STL 과제 보고서

2018180009 김시인

\*소스 코드 실행환경 : Visual Studio 2022, Release/x64, std::c++latest

저는 처음에 왜 교수님께서 하필 데이터를 바이너리파일로 만들었는지 이해하지 못했습니다.

하지만 파일을 읽어오고 강의노트를 다시한번 훑어보면서 그 이유를 알게 되었습니다.

>>연산자의 경우 int형을 받을 경우 문자를 숫자로 바꾸고 띄어쓰기도 체크하는 등 고등적인 연산을 하기 때문에 시간이 오래 걸립니다. 거기에 비해 바이너리 파일은 read()함수를 통해 CUP를 거치치 않고 (DMA)메모리에 직접 액세스하여 >>보다 훨씬 빠르게 작동합니다.

교수님께서 제공하시 “2022 STL 과제 파일”이 바이너리 파일이고 read()함수를 통해 읽었음에도 약 1.6초가 걸렸을 것을 감안하면 >> 연산을 통해 읽어올 경우 매우 오래 걸릴 것을 예상할 수 있고, 왜 교수님께서 바이너리 파일로 데이터를 주셨는지 이해할 수 있었습니다.

그리고 일반적으로 바이너리파일이 텍스트파일에 비해 용량이 적다는 점 또한 한몫한 것으로 생각됩니다.

사실 처음에 read를 사용했는데 오류가 발생하여 한참을 고민했습니다. 허무하게도 오류가 났던 이유는 Release모드가 아니라 Debug모드로 돌려서 에러가 났습니다. 왜 Debug모드는 에러가 났는지 처음에는 몰랐는데 sizeof(Player)를 해보고 나서 깨달았습니다.

Debug/x64모드의 경우 Player자료형은 64Byte의 메모리를 차지합니다. (Release의 경우 56Byte )

디버그 모드로 실행 시 디버그에 필요한 정보를 추가로 할당하기 때문에 size가 달라지게 되어 read를 했을 때 오류가 나게 되었습니다. 이번 과제를 하면서 이런 일을 겪음으로써 앞으로 다시는 이런 실수를 하지 않을 것 같습니다.

파일을 읽어올 때 Player의 데이터와 다음 Player의 데이터사이에 Player가 확보했던 데이터들이 저장되어 있는데 그 크기가 가변적이기 때문에 어떻게 읽어오면 좋을지 많이 고민했습니다.

여러가지 방식을 고민해본 끝에 수업에서 STRING.h에서 했던 방식을 응용하여 Player의 num만큼 새로 메모리를 할당하여, 할당한 곳에 read()를 통해 할당한 크기만큼 고속으로 읽어오는 방식을 사용했습니다.

2022.04.7 +) Player의 멤버변수를 public으로 쓰면서 main함수에서 player의 데이터를 만들어줬는데, 클래스의 변수를 private로 해야 된다는 것을 알고 Player의 멤버변수를 private안에 숨기고 따로 read함수를 작성하여 파일을 읽어오는 함수를 만들었습니다. 추가로 멤버변수를 읽어오는 get함수들을 추가했습니다. Name의 경우 메모리가 커서 복사생성할 경우 오래걸릴 수 있기 때문에 반환타입이 const string&인 getNameRef()함수로 만들었습니다.

저 같은 경우 데이터를 전역 변수에 있는 array<Player, 2’000’000> 자료형에 저장하였 습니다.

그렇게 한 이유는 첫번째로 자료의 개수가 정확히200만개로 정해져 있으면 추가되거나 삭제되지 않는다는 조건이 있기 때문입니다. 그리고 전역으로 선언한 이유는 함수(main포함) 내에서 배열크기가 큰 배열을 선언할 수 없기 때문입니다.

큰 배열을 선언할 수 없는(안되는) 이유 [[출처]](https://ko.wikibooks.org/wiki/C_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D_%EC%9E%85%EB%AC%B8/%EB%8D%B0%EC%9D%B4%ED%84%B0_%EB%B0%B0%EC%97%B4)[[1]](#footnote-1) :

* 함수에서 크기가 큰 배열을 선언할 경우 함수가 호출될 때마다 stack에 큰 크기의 메모리를 잡았다가 해제해야 하기 때문에 프로그램의 속도를 매우 느리게 만들 수 있다.
* 이론적으로 스택 지역에 변수가 저장되는 영역의 크기는 제한되지 않지만 시스템에 따라 하드웨어적인 제한이나 스택 운영의 효율을 높이기 위해 스택의 크기를 제한하기도 한다.

여기서 생각해 볼 것은 만약 데이터가 추가되거나 삭제될 수 있는 경우인데 그 경우 vector를 사용하면 될 것 같다고 생각했습니다. 사용할 때 주의할 점으로 vector에 자료를 push\_back할 때 vector의 capacity를 초과할 경우 새로 메모리를 할당하고 복사하는 과정을 거치면서 추가적인 시간이 듭니다. Capacity가 0인 상황에서 데이터를 200만개를 넣을 경우 넣는 과정에서 수십 번 메모리를 재할당하는 경우가 생기기 때문에, vector의 reserve()함수를 통해 처음부터 200만의 capacity를 할당해 놓고 데이터를 읽어오면 훨씬 효율적이라고 생각을 했습니다.

2022.04.7 +) emplace\_back을 배우고 난 후 puch\_back보단 emplace\_back을 사용하는 것이 이득인거 같습니다.

문제 1번과 2번의 출력결과(그림) :

텍스트이(가) 표시된 사진

자동 생성된 설명

문제 1번 출력결과(text):

**이름:zlzsdt , 아이디:1785636 , 점수:210159520, 자원수:582**

1번 문제의 경우 STL에서 배운 format을 써 보면서 array<>의 back()함수를 이용하여 쉽게 출력할 수 있었습니다.

문제 2번 출력결과(text):

**합계:600146737558800, 평균300073368.7794**

2번 문제의 경우 numeric에 있는 accumulate함수를 이용하여 풀었습니다.

이 문제를 풀면서 중요하게 생각했던 점은 두 가지입니다.

첫번째는 오버플로우의 가능성 입니다. 합계를 구하게 될 경우 값이 대략적으로

(-21억 \* 200만) ~ (+21억 \* 200만) 의 범위를 가지게 됩니다. (score가 int이며 음수일 가능까지 생각하여)

즉 int가 나타낼 수 있는 수의 범위를 초과할 수 있기 때문에 long long 써줘야 합니다.

두번째는 람다입니다.

accumulate함수를 이용해서 합을 구하려고 할 때 Player의 어떤 멤버번수를 사용할지 컴퓨터는 알 수 없기 때문에 사용자가 알려줘야 합니다. 이때 람다 함수를 이용했습니다.

Accumulate에 제공할 함수를 다른 곳에서 재사용할 일이 없으며 코드가 짧기 때문에 함수를 따로 정의하여 넘겨주기보다는 람다를 사용하는 것이 가독성 면에서 이득이 될 것이라고 생각했습니다. 람다를 작성할 때도 주의할 점이 인자로 Player가 아니라const Player& 로 받았다는 점입니다. Player 자료형은 그 크기가 크기 때문에 복사하여 사용하는 것보다 참조하는 것이 속도면에서 효율적이기 때문입니다.

문제 3번 출력결과:





3번 문제를 풀 때 주의할 점으로 파일을 ios::binary로 저장을 해야 불러올 때와 저장할 때 둘 다 같은 포맷이 되므로 정상적으로 save, load기능이 적용될 것입니다.

저장하는 방법은 플레이어들의 객체를 저장한 컨테이너의 반복자를 이용해 begin()부터 end()가 나올 때 까지 num이 999인지 하면서 Player에 만들어둔 write()함수를 그대로 이용했습니다.

친구와 서로 어떻게 구현했는지 상의를 해봤습니다. 친구의 경우 파일을 읽어오면서 동시에 num이 999인 객체를 확인하여 저장하였다고 했습니다. 그렇게 하면 굳이 루프를 한번 더 돌 필요가 없기 때문에 시간적인 측면에서 더 효율적인 코드라고 생각했습니다.

문제 4번 출력결과:

![텍스트, 배터리이(가) 표시된 사진

자동 생성된 설명]()

\*stable하게 정렬하지는 않았습니다.

이 문제를 풀기위해 어떻게 sort를 수행하면 좋을지 생각해봤습니다.

처음에는 똑같이 복사하여 id, name, score에 따라 정렬을 수행하는 것이었습니다. 하지만 그렇게 할 경우 차지하는 메모리가 엄청나게 늘어납니다. 특히 Player의 크기가 커질수록, 정렬할 기준이 많아질수록 그러한 문제는 더 크게 다가올 것입니다. 또한 Player의 정보를 수정하거나 추가, 삭제 기능까지 있을 경우 문제가 더 심각해집니다.

그래서 두번째로 생각한 방법이 포인터를 이용하는 것입니다. Player의 데이터가 크더라도 포인터는 항상 8byte(x64에서)이므로 Player의 크기에 영향을 받지 않습니다. 배열에 Player의 주소 값을 넣고 각각 id, name, score에 대해서 정렬을 시도했습니다. 이렇게 했을 경우 사용자에게 입력을 받았을 때 이진탐색을 통해 O(log n)시간만에 결과를 도출할 수 있으면 메모리 또한 아낄 수 있습니다.

정렬되어 있는 상태에서 lower\_bound와 upper\_bound를 같이 사용하면 찾고자 하는 값이 여러 개 여도 빠른 속도로 다 찾을 수 있었습니다. 이 함수를 사용하면서 해 맺던 점은



사진과 같이 lower\_bound와 upper\_bound의 compare함수가 받는 인자의 위치가 당연히 서로 같을 거라 생각하고 접근해서 오류의 원인을 찾는데 많을 시간을 소비해버린 것입니다.

이를 통해 함수를 사용할 때는 함수의 형식과 사용방법을 확실하게 숙지한후에 사용해야 한다는 것을 깨달았습니다.

이번에 과제를 통해서 프로그램 한 줄 한 줄마다 더 효율적인 방법은 없는지 고민하게 되고 여러가지 실험도 많이 해보게 되었습니다.

예를 들면 array<>의 operator= 하나를 쓸 때도 memcpy과 속도차이는 없는지 for문과 비교했을 때는 얼마의 차이가 있는지 확인해가며 자신이 짠 코드에 대해서도 “더 효율적인 방법은 없을까?”라는 물음을 습관적으로 할 수 있게 되었습니다.

1. https://ko.wikibooks.org/wiki/C\_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D\_%EC%9E%85%EB%AC%B8/%EB%8D%B0%EC%9D%B4%ED%84%B0\_%EB%B0%B0%EC%97%B4 [↑](#footnote-ref-1)