#include <iostream>

#include <vector>

#include <algorithm>

#include <cstdlib>

#include <ctime>

const int DISPENSER\_ROWS = 3; // Example value, adjust as needed

const int DISPENSER\_COLS = 5; // Example value, adjust as needed

const int NUM\_COLORS = 3; // Example value, adjust as needed

const int INITIAL\_POTIONS = 3; // Example value, adjust as needed

enum Color {

RED,

BLUE,

YELLOW,

// Add more colors as needed

};

struct Marble {

Color color;

};

class Potion {

public:

Color requiredColor;

bool isCompleted;

int points; // Add points for completed potions

Potion(Color color) : requiredColor(color), isCompleted(false), points(0) {}

};

class Player {

public:

std::vector<Marble> hand;

std::vector<Potion> potions;

int score; // Player's score

std::string name; // Added for player identification

Player() : score(0) {}

};

class PotionDispenser {

public:

std::vector<std::vector<Marble>> dispenserRows;

void initializeDispenser() {

// Initialize the dispenser rows with random marbles

for (int i = 0; i < DISPENSER\_ROWS; ++i) {

std::vector<Marble> row;

for (int j = 0; j < DISPENSER\_COLS; ++j) {

Marble marble;

marble.color = static\_cast<Color>(rand() % NUM\_COLORS);

row.push\_back(marble);

}

dispenserRows.push\_back(row);

}

}

};

Marble generateRandomMarble() {

// Generate a random Color for the marble

Color randomColor = static\_cast<Color>(rand() % NUM\_COLORS);

// Create and return the new marble

Marble newMarble = {randomColor};

return newMarble;

}

class PotionExplosionGame {

public:

PotionDispenser dispenser;

std::vector<Player> players;

int currentPlayerIndex;

PotionExplosionGame(int numPlayers) : currentPlayerIndex(0) {

// Initialize the dispenser

dispenser.initializeDispenser();

// Create players

for (int i = 0; i < numPlayers; ++i) {

Player player;

player.name = "Player " + std::to\_string(i + 1);

players.push\_back(player);

}

// Distribute initial potions to players

for (Player& player : players) {

for (int i = 0; i < INITIAL\_POTIONS; ++i) {

Potion potion(generateRandomMarble().color);

player.potions.push\_back(potion);

}

}

}

// Add other member functions as needed

void playTurn() {

// Implement the logic for playing a turn

}

// Add other member functions as needed

};

int main() {

// Seed for random number generation

std::srand(static\_cast<unsigned>(std::time(nullptr)));

// Create a PotionExplosionGame with, for example, 2 players

PotionExplosionGame game(2);

// Main game loop

while (true) {

// Display the game board

// game.displayGameBoard(); // Uncomment once implemented

// Play the turn for the current player

game.playTurn();

// Check if the game should end

// if (game.potionsStacksEmpty()) { // Uncomment once implemented

// break;

// }

}

// Display the final results and winner

// game.determineWinner(); // Uncomment once implemented

return 0;

}

UNCORRECTED CODE

#include <iostream>

#include <vector>

#include <algorithm>

#include <cstdlib>

#include <ctime>

enum Color {

RED,

BLUE,

YELLOW,

// Add more colors as needed

};

struct Marble {

Color color;

};

class Potion {

public:

Color requiredColor;

bool isCompleted;

int points; // Add points for completed potions

};

class Player {

public:

std::vector<Marble> hand;

std::vector<Potion> potions;

int score; // Player's score

};

class PotionDispenser {

public:

std::vector<std::vector<Marble>> dispenserRows;

void initializeDispenser() {

// Initialize the dispenser rows with random marbles

for (int i = 0; i < DISPENSER\_ROWS; ++i) {

std::vector<Marble> row;

for (int j = 0; j < DISPENSER\_COLS; ++j) {

Marble marble;

marble.color = static\_cast<Color>(rand() % NUM\_COLORS);

row.push\_back(marble);

}

dispenserRows.push\_back(row);

}

}

};

Marble generateRandomMarble() {

// Generate a random Color for the marble

Color randomColor = static\_cast<Color>(rand() % NUM\_COLORS); // Assuming NUM\_COLORS is the total number of colors

// Create and return the new marble

Marble newMarble = {randomColor};

return newMarble;

}

class PotionExplosionGame {

public:

PotionDispenser dispenser;

std::vector<Player> players;

int currentPlayerIndex;

PotionExplosionGame::PotionExplosionGame(int numPlayers) {

currentPlayerIndex = 0;

// Initialize the dispenser

dispenser.initializeDispenser();

// Create players

for (int i = 0; i < numPlayers; ++i) {

Player player;

players.push\_back(player);

}

// Distribute initial potions to players

for (Player& player : players) {

for (int i = 0; i < INITIAL\_POTIONS; ++i) {

Potion potion;

// Set required color and other potion properties

// ...

player.potions.push\_back(potion);

}

}

}

void PotionExplosionGame::playTurn() {

// Get the current player

Player& currentPlayer = players[currentPlayerIndex];

// Perform regular pick

performRegularPick(currentPlayer);

// Ask for help from the professor

askForHelp(currentPlayer);

// Display player's options and prompt for action

displayPlayerOptions(currentPlayer);

// Handle player's chosen actions (placing marbles, using potions, etc.)

handlePlayerActions(currentPlayer);

// Resolve potion effects

resolvePotionEffects();

// Check for completed potions

checkCompletedPotions();

// Refill the dispenser

refillDispenser();

// Switch to the next player

currentPlayerIndex = (currentPlayerIndex + 1) % players.size();

}

void PotionExplosionGame::triggerExplosion(int row, int col) {

// Check if an explosion occurs at the specified position

if (isExplosionTriggered(row, col)) {

// Collect exploded marbles

collectExplodedMarbles(row, col);

// Trigger additional explosions if any

while (isAdditionalExplosion()) {

collectExplodedMarbles(row, col);

}

}

}

bool PotionExplosionGame::isExplosionTriggered(int row, int col) {

// Check if there are at least two matching marbles in the specified row

if (dispenser.dispenserRows[row].size() >= 2) {

Marble& marble1 = dispenser.dispenserRows[row][col];

Marble& marble2 = dispenser.dispenserRows[row][col + 1];

// Check if the two marbles have the same color

if (marble1.color == marble2.color) {

return true; // Explosion occurs

}

}

return false; // No explosion

}

bool PotionExplosionGame::isAdditionalExplosion() {

bool additionalExplosion = false;

// Iterate through dispenser rows to check for additional explosions

for (int row = 0; row < dispenser.dispenserRows.size(); ++row) {

for (int col = 0; col < dispenser.dispenserRows[row].size() - 1; ++col) {

Marble& marble1 = dispenser.dispenserRows[row][col];

Marble& marble2 = dispenser.dispenserRows[row][col + 1];

// Check if two matching marbles are adjacent

if (marble1.color == marble2.color) {

additionalExplosion = true;

break;

}

}

if (additionalExplosion) {

break; // If one additional explosion is found, no need to check further

}

}

return additionalExplosion;

}

void resolvePotionEffects() {

// Implement the logic for resolving potion effects

}

void PotionExplosionGame::checkCompletedPotions() {

// Iterate through each player to check completed potions

for (Player& player : players) {

// Iterate through each potion of the player

for (Potion& potion : player.potions) {

if (!potion.isCompleted) {

// Check if the potion is completed by matching its color with the ingredients in the player's hand

bool isPotionCompleted = std::all\_of(player.hand.begin(), player.hand.end(),

[&potion](const Marble& marble) {

return marble.color == potion.requiredColor;

});

if (isPotionCompleted) {

// Mark the potion as completed

potion.isCompleted = true;

// Add any additional logic for handling completed potions if needed

}

}

}

}

}

void PotionExplosionGame::refillDispenser() {

// Iterate through each row of the dispenser

for (std::vector<Marble>& row : dispenser.dispenserRows) {

// Check if the row is not full

if (row.size() < 5) {

// Calculate the number of marbles needed to fill the row

int marblesToAdd = 5 - row.size();

// Add new marbles to the row

for (int i = 0; i < marblesToAdd; ++i) {

// Implement logic to add new marbles (random or from a predefined set)

row.push\_back(generateRandomMarble());

}

}

}

}

void PotionExplosionGame::endGame() {

// Check if the countdown stack is empty or if there are no more potions in the stacks

if (countdownStack.empty() || potionsStacksEmpty()) {

// End the game

determineWinner();

}

}

bool PotionExplosionGame::potionsStacksEmpty() {

// Check if all potion stacks are empty

for (const auto& potionStack : potionStacks) {

if (!potionStack.empty()) {

return false;

}

}

return true;

}

void PotionExplosionGame::determineWinner() {

// Calculate and display the final scores

calculateFinalScores();

displayFinalScores();

// Determine the winner

Player\* winner = findWinner();

// Display the winner

displayWinner(winner);

}

void PotionExplosionGame::calculateFinalScores() {

// Iterate through each player and calculate their final score

for (Player& player : players) {

int playerScore = 0;

// Add points from completed potions

for (const Potion& potion : player.potions) {

if (potion.isCompleted) {

playerScore += getPotionPoints(potion);

}

}

// Subtract points from helper tokens

playerScore -= getHelperTokenPoints(player);

// Add points from skill tokens

playerScore += getSkillTokenPoints(player);

// Set the player's final score

player.finalScore = playerScore;

}

}

int PotionExplosionGame::getPotionPoints(const Potion& potion) {

// Implement the logic to calculate points for a completed potion

// You may use the potion's required color or other properties

int points = 0;

// Switch case based on the required color of the potion

switch (potion.requiredColor) {

case RED:

points = 3; // Assign 3 points for a completed red potion

break;

case BLUE:

points = 2; // Assign 2 points for a completed blue potion

break;

case YELLOW:

points = 1; // Assign 1 point for a completed yellow potion

break;

// Add more cases for other colors if needed

default:

points = 0; // Default case, replace with appropriate value

break;

}

return points;

}

int PotionExplosionGame::getHelperTokenPoints(const Player& player) {

// Implement the logic to calculate points from helper tokens

// For example: return player.helperTokens.size() \* 2;

// Calculate points based on the number of helper tokens

int points = player.helperTokens.size() \* 2;

return points;

}

int PotionExplosionGame::getSkillTokenPoints(const Player& player) {

// Implement the logic to calculate points from skill tokens

// For example: return player.skillTokens.size() \* 5;

// Calculate points based on the number of skill tokens

int points = player.skillTokens.size() \* 5;

return points;

}

Player\* PotionExplosionGame::findWinner() {

// Find the player with the highest final score

auto maxScorePlayer = std::max\_element(players.begin(), players.end(),

[](const Player& a, const Player& b) {

return a.finalScore < b.finalScore;

});

return &(\*maxScorePlayer);

}

void PotionExplosionGame::displayFinalScores() {

// Implement the logic to display the final scores

std::cout << "Final Scores:\n";

// Print each player's name and final score

for (const Player& player : players) {

std::cout << "Player " << player.name << ": " << player.finalScore << " points\n";

}

}

void PotionExplosionGame::displayWinner(Player\* winner) {

// Implement the logic to display the winner

if (winner) {

std::cout << "Player " << winner->name << " wins!\n";

} else {

std::cout << "It's a tie!\n";

}

}

////////////////////////////////////////////////

void PotionExplosionGame::displayGameBoard() {

// Display dispenser rows

std::cout << "Dispenser:\n";

for (const auto& row : dispenser.dispenserRows) {

for (const Marble& marble : row) {

// Display each marble color

switch (marble.color) {

case RED:

std::cout << "R ";

break;

case BLUE:

std::cout << "B ";

break;

case YELLOW:

std::cout << "Y ";

break;

// Add more cases for other colors as needed

}

}

std::cout << '\n';

}

// Display players' potions

std::cout << "\nPlayers' Potions:\n";

for (const Player& player : players) {

std::cout << "Player " << player.name << ":\n";

for (const Potion& potion : player.potions) {

// Display each potion's information

std::cout << " Potion - Required Color: ";

switch (potion.requiredColor) {

case RED:

std::cout << "RED";

break;

case BLUE:

std::cout << "BLUE";

break;

case YELLOW:

std::cout << "YELLOW";

break;

// Add more cases for other colors as needed

}

std::cout << ", Completed: " << (potion.isCompleted ? "Yes" : "No") << '\n';

}

std::cout << '\n';

}

// Add more sections to display additional game board information as needed

}

};

int main() {

// Seed for random number generation

std::srand(static\_cast<unsigned>(std::time(nullptr)));

// Create a PotionExplosionGame with, for example, 2 players

PotionExplosionGame game(2);

// Main game loop

while (true) {

// Display the game board

game.displayGameBoard();

// Play the turn for the current player

game.playTurn();

// Check if the game should end

if (game.potionsStacksEmpty()) {

break;

}

}

// Display the final results and winner

game.determineWinner();

return 0;

}