

game.cpp

```

/*****
 *
 * game.cpp
 *
 * Stores all game objects
 * Controls the flow of the game
 *
 * Briano Goestiawan, 31482228
 *
 *****/

#include <iostream>
#include <sstream>
#include <algorithm>
#include "game.h"
#include "player.h"
#include "item.h"
#include "utils.h"
#define ROOM_ROWS 4
#define ROOM_COLS 3
#define ROOM_WIDTH 25 // Not including border
#define WINDOW_WIDTH 77 // Not including border
#define WINDOW_HEIGHT 41 // Not including border

using namespace std;
string readFile(string fileName);
string readInput(string prompt);

/***** PUBLIC METHODS *****/

// Constructor: initialize game variables and create game objects
Game::Game(string playerName, Difficulty difficulty) {
    // Create and allocate game objects
    this->createRooms();
    this->createItems();
    this->createSuspects();

    // Create player instance
    Room startingRoom = this->rooms[6];
    this->player = Player(playerName, &startingRoom);
    this->inventory = Room("Inventory", "");

    // Initialize variables
    this->view = VIEW_TOWER;
    this->difficulty = difficulty;
    this->moveCount = 0;
    this->searchCount = 0;
    this->questionCount = 0;
    this->gameOver = false;
    this->gameWin = false;
}

```

```

// Display help screen. Wait for user before continuing
void Game::showStoryLine() {
    clearScreen();
    cout << readFile("assets/story_line.txt");
    pause();
}

// Display help screen. Wait for user before continuing
void Game::showHelpScreen() {
    clearScreen();
    cout << readFile("assets/help_screen.txt");
    pause();
}

// Draw image to screen
void Game::displayView() {
    clearScreen();

    // Display different views depending on currently selected room
    switch (this->view) {
        case VIEW_TOWER:
            this->displayTower();
            break;
        case VIEW_ROOM:
            this->displayRoom();
            break;
        case VIEW_INVENTORY:
            this->displayInventory();
            break;
    }

    this->displayStatusBar();
}

// Validate and process user input
void Game::command() {
    // Anatomy of a command:
    //      command      argument (can be multiple words)
    //      \_____/      \_____/
    //
    // Read input from user
    string input = toLower(readInput("/> "));
    string command = "";
    string argument = "";
    int spaceIndex = input.find(' ');
    if (spaceIndex == string::npos) {
        command = input.substr(0, spaceIndex);
    } else {
        command = input.substr(0, spaceIndex);
        argument = input.substr(spaceIndex + 1);
    }

    // Help screen
    if (command == "help") this->showHelpScreen();
}

```

```

// Movement
else if (command == "left") this->move(DIR_LEFT);
else if (command == "right") this->move(DIR_RIGHT);
else if (command == "up") this->move(DIR_UP);
else if (command == "down") this->move(DIR_DOWN);

// View
else if (command == "tower") this->view = VIEW_TOWER;
else if (command == "room") this->view = VIEW_ROOM;
else if (command == "inv") this->view = VIEW_INVENTORY;
else if (command == "view") this->cycleView();

// Item Interactions
else if (command == "search") this->search();
else if (command == "pickup") this->pickup(argument);
else if (command == "drop") this->drop(argument);
else if (command == "examine") this->examine(argument);

// Supect Interactions
else if (command == "gather") this->gather();
else if (command == "question") this->question(argument);
else if (command == "accuse") this->accuse(argument);
else if (command == "stab") this->stab(argument);

// Utility commands
else if (command == "note") this->note(argument);
else if (command == "clear");
else if (command == "quit") this->confirmQuit();
else if (command == "easter") cout << "egg" << endl, pause();
else this->invalidCommand();
}

// Return true if player won else false
bool Game::getGameWin() {
    return this->gameWin;
}

// Return true if game is still running else false
bool Game::getGameOver() {
    return this->gameOver;
}

/***** PRIVATE METHODS *****/

// Generate room objects
void Game::createRooms() {
    // Create room instances and append to this->rooms
    this->rooms.push_back(Room("CONTROL CENTER", "assets/room_control_center.txt"));
    this->rooms.push_back(Room("OFFICE", "assets/room_office.txt"));
    this->rooms.push_back(Room("SPA", "assets/room_spa.txt"));
    this->rooms.push_back(Room("LABORATORY", "assets/room_laboratory.txt"));
    this->rooms.push_back(Room("LIBRARY", "assets/room_library.txt"));
    this->rooms.push_back(Room("GIFT SHOP", "assets/room_gift_shop.txt"));
}

```

```

this->rooms.push_back(Room("CAFETERIA", "assets/room_cafeteria.txt"));
this->rooms.push_back(Room("LOBBY", "assets/room_lobby.txt"));
this->rooms.push_back(Room("TOILET", "assets/room_toilet.txt"));
this->rooms.push_back(Room("SERVER ROOM", "assets/room_server_room.txt"));
this->rooms.push_back(Room("CAR PARK", "assets/room_car_park.txt"));
this->rooms.push_back(Room("PLUMBING ROOM", "assets/room_plumbing_room.txt"));

// Set one of the room to be the murder room
this->rooms[rand() % this->rooms.size()].setMurderRoom();

// Set room neighbours. for each room in rooms, set its left, right, up and
// down neighbouring room if possible (not wall)
for (int row = 0; row < ROOM_ROWS; row++) {
    for (int col = 0; col < ROOM_COLS; col++) {
        int index = row * ROOM_COLS + col;

        // Set left
        if (col > 0) {
            this->rooms[index].setNeighbour(DIR_LEFT, &this->rooms[index - 1]);
        }

        // Set right
        if (col < ROOM_COLS - 1) {
            this->rooms[index].setNeighbour(DIR_RIGHT, &this->rooms[index + 1]);
        }

        // Set up
        if (row > 0) {
            this->rooms[index].setNeighbour(DIR_UP, &this->rooms[index - ROOM_COLS]);
        }

        // Set down
        if (row < ROOM_ROWS - 1) {
            this->rooms[index].setNeighbour(DIR_DOWN, &this->rooms[index + ROOM_COLS]);
        }
    }
}

// Generate items objects
void Game::createItems() {
    // Create items
    this->items.push_back(Item("Knife", "assets/item_knife.txt"));
    this->items.push_back(Item("Fork", "assets/item_fork.txt"));
    this->items.push_back(Item("Stick", "assets/item_stick.txt"));
    this->items.push_back(Item("Scissors", "assets/item_scissors.txt"));
    this->items.push_back(Item("Bowling ball", "assets/item_bowling_ball.txt"));
    this->items.push_back(Item("Screwdriver", "assets/item_screwdriver.txt"));
    this->items.push_back(Item("Chair", "assets/item_chair.txt"));
    this->items.push_back(Item("Vase", "assets/item_vase.txt"));

    // Set one of the item to be the murder weapon
    this->items[rand() % this->items.size()].setMurderWeapon();
}

```

```

    // Put all item in items to a randomly selected room
    for (int i = 0; i < this->items.size(); i++) {
        this->items[i].setLocation(this->getRandomRoom());
    }
}

// Generate suspects objects
void Game::createSuspects() {
    // Create suspects
    this->suspects.push_back(Suspect("Anna"));
    this->suspects.push_back(Suspect("Bob"));
    this->suspects.push_back(Suspect("Charlie"));
    this->suspects.push_back(Suspect("Daniel"));
    this->suspects.push_back(Suspect("Emma"));
    this->suspects.push_back(Suspect("Felix"));
    this->suspects.push_back(Suspect("George"));

    // Put all suspects to a randomly selected room
    for (int i = 0; i < this->suspects.size(); i++) {
        this->suspects[i].setRoom(this->getRandomRoom());
    }

    // Create a vector of pointer to all suspects
    // Later use this to assign suspects as different types
    vector<Suspect*> suspectCollection;
    for (int i = 0; i < this->suspects.size(); i++) {
        suspectCollection.push_back(&this->suspects[i]);
    }

    // Shuffle suspect collection so suspect are assign different roles randomly
    for (int i = 0; i < suspectCollection.size(); i++) {
        Suspect *temp = suspectCollection[i];
        int randomIndex = rand() % suspectCollection.size();
        suspectCollection[i] = suspectCollection[randomIndex];
        suspectCollection[randomIndex] = temp;
    }

    // Temporary suspect pointer for assigning suspect types
    Suspect *tempSuspect;

    // Assign victim
    tempSuspect = (suspectCollection.back());
    suspectCollection.pop_back();
    tempSuspect->setType(SUS_VICTIM);

    // Assign killer
    tempSuspect = suspectCollection.back();
    suspectCollection.pop_back();
    tempSuspect->setType(SUS_KILLER);
    tempSuspect->setAlibi(suspectCollection[rand() % suspectCollection.size()]);

    // Set alibi pairs except for the last in the list
    while (suspectCollection.size() > 1) {
        Suspect *suspectA = suspectCollection.back();

```

```

        suspectCollection.pop_back();
        Suspect *suspectB = suspectCollection.back();
        suspectCollection.pop_back();
        suspectA->setAlibi(suspectB);
        suspectB->setAlibi(suspectA);
    }
}

// Cycle through all the different views
void Game::cycleView() {
    switch (this->view) {
        case VIEW_TOWER:
            this->view = VIEW_ROOM;
            break;
        case VIEW_ROOM:
            this->view = VIEW_INVENTORY;
            break;
        case VIEW_INVENTORY:
            this->view = VIEW_TOWER;
            break;
    }
}

// Display the tower including the player character where it is located in the tower
void Game::displayTower() {
    // Print tower roof
    cout << " +-----+ " << endl;
    cout << " / \ " << endl;
    cout << " /          B R U M P   T O W E R          \ " << endl;
    cout << "| | " << endl;
    cout << "+-----+" << endl;

    // For each floor in the tower
    for (int row = 0; row < ROOM_ROWS; row++) {

        // Status bar on top of each room
        cout << '|';
        for (int col = 0; col < ROOM_COLS; col++) {
            // Display a * for each item in room
            Room *currentRoom = &this->rooms[row * ROOM_COLS + col];
            string itemString = "";
            if (currentRoom->getItemHidden()) {
                itemString = "? ";
            } else {
                for (int i = 0; i < this->items.size(); i++) {
                    if (items[i].getLocation() == currentRoom) {
                        itemString += "* ";
                    }
                }
            }
        }
    }

    // Display the first letter of each suspect in the room
    string suspectString = "";
    for (int i = 0; i < this->suspects.size(); i++) {

```

```

        if (this->suspects[i].getRoom() == currentRoom) {
            suspectString += ' ' + this->suspects[i].getName().substr(0, 1);
        }
    }

    // Example display format: | A B C D E F G * * * * * |
    int blankCount = ROOM_WIDTH - itemString.length() - suspectString.length();
    cout << fixedWidth(suspectString + fixedWidth("", ' ', blankCount) + itemString, ' ', ROOM_WIDTH);
    cout << '
';

    // Display the main section of the room (where the character might be)
    for (int i = 0; i < 6; i++) {
        cout << '|';
        for (int col = 0; col < ROOM_COLS; col++) {
            string content = "";

            Room *currentRoom = &this->rooms[row * ROOM_COLS + col];
            if (currentRoom == this->player.getRoom()) {
                content = this->player.getImage()[i];
            }

            cout << fixedWidth(content, ' ', ROOM_WIDTH) << '|';
        }
        cout << '
';
    }

    // Print names of rooms in the current floor
    cout << '|';
    for (int col = 0; col < ROOM_COLS; col++) {
        string roomName = this->rooms[row * ROOM_COLS + col].getName();
        cout << fixedWidth(" " + roomName, ' ', ROOM_WIDTH) << '|';
    }
    cout << '
';

    cout << "+-----+" << endl;
}

// Display the room where the player is in
void Game::displayRoom() {
    Room *currentRoom = this->player.getRoom();

    // Display the current room's image and description
    cout << currentRoom->getImage();
    cout << "
|" << endl;

    // Display each item that exists in the current room
    string itemsString = " ITEMS: ";
    if (currentRoom->getItemHidden()) {
        itemsString += '?';
    }
}

```

```

    } else {
        vector<Item *> items;
        for (int i = 0; i < this->items.size(); i++) {
            if (this->items[i].getLocation() == currentRoom) {
                items.push_back(&this->items[i]);
            }
        }
        if (items.size() > 0) {
            itemsString += items[0]->getName();
            for (int i = 1; i < items.size(); i++) {
                itemsString += ", " + items[i]->getName();
            }
        } else {
            itemsString += '-';
        }
    }
    cout << '|' << fixedWidth(itemsString, ' ', WINDOW_WIDTH) << '|' << endl;
    cout << " |" << endl;

    // Display all suspect in the current room
    string suspectsString = "  SUSPECTS: ";
    vector<Suspect *> suspects;
    for (int i = 0; i < this->suspects.size(); i++) {
        if (this->suspects[i].getRoom() == currentRoom) {
            suspects.push_back(&this->suspects[i]);
        }
    }
    if (suspects.size() > 0) {
        suspectsString += suspects[0]->getName();
        for (int i = 1; i < suspects.size(); i++) {
            suspectsString += ", " + suspects[i]->getName();
        }
    } else {
        suspectsString += '-';
    }
    cout << '|' << fixedWidth(suspectsString, ' ', WINDOW_WIDTH) << '|' << endl;
    cout << " |" << endl;

    cout << "+-----+" << endl;
}

// Display player's inventory full screen
void Game::displayInventory() {
    cout << "+-----+" << endl;

    // Print title
    cout << '|' << fixedWidth("", ' ', WINDOW_WIDTH) << '|' << endl;
    cout << '|' << fixedWidth("  INVENTORY", ' ', WINDOW_WIDTH) << '|' << endl;

    // Print items in player's inventory
    for (int i = 0; i < this->getInventory().size(); i++) {
        cout << '|' << fixedWidth("", ' ', WINDOW_WIDTH) << '|' << endl;
        cout << '|' << fixedWidth("  - " + this->getInventory()[i]->getName(), ' ', WINDOW_WIDTH) << '|'
    }
}

```



```

    // Print blank lines
    for (int i = 0; i < WINDOW_HEIGHT - this->getInventory().size() * 2 - 2; i++) {
        if (i == 1 && this->getInventory().size() == 0) {
            // Print empty. if nothing in inventory
            cout << '|' << fixedWidth(" Empty.", ' ', WINDOW_WIDTH) << '|' << endl;
        } else {
            cout << '|' << fixedWidth("", ' ', WINDOW_WIDTH) << '|' << endl;
        }
    }

    cout << "+-----+" << endl;
}

// Get confirmation from user if they want to quit
void Game::confirmQuit() {
    string answer = readInput("Are you sure? (Y/n) ");
    if (answer == "y" || answer == "Y") {
        this->gameOver = true;
    }
}

// If the user puts an invalid command suggest them to read the help screen
void Game::invalidCommand() {
    cout << "Get some '/*> help" << endl;
    pause();
}

// Display a response from the suspect named suspectName
void Game::question(string suspectName) {
    // If questioned more than 3 times in nightmare mode, end the game
    if (this->difficulty == DIFF_NIGHTMARE && this->questionCount >= 3) {
        cout << "Too many questions" << endl;
        pause();
        this->gameOver = true;
        return;
    }

    Suspect *suspect = this->searchSuspect(suspectName);

    // Let user know if they misspelled suspectName
    if (suspect == NULL) {
        cout << "Suspect named '" << suspectName << "' not found" << endl;
        pause();
        return;
    }

    // Let user know suspect not in room
    if (suspect->getRoom() != this->player.getRoom()) {
        cout << suspectName << "not in this room" << endl;
        pause();
        return;
    }
}

```

```

        // Let suspect talk and pause so user can read message
        suspect->talk(this->player.getName());
        this->questionCount++;
        pause();
    }

    // Move all suspect to the room where the player is in
    void Game::gather() {
        // Block this command if in Hard or Nightmare difficulty
        if (this->difficulty == DIFF_HARD || this->difficulty == DIFF_NIGHTMARE) {
            cout << "gather command not enabled in this difficulty" << endl;
            pause();
            return;
        }

        // Move all suspect to player's current room
        for (int i = 0; i < this->suspects.size(); i++) {
            if (&this->suspects[i] != this->getVictim()) {
                this->suspects[i].setRoom(this->player.getRoom());
            }
        }
    }

    // Return a pointer to a random room
    Room *Game::getRandomRoom() {
        return &this->rooms[rand() % this->rooms.size()];
    }

    // Return a pointer to the room named roomName and return NULL if not found
    Room *Game::searchRoom(std::string roomName) {
        for (int i = 0; i < this->rooms.size(); i++) {
            if (toLower(this->rooms[i].getName()) == toLower(roomName)) {
                return &this->rooms[i];
            }
        }
        return NULL;
    }

    // Return a pointer to the suspect named suspectName and return NULL if not found
    Suspect *Game::searchSuspect(string suspectName) {
        for (int i = 0; i < this->suspects.size(); i++) {
            if (toLower(this->suspects[i].getName()) == toLower(suspectName)) {
                return &this->suspects[i];
            }
        }
        return NULL;
    }

    // Return a pointer to an item named itemName and return NULL if not found
    Item *Game::searchItem(string itemName) {
        for (int i = 0; i < this->items.size(); i++) {
            if (toLower(this->items[i].getName()) == toLower(itemName)) {
                return &this->items[i];
            }
        }
    }

```

```

    }
    return NULL;
}

// Search for item in room add to inventory if found
void Game::pickup(string itemName) {
    Item *item = this->searchItem(itemName);

    // Let user know if item not found
    if (item == NULL || item->getLocation() != this->player.getRoom()) {
        cout << "item named '" << itemName << "' not found" << endl;
        pause();
        return;
    }

    // Point item location to player's inventory
    item->setLocation(&this->inventory);

    // If easy mode, picking up all items in the tower results in a win
    if (this->difficulty == DIFF_EASY && this->getInventory().size() == this->items.size()) {
        this->gameWin = true;
        this->gameOver = true;
    }
}

// Drop item named itemName to player's room if found in inventory
void Game::drop(string itemName) {
    Item *item = this->searchItem(itemName);

    if (item && item->getLocation() == &this->inventory) {
        item->setLocation(this->player.getRoom());
    }
}

// Display item named itemName in player's inventory if exists
void Game::examine(string itemName) {
    // calling //> examine with no argument implies examine all item
    if (itemName == "") {
        this->examineAll();
    }

    Item *item = this->searchItem(itemName);

    // Let user know if item not found
    if (item == NULL || item->getLocation() != &this->inventory) {
        cout << "Item named '" << itemName << "' not found" << endl;
        return;
    }

    this->displayItem(item);
}

// Examine all item in player's inventory
void Game::examineAll() {

```

```

    for (int i = 0; i < this->getInventory().size(); i++) {
        displayItem(this->getInventory()[i]);
    }
}

// Display an item in fullscreen
void Game::displayItem(Item *item) {
    clearScreen();
    cout << item->getImage();
    cout << "|" << endl;
    cout << '|' << fixedWidth("  ITEM NAME: " + item->getName(), ' ', WINDOW_WIDTH) << '|' << endl;
    cout << "|" << endl;
    cout << '|' << fixedWidth("  LOCATION: " + item->getLocation()->getName(), ' ', WINDOW_WIDTH) << '|' << endl;
    cout << "|" << endl;
    cout << "+-----+" << endl;
    pause();
}

// Accuse the killer, result in ending the game as won or lose
void Game::accuse(string suspectName) {
    if (this->difficulty == DIFF_NIGHTMARE) {
        cout << "Sorry, accusing is not an option in nightmare mode" << endl;
        return;
    }

    Suspect *suspect = this->searchSuspect(suspectName);

    // Let player know if suspect not in same room
    if (suspect == NULL || suspect->getRoom() != player.getRoom()) {
        cout << "suspect named '" << suspectName << "' not found in room" << endl;
        pause();
        return;
    }

    // Get room name from user and check if room exists
    string roomName = readInput("Where did the murder took place? ");
    Room *room = this->searchRoom(roomName);
    if (room == NULL) {
        cout << "room named '" << roomName << "' not found in tower" << endl;
        pause();
        return;
    }

    // Get item name from user and check if item exists
    string itemName = readInput("What weapon did the murderer used? ");
    Item *item = searchItem(itemName);
    if (item == NULL) {
        cout << "item named '" << itemName << "' not found in tower" << endl;
        pause();
        return;
    }

    // Default to win, but will be set to false if any of the following
    // requirments is not met

```

```

    this->gameWin = true;

    // Check if player accuse the killer
    if (suspect != this->getKiller()) {
        cout << "Accused the wrong suspect. killer was " << this->getKiller()->getName() << endl;
        this->gameWin = false;
    }

    // Check if player told the right murder room
    if (room != this->getMurderRoom()) {
        cout << "Told wrong room. murder room was " << this->getMurderRoom()->getName() << endl;
        this->gameWin = false;
    }

    // Check if player told the right murder weapon
    if (item != this->getMurderWeapon()) {
        cout << "Told wrong item. murder weapon was " << this->getMurderWeapon()->getName() << endl;
        this->gameWin = false;
    }

    // Pause so player can read their mistakes
    if (!this->gameWin) {
        pause();
    }

    // Signal to end the game
    this->gameOver = true;
}

// Return a list of item pointer of items in inventory
vector<Item *> Game::getInventory() {
    vector<Item *> inventoryItems;

    for (int i = 0; i < this->items.size(); i++) {
        // Only add to vector if item location is player's inventory
        if (items[i].getLocation() == &this->inventory) {
            inventoryItems.push_back(&items[i]);
        }
    }

    return inventoryItems;
}

// Move to room besides current player room if it exists
void Game::move(Direction direction) {
    // If move more than 20 times in hard mode, end the game
    if (this->difficulty == DIFF_HARD && this->moveCount >= 20) {
        cout << "Too much movement. leg break" << endl;
        pause();
        this->gameOver = true;
        return;
    }

    // Move the player

```

```

Room *destination = this->player.getRoom()->getNeighbour(direction);
if (destination != NULL) {
    this->player.setRoom(destination);
    this->moveCount++;
}

// Randomly move the suspects (except for the victim)
for (int i = 0; i < this->suspects.size(); i++) {
    if (&this->suspects[i] != this->getVictim() && rand() % 2 < 1) {
        Direction randomDirection = static_cast<Direction>(rand() % 4);
        this->suspects[i].move(randomDirection);
    }
}

// Search the current room the player is in
void Game::search() {
    this->player.getRoom()->search();
    this->searchCount++;
}

// Stab a suspect for the nightmare mode
void Game::stab(string suspectName) {
    // Check if game is in nightmare difficulty and exit function if not
    if (this->difficulty != DIFF_NIGHTMARE) {
        cout << "Stabbing only in nightmare mode." << endl;
        pause();
        return;
    }

    // Check if player have the knife
    Item *foundKnife = this->searchItem("knife");
    if (foundKnife->getLocation() != &this->inventory) {
        cout << "You don't have a knife..." << endl;
        pause();
        return;
    }

    // Find suspect based on user input
    Suspect *suspect = this->searchSuspect(suspectName);

    // Let the user know they misspelled the suspectName
    if (suspect == NULL) {
        cout << "Suspect named '" << suspectName << "' not found in this room" << endl;
        pause();
        return;
    }

    // Let the user know that suspect is not in the room
    if (suspect->getRoom() != this->player.getRoom()) {
        cout << suspectName << "is not in this room" << endl;
        pause();
        return;
    }
}

```

```

    // Let the user know they killed an innocent person or win the game
    if (suspect == this->getKiller()) {
        this->gameWin = true;
    } else {
        cout << "Congratulation, you stabbed an innocent person.." << endl;
        pause();
    }

    // Susccessfully killed the killer
    this->gameOver = true;
}

// Display notes if content is empty else add content to notes
void Game::note(string content) {
    if (content != "") {
        // Append content to notes
        this->notes.push_back(content);
    } else {
        // Display notes
        clearScreen();
        cout << "+-----+" << endl;

        // Print each line in notes
        for (int i = 0; i < this->notes.size(); i++) {
            cout << "| " << fixedWidth(this->notes[i], ' ', WINDOW_WIDTH - 4) << " |" << endl;
        }

        // Print blank lines to fill the screen
        int blankCount = WINDOW_HEIGHT - 2 * this->notes.size();
        for (int i = 0; i < blankCount; i++) {
            cout << "| " << endl;
        }

        cout << "+-----+" << endl;
        pause();
    }
}

// Return a pointer to the killer suspect
Suspect *Game::getKiller() {
    for (int i = 0; i < this->suspects.size(); i++) {
        if (this->suspects[i].getType() == SUS_KILLER) {
            return &suspects[i];
        }
    }
    return &this->suspects[0];
}

// Return a pointer to the victim suspect
Suspect *Game::getVictim() {
    for (int i = 0; i < this->suspects.size(); i++) {
        if (this->suspects[i].getType() == SUS_VICTIM) {

```

```

        return &suspects[i];
    }
}
return &this->suspects[0];
}

// Display the status info bar
void Game::displayStatusBar() {
    // Left column: show move count
    string leftColumn = " MOVE: ";
    leftColumn += to_string(this->moveCount);
    leftColumn = fixedWidth(leftColumn, ' ', ROOM_WIDTH);

    // Mid column: show player name
    string midColumn = " SEARCH: ";
    midColumn += to_string(this->searchCount);
    midColumn = fixedWidth(midColumn, ' ', ROOM_WIDTH);

    // Right column: show question count
    string rightColumn = " QUESTIONS: ";
    rightColumn += to_string(this->questionCount);
    rightColumn = fixedWidth(rightColumn, ' ', ROOM_WIDTH);

    cout << '|' << leftColumn << '|' << midColumn << '|' << rightColumn << '|' << endl;
    cout << "+-----+" << endl;
}

// Return a pointer to the murder room
Room *Game::getMurderRoom() {
    for (int i = 0; i < this->rooms.size(); i++) {
        if (this->rooms[i].getMurderRoom()) {
            return &this->rooms[i];
        }
    }
    return &this->rooms[0];
}

// Return a pointer to the murder room
Item *Game::getMurderWeapon() {
    for (int i = 0; i < this->items.size(); i++) {
        if (this->items[i].getMurderWeapon()) {
            return &this->items[i];
        }
    }
    return &this->items[0];
}

```

game.h

```

/*****
 *
 * game.h
 *
 */

```



```

* Stores all game objects
* Controls the flow of the game
*
* Briano Goestiawan, 31482228
*
*****/

#pragma once
#include "player.h"
#include "suspect.h"
#include "room.h"
#include "item.h"
#include <string>
#include <vector>

enum View {
    VIEW_TOWER,
    VIEW_ROOM,
    VIEW_INVENTORY
};

enum Difficulty {
    DIFF_EASY,      // Pickup all items as fast as possible. No time limit
    DIFF_MEDIUM,    // Accuse killer, murder room and weapon
    DIFF_HARD,      // Medium requirments but only have 10 moves
    DIFF_NIGHTMARE // 3 room search, 3 suspect questions, stab the killer
};

class Game {
public:
    Game(std::string playerName, Difficulty difficulty);
    void showStoryLine();
    void showHelpScreen();
    void displayView();
    void command();
    bool getGameWin();
    bool getGameOver();

private:
    // Game objects
    std::vector<Room> rooms;
    std::vector<Item> items;
    std::vector<Suspect> suspects;
    std::vector<std::string> notes;
    Player player;
    Room inventory;

    // Private variables
    View view;
    Difficulty difficulty;
    int moveCount;
    int searchCount;
    int questionCount;
    bool gameOver;
};

```

```

    bool gameWin;

    // Helper functions to get object pointers
    std::vector<Item *> getInventory();
    Room *getRandomRoom();
    Room *searchRoom(std::string roomName);
    Suspect *searchSuspect(std::string suspectName);
    Item *searchItem(std::string itemName);
    Suspect *getVictim();
    Suspect *getKiller();
    Room *getMurderRoom();
    Item *getMurderWeapon();

    // Create game objects
    void createRooms();
    void createItems();
    void createSuspects();

    // Display methods
    void displayTower();
    void displayRoom();
    void displayInventory();
    void displayStatusBar();
    void displayItem(Item *item);

    // Command methods
    void cycleView();
    void confirmQuit();
    void invalidCommand();
    void question(std::string suspectName);
    void gather();
    void pickup(std::string itemName);
    void drop(std::string itemName);
    void examine(std::string itemName);
    void accuse(std::string suspectName);
    void move(Direction direction);
    void search();
    void stab(std::string suspectName);
    void note(std::string);
    void examineAll();
};

```

item.cpp

```

/*****
 *
 * item.cpp
 *
 * Represents an item object
 * Might be the murder weapon
 *
 * Briano Goestiawan, 31482228
 *
 *****/

```

```

*****/

#include <iostream>
#include "item.h"
#include "utils.h"
#define WINDOW_WIDTH 77 // Not including border

using namespace std;

// Initialize class variables
Item::Item(string name, string imagePath) {
    this->name = name;
    this->image = readFile(imagePath);
    this->murderWeapon = false;
}

// Return the name of the item
string Item::getName() {
    return this->name;
}

// Return a pointer to the location of the item
Room *Item::getLocation() {
    return this->location;
}

// Set the location of the item
void Item::setLocation(Room *room) {
    this->location = room;
}

// Return the item image as an ascii art string
string Item::getImage() {
    return this->image;
}

// Return true if item is the murder weapon else false
bool Item::getMurderWeapon() {
    return this->murderWeapon;
}

// Set murder room and add blood stains to item image
void Item::setMurderWeapon() {
    this->murderWeapon = true;

    // Replace random column of every other line with "BLOOD"
    string blood = "BLOOD";
    for (int i = 1; i <= 34; i++) {
        if (i % 2 == 0) {
            int possible = WINDOW_WIDTH - blood.length() + 1;
            int position = i * 80 + 1 + rand() % possible;
            this->image.replace(position, blood.length(), blood);
        }
    }
}

```

```
}
```

item.h

```
/*  
 *  
 * item.h  
 *  
 * Represents an item object  
 * Might be the murder weapon  
 *  
 * Briano Goestiawan, 31482228  
 */  
*****/  
  
#pragma once  
#include <string>  
#include "room.h"  
  
class Item {  
    public:  
        Item(std::string name, std::string imagePath);  
        std::string getName();  
        std::string getImage();  
        Room *getLocation();  
        void setLocation(Room *room);  
        bool getMurderWeapon();  
        void setMurderWeapon();  
  
    private:  
        std::string name;  
        std::string image;  
        bool murderWeapon;  
        Room *location;  
};
```

main.cpp

```
/*  
 *  
 * main.cpp  
 *  
 * Controls the flow of the application  
 * Handle menu interactions  
 * Difficulty option  
 * View leaderboard functionality  
 *  
 * Briano Goestiawan, 31482228  
 */  
*****/  
  
#include <iostream>  
#include <ctime>
```

```

#include <string>
#include "game.h"
#include "utils.h"
#define WINDOW_WIDTH 77 // Not including border
#define WINDOW_HEIGHT 41 // Not including border

using namespace std;

bool hasExit = false; // true while application is running else exit
Difficulty difficulty = DIFF_MEDIUM;

void mainMenu();
void runGame();
void changeDifficulty();
void showLeaderboards();
void showLeaderboard(Difficulty difficulty);
void addToLeaderboard(int timeSeconds, std::string playerName);
std::string difficultyString(Difficulty difficulty);
std::string leaderboardFileName(Difficulty difficulty);

// Function call graph: main -> mainMenu -> startGame
int main() {
    // Seed random with current time
    srand(time(NULL));

    // Run main menu until the user exits
    while (!hasExit) {
        mainMenu();
    }
}

// Show list of actions to user, run specific actions based on what the user input
void mainMenu() {
    clearScreen();

    // Display main menu screen
    cout << readFile("assets/cover_screen.txt");
    string diff = fixedWidth(difficultyString(difficulty), ' ', 14);
    cout << " | " << endl;
    cout << " | 1. Start game | 3. Show leaderboard | " << endl;
    cout << " | " << endl;
    cout << " | 2. Change difficulty " << diff << " | 4. Exit | " << endl;
    cout << " | " << endl;
    cout << "+-----+" << endl;

    // Get option from user. keep asking until get valid option
    int option;
    do {
        option = readInputInt("Pick one option (1-4): ");
    } while(option < 1 || option > 4);

    // Call the appropriate functions based on option the user selects
    switch (option) {
        case 1:

```

```

        runGame();
        break;
    case 2:
        changeDifficulty();
        break;
    case 3:
        showLeaderboards();
        break;
    case 4:
        hasExit = true;
    }
}

// Start game
void runGame() {
    // Get player name from user. Ask again if user put blank
    string playerName;
    do {
        playerName = readInput("Enter player name: ");
    } while (playerName == "");

    // Run the game while keeping track of the time
    int gameStartTimeSeconds = time(NULL);
    Game game(playerName, difficulty);

    // Show story line and help screen before running the main game loop
    game.showStoryLine();
    game.showHelpScreen();

    // Main game loop
    while (!game.getGameOver()) {
        game.displayView();
        game.command();
    }

    // Display end screen congratulating or ridiculing the player
    // depending on if they win or lose. Show time played
    clearScreen();
    if (game.getGameWin()) {
        cout << readFile("assets/you_win.txt");
    } else {
        cout << readFile("assets/game_over.txt");
    }

    // Display the time played
    int playedTimeSeconds = time(NULL) - gameStartTimeSeconds;
    cout << " | " << endl;
    cout << " | " + fixedWidth("  TIME: " + toHourMinuteSeconds(playedTimeSeconds), ' ', WINDOW_WIDTH) << endl;
    cout << " | " << endl;
    cout << "+-----+" << endl;
    pause();

    // Add time to leaderboard if win
    if (game.getGameWin()) addToLeaderboard(playedTimeSeconds, playerName);
}

```

```

}

// Show the difficulty options to the player than prompt the player to pick one difficulty level
void changeDifficulty() {
    // Display difficulty options
    clearScreen();
    cout << readFile("assets/difficulty_options.txt");

    // Get a valid option from user. Keep asking until user input valid option
    int option;
    do {
        option = readInputInt("Pick an option (1-4):");
    } while (option < 1 || option > 4);

    // Set difficulty based on user input
    difficulty = static_cast<Difficulty>(option - 1);
}

// Show all leaderboards one by one
void showLeaderboards() {
    showLeaderboard(DIFF_EASY);
    showLeaderboard(DIFF_MEDIUM);
    showLeaderboard(DIFF_HARD);
    showLeaderboard(DIFF_NIGHTMARE);
}

// Display leaderboard. Wait for user before continuing
void showLeaderboard(Difficulty difficulty) {
    clearScreen();

    // Print leaderboard heading including difficulty category
    cout << "+-----+" << endl;
    cout << "|                                     |" << endl;
    cout << '|' << fixedWidth("  LEADERBOARD " + difficultyString(difficulty), ' ', WINDOW_WIDTH) << '|';

    // Load the leaderboard and split it by newline into a vector of strings
    vector<string> leaderboard = stringSplit(readFile(leaderboardFileName(difficulty)), '\n');

    // Print every entry in leaderboard
    for (int i = 0; i < leaderboard.size(); i++) {
        int spaceIndex = leaderboard[i].find(' ');
        string time = toHourMinuteSeconds(stoi(leaderboard[i].substr(0, spaceIndex)));
        string playerName = leaderboard[i].substr(spaceIndex + 1);
        int dotCount = WINDOW_WIDTH - playerName.length() - time.length() - 6;
        cout << "|                                     |" << endl;
        cout << "|  " << playerName << ' ' << fixedWidth("", '.', dotCount) << ' ' << time << " |" << endl;
    }

    // Print blank lines so it reaches the window height
    int blankLineCount = WINDOW_HEIGHT - leaderboard.size() * 2 - 2;
    for (int i = 0; i < blankLineCount; i++) {
        cout << "|                                     |" << endl;
    }
}

```

```

        cout << "+-----+" << endl;
        pause();
    }

    // Add a new entry to leaderboard, insert in correct position sorted in ascending order by time
    void addToLeaderboard(int timeSeconds, string playerName) {
        vector<string> leaderboard = stringSplit(readFile(leaderboardFileName(difficulty)), '
');
        string entry = to_string(timeSeconds) + ' ' + playerName;

        if (leaderboard.size() > 0) {
            // If leaderboard is not empty, insert entry to correct location
            for (int i = 0; i < leaderboard.size(); i++) {
                int spaceIndex = leaderboard[i].find(' ');
                int time = stoi(leaderboard[i].substr(0, spaceIndex));
                if (timeSeconds < time) {
                    leaderboard.insert(leaderboard.begin() + i, entry);
                    break;
                }
            }
        } else {
            // Else append to leaderboard as first entry
            leaderboard.push_back(entry);
        }

        writeFile(leaderboardFileName(difficulty), stringJoin(leaderboard));
    }

    // Maps difficulty enum type to its string form
    string difficultyString(Difficulty difficulty) {
        switch (difficulty) {
            case DIFF_EASY:
                return "[EASY]";
            case DIFF_MEDIUM:
                return "[MEDIUM]";
            case DIFF_HARD:
                return "[HARD]";
            case DIFF_NIGHTMARE:
                return "[NIGHTMARE]";
        }
    }

    // Maps difficulty enum to file path of its leaderboard collection
    string leaderboardFileName(Difficulty difficulty) {
        switch (difficulty) {
            case DIFF_EASY:
                return "assets/leaderboard_easy.txt";
            case DIFF_MEDIUM:
                return "assets/leaderboard_medium.txt";
            case DIFF_HARD:
                return "assets/leaderboard_hard.txt";
            case DIFF_NIGHTMARE:
                return "assets/leaderboard_nightmare.txt";
        }
    }

```



```

    }
}

```

player.cpp

```

/*****
 *
 * player.cpp
 *
 * Represents the player in the game
 * Stores player's name, location and image
 *
 * Briano Goestiawan, 31482228
 *
 *****/

#include <iostream>
#include "player.h"
#include "room.h"
#include "utils.h"

using namespace std;

// Default constructor, initialize name and room to NULL
Player::Player() {
    this->name = "";
    this->room = NULL;
}

// Initialize player variable and set player image
Player::Player(string name, Room *startingRoom) {
    this->name = name;
    this->room = startingRoom->getNeighbour(DIR_RIGHT);

    // Load the image of the player from file
    image = stringSplit(readFile("assets/player.txt"), '
');
}

// Return the player's name
string Player::getName() {
    return this->name;
}

// Return the player's image string vector
vector<string> Player::getImage() {
    return this->image;
}

// Set the player's location
void Player::setRoom(Room *room) {
    this->room = room;
}

```

```
// Get player's location as a pointer to a room
Room *Player::getRoom() {
    return this->room;
}
```

player.h

```

/*****
 *
 * player.h
 *
 * Represents the player in the game
 * Stores player's name, location and image
 *
 * Briano Goestiawan, 31482228
 *
 *****/

#pragma once
#include <string>
#include <vector>
#include "item.h"
#include "room.h"

class Player {
public:
    Player();
    Player(std::string name, Room *startingRoom);
    std::string getName();
    std::vector<std::string> getImage();
    void setRoom(Room *room);
    Room *getRoom();

private:
    std::string name;
    std::vector<std::string> image;
    Room *room;
};

```

room.cpp

```

/*****
 *
 * room.cpp
 *
 * Represents the room object
 * Player, item and suspect all points to a room
 * Can access neighbouring rooms in all 4 directions
 *
 * Briano Goestiawan, 31482228
 *
 *****/

```

```

#include "room.h"
#include "utils.h"
#define WINDOW_WIDTH 77 // Not including border

using namespace std;

// Default constructor so can be invoked from game
Room::Room() { }

// Initialize room name and load image from file
Room::Room(string name, string imagePath) {
    // Initialize class variables
    this->name = name;
    this->image = readFile(imagePath);
    this->itemHidden = true;
    this->murderRoom = false;

    // Initialize neighbours to NULL
    this->neighbour[DIR_LEFT] = NULL;
    this->neighbour[DIR_RIGHT] = NULL;
    this->neighbour[DIR_UP] = NULL;
    this->neighbour[DIR_DOWN] = NULL;
}

// Return the room name
string Room::getName() {
    return this->name;
}

// Return the room image as a ascii art string
string Room::getImage() {
    return this->image;
}

// Return true if room have not been search
bool Room::getItemHidden() {
    return this->itemHidden;
}

// Set item to not hidden if room is searched
void Room::search() {
    this->itemHidden = false;
}

// Return true if room is the murder room else false
bool Room::getMurderRoom() {
    return this->murderRoom;
}

// Set murder room and add blood to room
void Room::setMurderRoom() {
    this->murderRoom = true;
}

```

```

        // Replace random column of every other line with "BLOOD"
        string blood = "BLOOD";
        for (int i = 1; i <= 33; i++) {
            if (i % 2 == 0) {
                int posible = WINDOW_WIDTH - blood.length() + 1;
                int position = i * 80 + 1 + rand() % posible;
                this->image.replace(position, blood.length(), blood);
            }
        }
    }

    // Set neighbour room in direction
    void Room::setNeighbour(Direction direction, Room *room) {
        this->neighbour[direction] = room;
    }

    // Get neighbouring room provided a direction
    Room *Room::getNeighbour(Direction direction) {
        return this->neighbour[direction];
    }
}

```

room.h

```

/*****
 *
 * room.h
 *
 * Represents the room object
 * Player, item and suspect all points to a room
 * Can access neighbouring rooms in all 4 directions
 *
 * Briano Goestiawan, 31482228
 *
 *****/

#pragma once
#include <string>
#include <vector>
#include <map>

enum Direction {
    DIR_LEFT,
    DIR_RIGHT,
    DIR_UP,
    DIR_DOWN
};

class Room {
public:
    Room();
    /* Room(std::string name); */
    Room(std::string name, std::string imagePath);
}

```

```

        std::string getName();
        std::string getImage();
        bool getItemHidden();
        void search();
        bool getMurderRoom();
        void setMurderRoom();

        // Neighbour
        void setNeighbour(Direction direction, Room *room);
        Room *getNeighbour(Direction direction);

    private:
        std::string name;
        std::string image;
        bool itemHidden;
        bool murderRoom;
        std::map<Direction, Room*> neighbour; // Pointer to neighbouring room in all four direction
};

```

suspect.cpp

```

/*****
 *
 * suspect.cpp
 *
 * Represents the suspects in the game
 * A suspect might be a killer, victim or neither (normal)
 * Provides interface to get and set fields
 *
 * Briano Goestiawan, 31482228
 *
 *****/

#include <iostream>
#include "suspect.h"
#include "utils.h"

using namespace std;

// Suspect constructor
Suspect::Suspect(string name) {
    this->name = name;
    this->type = SUS_NORMAL;
    this->alibi = NULL;
}

// Return the name of the suspect
string Suspect::getName() {
    return this->name;
}

// Return a pointer to suspect's location
Room *Suspect::getRoom() {

```

```

        return this->room;
    }

    // Set the location of the suspect
    void Suspect::setRoom(Room *room) {
        this->room = room;
    }

    // Get the type of suspect
    SuspectType Suspect::getType() {
        return this->type;
    }

    // Mutator method to set the type of the suspect
    void Suspect::setType(SuspectType type) {
        this->type = type;
    }

    // Set the alibi of the suspect
    void Suspect::setAlibi(Suspect *alibi) {
        this->alibi = alibi;
    }

    // Display messages from the suspect
    void Suspect::talk(string playerName) {
        switch (this->type) {
            // Normal or killer acts the same way by mentioning their alibi or
            // stating that their alone
            case SUS_NORMAL:
            case SUS_KILLER:
                cout << "Hi " << playerName << ", ";
                if (this->alibi == NULL) {
                    cout << "I was alone" << endl;
                } else {
                    cout << "I was with " << this->alibi->getName() << endl;
                }
                break;
            // Dead
            case SUS_VICTIM:
                cout << "X_X" << endl;
                break;
        }
    }

    // Move to neighbouring room based on given direction
    void Suspect::move(Direction direction) {
        Room *destination = this->room->getNeighbour(direction);
        // Move to destination if it exists
        if (destination) this->room = destination;
    }

```

suspect.h

```
/*
 *
 * suspect.h
 *
 * Represents the suspects in the game
 * A suspect might be a killer, victim or neither (normal)
 * Provides interface to get and set fields
 *
 * Briano Goestiawan, 31482228
 */

#pragma once
#include <string>
#include "room.h"

enum SuspectType {
    SUS_NORMAL,
    SUS_KILLER,
    SUS_VICTIM
};

class Suspect {
public:
    Suspect(std::string name);
    std::string getName();
    SuspectType getType();
    void setType(SuspectType type);
    void setAlibi(Suspect *alibi);
    void talk(std::string playerName);
    void setRoom(Room *room);
    Room *getRoom();
    void move(Direction direction);

private:
    std::string name;
    SuspectType type;
    Suspect *alibi;
    Room *room;
};
```

utils.cpp

```
/*
 *
 * utils.cpp
 *
 * A collection of helper functions to avoid repetition
 *
 * Briano Goestiawan, 31482228
 */
```

```

#include <fstream>
#include <iostream>
#include <vector>
#include <sstream>
#include "utils.h"
#ifdef _WIN32
#define CLEAR "cls"
#else
#define CLEAR "clear"
#endif

using namespace std;

// Returns the content of a file specified by fileName
string readFile(string fileName) {
    ifstream file;
    file.open(fileName);

    // If file failed to open
    if (!file.is_open()) {
        cout << "ERROR: cannot open file: " << fileName << endl;
        return "";
    }

    // Append each line in file to content
    string content;
    string line;
    getline(file, content);
    while (!file.eof()) {
        getline(file, line);
        content += '
' + line;
    }
    file.close();

    return content;
}

// Write content to file named fileName, overwrite if it exists
void writeFile(string fileName, string content) {
    ofstream file;
    file.open(fileName);

    // If file failed to open
    if (!file.is_open()) {
        cout << "ERROR: cannot open file" << fileName << endl;
        return;
    }

    file << content;
    file.close();
}

```



```

// Split string to vector of strings based on character
vector<string> stringSplit(string content, char character) {
    vector<string> lines;

    // Push to lines when content matches character
    int prev = 0;
    for (int i = 0; i < content.length(); i++) {
        if (content[i] == character) {
            lines.push_back(content.substr(prev, i - prev));
            prev = i + 1;
        }
    }

    // Check if last element if delimited else ignore
    string end = content.substr(prev);
    if (end.length() > 0) {
        lines.push_back(end);
    }

    return lines;
}

// Join vector of string into one string used character
string stringJoin(vector<string> lines, char character) {
    string result = "";

    for (int i = 0; i < lines.size(); i++) {
        result += lines[i] + character;
    }

    return result;
}

// Print prompt to screen then read and return input line
string readInput(string prompt) {
    cout << prompt;
    string input;
    getline(cin, input);
    return input;
}

// Print prompt to screen then read input as int. Repeat until user input an int
int readInputInt(string prompt) {
    string input;
    do {
        input = readInput("Pick one option (1-4): ");
    } while (!isInteger(input));
    return stoi(input);
}

// Pause the control flow until the user press enter
void pause() {
    readInput("Press Enter to continue ");
}

```

```

// Cross platform clear command
void clearScreen() {
    system(CLEAR);
}

// Return true if value is an integer string else return false
bool isInteger(std::string value) {
    // Empty string is not an integer
    if (value.length() <= 0) {
        return false;
    }

    // Set first index to be checked to allow negative integers
    int firstDigitIndex = 0;
    if (value[0] == '-')
        firstDigitIndex = 1;

    // Check if any characters is not a digit
    for (int i = firstDigitIndex; i < value.length(); i++)
        if (value[i] < '0' or value[i] > '9')
            return false;

    // It survived all the previous tests. It must be an integer
    return true;
}

// Returns a copy of value with all the uppercase characters replaced with its lowercase equivalent
string toLower(string str) {
    for (int i = 0; i < str.length(); i++) {
        if (str[i] >= 'A' && str[i] <= 'Z') {
            str[i] = str[i] + 'a' - 'A';
        }
    }
    return str;
}

// Returns a string with a fixed width and align text to the left
string fixedWidth(string text, char symbol, int width) {
    string output = "";

    for (int i = 0; i < width; i++) {
        if (i < text.length()) {
            output += text[i];
        } else {
            output += symbol;
        }
    }

    return output;
}

// Returns seconds time in hour:min:sec string format
string toHourMinuteSeconds(int seconds) {

```

```

    int hour = seconds / 60 / 60;
    int min = seconds / 60 % 60;
    int sec = seconds % 60;
    stringstream output;
    output << hour << ':' << min << ':' << sec;
    return output.str();
}

```

utils.h

```

/*****
 *
 *  utils.cpp
 *
 *  A collection of helper functions to avoid repetition
 *
 *  Briano Goestiawan, 31482228
 *
 *****/

#pragma once
#include <string>

// Returns the content of a file specified by fileName
std::string readFile(std::string fileName);

// Write content to file named fileName, overwrite if it exists
void writeFile(std::string fileName, std::string content);

// Split string to vector of strings based on character
std::vector<std::string> stringSplit(std::string content, char character = ' ');

// Join vector of string into one string used character
std::string stringJoin(std::vector<std::string> lines, char character = ' ');

// Print prompt to screen then read and return input line
std::string readInput(std::string prompt = "");

// Print prompt to screen then read input as int. Repeat until user input an int
int readInputInt(std::string prompt = "");

// Pause the control flow until the user press enter
void pause();

// Clear the output screen
void clearScreen();

// Return true if value is an integer string else return false
bool isInteger(std::string value);

// Returns a copy of value with all the uppercase characters replaced with its lowercase equivalent

```

```
std::string toLower(std::string value);

// Returns a string with a fixed width and align text to the left
std::string fixedWidth(std::string text, char symbol, int width);

// Returns seconds time in hour:min:sec string format
std::string toHourMinuteSeconds(int seconds);
```