skipped coordinate, then (u, v) originates from the bottom-right diagonal cell.

```
void HirshbergEDistance::align(
    const std::string& geneX, const std::string& geneY, const std::pair<size_t, size_t>& offset) {
    const auto xLength = geneX.length();
    const auto yLength = geneY.length();
    const auto xHalfLength = xLength / 2;

    if (xLength <= 2 || yLength <= 2) {
        // Use standard alignment
        NeedlemanWunschEDistance eDistance{ geneX, geneY };
        eDistance.optDistance();
        const auto* const matrix = eDistance.matrix();

        size_t i = 0;
        size_t j = 0;
        while (i + j < xLength + yLength) {
            insertArrowPathCoordinate({ i + offset.first, j + offset.second });
            const auto val = matrix[i][j];
            const auto isFromBottom = i < xLength && val - 2 == matrix[i + 1][j];
            const auto isFromRight = j < yLength && val - 2 == matrix[i][j + 1];
            i += static_cast<int>(!isFromRight);
            j += static_cast<int>(!isFromBottom);
        }

        return;
    }

    const auto xHalf1 = geneX.substr(0, xHalfLength);
    const auto xHalf2 = geneX.substr(xHalfLength);
    const auto yPrefix = allYPrefixCosts(xHalf1, geneY);
    const auto ySuffix = allYSuffixCosts(xHalf2, geneY);

    int bestCost = static_cast<int>(xLength + yLength) << 1;
    size_t bestQ = 0;
    for (size_t q = 0; q < yLength; ++q) {
        const auto costSum = yPrefix[q] + ySuffix[q];
        if (costSum < bestCost) {
            bestCost = costSum;
            bestQ = q;
        }
    }

    // Add the coordinate to the arrow path
    const std::pair coordinate = { xHalfLength + offset.first, bestQ + offset.second };
    insertArrowPathCoordinate(coordinate);

    // Recursively find the other coordinates of the arrow path
    align(xHalf1, geneY.substr(0, bestQ), offset);
    align(xHalf2, geneY.substr(bestQ), coordinate);
}
```

Since we compute the yPrefix and ySuffix in separate functions, the space complexity of this function is $O(n)$. The time complexity, however, is $O(n^2)$.

With the **align** method, we can implement the **optDistance** in a straightforward way. Once the two genes are aligned, we compute the total cost by traversing the complete arrow path, adhering to its previously defined definition.

```
int HirshbergEDistance::optDistance() {
    static constexpr std::pair<size_t, size_t> ZERO_PAIR(0, 0);

    // Initialize arrowPath
    const auto xLength = m_geneX.length();
    const auto yLength = m_geneY.length();
    for (size_t i = 0; i < xLength + yLength; ++i) {
        arrowPath.emplace_back(0, 0);
    }
    arrowPath.emplace_back(xLength, yLength);

    // Align the two genes
    align(m_geneX, m_geneY, { 0, 0 });

    // Find the total cost according to the arrow path
    const auto length = m_geneX.length() + m_geneY.length();
    int totalCost = 0;
    for (size_t i = 0; i < length; ++i) {
        if (arrowPath[i] == ZERO_PAIR && i != 0) {
            continue;
        }

        if (arrowPath[i + 1] != ZERO_PAIR) {
            // Either from right or from bottom
            totalCost += 2;
            // From diagonal
        } else {
            const auto xChar = m_geneX.at(arrowPath[i].first);
            const auto yChar = m_geneY.at(arrowPath[i].second);
            totalCost += penalty(xChar, yChar);
        }
    }

    return totalCost;
}
```

The **alignment** is also easy to implement based on the arrow path's definition.

```
std::string HirshbergEDistance::alignment() const {
    static constexpr auto CHAR_GAP = '-';
    static constexpr auto CHAR_SPACE = ' ';
    static constexpr std::pair<size_t, size_t> ZERO_PAIR(0, 0);

    const auto xLength = m_geneX.length();
    const auto yLength = m_geneY.length();
    std::ostringstream ostringstream;
    for (size_t i = 0; i < m_geneX.length() + m_geneY.length(); ++i) {
        if (arrowPath[i] == ZERO_PAIR && i != 0) {
            continue;
        }

        const auto [xIndex, yIndex] = arrowPath[i];
        const auto isFromRightOrBottom = arrowPath[i + 1] != ZERO_PAIR;
        const auto isFromRight =
            isFromRightOrBottom && arrowPath[i].first == arrowPath[i + 1].first;
        const auto isFromBottom = isFromRightOrBottom && !isFromRight;
        const auto xChar = xIndex < xLength && !isFromRight ? m_geneX.at(xIndex) : CHAR_GAP;
        const auto yChar = yIndex < yLength && !isFromBottom ? m_geneY.at(yIndex) : CHAR_GAP;
        const auto cost = isFromRightOrBottom ? 2 : penalty(xChar, yChar);

        // Output: "<xChar> <yChar> <cost>\n"
        ostringstream << xChar << CHAR_SPACE << yChar << CHAR_SPACE << cost << std::endl;
    }

    return ostringstream.str();
}

void HirshbergEDistance::insertArrowPathCoordinate(const std::pair<size_t, size_t>& coordinate) {
    arrowPath[coordinate.first + coordinate.second] = coordinate;
}
```

A bunch of tests are established to ensure program's correctness.

```
// <test.cpp>
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main

#include <sstream>
#include <boost/test/unit_test.hpp>
#include "EDistance.hpp"

```

```
// Checks if `EDistance::penalty` works correctly.
BOOST_AUTO_TEST_CASE(testPenalty) {
    BOOST_REQUIRE_EQUAL(EDistance::penalty('a', 'a'), 0);
    BOOST_REQUIRE_EQUAL(EDistance::penalty('a', 'B'), 1);
}

// Checks if `EDistance::min3` works correctly.
BOOST_AUTO_TEST_CASE(testMin3) {
    BOOST_REQUIRE_EQUAL(EDistance::min3(5, 4, 3), 3);
    BOOST_REQUIRE_EQUAL(EDistance::min3(9, 12, 9), 9);
    BOOST_REQUIRE_EQUAL(EDistance::min3(-3, 6, 0), -3);
    BOOST_REQUIRE_EQUAL(EDistance::min3(7, 7, 7), 7);
}

// Checks if `EDistance::optDistance` works correctly; using the example in the instructions.
BOOST_AUTO_TEST_CASE(testOptDistance1) {
    EDistance eDistance{ "AACAGTTACC", "TAAGGTCA" };
    BOOST_REQUIRE_EQUAL(eDistance.optDistance(), 7);
}

// Checks if `EDistance::optDistance` works correctly; using 'endgaps7.txt'.
BOOST_AUTO_TEST_CASE(testOptDistance2) {
    EDistance eDistance{ "atattat", "tattata" };
    BOOST_REQUIRE_EQUAL(eDistance.optDistance(), 4);
}

// Checks if the return value of `EDistance::optDistance` the same as the summation of costs from
// `EDistance::alignment`; using the example in the instructions.
BOOST_AUTO_TEST_CASE(testAlignmentCostSum1) {
    EDistance eDistance{ "AACAGTTACC", "TAAGGTCA" };
    const auto editDistance = eDistance.optDistance();

    // Get the summation of costs
    int costSum = 0;
    std::istringstream istringstream(eDistance.alignment());
    std::string line;
    while (std::getline(istringstream, line)) {
        costSum += line.at(4) - '0';
    }

    BOOST_REQUIRE_EQUAL(editDistance, costSum);
}

// Checks if the return value of `EDistance::optDistance` the same as the summation of costs from
// `EDistance::alignment`; using 'endgaps7.txt'.
BOOST_AUTO_TEST_CASE(testAlignmentCostSum2) {
    EDistance eDistance{ "atattat", "tattata" };
    const auto editDistance = eDistance.optDistance();

    // Get the summation of costs
    int costSum = 0;
    std::istringstream istringstream(eDistance.alignment());
    std::string line;
    while (std::getline(istringstream, line)) {
        costSum += line.at(4) - '0';
    }
```

```cpp

    BOOST_REQUIRE_EQUAL(editDistance, costSum);
}

// Checks if the two columns are correct. The first column should be the first string, while the
// second column should be the second string.
BOOST_AUTO_TEST_CASE(testAlignmentColumns) {
    static constexpr auto CHAR_GAP = '-';

    const std::string geneX = "AACAGTTACC";
    const std::string geneY = "TAAGGTCA";
    EDistance eDistance{ geneX, geneY };
    eDistance.optDistance();

    std::ostringstream geneXActual;
    std::ostringstream geneYActual;
    std::istringstream istringstream(eDistance.alignment());
    std::string line;
    while (std::getline(istringstream, line)) {
        const auto charX = line.at(0);
        const auto charY = line.at(2);
        if (charX != CHAR_GAP) {
            geneXActual << charX;
        }
        if (charY != CHAR_GAP) {
            geneYActual << charY;
        }
    }

    BOOST_REQUIRE_EQUAL(geneX, geneXActual.str());
    BOOST_REQUIRE_EQUAL(geneY, geneYActual.str());
}
```

7 PS6 - Random Writer

7.1 Discussion

In PS6, I created a program that serves as a random text generator utilizing the Markov model. It operates by accepting two parameters: the order *k* and the desired length of the generated text. Additionally, it necessitates training data input. The program then proceeds to construct a Markov model based on this training data, subsequently producing text of the specified length.

I've significantly revamped this project through an aggressive refactor. The key enhancement lies in the creation of the **SymbolTable**, a template class designed to accommodate two distinct generics. This essential component functions as a repository for k-grams (also refers to as ”symbol”) along with their associated subsequent characters (also refers to as ”token”) and frequencies.

Through this project, I immersed myself in the concept of Markov chains, delving into the creation of a symbol table of k-grams and mastering its application. Furthermore, I gained invaluable insights into template classes. This project marked my first attempt of crafting a template class independently, opening up a new realm of possibilities in my programming journey.

7.2 Achievements

The following content is used to generate a symbol table for the markov model, and it is saved in the file **romeo.txt**.

```
Two households, both alike in dignity
(In fair Verona, where we lay our scene),
From ancient grudge break to new mutiny,
Where civil blood makes civil hands unclean.
From forth the fatal loins of these two foes
A pair of star-crossed lovers take their life;
Whose misadventured piteous overthrows
Doth with their death bury their parents' strife.
The fearful passage of their death-marked love
And the continuance of their parents' rage,
Which, but their children's end, naught could remove,
Is now the two hours' traffic of our stage;
The which, if you with patient ears attend,
What here shall miss, our toil shall strive to mend.
```

After running the following command:

```
./TextWriter 3 200 < romeo.txt
```

the console outputs a short text:

```
Two forth with with pair parents' stage of our scene),From ance of their patientured love,Is now their
↪  patientured loins overthrowsDoth with the which, both the the fatal loveAnd the we lay our strife
```

I've also developed a **WordWriter** class that generates a symbol table based on words rather than individual characters. Likewise, we run the program with *romeo.txt* and the command below:

```
./WordWriter 2 50 < romeo.txt
```

We will get the following from the console.

```
Two households, both alike in dignity (In fair Verona, where we lay our scene), From ancient grudge break to new mutiny, Where civil blood makes civil hands unclean. From forth the fatal loins of these two foes A pair of star-crossed lovers take their life; Whose misadventured piteous overthrows Doth
```

7.3 Codebase

```
# C++ Compiler
COMPILER = g++

# C++ Flags (C++ version: 20)
CFLAGS = --std=c++20 -Wall -Werror -pedantic -g

# Libraries
LIB = -lboost_unit_test_framework

# Code source directory
SRC = ./

# Hpp files (dependencies)
DEPS = $(SRC)RandWriter.hpp \
           $(SRC)SymbolTable.hpp

# Static library
STATIC_LIB = TextWriter.a

# The object files that the static library includes
STATIC_LIB_OBJECTS = $(SRC)RandWriter.o

# Program
PROGRAM = TextWriter

# Program object files
MAIN_OBJECTS = $(SRC)TextWriter.o

# Test program
TEST_PROGRAM = test

# Test object files
TEST_OBJECTS = $(SRC)test.o

# WordWriter program
WORD_WRITER_PROGRAM = WordWriter

# WordWriter object files
WORD_WRITER_PROGRAM_OBJECTS = $(SRC)WordWriter.o $(SRC)AdvancedTextWriter.o

all: $(PROGRAM) $(TEST_PROGRAM)

$(SRC)%.o: $(SRC)%.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

$(PROGRAM): $(MAIN_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

```

```
$(STATIC_LIB): $(STATIC_LIB_OBJECTS)
        ar rcs $@ $^

$(TEST_PROGRAM): $(TEST_OBJECTS) $(STATIC_LIB)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

$(WORD_WRITER_PROGRAM): $(WORD_WRITER_PROGRAM_OBJECTS)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

clean:
        rm -f $(SRC)*.o $(PROGRAM) $(STATIC_LIB) $(TEST_PROGRAM) $(WORD_WRITER_PROGRAM)

lint:
        cpplint *.hpp *.cpp

boost: $(TEST_PROGRAM)
        ./$(TEST_PROGRAM)

run-input17: $(PROGRAM)
        ./$(PROGRAM) 2 11 < input17.txt

run-romeo: $(PROGRAM)
        ./$(PROGRAM) 3 200 < romeo.txt

run-tomsawyer: $(PROGRAM)
        ./$(PROGRAM) 8 400 < tomsawyer.txt

run-word-romeo: $(WORD_WRITER_PROGRAM)
        ./$(WORD_WRITER_PROGRAM) 2 50  < romeo.txt

run-word-tomsawyer: $(WORD_WRITER_PROGRAM)
        ./$(WORD_WRITER_PROGRAM) 2 150  < tomsawyer.txt
```

I developed a main function that accepts two arguments: (1) the order k of the Markov model, and (2) the desired length L of the generated text. Utilizing the thoroughly tested **RandWriter**, the program generates a text of length L.

```cpp
// <TextWriter.cpp>
#include <iostream>
#include <sstream>
#include <string>
#include "RandWriter.hpp"
#include "SymbolTable.hpp"

/**
 * @brief Starts the universe simulation.
 * @param argc The number of arguments.
 * @param argv The arguments vector. This program requires two arguments:
 * 1. k (int): The order of the Markov model.
 * 2. L (int): The length of the generated text.
 */
int main(const int argc, const char* argv[]) {
    if (argc != 3) {
        std::cerr << "Too many or too few arguments!" << std::endl;
        return 1;
    }

    // Arguments
    const auto orderK = std::stoi(argv[1]);
    const auto generatedTextLength = std::stoi(argv[2]);

    // Read the input text from standard input
    std::string line{ "0" };
    std::ostringstream textStream;
    while (!line.empty() && !std::cin.eof()) {
        getline(std::cin, line);
        textStream << line;
    }
    const std::string text = textStream.str();

    // Create a RandWriter instance and generate a text
    RandWriter randWriter{ text, static_cast<size_t>(orderK) };
    const auto generatedText = randWriter.generate(text.substr(0, orderK), generatedTextLength);
    std::cout << generatedText;
    // std::cout << std::endl << randWriter;

    return 0;
}
```

First of all, I created a **RandWriter** class:

```cpp
// <RandWriter.hpp>
#ifndef RANDWRITTEN_H
#define RANDWRITTEN_H

#include <array>
#include <string>
#include <unordered_map>
#include "SymbolTable.hpp"

/**
 * @brief A class for generating text using a Markov model. This class provides functionalities to
 * create and utilize a Markov model of order k from a given text, allowing the generation of new
 * text based on the learned patterns.
 */
class RandWriter {
 public:
    /**
     * @brief Constructs a RandWriter object with a Markov model of order k. Creates a Markov model
     * of order k from the provided text. The order k represents the number of preceding characters
     * considered for predicting the next character.
     * @param text The text used to create the Markov model.
     * @param k The order of the Markov model.
     */
    RandWriter(const std::string& text, size_t k);

    /**
     * @brief Returns the order of the Markov model.
     * @return The order of the Markov model.
     */
    [[nodiscard]] size_t orderK() const;

    /**
     * @brief Returns the number of occurrences of a specific k-gram in the text.
     * @param kgram The k-gram to search for.
     * @return The frequency of the k-gram in the text.
     * @throw std::invalid_argument If the length of kgram is not equal to the order of the Markov
     * model.
     */
    int freq(const std::string& kgram) const;

    /**
     * @brief Returns the number of times a character follows a specific k-gram
     * in the text.
     * @param kgram The k-gram preceding the character.
     * @param c The character to check for.
     * @return The frequency of character c following the k-gram.
     * @throw std::invalid_argument If the length of kgram is not equal to the order of the Markov
     * model.
     */
    int freq(const std::string& kgram, char c) const;

    /**
     * @brief Generates a string of specified length using the Markov chain. Generates a string of
     * length L characters by simulating a trajectory through the Markov chain learned from the
     * input text. The initial k characters of the generated string are provided as the argument
     * kgram.
     * @param kgram The initial k characters of the generated string.
     * @param L The length of the generated string.
     * @return The generated string.
     * @throw std::invalid_argument If the length of kgram is not equal to the order of the Markov
     * model.
     */
    std::string generate(const std::string& kgram, size_t L);

    /**
     * @brief Returns a random character followed by a specified k-gram .
     * @param kgram The k-gram.
     * @return a random character followed by a specified k-gram.
     */
    // ReSharper disable once CppFunctionIsNotImplemented
    char kRand(const std::string& kgram);

    /**
     * @brief Overloading of "<<" for this class. Outputs the symbol table, including kgrams, k+1
     * grams and their frequencies.
     */
    friend std::ostream& operator<<(std::ostream& os, const RandWriter& randWriter);

private:
    /**
     * @brief Checks if a k-gram is of length k, which is the order of the Markov model.
     * @param kgram The k-gram to check.
     */
    void checkKgram(const std::string& kgram) const;

    /**
     * Returns a random number between 1 (included) and the given total frequency (excluded).
     * @param totalFreq The total frequency of all characters.
     * @return A random number between 1 (included) and the given total frequency (excluded).
     */
    [[nodiscard]] static int getRandomNumber(int totalFreq);

    /**
     * @brief The order of the Markov model.
     */
    size_t order_k_;

    /**
     * @brief The symbol table used to store kgrams and the frequencies of next characters.
     */
    SymbolTable<std::string, char> symbol_table_;
};

#endif

<RandWriter.cpp>
#include "RandWriter.hpp"
#include <iostream>
#include <random>

RandWriter::RandWriter(const std::string& text, const size_t k) : order_k_(k) {
    // Check if order k is greater than the length of the given text
    if (k > text.size()) {
        throw std::invalid_argument("The order k should be less then the size of the text!");
    }

    // Use slide window technique to scan all k-grams in the given text
    const auto circularText = text + text.substr(0, k);
    for (size_t i = 0; i < circularText.length() - k; ++i) {
        const auto kgram = circularText.substr(i, k);
        const auto nextChar = circularText.at(i + k);
        symbol_table_.increment(kgram, nextChar);
    }
}

size_t RandWriter::orderK() const { return order_k_; }

int RandWriter::freq(const std::string& kgram) const {
    checkKgram(kgram);
    return symbol_table_.frequencyOf(kgram);
}

int RandWriter::freq(const std::string& kgram, const char c) const {
    checkKgram(kgram);
    return symbol_table_.frequencyOf(kgram, c);
}

std::string RandWriter::generate(const std::string& kgram, const size_t L) {
    checkKgram(kgram);

    std::string generatedText = kgram;
```

```cpp
    for (size_t i = 0; i < L - order_k_; ++i) {
        const auto& lastKgram = generatedText.substr(i, order_k_);
        const auto nextChar = kRand(lastKgram);
        generatedText.push_back(nextChar);
    }

    return generatedText;
}

// ReSharper disable once CppMemberFunctionMayBeConst
char RandWriter::kRand(const std::string& kgram) {
    const auto totalFrequency = symbol_table_.frequencyOf(kgram);
    const auto randomIndex = getRandomNumber(totalFrequency);
    const auto frequencyTable = symbol_table_.frequencyMapOf(kgram);

    int cumulative_frequency = 0;
    for (const auto& [c, frequency] : frequencyTable) {
        cumulative_frequency += frequency;
        if (cumulative_frequency >= randomIndex) {
            return c;
        }
    }

    return '\0';
}

std::ostream& operator<<(std::ostream& os, const RandWriter& randWriter) {
    static const std::string INDENT = "--- ";

    randWriter.symbol_table_.traverse(
        [&](auto kgram, auto totalFreq) { os << kgram << ": " << totalFreq << std::endl; },
        [&](auto c, auto freq) { os << INDENT << c << ": " << freq << std::endl; });

    return os;
}

void RandWriter::checkKgram(const std::string& kgram) const {
    if (kgram.length() != order_k_) {
        throw std::invalid_argument("Invliad k-gram: " + kgram);
    }
}

int RandWriter::getRandomNumber(const int totalFreq) {
    static std::random_device randomDevice;
    static std::mt19937 gen(randomDevice());

    if (totalFreq <= 0) {
        return 0;
    }

    std::uniform_int_distribution distribution(1, totalFreq);
    return distribution(gen);
}
```

```cpp
// <SymbolTable.hpp>
#ifndef SYMBOLTABLE_HPP
#define SYMBOLTABLE_HPP

#include <functional>
#include <unordered_map>

using std::unordered_map;

/**
 * @brief SymbolTable class template for managing symbol frequencies.
 * @tparam S The type of symbols.
 * @tparam T The type of tokens.
 */
template <typename S, typename T>
class SymbolTable {
 public:
    /**
     * @brief Increments the frequency count for a symbol-next_token pair.
     * @param symbol The symbol to increment the frequency for.
     * @param next_token The token associated with the symbol.
     */
    void increment(S symbol, T next_token);

    /**
     * @brief Retrieves the total frequency count of a given symbol.
     * @param symbol The symbol to retrieve the frequency count for.
     * @return The total frequency count of the given symbol.
     */
    [[nodiscard]] int frequencyOf(S symbol) const;

    /**
     * @brief Retrieves the frequency count of a specific symbol-next_token pair.
     * @param symbol The symbol to retrieve the frequency count for.
     * @param next_token The symbol to retrieve the frequency count for.
     * @return The frequency count of the given symbol-next_token pair.
     */
    [[nodiscard]] int frequencyOf(S symbol, T next_token) const;

    /**
     * @brief Gets the frequency map of symbols.
     * @return The frequency map of symbols.
     */
    unordered_map<S, int> frequencyMap() const;

    /**
     * @brief Gets the frequency map of tokens associated with a specific symbol.
     * @param symbol The symbol to retrieve the frequency map for.
     * @return The frequency map of tokens associated with the given symbol.
     */
    unordered_map<T, int> frequencyMapOf(S symbol) const;

    /**
     * Traverses through the symbol table, invoking provided callbacks for symbol and token
     * information.
     * @param symbolCallback A callback function accepting a symbol and its total frequency;
     * @param tokenCallback A callback function accepting a token and its frequency;
     */
    void traverse(
        std::function<void(S, int)> symbolCallback,
        std::function<void(T, int)> tokenCallback) const;

 private:
    /**
     * @brief Map to store symbols and their corresponding frequency maps.
     */
    unordered_map<S, unordered_map<T, int>> frequency_table_;

    /**
     * @brief Map to store the total frequency of each token.
     */
    unordered_map<S, int> frequency_map_;
};

template <typename S, typename T>
void SymbolTable<S, T>::increment(S symbol, T next_token) {
    // Increment frequency for the symbol-next_token pair
    ++frequency_table_[symbol][next_token];

    // Increment total frequency for the symbol
    ++frequency_map_[symbol];
}

template <typename S, typename T>
int SymbolTable<S, T>::frequencyOf(S symbol) const {
    const auto entry = frequency_map_.find(symbol);
    return entry == frequency_map_.end() ? 0 : entry->second;
}

template <typename S, typename T>
int SymbolTable<S, T>::frequencyOf(S symbol, T next_token) const {
    // Return 0 if the given symbol does not exist
    const auto entry = frequency_table_.find(symbol);
    if (entry == frequency_table_.end()) {
        return 0;
    }

    const auto frequencyMap = entry->second;
    const auto frequencyEntry = frequencyMap.find(next_token);
    return frequencyEntry == frequencyMap.end() ? 0 : frequencyEntry->second;
}

template <typename S, typename T>
unordered_map<S, int, std::hash<S>, std::equal_to<S>> SymbolTable<S, T>::frequencyMap() const {
    return frequency_map_;
}

template <typename S, typename T>
unordered_map<T, int, std::hash<T>, std::equal_to<T>>
SymbolTable<S, T>::frequencyMapOf(S symbol) const {
    return frequency_table_.at(symbol);
}
```

```cpp

template <typename S, typename T>
void SymbolTable<S, T>::traverse(
    std::function<void(S, int)> symbolCallback, std::function<void(T, int)> tokenCallback) const {
    for (auto const& [symbol, totalFrequency] : frequency_map_) {
        symbolCallback(symbol, totalFrequency);
        for (auto const& [token, frequency] : frequencyMapOf(symbol)) {
            tokenCallback(token, frequency);
        }
    }
}

#endif
```

By implementing the class as a template, I've ensured its versatility, enabling it to be utilized seamlessly by both **RandWriter** and **WordWriter**.

```cpp
// In RandWriter
SymbolTable<std::string, char> symbol_table_;

// In WordWriter
SymbolTable<std::string, std::string> symbol_table_;
```

```cpp
// <WordWriter.hpp>
#ifndef WORDWRITER_HPP
#define WORDWRITER_HPP

#include <string>
#include <vector>
#include "SymbolTable.hpp"

class WordWriter {
 public:
    /**
     * @brief Constructs a RandWriter object with a Markov model of order k. Creates a Markov model
     * of order k from the provided text. The order k represents the number of preceding characters
     * considered for predicting the next character.
     * @param text The text used to create the Markov model.
     * @param k The order of the Markov model.
     */
    WordWriter(const std::string& text, size_t k);

    /**
     * @brief Generates a string of specified length using the Markov chain. Generates a string of
     * length L characters by simulating a trajectory through the Markov chain learned from the
     * input text. The initial k characters of the generated string are provided as the argument
     * kgram.
     * @param kgram The initial k characters of the generated string.
     * @param L The length of the generated string.
     * @return The generated string.
     * @throw std::invalid_argument If the length of kgram is not equal to the order of the Markov
     * model.
     */
    std::string generate(const std::vector<std::string>& kgram, size_t L) const;

    /**
     * @brief Returns a random character followed by a specified k-gram .
     * @param kgram The k-gram.
     * @return a random character followed by a specified k-gram.
     */
    std::string kRand(const std::vector<std::string>& kgram) const;

    /**
     * @brief Overloading of "<<" for this class. Outputs the symbol table, including kgrams, k+1
     * grams and their frequencies.
     */
    friend std::ostream& operator<<(std::ostream& os, const WordWriter& wordWriter);

 private:
    /**
     * Returns a random number between 1 (included) and the given total frequency (excluded).
     * @param totalFreq The total frequency of all characters.
     * @return A random number between 1 (included) and the given total frequency (excluded).
     */
    [[nodiscard]] static int getRandomNumber(int totalFreq);

    /**
     * @brief The order of the Markov model.
     */
    size_t order_k_;

    SymbolTable<std::string, std::string> symbol_table_;
};

#endif

// <WordWriter.cpp>
#include "WordWriter.hpp"
#include <iostream>
#include <random>
#include <sstream>

std::string to_string(const std::vector<std::string>& stringVector) {
    std::ostringstream stringStream;
    const auto maxIndex = stringVector.size() - 1;
    for (int i = 0; i <= maxIndex; ++i) {
        stringStream << stringVector.at(i);
        if (i != maxIndex) {
            stringStream << ' ';
        }
    }

    return stringStream.str();
}

WordWriter::WordWriter(const std::string& text, const size_t k) : order_k_(k) {
    // Check if order k is greater than the length of the given text
    if (k > text.size()) {
        throw std::invalid_argument("The order k should be less then the size of the text!");
    }

    // Split the text into tokens
    std::istringstream ss(text);
    std::string token;
    std::vector<std::string> token_vector;
    while (std::getline(ss, token, ' ')) {
        token_vector.push_back(token);
    }

    for (int i = 0; i < k; ++i) {
        token_vector.push_back(token_vector.at(i));
    }

    // Use slide window technique to scan all k-grams in the given text
    for (size_t i = 0; i < token_vector.size() - k; ++i) {
        std::vector<std::string> kgram;
        for (size_t j = 0; j < k; ++j) {
            kgram.push_back(token_vector.at(i + j));
        }
        const auto nextToken = token_vector.at(i + k);
        symbol_table_.increment(to_string(kgram), nextToken);
    }
}

std::string WordWriter::generate(const std::vector<std::string>& kgram, const size_t L) const {
    std::vector<std::string> generatedTextVector = kgram;
    for (size_t i = 0; i < L - order_k_; ++i) {
        std::vector<std::string> lastKGram;
        for (size_t j = 0; j < order_k_; ++j) {
            lastKGram.push_back(generatedTextVector.at(i + j));
        }
        const auto nextToken = kRand(lastKGram);
        generatedTextVector.push_back(nextToken);
    }

    std::ostringstream generatedTextStream;
    for (const auto& token : generatedTextVector) {
        generatedTextStream << token << ' ';
    }

    return generatedTextStream.str();
}

std::string WordWriter::kRand(const std::vector<std::string>& kgram) const {
    const auto kgramString = to_string(kgram);
    const auto totalFrequency = symbol_table_.frequencyOf(kgramString);
    const auto randomIndex = getRandomNumber(totalFrequency);
    const auto frequencyTable = symbol_table_.frequencyMapOf(kgramString);

    int cumulative_frequency = 0;
    for (const auto& [token, frequency] : frequencyTable) {
        cumulative_frequency += frequency;
        if (cumulative_frequency >= randomIndex) {
            return token;
        }
    }
```

```cpp

    return "";
}

int WordWriter::getRandomNumber(const int totalFreq) {
    static std::random_device randomDevice;
    static std::mt19937 gen(randomDevice());

    if (totalFreq <= 0) {
        return 0;
    }

    std::uniform_int_distribution distribution(1, totalFreq);
    return distribution(gen);
}

std::ostream& operator<<(std::ostream& os, const WordWriter& wordWriter) {
    static const std::string INDENT = "--- ";

    wordWriter.symbol_table_.traverse(
        [&](auto kgram, auto totalFreq) { os << kgram << ": " << totalFreq << std::endl; },
        [&](auto word, auto freq) { os << INDENT << word << ": " << freq << std::endl; });

    return os;
}
```

The **AdvancedTextWriter** file creates a program based on the **WordWriter**.

```cpp
// <AdvancedTextWriter.cpp>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include "WordWriter.hpp"

/**
 * @brief Starts the universe simulation.
 * @param argc The number of arguments.
 * @param argv The arguments vector. This program requires two arguments:
 * 1. k (int): The order of the Markov model.
 * 2. L (int): The length of the generated text.
 */
int main(const int argc, const char* argv[]) {
    if (argc != 3) {
        std::cerr << "Too many or too few arguments!" << std::endl;
        return 1;
    }

    // Arguments
    const auto orderK = std::stoi(argv[1]);
    const auto generatedTextLength = std::stoi(argv[2]);

    // Read the input text from standard input
    std::string line{ "0" };
    std::ostringstream textStream;
    while (!line.empty() && !std::cin.eof()) {
        getline(std::cin, line);
        textStream << line << " ";
    }
    const std::string text = textStream.str();

    // Get the first kgram
    std::vector<std::string> kgram;
    std::istringstream ss(text);
    std::string token;
    std::vector<std::string> token_vector;
    for (int i = 0; i < orderK; ++i) {
        ss >> token;
        kgram.push_back(token);
    }

    // Create a RandWriter instance and generate a text
    WordWriter word_writer{ text, static_cast<size_t>(orderK) };
    const auto generatedText = word_writer.generate(kgram, generatedTextLength);
    std::cout << generatedText;

    return 0;
}
```

I established some essential test cases to ensure the **RandWriter** class functions correctly.

```cpp
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Main

#include <fstream>
#include <sstream>
#include <string>
#include <boost/test/unit_test.hpp>
#include "RandWriter.hpp"

/**
 * Reads content from a file.
 * @param filename The path of the file to read.
 * @return All the content of the file.
 */
std::string readFileContent(const std::string& filename) {
    std::ifstream ifstream(filename);
    std::ostringstream contentStream;
    std::string line;
    while (std::getline(ifstream, line)) {
        contentStream << line;
    }

    ifstream.close();

    return contentStream.str();
}

// Checks if `RandWriter::freq(const std::string&)` works correctly.
BOOST_AUTO_TEST_CASE(testFreq1) {
    const RandWriter randWriter{ "gagggagaggcgagaaa", 2 };
    BOOST_REQUIRE_EQUAL(randWriter.freq("ga"), 5);
    BOOST_REQUIRE_EQUAL(randWriter.freq("ag"), 5);
    BOOST_REQUIRE_EQUAL(randWriter.freq("gg"), 3);
    BOOST_REQUIRE_EQUAL(randWriter.freq("gc"), 1);
    BOOST_REQUIRE_EQUAL(randWriter.freq("cg"), 1);
    BOOST_REQUIRE_EQUAL(randWriter.freq("aa"), 2);
}

// Checks if `RandWriter::freq(const std::string&, char)` works correctly.
BOOST_AUTO_TEST_CASE(testFreq2) {
    const RandWriter randWriter{ "gagggagaggcgagaaa", 2 };
    BOOST_REQUIRE_EQUAL(randWriter.freq("ga", 'g'), 4);
    BOOST_REQUIRE_EQUAL(randWriter.freq("ga", 'a'), 1);
    BOOST_REQUIRE_EQUAL(randWriter.freq("ga", 'c'), 0);
    BOOST_REQUIRE_EQUAL(randWriter.freq("ag", 'g'), 2);
    BOOST_REQUIRE_EQUAL(randWriter.freq("ag", 'a'), 3);
    BOOST_REQUIRE_EQUAL(randWriter.freq("aa", 'a'), 1);
    BOOST_REQUIRE_EQUAL(randWriter.freq("aa", 'g'), 1);
}

// Checks if `RandWriter::freq(const std::string&)` works correctly. It should return 0 if the
// provided k-gram is not presented in the text.
BOOST_AUTO_TEST_CASE(testFreq3) {
    const RandWriter randWriter{ "gagggagaggcgagaaa", 2 };
    BOOST_REQUIRE_EQUAL(randWriter.freq("cc"), 0);
    BOOST_REQUIRE_EQUAL(randWriter.freq("ac"), 0);
}

// Checks if `RandWriter::freq(const std::string&, char)` works correctly. It should return 0 if the
// provided k-gram is not presented in the text or the frequency of a character is 0.
BOOST_AUTO_TEST_CASE(testFreq4) {
    const RandWriter randWriter{ "gagggagaggcgagaaa", 2 };
    BOOST_REQUIRE_EQUAL(randWriter.freq("ga", 'c'), 0);
    BOOST_REQUIRE_EQUAL(randWriter.freq("cg", 'g'), 0);
    BOOST_REQUIRE_EQUAL(randWriter.freq("cc", 'c'), 0);
    BOOST_REQUIRE_EQUAL(randWriter.freq("ac", 'g'), 0);
}

// Checks if `RandWriter::freq(const std::string&)` throws an exception when the provided k-gram
// does not appear in the original text.
BOOST_AUTO_TEST_CASE(testWrongKgram1) {
    const RandWriter randWriter{ "gagggagaggcgagaaa", 2 };
    BOOST_REQUIRE_THROW(std::cout << randWriter.freq("gag"), std::invalid_argument);
}

// Checks if `RandWriter::freq(const std::string&, char)` throws an exception when the provided
// k-gram does not appear in the original text.
BOOST_AUTO_TEST_CASE(testWrongKgram2) {
    const RandWriter randWriter{ "gagggagaggcgagaaa", 2 };
    BOOST_REQUIRE_THROW(std::cout << randWriter.freq("gag", 'g'), std::invalid_argument);
}

BOOST_AUTO_TEST_CASE(testOrderKFail) {
    BOOST_REQUIRE_THROW(const RandWriter randWriter("abc", 4), std::invalid_argument);
}
```

```cpp

BOOST_AUTO_TEST_CASE(testKRand1) {
    RandWriter randWriter{ "gagggagaggcgagaaa", 2 };
    for (int i = 0; i < 2 << 3; ++i) {
        BOOST_REQUIRE_EQUAL(randWriter.kRand("gc"), 'g');
    }
}

BOOST_AUTO_TEST_CASE(testKRand2) {
    RandWriter randWriter{ "gagggagaggcgagaaa", 3 };
    for (int i = 0; i < 2 << 3; ++i) {
        const auto nextChar = randWriter.kRand("gag");
        BOOST_REQUIRE(nextChar == 'g' || nextChar == 'a');
    }
}

// Checks if `RandWriter::generate()` works correctly: (1) The length of the generated text should
// be equal to the given length, and (2) the first k characters should be equal to the first k-gram.
BOOST_AUTO_TEST_CASE(testGenerate1) {
    constexpr int ORDER_K = 1;
    constexpr size_t LENGTH = 100;

    const std::string content = readFileContent("romeo.txt");
    RandWriter randWriter{ content, ORDER_K };
    const auto firstKGram = content.substr(0, ORDER_K);
    const auto generatedText = randWriter.generate(firstKGram, LENGTH);
    BOOST_REQUIRE_EQUAL(generatedText.size(), LENGTH);
    BOOST_REQUIRE_EQUAL(generatedText.substr(0, ORDER_K), firstKGram);
}

// Checks if `RandWriter::generate()` works correctly: (1) The length of the generated text should
// be equal to the given length, and (2) the first k characters should be equal to the first k-gram.
BOOST_AUTO_TEST_CASE(testGenerate2) {
    constexpr int ORDER_K = 3;
    constexpr size_t LENGTH = 300;

    const std::string content = readFileContent("tomsawyer.txt");
    RandWriter randWriter{ content, ORDER_K };
    const auto firstKGram = content.substr(0, ORDER_K);
    const auto generatedText = randWriter.generate(firstKGram, LENGTH);
    BOOST_REQUIRE_EQUAL(generatedText.size(), LENGTH);
    BOOST_REQUIRE_EQUAL(generatedText.substr(0, ORDER_K), firstKGram);
}

// Checks if `RandWriter::kRand()` follows a correct distribution.
BOOST_AUTO_TEST_CASE(testKRandDistribution) {
    constexpr int NUMBER_OF_TIMES = 10000;
    constexpr double PROBABILITY_OF_G = 0.8;
    constexpr float TOLERANCE = 10.F;

    RandWriter randWriter{ "gagggagaggcgagaaa", 2 };
    int numberOfG = 0;
    for (int i = 0; i < NUMBER_OF_TIMES; ++i) {
        const auto nextChar = randWriter.kRand("ga");
        if (nextChar == 'g') {
            ++numberOfG;
        }
    }

    const auto actualProbabilityOfG = static_cast<double>(numberOfG) / NUMBER_OF_TIMES;
    BOOST_REQUIRE_CLOSE(actualProbabilityOfG, PROBABILITY_OF_G, TOLERANCE);
}
```

8 PS7 - Kronos Time Clock

8.1 Discussion

This project entails developing a program designed to scan through log files in their entirety. The objective is to generate a comprehensive report file that chronologically details every instance when the device and its associated services underwent restart procedures.

In this project, I revisited the utilization of regular expressions. It brought me back to four years ago when I was a novice backend developer using PHP, first learning about them. Reflecting on this experience, I recalled the challenges of mastering regular expressions, but also the camaraderie I shared with my colleagues in China during that time.

8.2 Achievements

Below is an excerpt from a generated report:

```
Device Boot Report

InTouch log file: device5_intouch.log
Lines Scanned: 41855

Device boot count: initiated = 25, completed: 10


=== Device boot ===
31063(device5_intouch.log): 2014-01-26 09:55:07 Boot Start
31176(device5_intouch.log): 2014-01-26 09:58:04 Boot Completed
        Boot Time: 177000ms

Services
        Logging
                Start: 31079(device5_intouch.log)
                Completed: 31080(device5_intouch.log)
                Elapsed Time: 268 ms
        DatabaseInitialize
                Start: 31081(device5_intouch.log)
                Completed: 31109(device5_intouch.log)
                Elapsed Time: 44465 ms
        MessagingService
                Start: 31110(device5_intouch.log)
                Completed: 31112(device5_intouch.log)
                Elapsed Time: 5647 ms
        HealthMonitorService
                Start: 31126(device5_intouch.log)
                Completed: 31127(device5_intouch.log)
                Elapsed Time: 234 ms
        Persistence
                Start: 31128(device5_intouch.log)
                Completed: 31129(device5_intouch.log)
                Elapsed Time: 21260 ms
        ConfigurationService
                Start: 31130(device5_intouch.log)
                Completed: 31131(device5_intouch.log)
                Elapsed Time: 0 ms
        CacheService
                Start: 31132(device5_intouch.log)
                Completed: 31133(device5_intouch.log)
                Elapsed Time: 828 ms
        ThemingService
                Start: 31134(device5_intouch.log)
                Completed: 31135(device5_intouch.log)
                Elapsed Time: 0 ms
        PortConfigurationService
                Start: 31136(device5_intouch.log)
                Completed: 31137(device5_intouch.log)
                Elapsed Time: 81 ms
        LandingPadService
                Start: 31138(device5_intouch.log)
                Completed: 31139(device5_intouch.log)
                Elapsed Time: 1 ms
        DeviceIOService
                Start: 31140(device5_intouch.log)
                Completed: 31141(device5_intouch.log)
                Elapsed Time: 43 ms
        StagingService
                Start: 31142(device5_intouch.log)
                Completed: 31143(device5_intouch.log)
                Elapsed Time: 6734 ms
        GateService
                Start: 31144(device5_intouch.log)
                Completed: 31145(device5_intouch.log)
                Elapsed Time: 2 ms
        AVFeedbackService
                Start: 31146(device5_intouch.log)
                Completed: 31147(device5_intouch.log)
                Elapsed Time: 161 ms
        ReaderDataService
                Start: 31148(device5_intouch.log)
                Completed: 31149(device5_intouch.log)
                Elapsed Time: 2 ms
        BellService
                Start: 31150(device5_intouch.log)
                Completed: 31151(device5_intouch.log)
                Elapsed Time: 1 ms
        StateManager
                Start: 31152(device5_intouch.log)
                Completed: 31153(device5_intouch.log)
                Elapsed Time: 1584 ms
        OfflineSmartviewService
                Start: 31154(device5_intouch.log)
                Completed: 31155(device5_intouch.log)
                Elapsed Time: 14 ms
        DatabaseThreads
                Start: 31156(device5_intouch.log)
                Completed: 31166(device5_intouch.log)
                Elapsed Time: 4548 ms
        ProtocolService
                Start: 31157(device5_intouch.log)
                Completed: 31171(device5_intouch.log)
                Elapsed Time: 12305 ms
        SoftLoadService
                Start: 31159(device5_intouch.log)
                Completed: 31164(device5_intouch.log)
                Elapsed Time: 2617 ms
        WATCHDOG
                Start: 31160(device5_intouch.log)
                Completed: 31163(device5_intouch.log)
                Elapsed Time: 358 ms
        DiagnosticsService
                Start: 31161(device5_intouch.log)
                Completed: 31162(device5_intouch.log)
                Elapsed Time: 179 ms
        BiometricService
                Start: 31189(device5_intouch.log)
                Completed: 31191(device5_intouch.log)
                Elapsed Time: 2371 ms
```

8.3 Codebase

```makefile
# C++ Compiler
COMPILER = g++

# C++ Flags (C++ version: 20)
CFLAGS = --std=c++20 -Wall -Werror -pedantic -g

# Libraries
LIB = -lboost_unit_test_framework -lboost_date_time

# Code source directory
SRC = ./

# Hpp files (dependencies)
DEPS =

# Program
PROGRAM = ps7

# Program object files
MAIN_OBJECTS = $(SRC)main.o

all: $(PROGRAM)

$(SRC)%.o: $(SRC)%.cpp $(DEPS)
        $(COMPILER) $(CFLAGS) -c $<

$(PROGRAM): $(MAIN_OBJECTS)
        $(COMPILER) $(CFLAGS) -o $@ $^ $(LIB)

clean:
        rm -f $(SRC)*.o $(PROGRAM) $(STATIC_LIB) $(TEST_PROGRAM)

lint:
        cpplint *.hpp *.cpp

run: $(PROGRAM)
        ./$(PROGRAM) logs/device5_intouch.log
```

```makefile

run-all: $(PROGRAM)
	./$(PROGRAM) logs/device1_intouch.log
	./$(PROGRAM) logs/device2_intouch.log
	./$(PROGRAM) logs/device3_intouch.log
	./$(PROGRAM) logs/device4_intouch.log
	./$(PROGRAM) logs/device5_intouch.log
	./$(PROGRAM) logs/device6_intouch.log
```

To implement the program, I first created a **LogPattern** namespace to contain all regular expressions:

- **SERVER_START**: It matches datetime string in the format of *"YYYY-MM-DD HH:MM:SS"* and a fixed string in the server start line.
- **SERVER_COMPLETE**: It matches datetime string in the format of *"YYYY-MM-DD HH:MM:SS"*, three digits representing milliseconds, and a fixed string in the completion line.
- **SERVICE_START**: It matches a fixed string *"Starting Service."*, the name and version of the service.
- **SERVICE_COMPLETE**: It matches a fixed string *"Service started successfully."*, the name and version of the service, and the completion time in milliseconds.

```cpp
// <main.cpp>
#include <algorithm>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <regex>
#include <string>
#include <boost/date_time/posix_time/posix_time.hpp>

using boost::posix_time::ptime;

/**
 * @brief Log file regex patterns.
 */
namespace LogPattern {

// Match datetime string in the format of "YYYY-MM-DD HH:MM:SS"
constexpr auto* datetimePatternString = R"((\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}))";

// Match server start line
const std::regex
    SERVER_START(std::string(datetimePatternString) + R"(: \(log\.c\.166\) server started.*)");

// Match server boot complete line
const std::regex SERVER_COMPLETE(
    std::string(datetimePatternString) +
    R"(\.\d{3}:INFO:oejs\.AbstractConnector:Started SelectChannelConnector@0\.0\.0\.0:9080.*)");

// Match service start line
const std::regex SERVICE_START(R"(Starting Service\.\s*(\w*).*)");

// Match service complete line
const std::regex
    SERVICE_COMPLETE(R"(Service started successfully\.\s*(\w*)\s+\S*\s*\((\d+) ms\).*)");

}  // namespace LogPattern
```

Next, I established four structs in hierarchy:

- **LogEntry**: It contains metadata about a log file, such as its name.
- **BootLogEntry**: It extends **LogEntry** to include information about boot processes, such as the start and completion line numbers.
- **ServiceBootLogEntry**: It represents a log entry specific to the boot process of a service. Extends **LogEntry** to include details like service name and elapsed time.
- **DeviceBootLogEntry**: It represents a log entry specific to the boot process of a server. Extends **BootLogEntry** to include start and completion date times, as well as a collection of **ServiceBootLogEntry** instances.

```
struct LogEntry {
    // Name of the log file
    std::string filename;
};

struct BootLogEntry : LogEntry {
    // Boot start log line number
    int start_line_number = 0;

    // Boot completed log line number
    int completed_line_number = 0;
};

struct ServiceBootLogEntry : BootLogEntry {
    // The name of the service
    std::string service_name;

    // Elapsed time
    int elapsed_time = -1;
};

struct DeviceBootLogEntry : BootLogEntry {
    // Boot start datetime
    ptime start_datetime;

    // Boot completed datetime
    ptime completed_datetime;

    // Services log entries
    std::vector<ServiceBootLogEntry> service_log_entries;
};
```

Following this, five auxiliary functions have been implemented to improve code readability and maintainability. Each function is thoroughly commented to provide clarity on its purpose and functionality.

```
/**
 * @brief Traverses the lines of a file, invoking a callback function for each line. This function
 * reads each line from the provided input file stream and invokes the specified callback function
 * for each line. The callback function is called with two parameters: the current line as a string
 * and its corresponding line number.
```

```cpp
 * @param ifstream The input file stream to traverse.
 * @param callback The callback function to invoke for each line, taking a string representing the
 * line content and an integer representing the line number as parameters.
 * @return The total line number of the file.
 */
int traverseFile(
    std::ifstream& ifstream, const std::function<void(const std::string&, int)>& callback);

/**
 * @brief Finds matches in a given string using a regular expression pattern. This function searches
 * for matches of the specified regular expression pattern within the provided string. If one or
 * more matches are found, the callback function is invoked for each match, providing the matched
 * substring and its index within the original string as parameters.
 * @param string The string to search for matches.
 * @param pattern The regular expression pattern to match against the string.
 * @param callback The callback function to invoke for each match, taking two parameters: the
 * matched substring and its index within the original string.
 */
void match(
    const std::string& string,
    const std::regex& pattern,
    const std::function<void(const std::string&, int)>& callback);

/**
 * @brief Parses a datetime string into a Boost ptime object. This function takes a datetime string
 * in the format "YYYY-MM-DD HH:MM:SS" and converts it into a Boost ptime object. If the parsing
 * fails, an invalid ptime object is returned.
 * @param datetimeString The datetime string to parse.
 * @return A Boost ptime object representing the parsed datetime. If the parsing fails, an invalid
 * ptime object is returned.
 */
ptime parseDatetime(const std::string& datetimeString);

/**
 * @brief Converts a Boost ptime object into a string representation. This function converts the
 * specified Boost ptime object into a string representation using the format "YYYY-MM-DD HH:MM:SS".
 * The conversion is performed by utilizing a custom time_facet imbued with the desired format.
 * @param datetime The Boost ptime object to convert into a string.
 * @return A string representation of the specified datetime in the format "YYYY-MM-DD HH:MM:SS".
 */
std::string toString(const ptime& datetime);

/**
 * @brief Outputs device boot log items in a particular format.
 * @param ostream The output stream.
 * @param deviceBootLogItems The device boot log items to output.
 * @param filename The name of the scanned file.
 * @param numScannedLine The number of scanned lines.
 */
void outputDeviceBootLogItems(
    std::ostream& ostream,
    const std::vector<DeviceBootLogEntry>& deviceBootLogItems,
    const std::string& filename,
    int numScannedLine);

int traverseFile(
    std::ifstream& ifstream, const std::function<void(const std::string&, int)>& callback) {
    int line_number = 1;
    std::string line;
    while (getline(ifstream, line)) {
        callback(line, line_number);
        ++line_number;
    }

    ifstream.close();

    return line_number - 1;
}

void match(
    const std::string& string,
    const std::regex& pattern,
    const std::function<void(const std::string&, int)>& callback) {
    std::smatch matches;
    if (std::regex_search(string, matches, pattern)) {
        int index = 0;
        for (const auto& matchItem : matches) {
            callback(matchItem, index++);
        }
    }
}

ptime parseDatetime(const std::string& datetimeString) {
    // Define the format of the datetime string
    static const std::locale locale(
        std::locale::classic(), new boost::posix_time::time_input_facet("%Y-%m-%d %H:%M:%S"));

    std::stringstream ss{ datetimeString };
    ss.imbue(locale);
    ptime datetime;

    return ss >> datetime ? datetime : ptime{};
}

std::string toString(const ptime& datetime) {
    static const std::locale locale(
        std::locale::classic(), new boost::posix_time::time_facet("%Y-%m-%d %H:%M:%S"));

    std::stringstream ss;
    ss.imbue(locale);
    ss << datetime;

    return ss.str();
}

void outputDeviceBootLogItems(
    std::ostream& ostream,
    const std::vector<DeviceBootLogEntry>& deviceBootLogItems,
    const std::string& filename,
    const int numScannedLine) {
    // Initiated boot count and completed boot count
    const auto initiatedBootCount = static_cast<int64_t>(deviceBootLogItems.size());
```

```cpp
    const auto completedBootCount = std::count_if(
        deviceBootLogItems.cbegin(), deviceBootLogItems.cend(),
        [](const DeviceBootLogEntry& entry) { return entry.completed_line_number > 0; });

    // Print header
    ostream << "Device Boot Report" << std::endl << std::endl;
    ostream << "InTouch log file: " << filename << std::endl;
    ostream << "Lines Scanned: " << numScannedLine << std::endl << std::endl;
    ostream << "Device boot count: initiated = " << initiatedBootCount
            << ", completed: " << completedBootCount << std::endl
            << std::endl
            << std::endl;

    const auto serviceBootLogEntryHandler = [&](const ServiceBootLogEntry& entry) {
        ostream << "\t" << entry.service_name << std::endl;
        if (entry.start_line_number > 0) {
            ostream << "\t\t Start: " << entry.start_line_number << "(" << entry.filename << ")"
                    << std::endl;
        } else {
            ostream << "\t\t Start: Not started(" << entry.filename << ")" << std::endl;
        }

        if (entry.completed_line_number > 0) {
            ostream << "\t\t Completed: " << entry.completed_line_number << "(" << entry.filename
                    << ")" << std::endl;
            ostream << "\t\t Elapsed Time: " << entry.elapsed_time << " ms" << std::endl;
        } else {
            ostream << "\t\t Completed: Not Completed(" << entry.filename << ")" << std::endl;
            ostream << "\t\t Elapsed Time: " << std::endl;
        }
    };

    bool isFirstEntry = true;
    const auto deviceBootLogEntryHandler = [&](const DeviceBootLogEntry& entry) {
        if (isFirstEntry) {
            isFirstEntry = false;
        } else {
            ostream << std::endl;
        }

        // Print the boot log entry
        ostream << "=== Device boot ===" << std::endl;
        ostream << entry.start_line_number << "(" << entry.filename
                << "): " << toString(entry.start_datetime) << " Boot Start" << std::endl;
        if (entry.completed_line_number > 0) {
            ostream << entry.completed_line_number << "(" << entry.filename
                    << "): " << toString(entry.completed_datetime) << " Boot Completed"
                    << std::endl;
            const auto duration = entry.completed_datetime - entry.start_datetime;
            ostream << "\tBoot Time: " << duration.total_milliseconds() << "ms" << std::endl;
        } else {
            ostream << "**** Incomplete boot **** " << std::endl;
        }

        // Print service entries
        ostream << std::endl << "Services" << std::endl;
        const auto& service_log_entries = entry.service_log_entries;
        std::for_each(
            service_log_entries.cbegin(), service_log_entries.cend(), serviceBootLogEntryHandler);
    };
    std::for_each(
        deviceBootLogItems.cbegin(), deviceBootLogItems.cend(), deviceBootLogEntryHandler);
}
```

With all these preparations in place, the main file is quite straightforward. It accepts the path of a log file as input and generates a comprehensive report file with the same name but with a ”.rpt” suffix.

```
/**
 * @brief Starts the universe simulation.
 * @param argc The number of arguments.
 * @param argv The arguments array. This program requires one argument:
 * 1. The path of the log file.
 */
int main(const int argc, const char* argv[]) {
    constexpr auto STATUS_ERROR = 1;
    // Names of all services that are started in every boot
    const std::vector<std::string> serviceNameVector = { "Logging",
                                                         "DatabaseInitialize",
                                                         "MessagingService",
                                                         "HealthMonitorService",
                                                         "Persistence",
                                                         "ConfigurationService",
                                                         "CacheService",
                                                         "ThemingService",
                                                         "PortConfigurationService",
                                                         "LandingPadService",
                                                         "DeviceIOService",
                                                         "StagingService",
                                                         "GateService",
                                                         "AVFeedbackService",
                                                         "ReaderDataService",
                                                         "BellService",
                                                         "StateManager",
                                                         "OfflineSmartviewService",
                                                         "DatabaseThreads",
                                                         "ProtocolService",
                                                         "SoftLoadService",
                                                         "WATCHDOG",
                                                         "DiagnosticsService",
                                                         "BiometricService" };

    // Check the arguments
    if (argc != 2) {
        std::cerr << "Too many or too few arguments!" << std::endl;
        return STATUS_ERROR;
    }

    // Try to open the file; exit the program if failing
    std::filesystem::path filepath{ argv[1] };
    std::string filename = filepath.filename().string();
    std::ifstream fileStream{ filepath };
    if (!fileStream.is_open()) {
        std::cerr << "Fail to open file: " << filepath << std::endl;
        return STATUS_ERROR;
    }

    std::vector<DeviceBootLogEntry> device_boot_log_entries;
    const int numScannedLine = traverseFile(fileStream, [&](auto& line, auto line_number) {
        // Match "server start" lines
        match(line, LogPattern::SERVER_START, [&](auto datetimeString, auto index) {
            if (index != 1) {
                return;
            }

            // Add absent services to previous device boot log entry
            if (!device_boot_log_entries.empty()) {
                auto& previous_entry = device_boot_log_entries.back();
                auto& service_log_entries = previous_entry.service_log_entries;
                for (const auto& serviceName : serviceNameVector) {
                    bool found = false;
                    for (const auto& entry : service_log_entries) {
                        if (entry.service_name == serviceName) {
                            found = true;
                            break;
                        }
                    }

                    if (!found) {
                        ServiceBootLogEntry new_entry;
                        new_entry.filename = filename;
                        new_entry.service_name = serviceName;
                        service_log_entries.push_back(new_entry);
                    }
                }
            }

            // Add new boot log entry
            DeviceBootLogEntry log_entry;
            log_entry.filename = filename;
            log_entry.start_line_number = line_number;
            log_entry.start_datetime = parseDatetime(datetimeString);
            device_boot_log_entries.push_back(log_entry);
        });

        // Match "server complete" lines
        match(line, LogPattern::SERVER_COMPLETE, [&](auto datetime_string, auto index) {
            if (index == 1 && !device_boot_log_entries.empty()) {
                auto& log_entry = device_boot_log_entries.back();
                log_entry.completed_line_number = line_number;
                log_entry.completed_datetime = parseDatetime(datetime_string);
            }
        });

        if (device_boot_log_entries.empty()) {
            return;
        }

        auto& service_boot_log_entries = device_boot_log_entries.back().service_log_entries;

        // Match "service start" lines
        match(line, LogPattern::SERVICE_START, [&](auto service_name, auto index) {
            if (index == 1) {
                ServiceBootLogEntry log_entry;
                log_entry.filename = filename;
                log_entry.start_line_number = line_number;
                log_entry.service_name = service_name;
                service_boot_log_entries.push_back(log_entry);
            }
        });

        // Match "service complete" lines
        std::string service_name;
        match(line, LogPattern::SERVICE_COMPLETE, [&](auto matched_string, auto index) {
            if (index == 1) {
                service_name = matched_string;
            } else if (index == 2) {
                for (auto& entry : service_boot_log_entries) {
                    if (!service_name.empty() && entry.service_name == service_name) {
                        entry.completed_line_number = line_number;
                        entry.elapsed_time = std::stoi(matched_string);
                    }
                }
            }
        });
    });

    // Output all device boot log items to .rpt file
    std::filesystem::path reportFilepath{ filepath.string() + ".rpt" };
    std::ofstream outputStream{ reportFilepath };
    outputDeviceBootLogItems(outputStream, device_boot_log_entries, filename, numScannedLine);
    outputStream.close();

    return 0;
}
```