C++ 프로그래밍및실습

사설 게임 기록분석

최종 보고서

제출일자: 2024/12/22
제출자명: 배재일
제출자학번: 215561

1. 프로젝트 목표 (16 pt)

1) 배경 및 필요성 (14 pt)

리그오브레전드(LoL)는 나와 비슷한 나이대의 사람들이 즐겨 플레이하는 인기 게임으로, 솔로 플레이어에게는 랭크전에서 즐거운 경험을 제공한다. 하지만 많은 플레이어들에게 친구들끼리 두팀으로 나누어 플레이하는 사설 게임이 진정한 리그오브레전드의 재미라고 여겨진다. 이러한 사설 게임에서는 팀원과 함께 전략을 구상하며 문제점을 피드백하며 정말 재미있는 경험을 공유할 수 있다. 이러한 사설 게임의 가치에도 불구하고, 리그오브레전드를 운영하는 Riot Games는 2021년 이후로 사설 경기의 전적을 제공하지 않고 있다.

게임 통계에 접근할 수 있는 것은 또 다른 즐거움을 더해 준다고 생각한다. 이는 플레이어가 자신의 성과를 확인하고 개선할 부분을 깨닫고, 친구들과 게임에 대해 여러가지 관점으로 이야기할 수 있게 해준다. 이러한 데이터가 없다면 플레이어들은 위와 같은 즐거움을 놓치게 된다. 이러한 문제를 해결하기 위해 이러한 프로그램을 구상했다.

2) 프로젝트 목표

본 프로젝트의 목표는 각 경기의 모든 플레이어의 전적을 기록하고, 이를 통해 개인과 팀 단위의 지표를 분석할 수 있는 프로그램을 목표로 한다.

3) 차별점

리그 오브 레전드에서 게임 전적을 기록하는 대부분의 기존 사이트는 공식 랭크 게임의 통계와 순위에만 초점을 맞추고 있다. 이러한 플랫폼은 공식 랭크 게임과 모든 플레이어가 포함된 공식 랭킹에 대한 정보를 제공하지만, 친구나 가까운 지인과 함께 플레이하는 사설 게임의 전적을 알려주지 않고 있다.

이 프로그램은 친구들 또는 지인과의 사설 게임의 전적을 기록하고 친구/지인 그룹에서 나의 위치를 파악할 수 있는 점에서 차별점이 있다.

2. 기능 계획

1) 기능 1: 경기 데이터 입력

- 설명: 각 경기의 모든 플레이어의 전적을 입력 받는 기능

(1) 세부 기능 1: 플레이어 정보 수집

- 설명: 각 경기마다 각 플레이어의 이름, 역할, 챔피언, 킬, 데스, 어시스트 수 등 등을 입력 받는다

2) 기능 2 : 지표 분석

-설명: 입력된 데이터에 대한 기본 통계 분석

(1) 세부 기능 1: 개인의 평균 지표 분석

- 설명: 모든 경기의 데이터에서 평균 킬, 데스, 어시스트 등 관련 데이터를 계산하여 출력한다.

(2) 세부 기능 2: 각각의 챔피언 별 지표 분석

- 설명: 입력된 경기 결과를 바탕으로 챔피언별 지표를 계산하고 출력한다.

3) 기능 3: 기록 조회

- 설명: 특정 플레이어의 경기 기록을 조회하는 기능

(1) 세부 기능 1: 플레이어 전적 조회 + 랭킹

설명: 사용자가 입력한 플레이어의 이름을 통해 해당 플레이어의 모든 경기 기록을 출력한다. 또한 각각의 랭킹을 조회한다

(2) 세부 기능 2: 특정 경기 분석

설명: 특정 경기에 참여한 모든 플레이어의 지표를 출력한다.

3.

1) 기능 구현

(1) 데이터 입력 기능

- 입출력 : match ID = 매치 이름 name = 이름, role = 포지션, champion =챔피언 ,kll= 킬, death= 데스, assist =어시스트 을 입력

- 설명 : 한 번에 10명의 플레이어에 대한 데이터를 입력하며 또한 다음 데이터를 연속으로 적을지 선택한다

- 적용된 배운 내용:

반복문 (do-while): 10명의 데이터를 입력할 수 있도록 do-while문을 사용했다

조건문 (if): 사용자가 매치를 계속 추가할지 말지를 결정하는 부분에서 if 조건문을 사용하여 'y'일 경우 계속 입력받고, 'n'일 경우 종료된다

- 코드 스크린샷

```
int main() {
    StatsTracker tracker;
    char moreData;

    do {
        // 한 줄로 모든 데이터 입력받기
        string inputLine;
        cout << "Enter match ID (e.g., match1, match2, etc.): ";
        string matchID;
        cin >> matchID;
        cin.ignore();

        // 10명의 데이터 입력
        for (int i = 1; i <= 10; ++i) {
            string playerName, role, champion;
            int kills, deaths, assists;

            cout << "Enter data for Player " << i << " (format: Name Role Champion Kills Deaths Assists): ";
            getline(cin, inputLine);

            stringstream ss(inputLine);
            ss >> playerName >> role >> champion >> kills >> deaths >> assists;

            // 매치 데이터 추가
            tracker.addMatch(Match(playerName, role, champion, kills, deaths, assists));
        }

        // 새 매치를 더 입력할지 여부 묻기
        cout << "Do you want to enter another match? (y/n): ";
        cin >> moreData;
        cin.ignore();

    } while (moreData == 'y');

    return 0;
}
```

(4) 챔피언별 승률 계산

- 입출력:

championName: 승률을 계산할 챔피언의 이름, matches :각 경기 데이터를 저장하고 있으며, 각 경기에는 해당 챔피언(champion)과 해당 경기에서의 승패(win) 정보가 포함됨,

해당 챔피언의 **승률(%)**이 반환됨.

- 설명: 각 경기에서 해당 챔피언이 승리했는지 확인하고, 승리한 경기는 totalWins를 증가시키고, 전체 경기는 totalMatches를 증가시킨후 totalMatches가 0이 아닌 경우 (totalWins / totalMatches) * 100을 계산하여 승률을 반환한다.

만약 해당 챔피언이 한 경기도 출전하지 않았으면 승률은 0.0%로 반환한다

- 적용된 배운 내용: for 반복문을 사용하여 matches 벡터에 저장된 모든 경기 데이터를 순차적으로 확인하고 if 조건문을 사용하여 각 경기에서 해당 챔피언 이름과 일치하는 경기만을 찾았다

```
// 챔피언별 승률 계산
double calculateWinRateByChampion(const string& championName) {
    int totalMatches = 0;
    int totalWins = 0;

    for (const auto& match : matches) {
        if (match.champion == championName) {
            totalMatches++;
            if (match.win) {
                totalWins++;
            }
        }
    }

    if (totalMatches == 0) return 0.0; // 해당 챔피언의 경기 기록이 없을 때
    return static_cast<double>(totalWins) / totalMatches * 100.0; // 승률(%)
}
```

(5) 플레이어의 전적조회

- 입출력:

playerName: 통계를 확인할 플레이어 이름.

matches: 경기 데이터를 저장한 벡터 totalKills, totalDeaths, totalAssists: 총 킬, 데

스, 어시스트 totalDamage: 총 데미지 totalPlaytime: 총 경기 시간을 저장한다.

matchCount: 플레이어가 참여한 경기 수를 저장

모든 경기에서 입력된 플레이어 이름을 찾아 통계를 모아서 킬, 데스, 어시스트, 데미지를 합치고, 참여한 경기 수를 세서 평균을 계산하여 결과로 평균 데미지와 평균 KDA를 반환한다. 만약 해당 챔피언이 한 경기도 출전하지 않았으면 승률은 0.0%로 반환한다

- 적용된 배운 내용: for 반복문을 사용하여 matches 벡터에 저장된 모든 경기 데이터를 순차적으로 확인하고 if 조건문을 사용하여 특정 플레이어의 데이터를 찾아내었다.

```
void displayPlayerStats(const string& playerName) const {
    int totalKills = 0, totalDeaths = 0, totalAssists = 0;
    int totalDamage = 0, totalPlaytime = 0;
    int matchCount = 0;

    for (const auto& match : matches) {
        auto processTeam = [&](const vector<PlayerStats>& team) {
            for (const auto& player : team) {
                if (player.playerName == playerName) {
                    totalKills += player.kills;
                    totalDeaths += player.deaths;
                    totalAssists += player.assists;
                    totalDamage += player.damage;
                    totalPlaytime += match.playtime;
                    matchCount++;
                }
            }
        };

        processTeam(match.blueTeam);
        processTeam(match.redTeam);
    }

    if (matchCount == 0) {
        cout << "No stats found for player: " << playerName << "\n";
        return;
    }

    double avgDamagePerMinute = totalPlaytime > 0 ? static_cast<double>(totalDamage) / totalPlaytime : 0.0;
    double avgKDA = totalDeaths == 0 ? totalKills + totalAssists : static_cast<double>(totalKills + totalAssists) / totalDeaths;

    cout << "\n[Player Statistics: " << playerName << "]\n";
    cout << "Matches Played: " << matchCount << "\n";
    cout << "Average Damage per Minute: " << avgDamagePerMinute << "\n";
    cout << "Average KDA: " << avgKDA << "\n";
}
```

(6) 챔피언별 승률조회

- 입출력:

championName: 승률을 계산할 챔피언의 이름. matches: 여러 경기 데이터를 저장하는 벡터

모든 경기를 확인하며 특정 챔피언이 출전했는지 찾아내어 출전한 경기 중 승리한 경기 수와 총 경기 수를 계산한다 승률(%)을 계산하여 출력합니다. 출전 기록이 없으면 "데이터 없음" 메시지를 출력.

- 적용된 배운 내용: for 반복문을 사용하여 matches 벡터에 저장된 모든 경기 데이터를 순차적으로 확인하고 if 조건문을 사용하여 특정 챔피언의 데이터를 찾아내었다.

```
    void displayChampionWinRate(const string& championName) const {
        int totalGames = 0, totalWins = 0;

        for (const auto& match : matches) {
            auto processTeam = [&](const vector<PlayerStats>& team, bool isBlueWin) {
                for (const auto& player : team) {
                    if (player.champion == championName) {
                        totalGames++;
                        if (isBlueWin) totalWins++;
                    }
                }
            };

            processTeam(match.blueTeam, match.blueTeamWin);
            processTeam(match.redTeam, !match.blueTeamWin);
        }

        if (totalGames == 0) {
            cout << "No stats found for champion: " << championName << "\n";
            return;
        }

        double winRate = static_cast<double>(totalWins) / totalGames * 100.0;
        cout << "\n[Champion Statistics: " << championName << "]\n";
        cout << "Win Rate: " << winRate << "%\n";
    }
};
```

)

(7) 여러가지 지표 계산

```cpp
// 이제 누적된 스탯 기반으로 평균 지표 계산
// (분당 데미지, 분당 CS, 분당 골드, 분당 시야점수, KDA, 킬관여율 등)
double avgDamagePerMinute   = (st.playtime > 0) ? static_cast<double>(st.damage) / st.playtime : 0.0;
double avgCSPerMinute       = (st.playtime > 0) ? static_cast<double>(st.cs)     / st.playtime : 0.0;
double avgGoldPerMinute     = (st.playtime > 0) ? static_cast<double>(st.gold)   / st.playtime : 0.0;
double avgVisionScorePerMin = (st.playtime > 0) ? static_cast<double>(st.visionScore) / st.playtime : 0.0;

// KDA
double avgKDA = (st.deaths == 0)
    ? (st.kills + st.assists)
    : static_cast<double>(st.kills + st.assists) / st.deaths;

// Kill Participation
double killParticipation = 0.0;
if (st.totalTeamKills > 0) {
    killParticipation = (static_cast<double>(st.kills + st.assists) / st.totalTeamKills) * 100.0;
}
```

- 입출력:

누적된 플레이어 통계 데이터 (AccumulatedStats 구조체)로부터 값을 입력받음.각 플레이어의 누적 킬, 어시스트, 데스, 데미지, CS, 골드, 시야 점수, 플레이 시간 출력: 계산된 분당 데미지, 분당 CS, 분당 골드, 분당 시야 점수, 평균 KDA, 킬 관여율(Kill Participation) 값을 반환.

플레이어의 누적 데이터(킬, 데스, 어시스트, 데미지 등)를 활용해 **평균 지표(KDA, 분당 데미지 등)**와 **킬 관여율(Kill Participation)**을 계산함.

- 적용된 배운 내용: -

for 반복문을 활용하여 matches 벡터에 저장된 모든 경기 데이터를 하나씩 확인하며, 각각의 경기에 대해 플레이어 데이터를 처리하고 이를 통해 모든 플레이어의 누적 데이터를 손쉽게 집계하고, 지표를 계산하는 기반을 마련했으며, if 조건문을 사용한 데이터 필터링if 조건문을 활용하여 특정 조건(예: 특정 챔피언, 플레이어 이름 등)에 해당하는 데이터를 찾아내고, .map을 활용한 데이터 누적 map<string, AccumulatedStats> 자료구조를 활용하여 플레이어 이름을 키(key)로 설정하고, 해당 플레이어의 모든 데이터를 누적함

(8) 개인 지표 랭킹

```cpp
map<string, AccumulatedStats> statMap; // <플레이어명, 누적스탯>

// 모든 Match를 순회하며 정보를 수집
for (const auto& match : matches) {
    int blueTeamKills = 0;
    for (auto &p : match.blueTeam) {
        blueTeamKills += p.kills;
    }
    int redTeamKills = 0;
    for (auto &p : match.redTeam) {
        redTeamKills += p.kills;
    }

    // Blue Team 누적
    for (const auto& player : match.blueTeam) {
        auto& entry = statMap[player.playerName];
        entry.kills       += player.kills;
        entry.deaths      += player.deaths;
        entry.assists     += player.assists;
        entry.damage      += player.damage;
        entry.cs          += player.cs;
        entry.gold        += player.gold;
        entry.visionScore += player.visionScore;
        entry.playtime    += match.playtime;
        entry.matchesPlayed += 1;
        entry.totalTeamKills += blueTeamKills;
    }

    // Red Team 누적
    for (const auto& player : match.redTeam) {
        auto& entry = statMap[player.playerName];
        entry.kills       += player.kills;
        entry.deaths      += player.deaths;
        entry.assists     += player.assists;
        entry.damage      += player.damage;
        entry.cs          += player.cs;
        entry.gold        += player.gold;
        entry.visionScore += player.visionScore;
        entry.playtime    += match.playtime;
        entry.matchesPlayed += 1;
```

```cpp
vector<pair<string, AccumulatedStats>> playerStatsVec(statMap.begin(), statMap.end());

auto getValueForMetric = [&](const AccumulatedStats& st) {
    switch (metricChoice) {
    case 1: // Total Kills
        return static_cast<double>(st.kills);
    case 2: // Total Damage
        return static_cast<double>(st.damage);
    case 3: // Average Damage per Minute
        if (st.playtime == 0) return 0.0;
        return static_cast<double>(st.damage) / st.playtime;
    case 4: // Average KDA
        if (st.deaths == 0) {
            return static_cast<double>(st.kills + st.assists);
        }
        return static_cast<double>(st.kills + st.assists) / st.deaths;
    case 5: // Average CS per Minute
        if (st.playtime == 0) return 0.0;
        return static_cast<double>(st.cs) / st.playtime;
    case 6: // Average Gold per Minute
        if (st.playtime == 0) return 0.0;
        return static_cast<double>(st.gold) / st.playtime;
    case 7: // Average Vision Score per Minute
        if (st.playtime == 0) return 0.0;
        return static_cast<double>(st.visionScore) / st.playtime;
    case 8: // Kill Participation (%)
        if (st.totalTeamKills == 0) return 0.0;
        return (static_cast<double>(st.kills + st.assists) / st.totalTeamKills) * 100.0;
    default: // 잘못된 입력 시 kills 기준
        return static_cast<double>(st.kills);
    }
};
```

```cpp
// 내림차순 정렬(높은 값이 1등)
sort(playerStatsVec.begin(), playerStatsVec.end(),
     [&](auto& a, auto& b) {
         return getValueForMetric(a.second) > getValueForMetric(b.second);
     });

// 출력
cout << "\n[Player Ranking]\n";
cout << left << setw(5) << "Rank"
     << left << setw(15) << "Player"
     << left << setw(10) << "Matches"
     << left << setw(15) << "MetricValue"
     << endl;
cout << "-----------------------------------------------------------\n";

int rank = 1;
for (auto& [playerName, st] : playerStatsVec) {
    double val = getValueForMetric(st);
    cout << left << setw(5) << rank
         << left << setw(15) << playerName
         << left << setw(10) << st.matchesPlayed
         << left << setw(15) << val
         << endl;
    rank++;
}
```

- 입출력: 순위를 매길 지표 선택 (예: 총 킬, 평균 KDA, 분당 CS 등) 출력값: 선택한 지표 기준으로 정렬된 상위 플레이어 목록

std::map을 통해 각 플레이어의 누적 데이터를 저장. 각 경기 데이터를 순회하며 팀별 킬 수를 계산하고, 해당 데이터를 std::map에 추가시킨후 std::vector로 변환 후, std::sort와 람다 함수를 사용해 선택한 지표 기준으로 정렬하여 그 데이터를 기준으로 순위를 매김

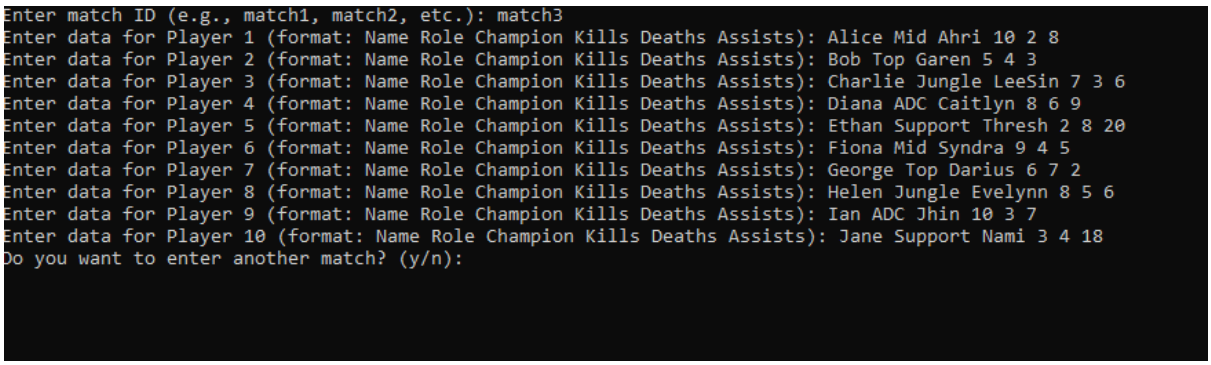
-적용된 배운 내용 : for 반복문을 사용하여 플레이어 데이터를 반복적으로 입력받아 vector<PlayerStats>에 저장. 벡터를 통해 플레이어 데이터를 효율적으로 관리하고, 이후 분석에서 손쉽게 접근 가능하게 했으며 PlayerStats 클래스와 Match 클래스를 활용해 데이터를 객체로 관리했다.

2) 테스트 결과

(1) 데이터 입력 기능

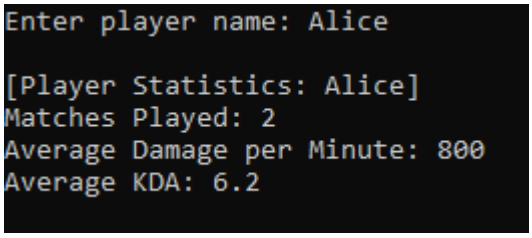
- 설명: 한 번에 10명의 플레이어에 대한 데이터를 입력하며 또한 다음 데이터를 연속으로 적을지 선택하는 기능

- 테스트 결과 스크린샷



(2) 플레이어의 전적조회

- 설명: 특정 플레이어의 kda, dpm등을 계산하여 보여줌



(3) 챔피언별 승률 조회

- 특정 챔피언의 승률을 조회한다

```
Choose an option: 3
Enter champion name: Ahri

[Champion Statistics: Ahri]
Win Rate: 50%
```

(4) 개인 지표 랭킹

- 각 지표마다 플레이어들의 랭킹을 보여줌

```
Choose an option: 6

[Ranking Menu]
1. Total Kills
2. Total Damage
3. Average Damage per Minute
4. Average KDA
5. Average CS per Minute
6. Average Gold per Minute
7. Average Vision Score per Minute
8. Kill Participation (%)
Choose a metric for ranking: 1

[Player Ranking]
Rank Player         Matches   MetricValue
------------------------------------------------------------
1    Henry          10        68
2    Bob            10        59
3    Charlie        10        59
4    David          10        58
5    Frank          10        52
6    Ginny          10        52
7    Isaac          10        48
8    Alice          10        30
9    Erin           10        14
10   Jane           10        10
```

4. 계획 대비 변경 사항

1) 기능2

- 이전 ~12/1

- 이후 ~12/15

- 사유: 지표를 호출하는 여러가지 함수를 만들었지만 이것을 정확하게 테스트 하기위해서는 기능3을 구현하는 과정에서 한번에 하는 편이 용이하다고 판단함

2) 지표분석

- 이전 원래는 **각각의 지표(예: 평균 KDA, 평균 분당 데미지)**를 계산하기 위해 별도의 함수를 만들어서 사용할 계획이었음.

calculateAverageDamagePerMinute() calculateAverageKDA()

와 같은 함수들을 각각 구현하고 호출할 계획이었음.

- 이후 지표를 보여주는 함수(displayPlayerStats) 안에서 각 지표를 바로 계산하고 처리하는 방식으로 변경함.

- 사유: 별도의 함수를 만들면 각 함수가 for 반복문을 사용하여 matches를 순회해야 함. 이 경우 여러 번 반복문이 실행되므로 비효율적일거라고 생각했으며 하나의 함수함수 작업하는 것이 더 직관적일거라고 생각하였음

3) 개인 지표 랭킹

- 이전 X

- 이후 O

- 사유 원래목적인 플레이어들의 비교를 하기위해 추가함

6. 느낀점

우선 이 프로젝트를 진행하면서 실제 데이터를 다루는 프로그램을 설계하고 구현하는 과정에서 다양한 어려움을 경험했다고 생각합니다 특히, 복잡한 데이터를 효율적으로 관리하고 분석하기 위해 적합한 데이터 구조를 설계하는 부분에서 어떤 방식을 써야하고 플레이어와 경기에 대한 정보를 어떻게 체계적으로 저장하고, 이후 분석 및 출력 과정에서 오류 또는 충돌 없이 효율적으로 사용할 수 있을지 설계하는 것이 쉽게 떠오르지 않았던 것 같습니다.

또한, 데이터를 분석하면서 발생할 수 있는 예외 처리나 오류 등을 방지하기 위위 고생했던 것 같습니다. 예를 들어, 플레이 시간이 0이거나, 특정 값이 비정상적으로 입력되는 경우를 처리하기 위해 많은 시행착오를 겪었으며 이러한 문제를 해결하기 위해 조건문과 반복문을 적극적으로 활용했지만, 디버깅 과정에서 예상치 못한 입력이나 계산 오류를 발견하고 수정해야 했을 때는 교수님께서 그때 그때 테스트 해보라는 말씀이 생각났던 기억이 납니다.

특히 랭킹 기능을 구현하는 과정에서는 특정 지표에 따라 데이터를 정렬하고, 이를 출력 형식으로 정리하는 작업이 생각보다 복잡했습니다. 특히, 다양한 지표(KDA, 분당 데미지, 킬 관여율 등)를 동적으로 계산하고, 이를 기준으로 데이터를 정렬하는 로직을 설계하는 데 있어 많은 시간이 소요되었습니다. std::map과 std::vector 같은 자료구조를 활용해 데이터를 관리하는 데 익숙하지 않아 초기에는 많은 어려움을 겪었지만, 이를 해결하면서 표준 라이브러리의 유용함을 느낄 수 있었습니다.

이러한 경험들을 겪으면서 프로그램을 만들 수 있었으며 비록 자의가 아닌 과제로 하게 되었지만 나만의 프로그램을 만든다는 과정이 저에게는 흥미로웠고 이렇게 해도 될까 저렇게 해도 될까 되게 고민이 많았던 과정이기도 했습니다. 지금 이 순간에는 이런 경험을 하게 되어 나름 뿌듯하고 이 강의가 아니었다면 제가 스스로 프로그램을 만들지 않았을 거라고 생각하기에 매우 고마움을 느낍니다