COMP315

Project Documentation

TRAPPED

Members	
1. Caitlin Fogg	223005053
2. Denica Chetty	223007201
3. Naseeha Osman	223005931
4. Firdous Mariam Khan	223051520
5. Sasthi Naidoo	223004483
6. Mohammed Saib	223007771
7. Rowan Naidoo	223087231
8. Keryn Scott	223009862
9. Tarika Sukdeoa	223010024

Table Of Contents

Introduction

Trapped is a game developed in C++ using SFML, it is designed to engage players in a choice-driven narrative while testing their knowledge on various topics. *Trapped* is structured like an escape room, where the player is kidnapped while visiting an escape room and held captive in a house which they must escape while being hunted by a killer. The player progresses through the story by picking the correct choices. Each choice will influence the storyline by offering different paths, challenges and potential endings.

The game starts with a main menu, allowing players to start or exit. The user interface features a textbox for narration and dialogue, while background images serve as visual representations of the game environment. Players interact by selecting options with the mouse, making choices that impact their escape.

Trapped incorporates an inventory system that enables players to collect items as they progress. The game tracks questions, where incorrect answers will lead to consequences such death. The game also has a text to speech feature, which enhances the player's experience and improves accessibility for players with reading difficulties. Audio elements enhance the players experience, making it more immersive.

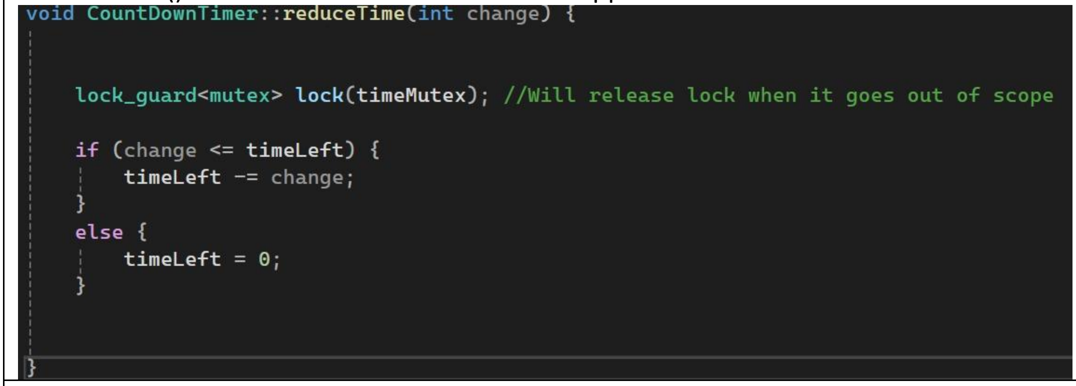
To ensure modularity and scalability, the game design incorporates structured software development. This project mixes storytelling, quiz challenges, and game development to create a fun, immersive experience that also uses key C++ programming techniques.

Programming Techniques

1. Function

Screenshot:

reduceTime () function from CountDownTimer.cpp



```
void CountDownTimer::reduceTime(int change) {


    lock_guard<mutex> lock(timeMutex); //Will release lock when it goes out of scope

    if (change <= timeLeft) {
        timeLeft -= change;
    }
    else {
        timeLeft = 0;
    }


}
```

Motivation:

The reduceTime() function is called every time the timer is reduced, caused by actions of the player. It ensures that the timeLeft variable is never below zero. The use of a mutex ensures mutual exclusion of the variable timeMutex. This allows only one thread to access and modify timeLeft at a time.

A lock_guard is used with timeMutex to automatically lock the mutex at the start of the function and release it when the scope ends. This ensures thread safety, maintaining safe and efficient access.

By encapsulating this logic within a function, it avoids unnecessary code repetition. It also improves modularity, readability, efficiency, and reusability of the code.

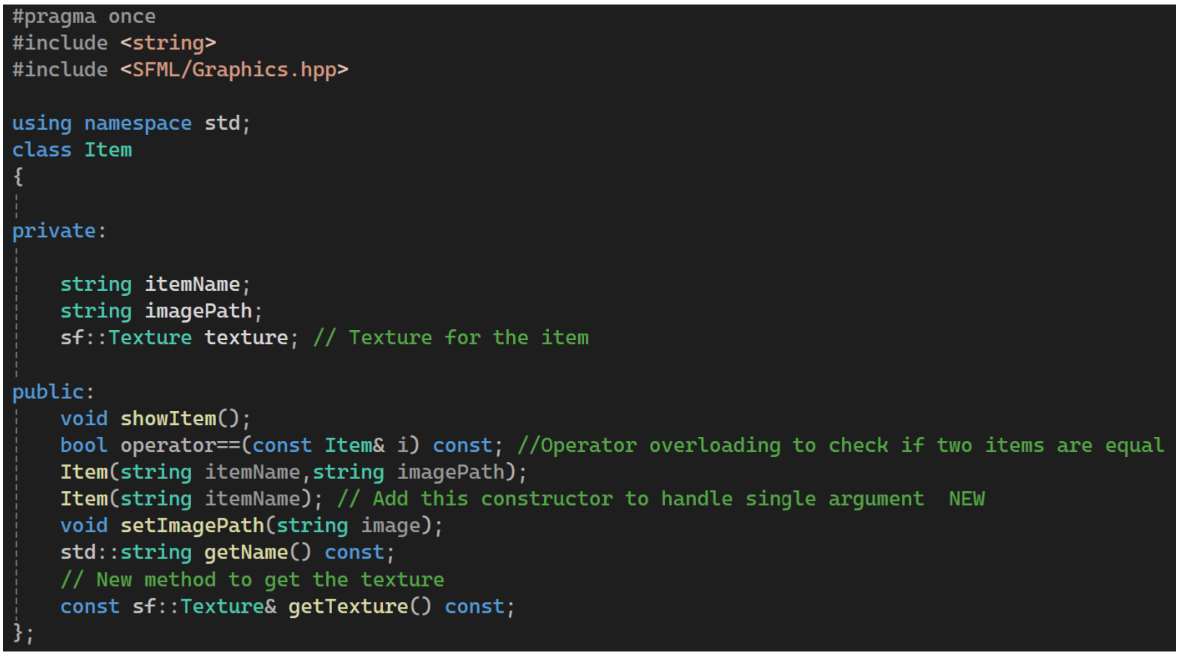
How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	Using the reduceTime() function prevents the need to duplicate logic, making the code more modular, efficient and easier to maintain. This improves clarity, encapsulation and reusability of the code by

		keeping the time reduction logic in one place.

2. Class

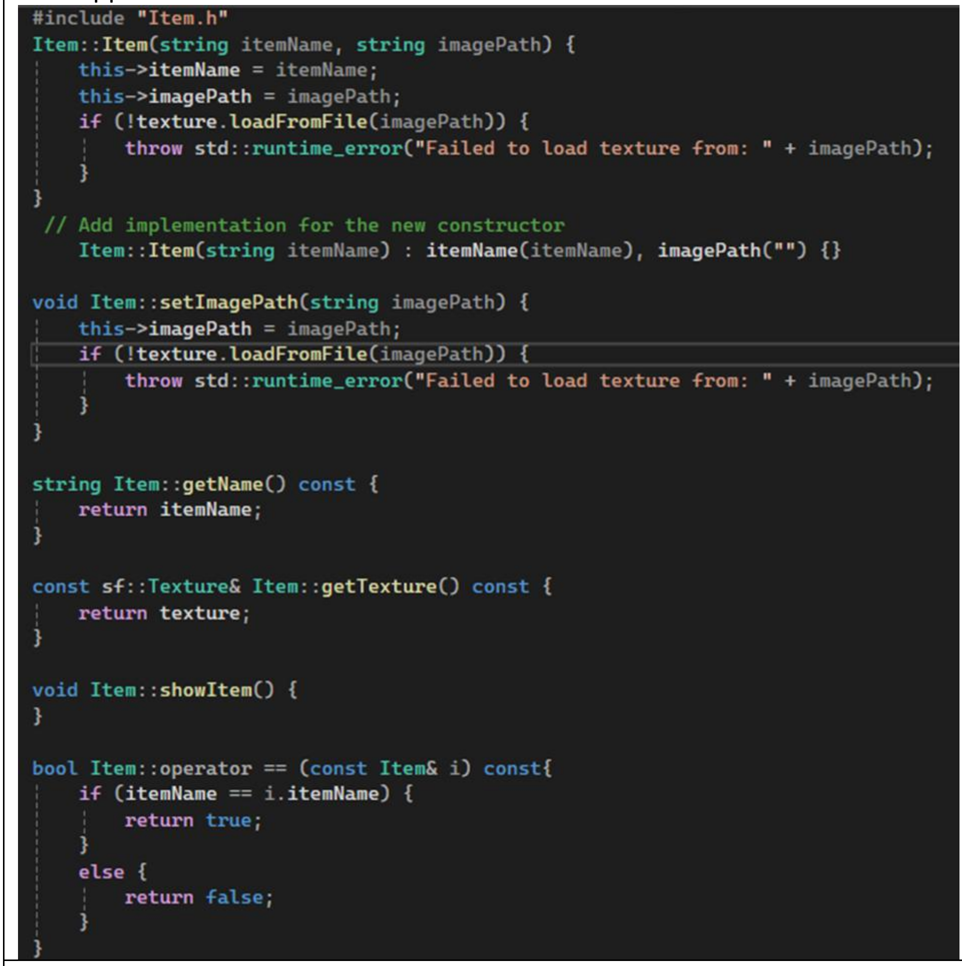
Screenshot:

Item.h



```
#pragma once
#include <string>
#include <SFML/Graphics.hpp>

using namespace std;
class Item
{

private:

    string itemName;
    string imagePath;
    sf::Texture texture; // Texture for the item

public:
    void showItem();
    bool operator==(const Item& i) const; //Operator overloading to check if two items are equal
    Item(string itemName,string imagePath);
    Item(string itemName); // Add this constructor to handle single argument  NEW
    void setImagePath(string image);
    std::string getName() const;
    // New method to get the texture
    const sf::Texture& getTexture() const;
};
```

Item.cpp



```cpp
#include "Item.h"
Item::Item(string itemName, string imagePath) {
    this->itemName = itemName;
    this->imagePath = imagePath;
    if (!texture.loadFromFile(imagePath)) {
        throw std::runtime_error("Failed to load texture from: " + imagePath);
    }
}
 // Add implementation for the new constructor
    Item::Item(string itemName) : itemName(itemName), imagePath("") {}

void Item::setImagePath(string imagePath) {
    this->imagePath = imagePath;
    if (!texture.loadFromFile(imagePath)) {
        throw std::runtime_error("Failed to load texture from: " + imagePath);
    }
}

string Item::getName() const {
    return itemName;
}

const sf::Texture& Item::getTexture() const {
    return texture;
}

void Item::showItem() {
}

bool Item::operator == (const Item& i) const{
    if (itemName == i.itemName) {
        return true;
    }
    else {
        return false;
    }
}
```

Motivation:

The Item class is used to store important details for an item, such as its name, image file path and texture. It handles loading the item's image into a texture and provides functions for comparing and managing items. The Item class prevents duplication of image loading code and simplifies item setup across the game. This makes it easier to pass and manage items for the inventory and rooms and reduces memory usage. It also allows the Inventory class to focus solely on managing collections of items, while the Item class handles individual item properties. This modular design improves readability, keeps the code well-organized, and makes it easier to modify or expand the game by updating the class only.

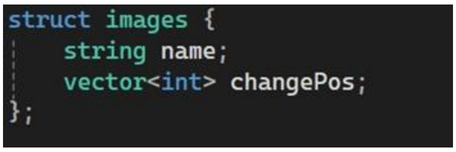
How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	The Item class is used in the inventory class to keep track of all the items collected and their details. It is also used in all the rooms to display collected items, allowing consistent item handling

		throughout the game. The item class centralizes the item logic to one place, keeping the code easier to organize and maintain.

3. Struct

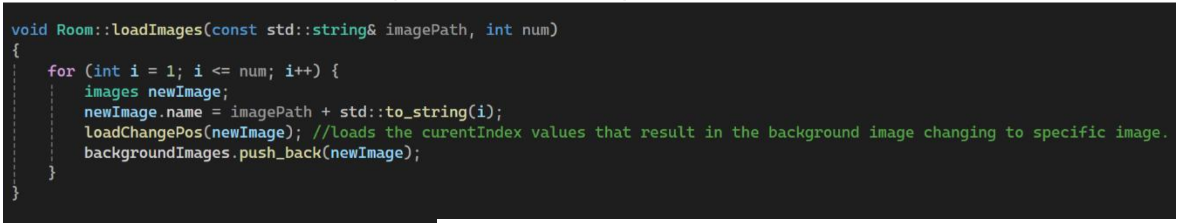
Screenshot:

Struct declaration

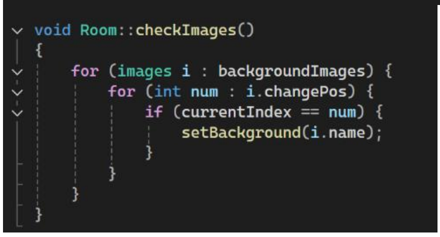


```
struct images {
    string name;
    vector<int> changePos;
};
```

The struct is used in the loadImages() and checkImages() functions



```
void Room::loadImages(const std::string& imagePath, int num)
{
    for (int i = 1; i <= num; i++) {
        images newImage;
        newImage.name = imagePath + std::to_string(i);
        loadChangePos(newImage); //loads the curentIndex values that result in the background image changing to specific image.
        backgroundImages.push_back(newImage);
    }
}
```



```
void Room::checkImages()
{
    for (images i : backgroundImages) {
        for (int num : i.changePos) {
            if (currentIndex == num) {
                setBackground(i.name);
            }
        }
    }
}
```

Motivation:

The struct images is used to efficiently group related data about images when there are multiple background images in a room.

The name in the struct is used to keep the name of the image and the vector keeps track of the question numbers where the screen will change to the relevant image.

The above functions implement the use of this struct.

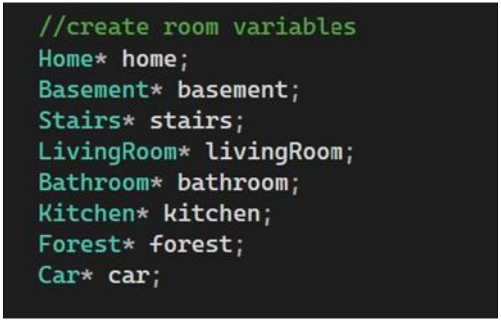
How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	This struct demonstrates the grouping of logically connected attributes into a reusable data type. Enhances code readability and makes it

		easier to manage the image logic.

4. Pointer

Screenshot:

Room pointers in Game.h



```
//create room variables
Home* home;
Basement* basement;
Stairs* stairs;
LivingRoom* livingRoom;
Bathroom* bathroom;
Kitchen* kitchen;
Forest* forest;
Car* car;
```

Motivation:

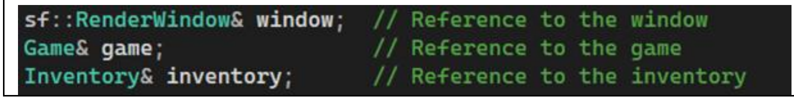
These pointers are all used for room objects to allow for dynamic memory management. By using these pointers we can create and delete rooms during runtime, which saves memory and enables transitions between scenes efficiently. This allows the use of base class pointers to interact with different room types polymorphically. Without pointers, the game would inefficiently use memory for every room at once, even if only one is in use.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	**X**	Pointers are used to dynamically allocate, access, and clean up room objects. They enable flexibility in managing game states and transitioning between scenes, while also supporting polymorphism through base class handling.

5. Reference

Screenshot:

Reference declarations in each derived room class



```
sf::RenderWindow& window;   // Reference to the window
Game& game;                 // Reference to the game
Inventory& inventory;       // Reference to the inventory
```

Motivation:

By using references, classes can directly access and modify the objects and allows any changes to be immediately reflected.

As shown above, a reference to window is created to ensure rendering occurs on the main game window. Any changes to the game's logic state are updated on the actual game instance, not a copy. The inventory reference lets the class interact directly with the player's inventory without creating separate copies, allowing changes to be made, keeping the item data consistent across the game.

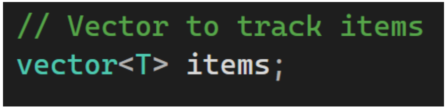
Using references prevents unnecessary duplication of objects and allows the game to be in sync, where the classes interact directly with the game. This also improves performance and helps avoid memory issues created when copying large objects like the game window.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	The references directly modify the original objects (e.g. game, inventory) which avoids unnecessary duplication. It allows synchronization for the game components to help it run smoothly, improving overall performance.

6. Vector

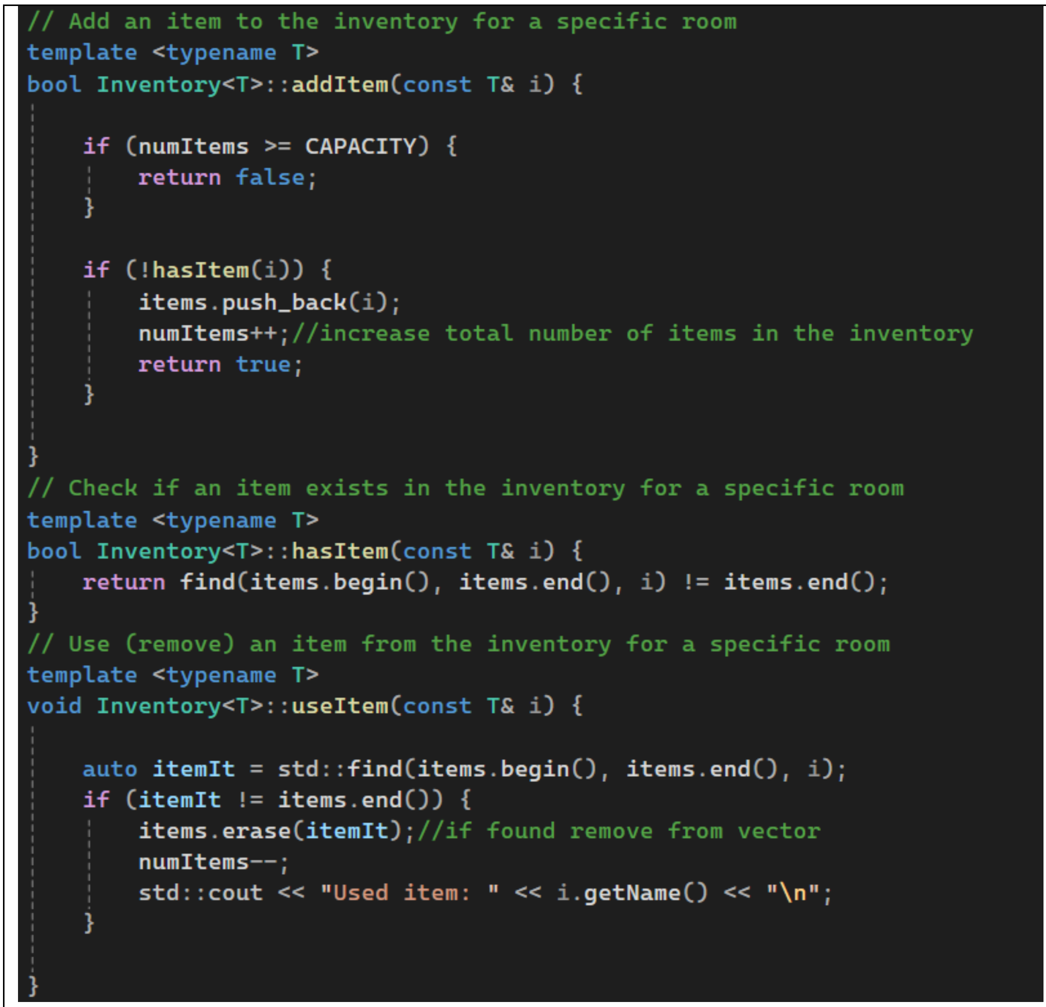
Screenshot:

Declaration in Inventory.h



```
// Vector to track items
vector<T> items;
```

Used in Inventory.cpp



```
// Add an item to the inventory for a specific room
template <typename T>
bool Inventory<T>::addItem(const T& i) {

    if (numItems >= CAPACITY) {
        return false;
    }

    if (!hasItem(i)) {
        items.push_back(i);
        numItems++;//increase total number of items in the inventory
        return true;
    }

}
// Check if an item exists in the inventory for a specific room
template <typename T>
bool Inventory<T>::hasItem(const T& i) {
    return find(items.begin(), items.end(), i) != items.end();
}
// Use (remove) an item from the inventory for a specific room
template <typename T>
void Inventory<T>::useItem(const T& i) {

    auto itemIt = std::find(items.begin(), items.end(), i);
    if (itemIt != items.end()) {
        items.erase(itemIt);//if found remove from vector
        numItems--;
        std::cout << "Used item: " << i.getName() << "\n";
    }

}
```

Motivation:

Vectors are used to manage the list of item objects in the inventory. Since the number of items, a player can currently hold is not fixed, vectors provide the flexibility of dynamic resizing. They allow efficient access and iteration, which is needed in the inventory class when checking for the existence of an item or removing it. Since vectors are part of the C++ STL they allow the use of iterators, which are used in functions such as addItem(), hasItem() and useItem() to perform operations on the inventory efficiently.

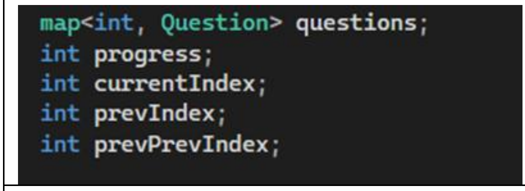
How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	**X**	Since the number of items a player has currently had in their inventory is not fixed, using a vector to store these is the most efficient way. It allows for dynamic resizing

		and easy iteration. This makes it suitable for managing inventory items.

7. Data Structures

Screenshot:

Declaration in Room.h



```
map<int, Question> questions;
int progress;
int currentIndex;
int prevIndex;
int prevPrevIndex;
```

Motivation:

A map, called questions, is used to store the questions in each room. Each question has a number, and each option is directed to a question. Questions are not in chronological order, so a map is used instead of a vector to allow us to control the keys we give each question.

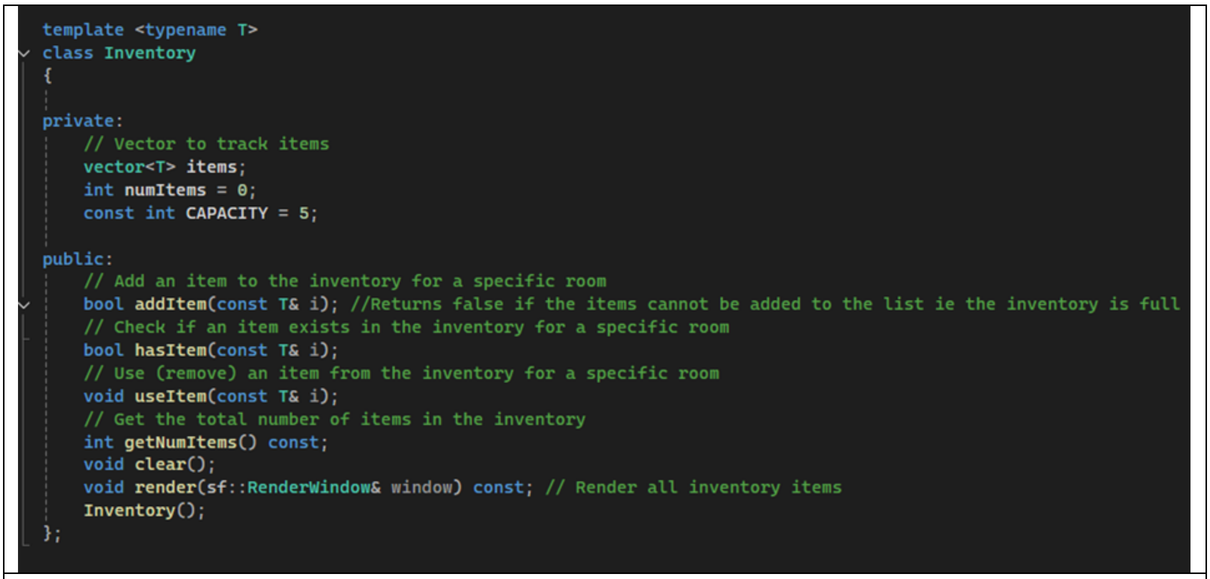
The use of a map allows fast and direct access using keys to retrieve the questions. The questions can be navigated easily, being able to go forward, backward and to specific questions, which is needed in the game as the player's choices leads to different paths in the storyline.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	The map allows the game to efficiently store and access the questions using keys. This supports the logic of the game where the player choices leads to different paths. It enables quick access and navigation to the questions, which is important for progression of the game.

8. Class Template

Screenshot:

From Inventory.h



```
template <typename T>
class Inventory
{

private:
    // Vector to track items
    vector<T> items;
    int numItems = 0;
    const int CAPACITY = 5;

public:
    // Add an item to the inventory for a specific room
    bool addItem(const T& i); //Returns false if the items cannot be added to the list ie the inventory is full
    // Check if an item exists in the inventory for a specific room
    bool hasItem(const T& i);
    // Use (remove) an item from the inventory for a specific room
    void useItem(const T& i);
    // Get the total number of items in the inventory
    int getNumItems() const;
    void clear();
    void render(sf::RenderWindow& window) const; // Render all inventory items
    Inventory();
};
```

Motivation:

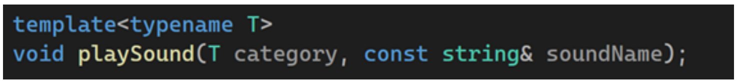
A class template is used for the Inventory class so it can handle different types of items, such as weapons or regular items, without rewriting the logic for each type. By templating the item type, it ensures the inventory class can be reused with various item types. The game can be easily expanded as more item types are introduced, avoiding unnecessary repetition.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	The class template allows the Inventory class to be reused in many other classes, where it can manage different item types without rewriting the logic. Makes the game more scalable and easier to manage.

9. Function Template

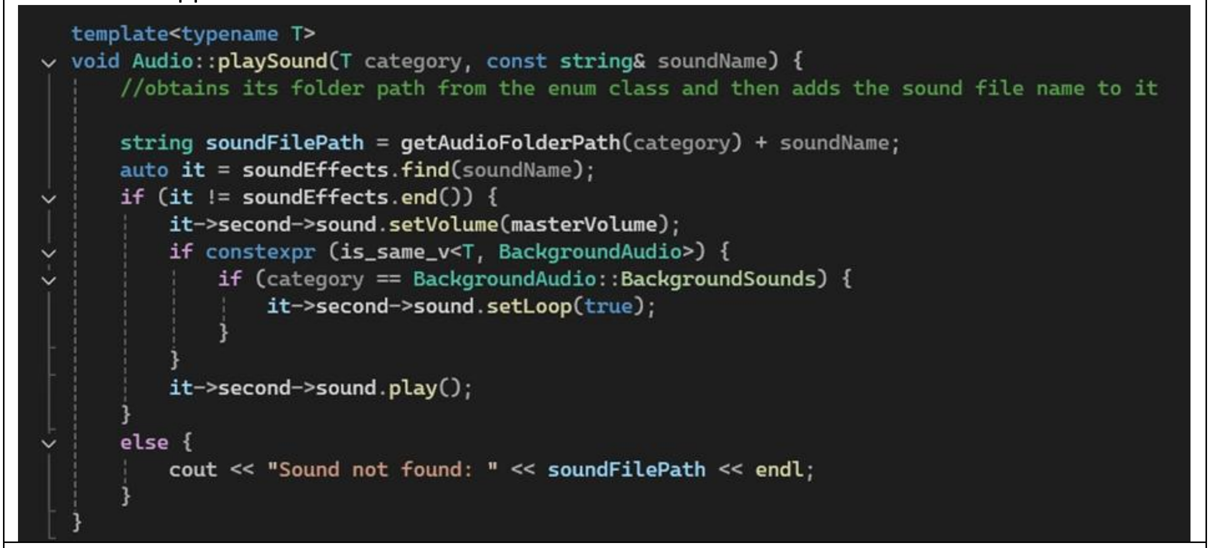
Screenshot:

From Audio.h



```
template<typename T>
void playSound(T category, const string& soundName);
```

From Audio.cpp



```
template<typename T>
void Audio::playSound(T category, const string& soundName) {
    //obtains its folder path from the enum class and then adds the sound file name to it

    string soundFilePath = getAudioFolderPath(category) + soundName;
    auto it = soundEffects.find(soundName);
    if (it != soundEffects.end()) {
        it->second->sound.setVolume(masterVolume);
        if constexpr (is_same_v<T, BackgroundAudio>) {
            if (category == BackgroundAudio::BackgroundSounds) {
                it->second->sound.setLoop(true);
            }
        }
        it->second->sound.play();
    }
    else {
        cout << "Sound not found: " << soundFilePath << endl;
    }
}
```

Motivation:

A function template is used for playSound to allow flexibility when handling different sound categories, such as sound effects and background music. Instead of writing multiple overloaded functions for each sound type, the use of a function template allows us to work with any enum type representing a sound category. This avoids unnecessary repetition.

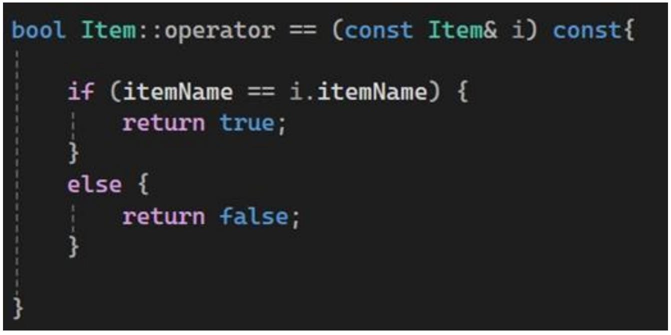
The audio system is now more scalable because when new categories are added, only a new enum type needs to be defined without needing to rewrite the logic. The template function reusable and easier to maintain.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	The use of the function template avoids unnecessary repetition by allowing playSound to work with multiple category types. Makes sound logic more scalable and easier to maintain. If another category needs to be added in the future, it is easier because a new enum type will be added, instead of needing hardcode more versions of the function.

10. Operator Overloading

Screenshot:

Overloading in Item.cpp



```
bool Item::operator == (const Item& i) const{

    if (itemName == i.itemName) {
        return true;
    }
    else {
        return false;
    }

}
```

Motivation:

Operator overloading was used instead of a regular method because comparing Item objects inside a vector, as done in find(), requires the == operator to be defined. The overloaded == operator allows two Item objects to be compared directly based on their itemName. Defining the operator was necessary for proper functionality. This avoids repeated string comparisons and encapsulates the logic within the Item class.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	**X**	Operator overloading was used to simplify item comparison. It enhances readability and reduces repetition.

11. Object Oriented Programming

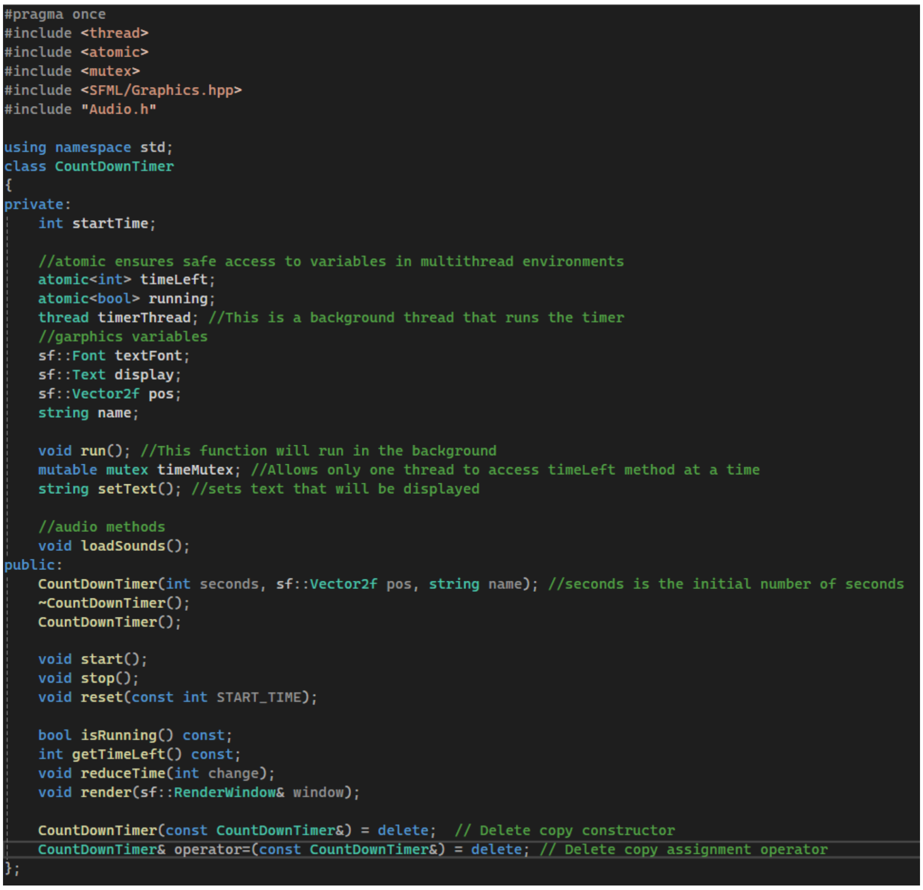
How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide evidence to support the claim
Not met		
Partially		
Completely	**X**	**Class:** Used to define different types of objects in the game. For example, we need a game class to manage the game engine and room classes to define different scenes. **Encapsulation:** Access control (private/public keywords) in each class access control keeps some variables private and provides controlled access through functions (like getters and setters). **Abstraction:** Defined interfaces to describe how game objects behave without exposing how it internally works. For example, declaring the update and render functions as virtual functions in the Room class allows all rooms to share common structure without knowing their specific behaviour. **Inheritance:** There is a generalization-specialization relationship between the base Room class and all specific rooms. These subclasses inherit the basic properties and functions of room and override them if needed. **Polymorphism:** The update and render functions behave differently depending on the type of object it is acting upon.

Additional Features

1. Timer

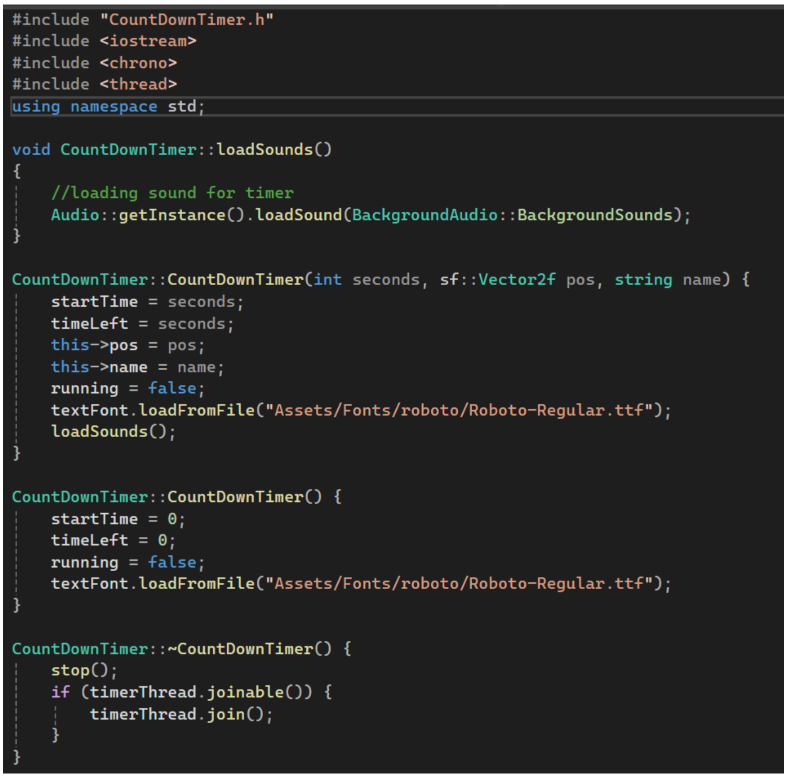
As the player progresses through the levels, the difficultly increases. A timer is introduced to increase pressure and urgency when answering questions.

CountDownTimer.h

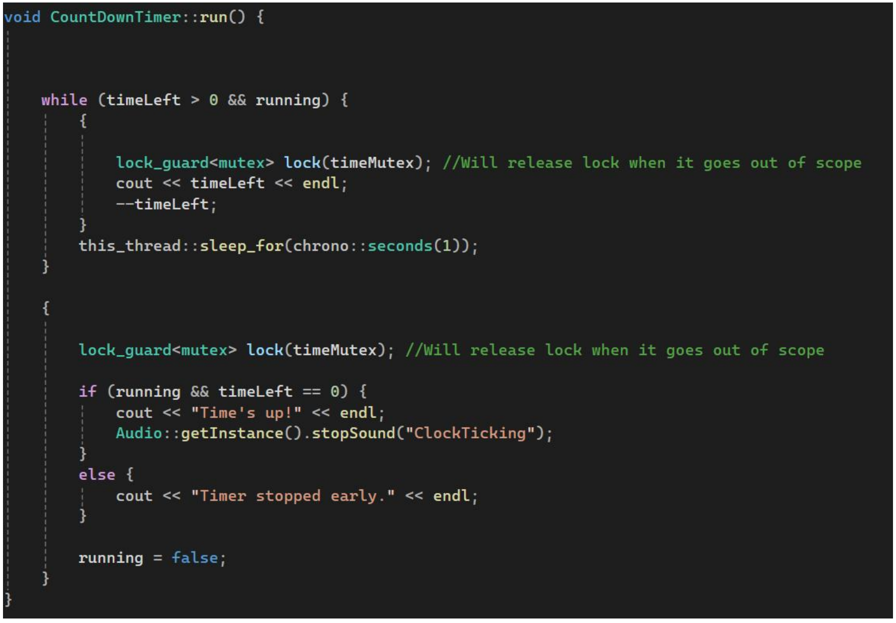


```
#pragma once
#include <thread>
#include <atomic>
#include <mutex>
#include <SFML/Graphics.hpp>
#include "Audio.h"

using namespace std;
class CountDownTimer
{
private:
    int startTime;

    //atomic ensures safe access to variables in multithread environments
    atomic<int> timeLeft;
    atomic<bool> running;
    thread timerThread; //This is a background thread that runs the timer
    //garphics variables
    sf::Font textFont;
    sf::Text display;
    sf::Vector2f pos;
    string name;

    void run(); //This function will run in the background
    mutable mutex timeMutex; //Allows only one thread to access timeLeft method at a time
    string setText(); //sets text that will be displayed

    //audio methods
    void loadSounds();
public:
    CountDownTimer(int seconds, sf::Vector2f pos, string name); //seconds is the initial number of seconds
    ~CountDownTimer();
    CountDownTimer();

    void start();
    void stop();
    void reset(const int START_TIME);

    bool isRunning() const;
    int getTimeLeft() const;
    void reduceTime(int change);
    void render(sf::RenderWindow& window);

    CountDownTimer(const CountDownTimer&) = delete;  // Delete copy constructor
    CountDownTimer& operator=(const CountDownTimer&) = delete; // Delete copy assignment operator
};
```

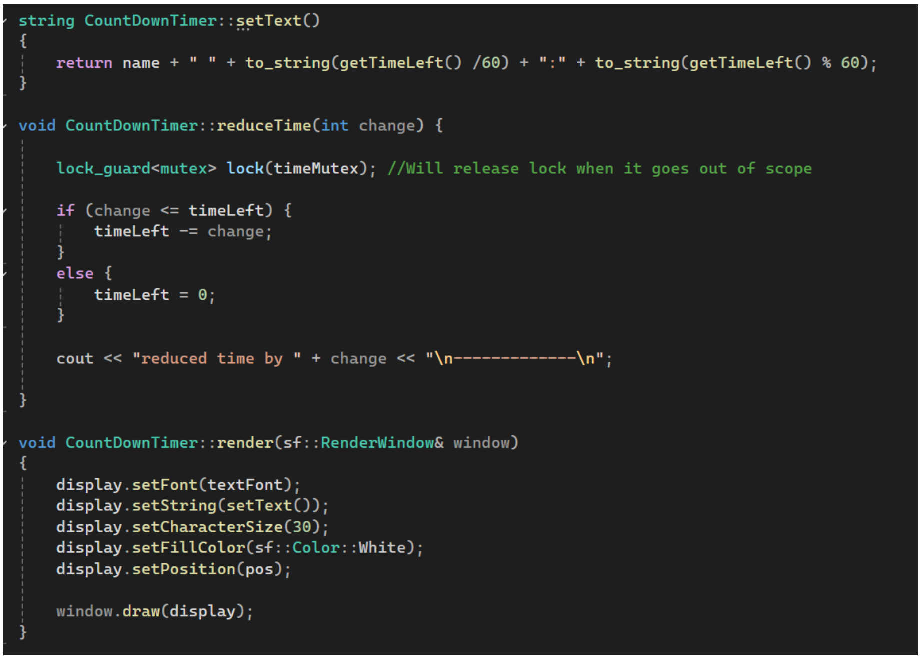
CountDownTimer.cpp



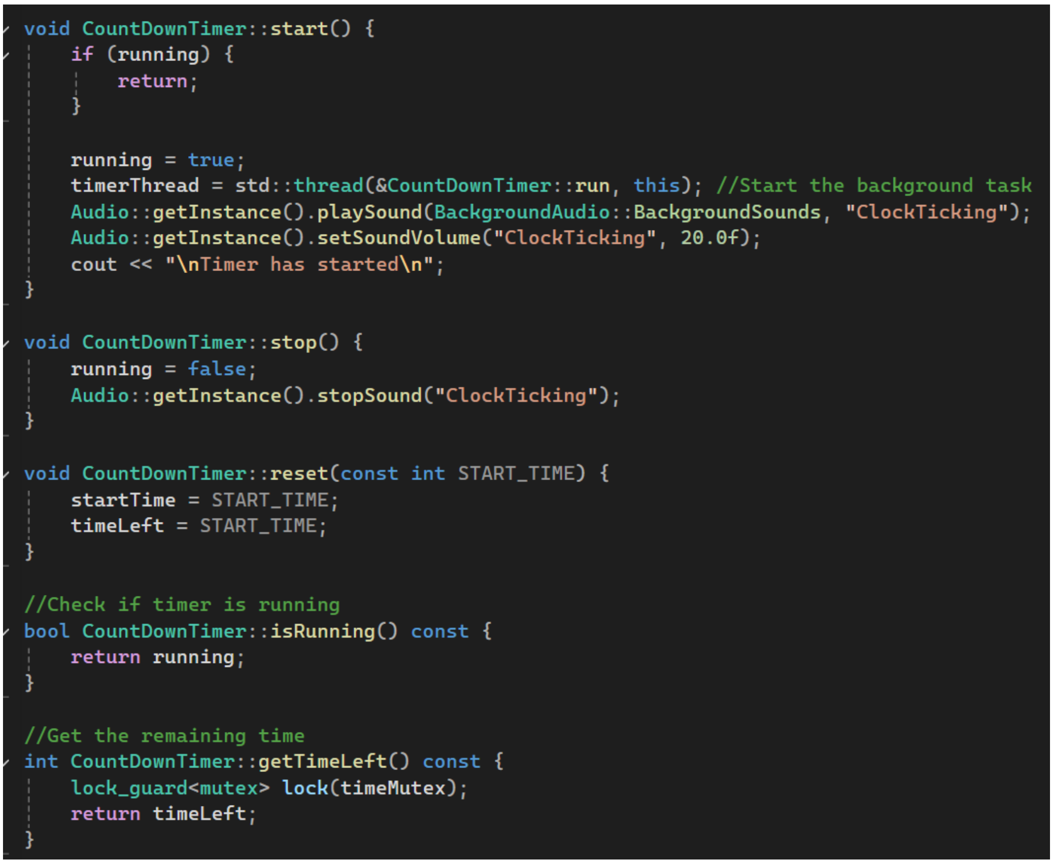
```cpp
#include "CountDownTimer.h"
#include <iostream>
#include <chrono>
#include <thread>
using namespace std;

void CountDownTimer::loadSounds()
{
    //loading sound for timer
    Audio::getInstance().loadSound(BackgroundAudio::BackgroundSounds);
}

CountDownTimer::CountDownTimer(int seconds, sf::Vector2f pos, string name) {
    startTime = seconds;
    timeLeft = seconds;
    this->pos = pos;
    this->name = name;
    running = false;
    textFont.loadFromFile("Assets/Fonts/roboto/Roboto-Regular.ttf");
    loadSounds();
}

CountDownTimer::CountDownTimer() {
    startTime = 0;
    timeLeft = 0;
    running = false;
    textFont.loadFromFile("Assets/Fonts/roboto/Roboto-Regular.ttf");
}

CountDownTimer::~CountDownTimer() {
    stop();
    if (timerThread.joinable()) {
        timerThread.join();
    }
}
```



```cpp
void CountDownTimer::run() {



    while (timeLeft > 0 && running) {
        {

            lock_guard<mutex> lock(timeMutex); //Will release lock when it goes out of scope
            cout << timeLeft << endl;
            --timeLeft;
        }
        this_thread::sleep_for(chrono::seconds(1));
    }

    {

        lock_guard<mutex> lock(timeMutex); //Will release lock when it goes out of scope

        if (running && timeLeft == 0) {
            cout << "Time's up!" << endl;
            Audio::getInstance().stopSound("ClockTicking");
        }
        else {
            cout << "Timer stopped early." << endl;
        }

        running = false;
    }
}
```



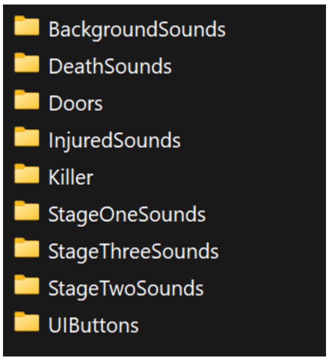
```
string CountDownTimer::setText()
{
    return name + " " + to_string(getTimeLeft() /60) + ":" + to_string(getTimeLeft() % 60);
}

void CountDownTimer::reduceTime(int change) {

    lock_guard<mutex> lock(timeMutex); //Will release lock when it goes out of scope

    if (change <= timeLeft) {
        timeLeft -= change;
    }
    else {
        timeLeft = 0;
    }

    cout << "reduced time by " + change << "\n-------------\n";

}

void CountDownTimer::render(sf::RenderWindow& window)
{
    display.setFont(textFont);
    display.setString(setText());
    display.setCharacterSize(30);
    display.setFillColor(sf::Color::White);
    display.setPosition(pos);

    window.draw(display);
}
```



```
void CountDownTimer::start() {
    if (running) {
        return;
    }

    running = true;
    timerThread = std::thread(&CountDownTimer::run, this); //Start the background task
    Audio::getInstance().playSound(BackgroundAudio::BackgroundSounds, "ClockTicking");
    Audio::getInstance().setSoundVolume("ClockTicking", 20.0f);
    cout << "\nTimer has started\n";
}

void CountDownTimer::stop() {
    running = false;
    Audio::getInstance().stopSound("ClockTicking");
}

void CountDownTimer::reset(const int START_TIME) {
    startTime = START_TIME;
    timeLeft = START_TIME;
}

//Check if timer is running
bool CountDownTimer::isRunning() const {
    return running;
}

//Get the remaining time
int CountDownTimer::getTimeLeft() const {
    lock_guard<mutex> lock(timeMutex);
    return timeLeft;
}
```

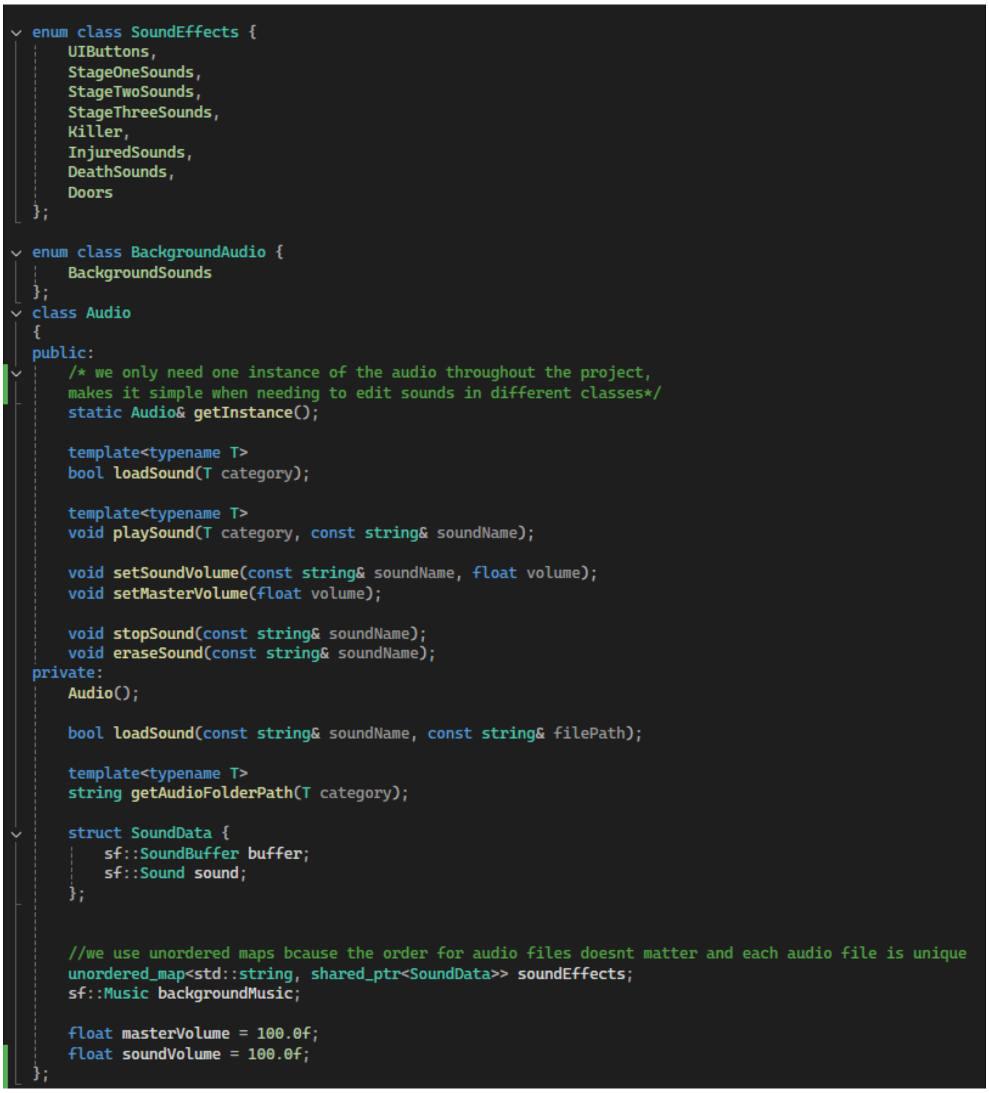
2. Audio

The game incorporates various audio elements, including background music, sound effects and interaction sounds. This enhances the player’s experience, making it more immersive and engaging.



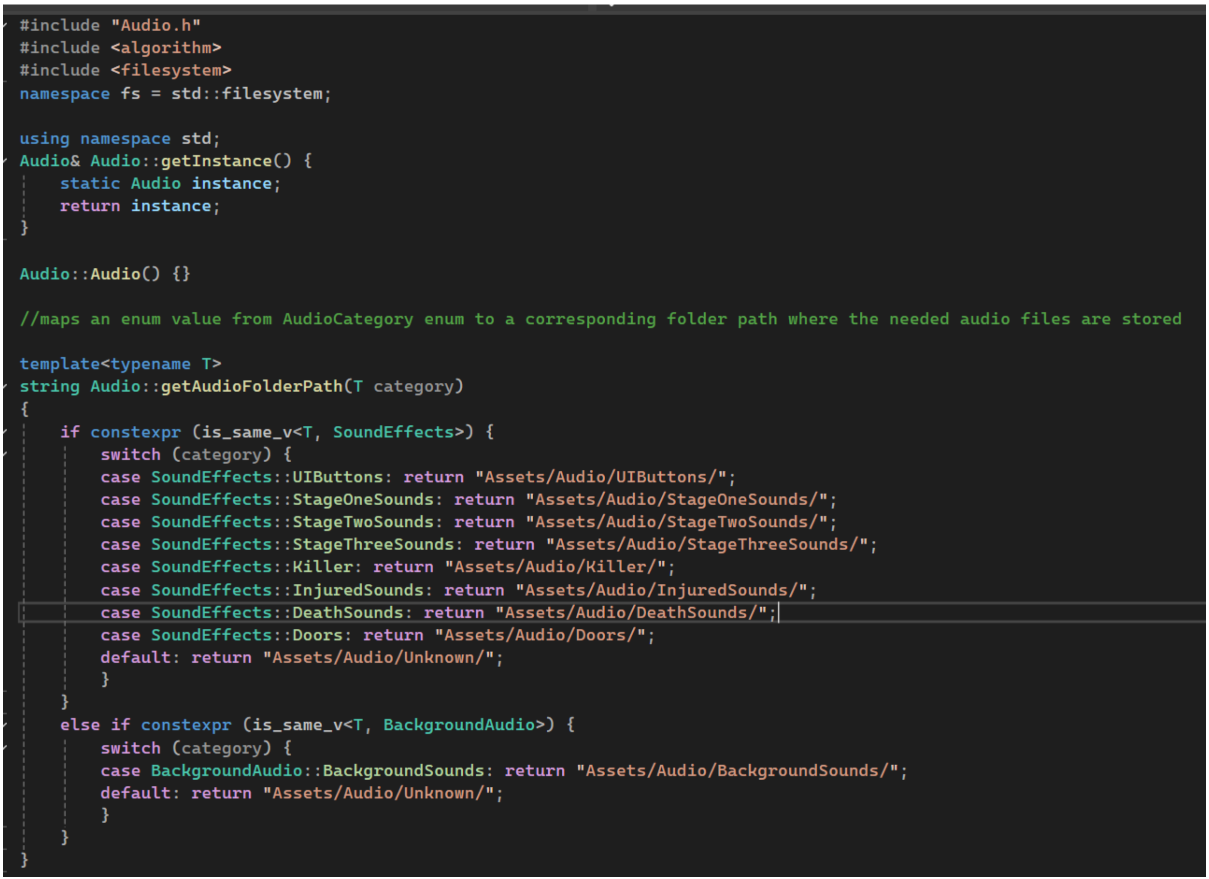
```
BackgroundSounds
DeathSounds
Doors
InjuredSounds
Killer
StageOneSounds
StageThreeSounds
StageTwoSounds
UIButtons
```

Audio.h

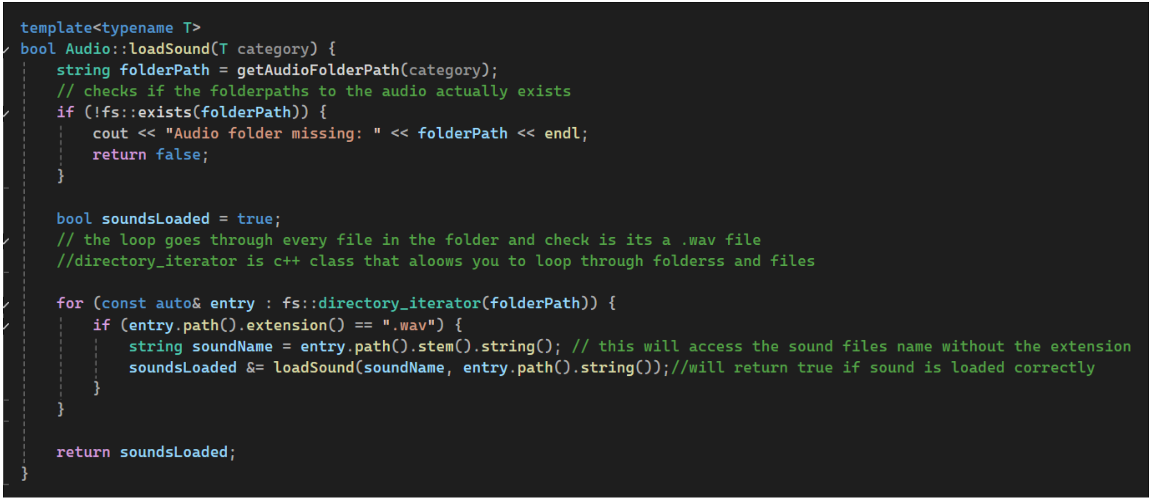


```
enum class SoundEffects {
    UIButtons,
    StageOneSounds,
    StageTwoSounds,
    StageThreeSounds,
    Killer,
    InjuredSounds,
    DeathSounds,
    Doors
};

enum class BackgroundAudio {
    BackgroundSounds
};
class Audio
{
public:
    /* we only need one instance of the audio throughout the project,
    makes it simple when needing to edit sounds in different classes*/
    static Audio& getInstance();

    template<typename T>
    bool loadSound(T category);

    template<typename T>
    void playSound(T category, const string& soundName);

    void setSoundVolume(const string& soundName, float volume);
    void setMasterVolume(float volume);

    void stopSound(const string& soundName);
    void eraseSound(const string& soundName);
private:
    Audio();

    bool loadSound(const string& soundName, const string& filePath);

    template<typename T>
    string getAudioFolderPath(T category);

    struct SoundData {
        sf::SoundBuffer buffer;
        sf::Sound sound;
    };


    //we use unordered maps bcause the order for audio files doesnt matter and each audio file is unique
    unordered_map<std::string, shared_ptr<SoundData>> soundEffects;
    sf::Music backgroundMusic;

    float masterVolume = 100.0f;
    float soundVolume = 100.0f;
};
```

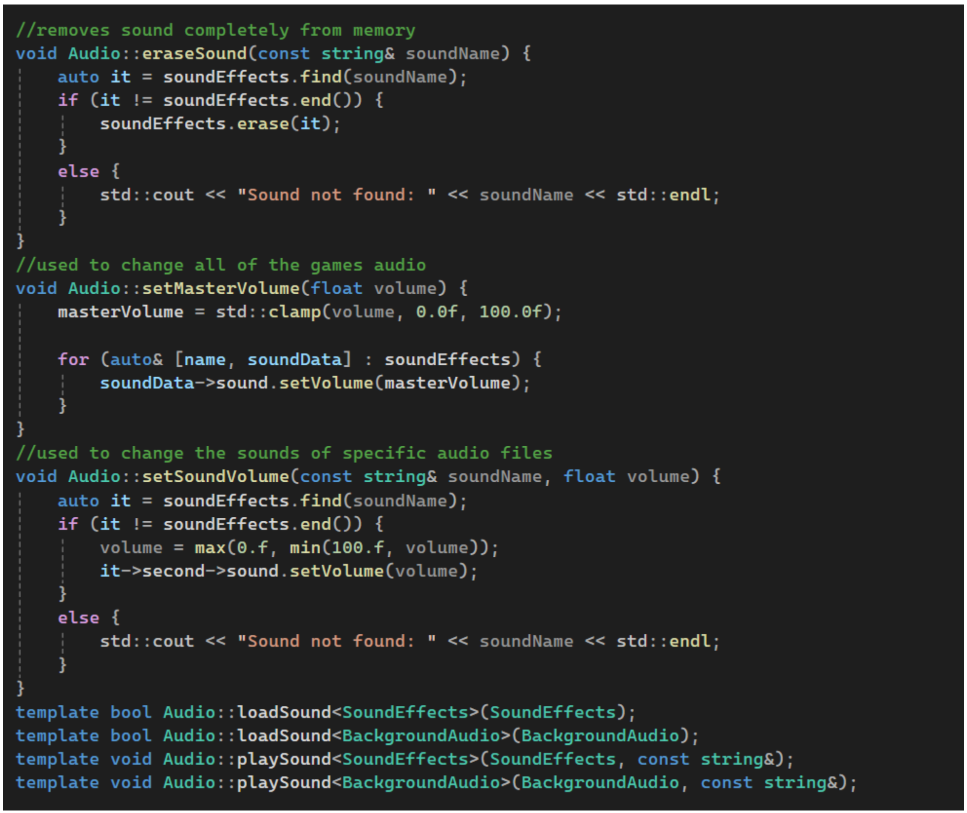
Audio.cpp



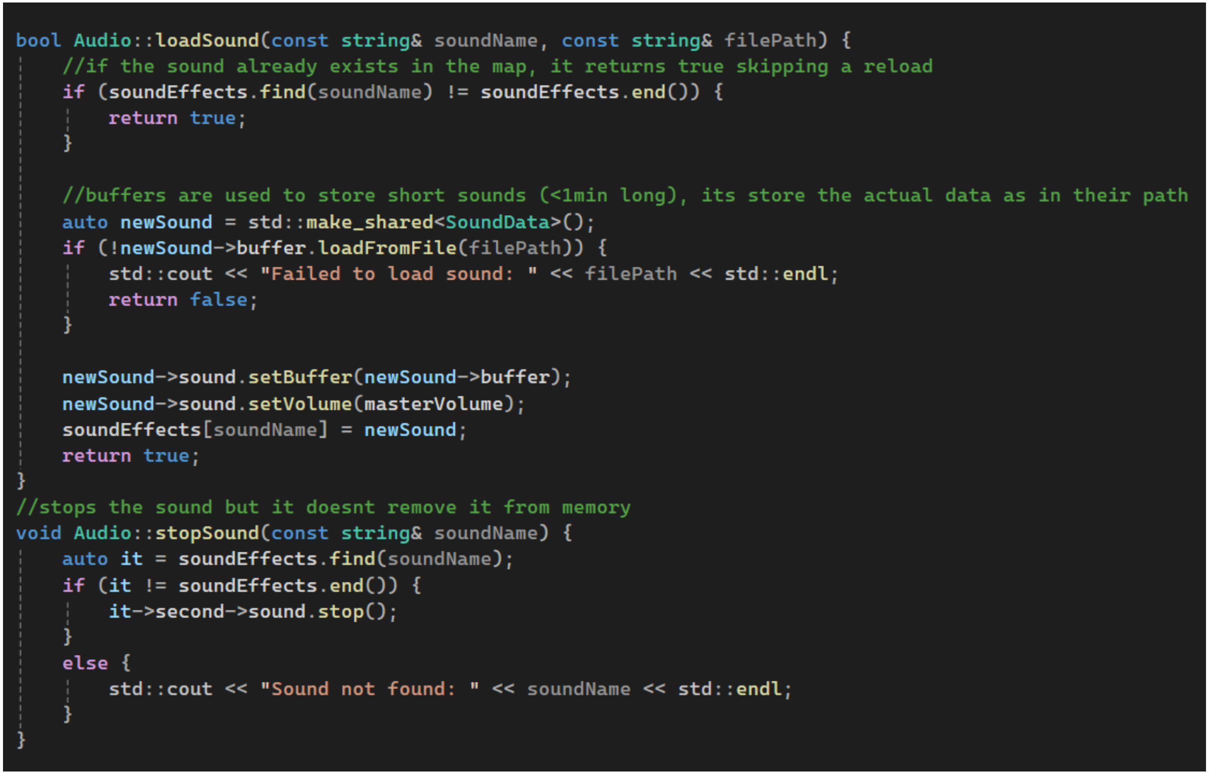
```cpp
#include "Audio.h"
#include <algorithm>
#include <filesystem>
namespace fs = std::filesystem;

using namespace std;
Audio& Audio::getInstance() {
    static Audio instance;
    return instance;
}

Audio::Audio() {}

//maps an enum value from AudioCategory enum to a corresponding folder path where the needed audio files are stored

template<typename T>
string Audio::getAudioFolderPath(T category)
{
    if constexpr (is_same_v<T, SoundEffects>) {
        switch (category) {
        case SoundEffects::UIButtons: return "Assets/Audio/UIButtons/";
        case SoundEffects::StageOneSounds: return "Assets/Audio/StageOneSounds/";
        case SoundEffects::StageTwoSounds: return "Assets/Audio/StageTwoSounds/";
        case SoundEffects::StageThreeSounds: return "Assets/Audio/StageThreeSounds/";
        case SoundEffects::Killer: return "Assets/Audio/Killer/";
        case SoundEffects::InjuredSounds: return "Assets/Audio/InjuredSounds/";
        case SoundEffects::DeathSounds: return "Assets/Audio/DeathSounds/";
        case SoundEffects::Doors: return "Assets/Audio/Doors/";
        default: return "Assets/Audio/Unknown/";
        }
    }
    else if constexpr (is_same_v<T, BackgroundAudio>) {
        switch (category) {
        case BackgroundAudio::BackgroundSounds: return "Assets/Audio/BackgroundSounds/";
        default: return "Assets/Audio/Unknown/";
        }
    }
}
```



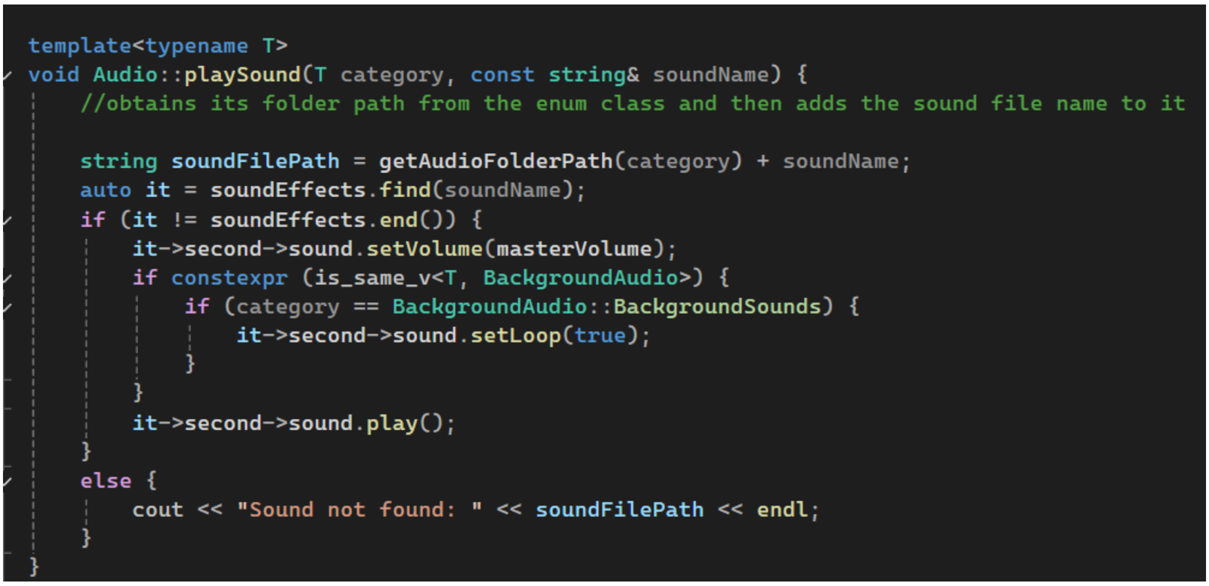
```cpp
template<typename T>
bool Audio::loadSound(T category) {
    string folderPath = getAudioFolderPath(category);
    // checks if the folderpaths to the audio actually exists
    if (!fs::exists(folderPath)) {
        cout << "Audio folder missing: " << folderPath << endl;
        return false;
    }

    bool soundsLoaded = true;
    // the loop goes through every file in the folder and check is its a .wav file
    //directory_iterator is c++ class that aloows you to loop through folderss and files

    for (const auto& entry : fs::directory_iterator(folderPath)) {
        if (entry.path().extension() == ".wav") {
            string soundName = entry.path().stem().string(); // this will access the sound files name without the extension
            soundsLoaded &= loadSound(soundName, entry.path().string());//will return true if sound is loaded correctly
        }
    }

    return soundsLoaded;
}
```



```cpp
//removes sound completely from memory
void Audio::eraseSound(const string& soundName) {
    auto it = soundEffects.find(soundName);
    if (it != soundEffects.end()) {
        soundEffects.erase(it);
    }
    else {
        std::cout << "Sound not found: " << soundName << std::endl;
    }
}
//used to change all of the games audio
void Audio::setMasterVolume(float volume) {
    masterVolume = std::clamp(volume, 0.0f, 100.0f);

    for (auto& [name, soundData] : soundEffects) {
        soundData->sound.setVolume(masterVolume);
    }
}
//used to change the sounds of specific audio files
void Audio::setSoundVolume(const string& soundName, float volume) {
    auto it = soundEffects.find(soundName);
    if (it != soundEffects.end()) {
        volume = max(0.f, min(100.f, volume));
        it->second->sound.setVolume(volume);
    }
    else {
        std::cout << "Sound not found: " << soundName << std::endl;
    }
}
template bool Audio::loadSound<SoundEffects>(SoundEffects);
template bool Audio::loadSound<BackgroundAudio>(BackgroundAudio);
template void Audio::playSound<SoundEffects>(SoundEffects, const string&);
template void Audio::playSound<BackgroundAudio>(BackgroundAudio, const string&);
```



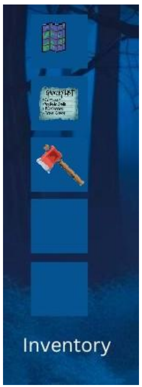
```cpp
bool Audio::loadSound(const string& soundName, const string& filePath) {
    //if the sound already exists in the map, it returns true skipping a reload
    if (soundEffects.find(soundName) != soundEffects.end()) {
        return true;
    }

    //buffers are used to store short sounds (<1min long), its store the actual data as in their path
    auto newSound = std::make_shared<SoundData>();
    if (!newSound->buffer.loadFromFile(filePath)) {
        std::cout << "Failed to load sound: " << filePath << std::endl;
        return false;
    }

    newSound->sound.setBuffer(newSound->buffer);
    newSound->sound.setVolume(masterVolume);
    soundEffects[soundName] = newSound;
    return true;
}
//stops the sound but it doesnt remove it from memory
void Audio::stopSound(const string& soundName) {
    auto it = soundEffects.find(soundName);
    if (it != soundEffects.end()) {
        it->second->sound.stop();
    }
    else {
        std::cout << "Sound not found: " << soundName << std::endl;
    }
}
```

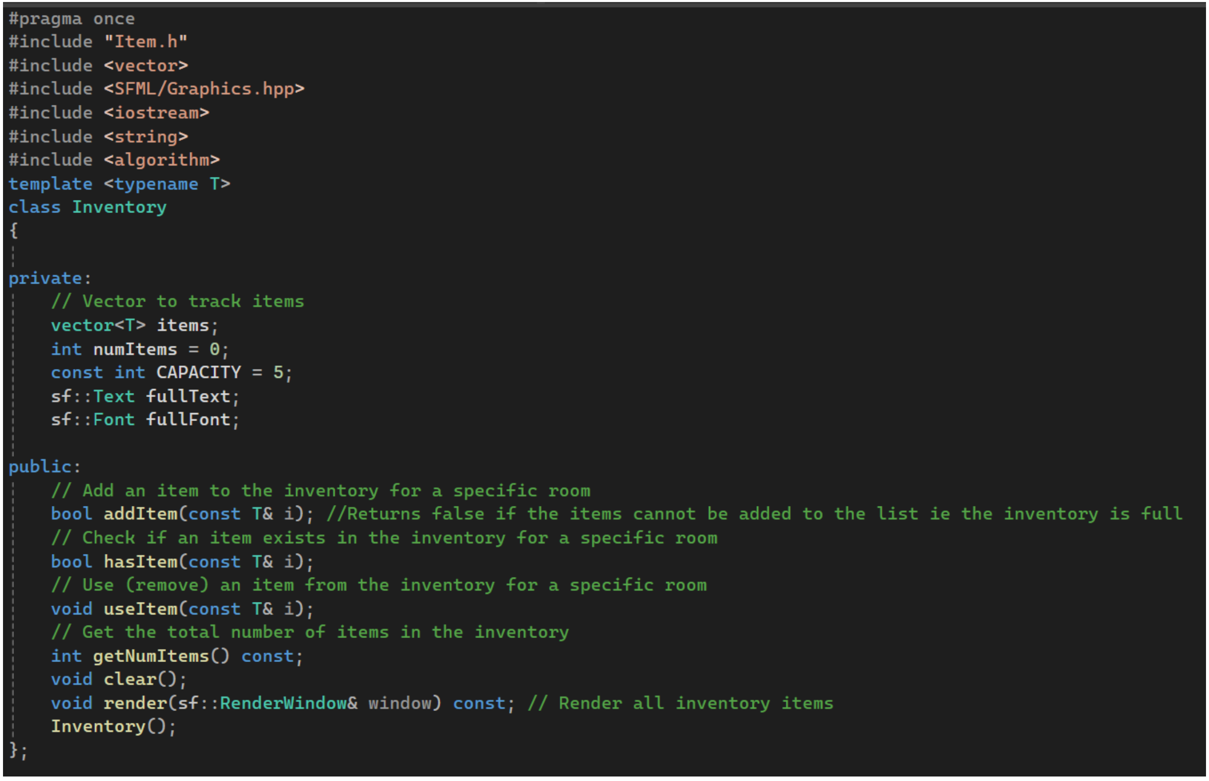


```
template<typename T>
void Audio::playSound(T category, const string& soundName) {
    //obtains its folder path from the enum class and then adds the sound file name to it

    string soundFilePath = getAudioFolderPath(category) + soundName;
    auto it = soundEffects.find(soundName);
    if (it != soundEffects.end()) {
        it->second->sound.setVolume(masterVolume);
        if constexpr (is_same_v<T, BackgroundAudio>) {
            if (category == BackgroundAudio::BackgroundSounds) {
                it->second->sound.setLoop(true);
            }
        }
        it->second->sound.play();
    }
    else {
        cout << "Sound not found: " << soundFilePath << endl;
    }
}
```

3. Inventory

An inventory system is used to track and manage the items the player collects throughout their escape attempt. The choices and success of the player's escape depends on the items they collect throughout the game.

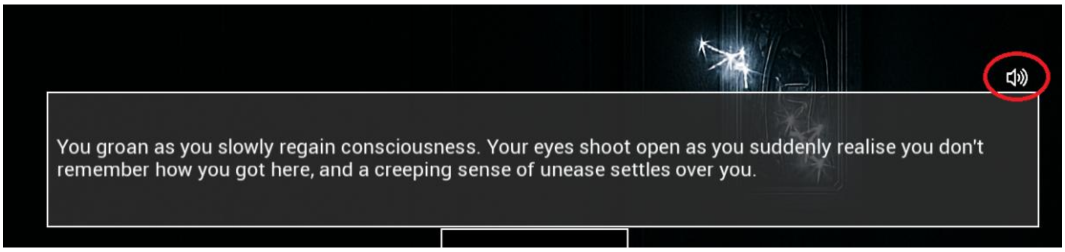




```
#pragma once
#include "Item.h"
#include <vector>
#include <SFML/Graphics.hpp>
#include <iostream>
#include <string>
#include <algorithm>
template <typename T>
class Inventory
{

private:
    // Vector to track items
    vector<T> items;
    int numItems = 0;
    const int CAPACITY = 5;
    sf::Text fullText;
    sf::Font fullFont;

public:
    // Add an item to the inventory for a specific room
    bool addItem(const T& i); //Returns false if the items cannot be added to the list ie the inventory is full
    // Check if an item exists in the inventory for a specific room
    bool hasItem(const T& i);
    // Use (remove) an item from the inventory for a specific room
    void useItem(const T& i);
    // Get the total number of items in the inventory
    int getNumItems() const;
    void clear();
    void render(sf::RenderWindow& window) const; // Render all inventory items
    Inventory();
};
```

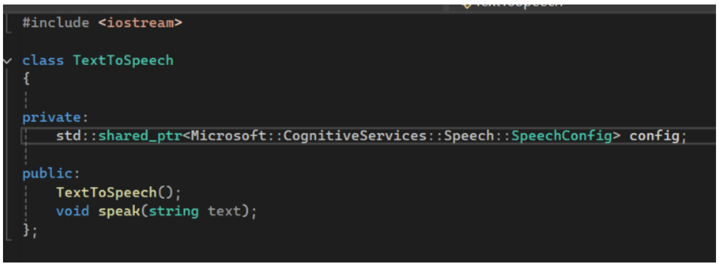
4. Text to speech

The game incorporates a text-to-speech accessibility feature using Microsoft Azure text to speech AI API. This allows dialogue and narration to be audibly spoken when the player selects the sound icon. It enhances the player's experience and improves accessibility for players with reading difficulties or visual impairments.



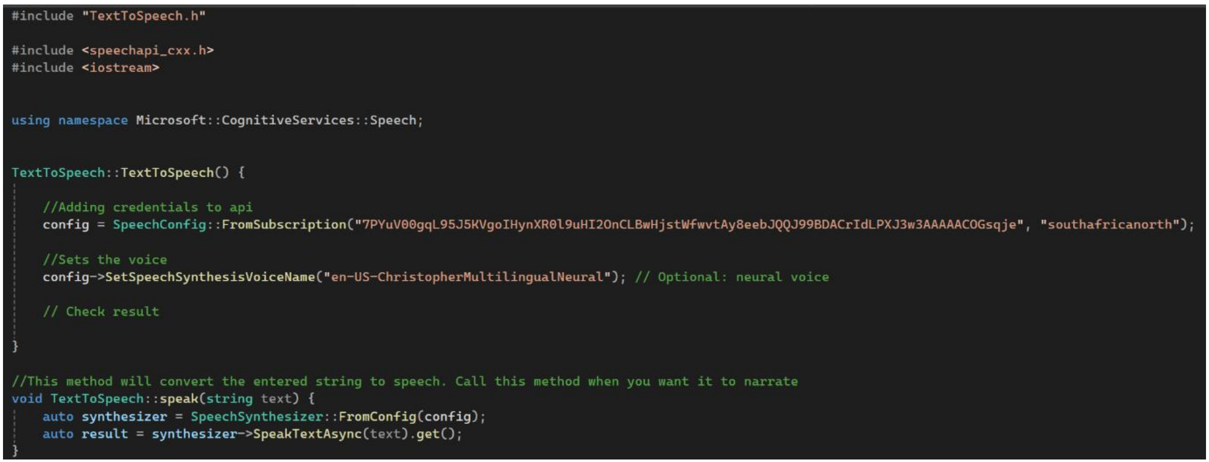
You groan as you slowly regain consciousness. Your eyes shoot open as you suddenly realise you don't remember how you got here, and a creeping sense of unease settles over you.

TextToSpeech.h

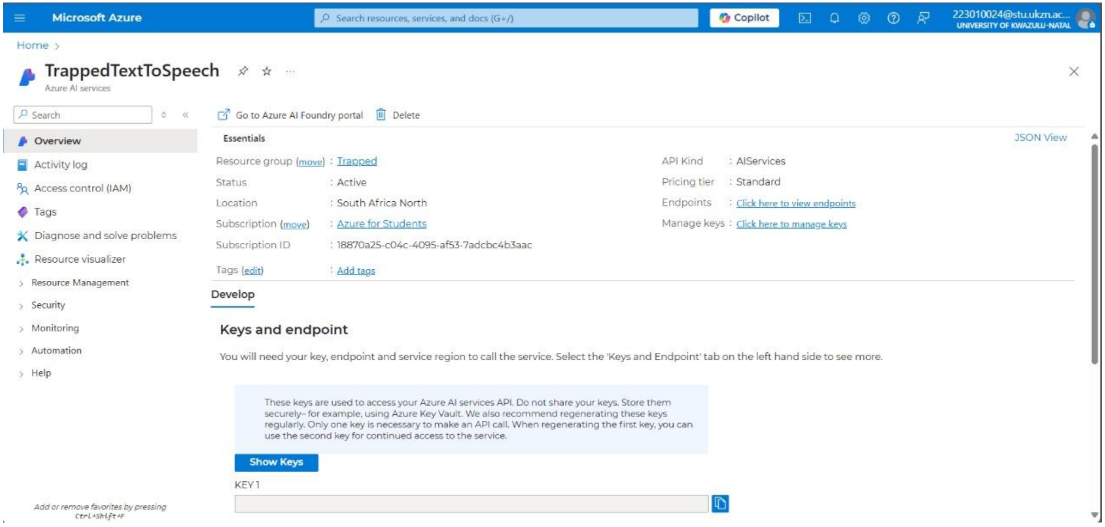


```cpp
#include <iostream>

class TextToSpeech
{

private:
    std::shared_ptr<Microsoft::CognitiveServices::Speech::SpeechConfig> config;

public:
    TextToSpeech();
    void speak(string text);
};
```

TextToSpeech.cpp



```cpp
#include "TextToSpeech.h"

#include <speechapi_cxx.h>
#include <iostream>


using namespace Microsoft::CognitiveServices::Speech;


TextToSpeech::TextToSpeech() {

    //Adding credentials to api
    config = SpeechConfig::FromSubscription("7PYuV00gqL95J5KVgoIHynXR0l9uHI2OnCLBwHjstWfwvtAy8eebJQQJ99BDACrIdLPXJ3w3AAAAACOGsqje", "southafricanorth");

    //Sets the voice
    config->SetSpeechSynthesisVoiceName("en-US-ChristopherMultilingualNeural"); // Optional: neural voice

    // Check result

}

//This method will convert the entered string to speech. Call this method when you want it to narrate
void TextToSpeech::speak(string text) {
    auto synthesizer = SpeechSynthesizer::FromConfig(config);
    auto result = synthesizer->SpeakTextAsync(text).get();
}
```



Resource group (move) : Trapped
Status : Active
Location : South Africa North
Subscription (move) : Azure for Students
Subscription ID : 18870a25-c04c-4095-af53-7adcbc4b3aac
Tags (edit) : Add tags

API Kind : AIServices
Pricing tier : Standard
Endpoints : Click here to view endpoints
Manage keys : Click here to manage keys

Keys and endpoint

You will need your key, endpoint and service region to call the service. Select the 'Keys and Endpoint' tab on the left hand side to see more.

These keys are used to access your Azure AI services API. Do not share your keys. Store them securely– for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.

KEY 1

References/Credits

Sourced from:

- Audio: https://freesound.org/ and https://www.zapsplat.com/
- Images: Generated using AI on https://www.canva.com/
- Item icons: outsourced to an independent freelance artist, Mr Tashiv Sooknandan
- Fonts: https://www.1001fonts.com/pixel-fonts.html
- API:
  - Documentation: https://learn.microsoft.com/en-us/azure/ai-services/speech-service/quickstarts/setup-platform?pivots=programming-language-cpp&tabs=windows%2Cubuntu%2Cdotnetcli%2Cjre%2Cmaven%2Cnodejs%2Cmac%2Cpypi
  - API: https://portal.azure.com/#@stuukznac.onmicrosoft.com/resource/subscriptions/18870a25-c04c-4095-af53-7adcbc4b3aac/resourceGroups/Trapped/providers/Microsoft.CognitiveServices/accounts/TrappedTextToSpeech/overview