**Overview of the Program**

This C++ program is designed to generate match fixtures for a set of 10 teams, ensuring that teams from the same location don't face each other until they have played teams from other towns. The program reads data from a CSV file, generates matchups for two legs, and saves the final fixture list to another CSV file. The code is organized into several functions to manage specific tasks such as loading teams, generating fixtures, and writing the results.

**Explanation of Each Section**

**1. Including Libraries**

#include <iostream>

#include <fstream>

#include <sstream>

#include <string>

#include <vector>

#include <unordered\_map>

#include <unordered\_set>

#include <algorithm>

* **Why we chose these libraries**:
  + iostream for input/output operations like printing to the console.
  + fstream to read and write files, which we need to handle our CSV files.
  + sstream for parsing the CSV file line by line.
  + vector to store dynamic lists of teams and matchups.
  + unordered\_map and unordered\_set for fast lookups and organizing data based on key-value pairs (e.g., teams grouped by their location).
  + algorithm for sorting or manipulating data later if needed.

**2. Defining the Team Structure**

struct Team {

std::string name;

std::string location;

std::string stadium;

};

* **Why we used a struct**: The Team struct is a simple way to group related data together. Each team has three pieces of information: its name, location, and the stadium where it plays. Using a struct makes it easy to store and pass around team details throughout the program. We could also use a class, but a struct keeps things straightforward for this task, since we're only storing data without needing more complex functionality.

**3. Loading Teams from the CSV**

std::vector<Team> loadTeamsFromCSV(const std::string& filename) {

std::vector<Team> teams;

std::ifstream file(filename);

std::cout << "Attempting to open file: " << filename << std::endl;

if (!file.is\_open()) {

std::cerr << "Error: Could not open file " << filename << std::endl;

return teams;

}

std::string line;

while (std::getline(file, line)) {

std::istringstream ss(line);

Team team;

std::getline(ss, team.name, ',');

std::getline(ss, team.location, ',');

std::getline(ss, team.stadium, ',');

teams.push\_back(team);

}

file.close();

return teams;

}

* **Why this part is crucial**: This function reads the teams.csv file and extracts the team data. The file is opened using ifstream, and we parse each line into a Team struct using stringstream. Each team's details are separated by commas, so we use getline with a comma delimiter to extract the name, location, and stadium.
* **Error handling**: The if (!file.is\_open()) checks whether the file could be opened. If not, the function returns an empty vector, ensuring the program doesn't crash.

**4. Writing Fixtures to CSV**

void writeFixturesToCSV(const std::vector<std::string>& fixtures, const std::string& filename) {

std::ofstream file(filename);

if (!file.is\_open()) {

std::cerr << "Error: Could not open file " << filename << " for writing." << std::endl;

return;

}

file << "Match,Leg,Location,Stadium\n";

for (const auto& line : fixtures) {

file << line << std::endl;

}

file.close();

std::cout << "Fixtures written to " << filename << std::endl;

}

* **Why this approach was chosen**: This function is designed to save the final match fixtures to a file. We use ofstream to write to a file, and it opens the file for writing. Each line of the fixtures is written to the file, with an initial CSV header (Match,Leg,Location,Stadium) to make the file readable.
* **Error handling**: As with file reading, the code checks if the file was opened successfully. If not, an error message is printed, preventing silent failures.

**5. Generating Fixtures**

void generateFixtures(const std::vector<Team>& teams) {

std::unordered\_map<std::string, std::vector<Team>> townTeams;

for (const auto& team : teams) {

townTeams[team.location].push\_back(team);

}

std::vector<std::pair<Team, Team>> fixtures;

std::unordered\_set<std::string> playedPairs;

std::vector<std::string> outputLines;

for (const auto& town : townTeams) {

const auto& townTeamList = town.second;

for (const auto& otherTown : townTeams) {

if (town.first == otherTown.first) continue;

for (const auto& homeTeam : townTeamList) {

for (const auto& awayTeam : otherTown.second) {

std::string pairKey = homeTeam.name + " vs " + awayTeam.name;

if (playedPairs.find(pairKey) == playedPairs.end()) {

fixtures.emplace\_back(homeTeam, awayTeam);

playedPairs.insert(pairKey);

}

}

}

}

}

* **Complicated part explained**: Here we group the teams by location using unordered\_map, where the location is the key, and the value is a list of teams from that location. This allows us to easily check which teams are from the same town.

The nested loops create matchups between teams from different towns. The playedPairs set ensures that the same match isn't generated more than once.

**6. Tracking Match Fixtures**

int totalWeeks = 0;

std::vector<std::string> leg1Lines, leg2Lines;

for (size\_t i = 0; i < fixtures.size(); ++i) {

if (i % 4 == 0) {

totalWeeks++;

if (totalWeeks > 40) break;

leg1Lines.push\_back("Weekend " + std::to\_string(totalWeeks) + ":");

}

const auto& match = fixtures[i];

std::string leg1Line = match.first.name + " (Home) vs " + match.second.name + " (Away),Leg 1," +

match.first.location + "," + match.first.stadium;

leg1Lines.push\_back(leg1Line);

}

for (size\_t i = 0; i < fixtures.size(); ++i) {

if (i % 4 == 0) {

totalWeeks++;

if (totalWeeks > 40) break;

leg2Lines.push\_back("Weekend " + std::to\_string(totalWeeks) + ":");

}

const auto& match = fixtures[i];

std::string leg2Line = match.second.name + " (Home) vs " + match.first.name + " (Away),Leg 2," +

match.second.location + "," + match.second.stadium;

leg2Lines.push\_back(leg2Line);

}

outputLines.insert(outputLines.end(), leg1Lines.begin(), leg1Lines.end());

outputLines.insert(outputLines.end(), leg2Lines.begin(), leg2Lines.end());

writeFixturesToCSV(outputLines, "fixtures.csv");

}

* **Why we limited weekends**: We limit the weekends to 40, with 4 matches each weekend. This ensures we don’t overload the schedule, keeping it manageable.
* **Leg 1 and Leg 2**: For each match, there are two legs. Leg 1 is when the team plays at home, and Leg 2 is when they reverse roles and play away. This simulates how real fixtures are often organized.

**7. Main Function**

int main() {

std::string input\_filename = "teams.csv";

std::vector<Team> teams = loadTeamsFromCSV(input\_filename);

if (teams.empty()) {

std::cout << "No teams loaded. Please check the teams.csv file." << std::endl;

} else {

std::cout << "Teams loaded successfully:" << std::endl;

for (const auto& team : teams) {

std::cout << "Name: " << team.name << ", Location: " << team.location << ", Stadium: " << team.stadium << std::endl;

}

generateFixtures(teams);

}

return 0;

}

* **Main logic**: This is the entry point of the program. It loads the teams from the CSV file and checks if the teams were loaded correctly. If they were, it generates the fixtures. This function acts as the controller of the entire process.

In summary, the code is designed to handle the entire process of loading team data, generating balanced matchups, and saving the results to a file. The approach is modular, with separate functions to handle each task, which makes the code easier