2023 STL 과제 보고서

게임공학과

2019182021

신민근

[과제 해결 방법]

- 파일 읽기

먼저 파일을 읽기 위해 바이너리 모드로 읽어와야 하니 해당 파일을 바이너리 모드로 열어 읽을 준비를 했다. 이후 파일을 읽기위해 ifstream객체의 read를 이용해 플레이어 객체의 수만큼 Player array에 읽어오면 쉽게 해결될 줄 알았다. 그런데 잘 되지 않아 살펴보니 이 파일이 작성될 때 write라는 함수로 작성이 되었고 객체 자신과 멤버변수 p가 가리키고 있던 내용을 write로 작성한 것을 깨달았다. 그래서 이 파일을 읽어 올 때 한 번에 읽어 올 것이 아니라 Player 객체 하나를 읽어 오고 그 다음 저장된 Player 객체의 p에 내용을 num개수만큼 동적할당을 통해 읽어오고 다음 객체를 읽는 방식으로 매번 반복하는 식으로 바꿔야 했다.

- 1번

array에 정보가 저장이 되었고 array의 크기도 알고 있으니 마지막 원소의 위치를 접근하는 것은 쉬웠다. 그리고 format을 이용해 출력하는 print() 멤버 함수를 만들어 마지막 객체의 정보를 출력하였다.

출력결과

1번 문제

이름:wuhwock , 아이디:2002867, 점수:8983734, 자원수:282

저장된글자:pztqolhqtnawyytaxhjugxzcjwcvvclwrtobgrxbqybnufcrkyuvtdjisleufptomrsqcgfbcxkizkixofzvtxhwzqvgzpiewpbvyzaannnzxkieofpnbnkuvclgxnekshvipzqtuahhotjdscspthopfrrjumpwhtipainyhvavhnsrlilldconiywmwrblkueajqbsmjrqaapxwinftrxidrsytupblyhquadugxxenschypnrrurzhigkoupasdkaeubxmzkawcnwevutazftrm

- 2번

처음에는 가장 큰 점수를 찾는 것을 알고리즘의 함수 max\_element를 이용하고 평균을 구하기 위해 점수의 합계를 구하는 것을 numeric헤더의 accumulate를 통해 해결하려 하였다. 그런데 이러한 방식은 array를 총 두 번 훑어보게 되는 것 같아 한번 훑어보되 점수의 합계와 최대값을 한번에 찾는 것이 더 효율적일 것 같아 직접 구하는 방식으로 바꾸었다. 시간의 차이가 나는지 확인해 보았는데 정확한지는 모르겠지만 평균적으로 3밀리초 정도차이가 나는데 이정도 차이면 큰 차이 같지는 않았다.

그러다 문득 max\_element를 사용하여 가장 큰 값을 구하면서 람다함수 내에서 sum에다가 하나씩 더해주어 한 번에 두가지 방식을 해결하는 것을 생각해서 구현해보았다. 이러한 방식으로 구현하니 속도는 직접 구현한 방식과 거의 똑같았다. 그리고 2번을 풀면서 생각해생 될 것 중 하나는 sum의 값의 자료형을 long long으로 해야지 오버플로우를 방지할 수 있었다.

출력결과

2번 문제

가장 큰 점수의 Player

이름:strbwah , 아이디:1911698, 점수:423456369, 자원수:293

저장된글자:pbfawcpcfsbrcjwftqsrcmbwlspqrzjsoastkehwjmwdiujppsrvlbmemrdsrphdefzbspreyqjnmyqumsdvnqasphxxuyndagfjpbmnobvqovsntmpfwptklqyaihmfnnjzjwzjjfcqtsqtpvgeldjyfcaiftuydeklisiixhqhrfqejvllkvfcbbzroibhuzhvdowstvolkuhbaasorjpodnzjmtvhugkezspafiwhpbibmguxzlmamxinwguvztddupqawsplygakfjrsfuuiyrnbwlrxjoqtx

모든 Player의 평균 점수:123468028

- 3번

3번의 경우 id가 서로 같은 객체를 찾아 텍스트 파일에 입력해야 하는데 정렬을 하지 않고 같은 id를 찾으려면 매번 id에 대해 array의 끝까지 비교해봐야 한다고 생각이 들어 id에 대해서 먼저 정렬을 시켰다.

Player의 소멸자에서 p에 대해서 delete[]를 시키고 있는데 sort를 시킬 경우 만약 복사생성자/이동생성자와 복사할당연산자/이동할당연산자에 대해 따로 정의해 두지 않는다면 sort시 소멸자가 발생하여 프로그램이 죽게 된다. 또한 복사생성자와 복사할당연산자에 대해서만 정의 해 놓아도 sort를 잘 돌릴 수 있지만 시간이 7초 정도 소요되는데 비해 이동생성자와 이동할당연산자를 통해 sort를 한다면 1초정도만에 정렬을 수행한다는 것을 통해서 차이가 매우 크다는 것을 알 수 있었다.

이제 문제로 돌아가서 정렬을 시킨 후 이제 파일 쓰기를 준비한다. 그리고 어떤 알고리즘을 쓸 수 있을까에 대해 생각해보았는데 일단 생각이 나지 않아 for문을 돌리기로 했다. 먼저 정렬을 시켰으니 array의 첫번째 인덱스의 id값을 이전 id값으로 저장하고 firstIndex라는 같은 id에 대한 첫 번째 인덱스번째를 0으로 설정해서 값을 비교해나가기 시작한다. 만약 id가 같으면 카운트를 늘려가면서 다른 id를 찾을 때까지 카운트를 늘려간다. 이후 다른 id가 나오게 되었을 때 카운트의 수가 2개 이상이라면 저장했던 firstIndex를 가져와서 같은 id의 카운트만큼 파일에다가 작성한다. 이때 모든 같은 id의 개수(allSameNum)를 같이 올려준다. 이후 카운트를 1로 초기화 시켜주고 시작 인덱스도 현재 다른id의 인덱스로 바꿔준 뒤에 끝까지 반복한다. 이때 만약 for문의 i가 마지막 인덱스를 가리키고 있다면 카운트에 따라 똑같은 처리를 한 뒤에 루프를 마친다. 이후allSameNum을 출력하고 3번을 끝냈다.

최대한 알고리즘을 찾아서 풀 방법을 생각하던 중에 나는 정렬을 시켰으니 find\_if를 쓰는 것이 좋을 것 같다는 생각이 들었다. 이전 iterator를 선언하고 이전 iterator부터 array의 end()까지 이전 iterator의 ID와 다른 것을 찾을 때까지 find\_if를 돌려서 다른 id를 가지는 iterator를 받는다. 그리고 이전 iteartor와 find\_if를 통해서 반환 받은 iterator사이의 distance를 구해 만약 1보다 크다면 이전 iterator부터 시작하는 for문을 통해 find\_if를 통해 구한 iterator전까지 출력을 시키고 동시에 모든 같은 id의 개수(allSameNum)를 1씩 추가한다. 그리고 이전 iterator의 값을 find\_if로 반환 받은 iterator로 바꾸고 find\_if가 반환한 iterator가 array의 end()일때까지 루프를 돌린다.

출력 결과

3번 문제

ID가 같은 모든 객체의 수:1149459

- 4번

4번의 경우 일단 array의 모든 객체에 대해 p를 정렬해야 했기에 그 부분에 대해서 먼저 간단한 for문과 sort를 이용해 구현했다. 이후에 ‘a’의 개수를 세는 것이 문제였는데 만약 count함수를 사용하게 된다면 간단하게 해결이 되지만 문제는 p가 정렬이 되어있기 때문에 ‘a’가 더 이상 없게되면 ‘a’의 개수에 대해서 셀 필요가 없어져 필요없는 연산을 추가적으로 한다는 문제가 있었다. 그래서 일단은 직접 for문을 돌려서 ‘a’가 아니라면 즉시 break시켜 10개 이상인 경우에 대해 개수를 추가시키는 방식으로 구현했다.

그러다 이것을 알고리즘으로 해결할 방법이 없을까를 생각하다 보니 정렬이 되어있으니 find\_if를 이용해서 ‘a’가 아닌 지점의 iterator를 받아오고 p의 시작점에서 해당 iterator의 거리를 distance로 구하면 되겠다는 생각이 들었다. 그래서 해보니 더 짧은 코드를 작성할 수 있었다.

출력 결과

4번 문제

a가 10개 이상인 Player의 수:983062

- 5번

먼저 기본적으로 3번의 정렬을 해야만 하는데 이 부분을 피할 방법이 있을까에 대해 생각해보았다. 그러나 array로 만들어 놓았고 추가적인 Plyer객체를 만들 수 없기에 3번의 정렬을 거치는 것은 피할 수 없다고 생각했다. 그렇게 생각을 한 후 기본적인 id 입력에 대해서 받고 만약 숫자가 아닌 것을 입력하면 프로그램이 종료되게 설정하였다. id입력시에는 id에 대해서 먼저 정렬을 수행한다. 이후 find\_if를 통해 찾는 id의 첫번째 iterator를 받아오게 한다. 만약 받아온 iterator가 array의 end()와 같다면 찾지 못한 것이니 continue를 시켜 다시 입력을 받도록 시킨다. 다르다면 iterator – 1과 iterator, iterator + 1에 대해서 출력을 시키고 iterator와 iterator+1의 id를 비교하여 같다면 iterator를 +1하여 출력을 반복한다. 만약 iterator와 iterator+1 다르거나 iterator + 1이 array의 end()와 같다면 루프를 빠져나온다. 이때 주의할 점으로는 처음 find\_if를 통해 받은 iterator가 array의 begin()과 같은지를 비교해서 같다면 그전의 정보의 대해서는 출력을 하지 않도록 시켜야한다. 또는 iterator+1이 array의 end()와 같다면 그 다음의 정보에 대해서는 출력을 하지 않도록 만들고 다음 id와 비교하는 루프를 돌지 않도록 해야한다.

그런데 이렇게 만들고 나서 find\_if를 쓰는 것이 비효율적이라고 생각이 들었다. 그 이유는 id에 대해 정렬을 시켰기 때문에 이제는 이진 탐색을 할 수가 있었기 때문이다. 그래서 처음에는 그냥 binary\_search를 사용할지에 대해 생각했지만 bool값을 리턴하기에 이전 정보와 뒤의 정보를 가져오는데에는 적합하지 않다고 생각했다. 그래서 iterator를 반환하는 이진탐색인 lower\_bound를 사용하기로 하였다. lower\_bound를 사용하여 검색의 효율를 조금은 높일 수 있었는데 이렇게 바꾸게 되면 기존에 있던 코드를 조금 바꿔야 했다. 처음 id정렬을 할 때 find\_if의 경우에는 같은 id에 대해 무조건 첫번째 iterator를 주기 때문에 다음 id에 대해 비교해 나가면 되었지만 lower\_bound의 경우에는 같은 id에 대해 첫번째를 준다는 보장이 없기에 앞과 뒤로 같은 id가 있는지를 비교해 나가야 했다. 그래서 lower\_bound로 받아온 iterator를 통해 앞으로 비교해나가 first iterator를 찾고 뒤를 비교해 나가 last iterator를 찾아서 for문을 통해 first부터 last까지 출력시키도록 하였다. 이때 first가 array의 begin()이 아니라면 first-1을 출력하고 last + 1이 array의 end()가 아니라면 last + 1을 출력하도록 해줘야한다. 그리고 lower\_bound를 쓰게되면 id를 찾지 못하게 되면 찾는 id에 가장 가까운 큰 값의 iterator를 반환하기 때문에 반환된 iterator가 array의 end()인지를 비교해 아니여야하며 iterator의 id값과 입력한 id값이 같아야 정보출력를 출력하고 둘 중 하나를 만족하지 못하면 다음루프로 이동해 다시 id값을 받게 해야 한다.

다음으로는 name정렬 이후에 id에 대해서 출력하는 부분이 문제였는데 id에 대한 정렬이 된 것이 아니기 때문에 이진 탐색을 쓰기에는 적합하지 않다고 생각을 했다. 그래서 이전에 id를 찾은 것들의 name을 따로 저장해서 name에 대해서 탐색을 할까에 대해 생각해보았지만 name들도 중복된 것이 있어 이 방법은 잘못되었다고 생각이 들었다. 그래서 결국 find\_if를 통해 탐색하기로 하였다. 그래서 find\_if를 통해 탐색을 하는데 begin부터 시작해서 탐색을 할 때 이제 찾은 첫번째 id에대한 iterator를 넘겨주니 다음 id를 찾기 위해서는 찾은 iteartor의 다음을 시작으로 검색을 돌려받는 iterator가 array의 end()일때까지 반복한다. 그리고 iteartor를 찾았을 때마다 앞, 현재, 뒤에대해서 출력을 시켜주고 출력전에 똑같이 앞과 뒤가 존재하는지도 체크를 해봐야한다. score정렬 부분 또한 name정렬부분과 동일하게 진행되게 하였다.