STL 과제 보고서 - 2021184018

한국공학대학교 게임공학과 3학년 STL 과제 보고서

1. 파일 읽기 및 컨테이너 저장 방식

std::ifstream을 통해 바이너리 파일 "2025 STL 과제 파일 - 2021184018"을 열고, Player::read() 멤버 함수를 이용하여 std::array<Player, 2'500'000>에 모든 데이터를 한 번에 읽었다.

바이너리 파일 구조에 맞게 Player 객체의 기본 정보(name, score, id, num)를 먼저 읽고

그 이후에 unique\_ptr<char[]> p로 관리되는 메모리 블록을 num 바이트만큼 추가로 읽는다.

반복문을 통해 250만 개 객체에 대해 각각의 멤버함수read()를 호출하여 저장하였다.

마지막 Player 객체는 Show() 함수를 통해 다음과 같이 출력되었다.



2. 최고 점수와 평균 점수 계산

모든 Player의 점수를 비교하는 람다 함수를 사용하여 max\_element를 사용해 최댓 값을 구했다.

처음에는 scoreMap에 저장하는 방식을 사용하였으나

4번 문항에서 원본 배열을 한번 정렬할 때, 복사하지 않고 참조를 한 scoreMap의 내부 값이 꼬여서 다시 만들어야 하는 문제가 있었다.

이 경우 O(nlogn)만큼 걸리는 std::map의 생성을 한번 더 하는 건 비효율적이라고 생각했기 때문에, max\_element로 구하는 것이 낫다고 판단했다.

평균 점수는 std::accumulate를 사용하여 operator+= 연산자 오버로딩으로 모든 Player의 score를 더해 계산하였다.

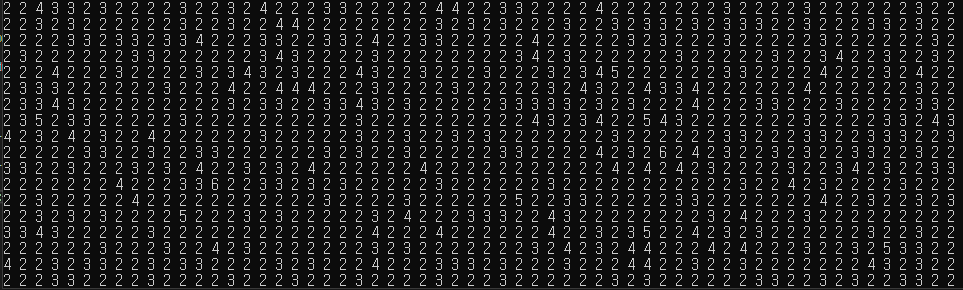


3. 같은 id를 가진 Player 탐색 및 저장

모든 Player를 id 기준으로 정렬한 후, 인접한 두 객체의 id가 같은 경우를 탐색하여 "같은아이디.txt"에 출력했다. 중복되는 id의 개수는 2개부터 전부 화면에 한 칸 간격으로 계속 출력되게 하였다.

중복되는 id의 개수들을 모두 합한다는 생각도 해보았으나

각 id 별로 얼마나 있는지를 출력하는 것이 문제와 더 부합한다고 생각하였다.



4. p가 가리키는 문자열 정렬 및 'a' 개수 기준 필터링

Player::getBuffer()를 이용해 unique\_ptr<char[]>가 가리키는 문자열을 std::string\_view로 받아 정렬에 사용하였다. 이후 std::count를 통해 'a'가 10개 이상 포함된 Player 객체의 수를 집계하였다.

초기에는 getBuffer의 반환 타입을 std::pair로 반환해서 문제를 해결하였는데,

string\_view로 반환하였을 때 수정은 불가능한 읽기 전용 이면서 pair에 비해 코드도 더 간결해졌고, 그래서 string\_view로 반환하였다.

출력 결과는 다음과 같다.



5. id 기반 검색 및 3가지 기준 탐색

이 문제에서는 2가지 std::map을 만들었다.

1. id를 저장하는 idmap
2. score를 저장하는 scoreMap

map을 사용하기로 결정한 이유는 다음과 같았다.

1. 생성의 결과가 동일한 값들의 본래 순서가 보장되는 stable한 정렬이 된다는 점
2. 탐색의 시간 복잡도가 O(logn)으로 array, vector에 비해 빠르다는 점
3. map의 생성 속도가 O(nlogn)으로, array, vector등의 정렬과 같은 시간 복잡도를 가지고 있다는 점
4. 중복되는 id들을 하나로 묶어서 관리할 수 있다는 점

초기에는 unordered\_map을 사용하고자 했지만,

1. 탐색 성능이O(1)로서 매우 빠른 성능을 보장하지만 정렬되지 않았기 때문에 “정렬된 기준”으로 자신의 앞, 뒤 값을 찾아내는 과정이 번거롭고, 그 방법이 알맞을 것이라는 확신이 없었다는 점
2. 탐색하는 코드도 복잡했다는 점

이 두가지 사유로 인해 사용하지 않기로 했고,

같은 id를 가진 Player들을 하나로 묶어서 관리하고자 했기에

multimap은 사용하지 않았다.

모든 map은 Player를 담을 수 있는 vector로 이루어진 value를 가지게 하였다

* 중복되는 id들을 하나로 묶어 관리하기 위해 vector를 사용하였고,

idmap의 contains 함수를 사용하여 id가 존재하는지 검사하였다.

* contains 함수는 검색을 통해 해당 id가 map에 존재한다면 true, 아니면 false를 리턴하는 함수이다.

일반 std::find를 사용하려는 생각도 했지만

find는 어떤 container를 받아도 반복자를 통해서 동작하도록 만들어진 함수인데

O(logn)으로 찾을 수 있는 map의 장점을 살려내지 못하기 때문에

map이 가지고 있는 contains함수를 사용하였다.

또한, id를 찾을 때는 최초로 찾아낸 id만을 기준으로 한다

일치하는 id가 있을 때의 출력 방식은 다음과 같다

1. id 기준 앞 뒤 원소 출력
2. name 기준 앞 뒤 원소 출력
3. Score 기준 앞 뒤 출력

<id 기준 앞 뒤 원소 출력>

id기준 앞, 뒤 원소를 출력할 때,

id의 바로 앞은 바로 앞 id값의 맨 뒤 원소이고,

바로 뒤는 그 id 값의 vector바로 다음 원소

혹은 뒤 id 값의 맨 앞 원소이다

동일한 id는 전부 출력하기 때문에

앞, 뒤로 찾은 값이 속한 vector의 모든 원소를 출력한다

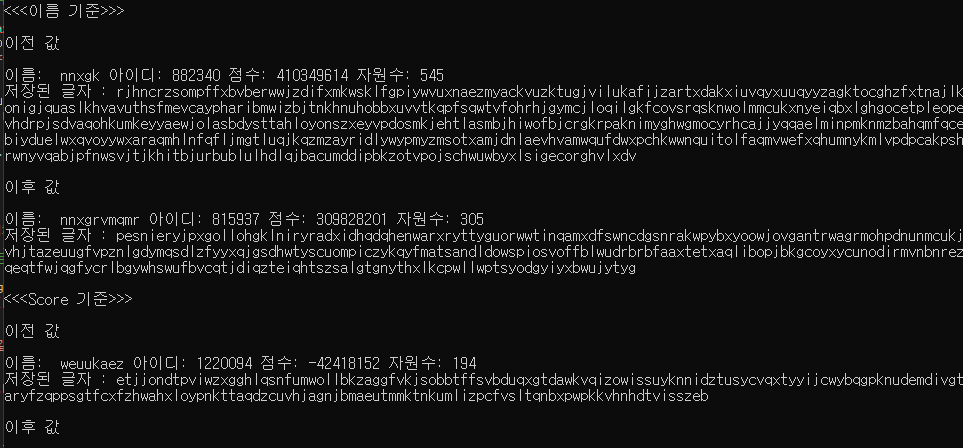
|  |  |  |  |
| --- | --- | --- | --- |
| prev |  |  |  |

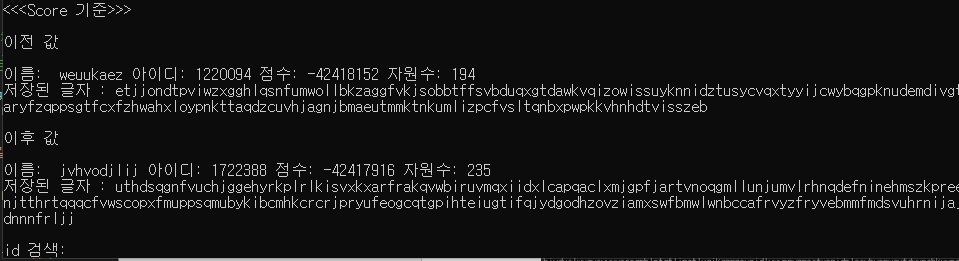
|  |  |  |  |
| --- | --- | --- | --- |
| id |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| next |  |  |  |

특히, 배열 내 포인터 차를 이용해 &player - players.data()를 통해 반복자 없이 인덱스를 계산하였고, STL 정렬 함수 및 std::map, std::vector<ref>를 적절히 혼합하여 탐색 성능을 확보하였다









효율성 평가 및 개선점

전체 데이터는 고정 크기의 std::array로 관리되므로, 반복적인 할당과정이 없고 메모리 오버헤드도 낮다. map<int, vector<ref>>를 활용한 그룹핑은 중복 탐색을 효율적으로 수행할 수 있게 해주며, 정렬 후 탐색에서는 std::prev, std::next를 적극 활용하여 포인터 및 반복자 연산을 잘 사용하였다.

개선 가능한 점:  
- std::cin 입력 실패 처리 추가 필요  
- Answer5에서 ++++itr처럼 비직관적인 표현보다는 std::next(itr, 2)가 더 가독성이 좋음  
- Player의 read()에서 name.data()에 바로 읽는 건 std::string의 정의상 안전하지 않음 (UB 가능)

과제를 하면서 느낀 점

이번 STL 과제를 통해 std::array, std::map, std::vector, unique\_ptr, string\_view, accumulate, sort, count 등 다양한 STL 기능을 실제 대규모 데이터에 적용해보는 경험을 할 수 있었다. 특히 바이너리 파일 입출력과 메모리 할당을 직접 다뤄보면서 C++의 자원 관리 기법과 STL의 반복자 기반 설계의 장점을 체감할 수 있었다. 250만 개 객체를 대상으로 수행된 고효율 데이터 처리 경험은 향후 게임 개발이나 실시간 시스템에서 큰 도움이 될 것이라 생각한다.