File Processing

Programming Project #1

담당 교수 : 박석

학번 : 20161566

이름 : 권형준

목차

0. 프로젝트 필요하다고 생각되는 가정

① Dat 내용에 대한 가정

② 입력에 대한 가정

1. GamePurchaseSystem 흐름도

2. 검색 방법과 그 방법이 효율적인 이유

① 방법 설명

② 사용(수정) 함수

③ 효율적인 이유

3. 삭제 방법과 그 방법이 효율적인 이유

① 방법 설명

② 사용(수정) 함수

③ 효율적인 이유

4. 삽입 방법과 그 방법이 효율적인 이유

① 방법 설명

② 사용(수정) 함수

③ 효율적인 이유

5. 수정 방법과 그 방법이 효율적인 이유

① 방법 설명

② 사용(수정) 함수

③ 효율적인 이유

6. 삭제에 따른 파일 compaction전략

7. 파일과 자료구조 설명

① Main.h, main.cpp

② Member.h, member.cpp

③ Game.h, game.cpp

④ Purchase.h, purchase.cpp

⑤ Recfile.h

⑥ Buffile.h, bufflie.cpp

⑦ Iobuffer.h

⑧ Varlen.h varlen.cpp

⑨ ".dat" file

8. 구현한 클래스의 다이어그램

9. 6장 연습문제

① 21번, 23번

② 22번, 24번

③ 25번

0. 프로젝트 필요하다고 생각되는 가정

① 개발 환경 가정

- Visual studio 2019를 사용하였다.

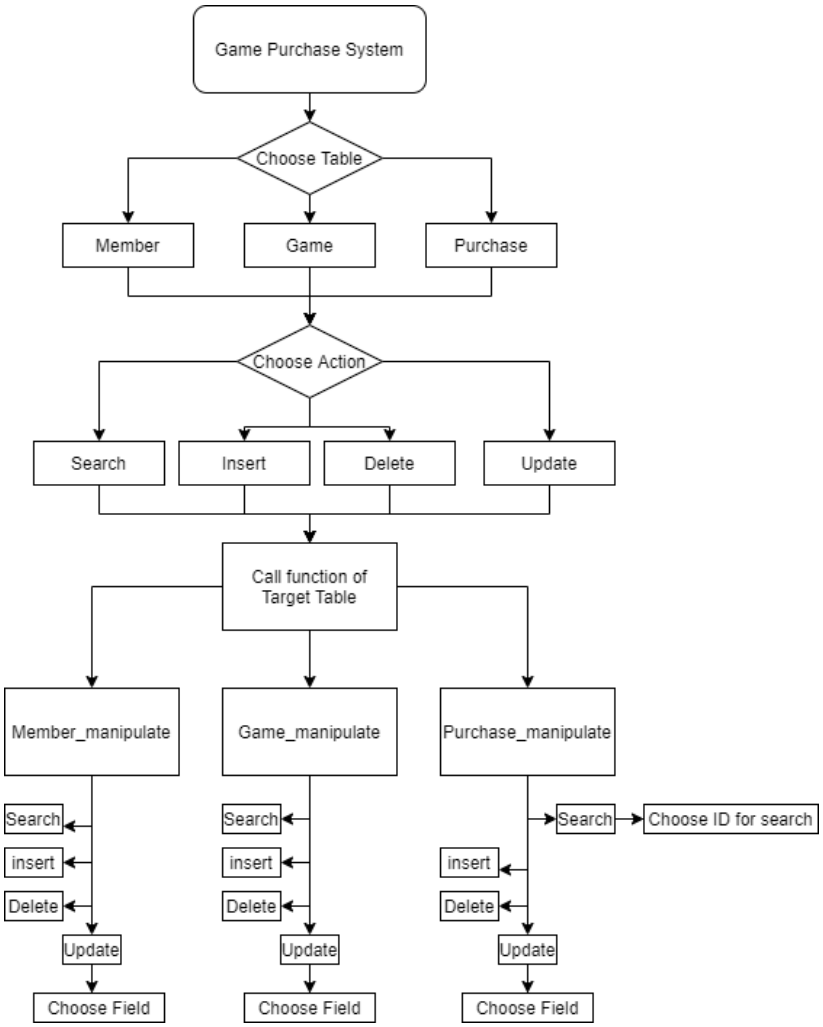
② Dat 내용에 대한 가정

- 장소는 실제로 존재하지 않을 수 있다.
- 날짜의 상대성을 고려하지 않았다. (예: 발매 2020, 구입 2010)
- 삭제된 Record는 프로그램이 종료돼도 삭제된 상태로 유지된다.

③ 입력에 대한 가정

- 입력 형식을 지켜서 입력해야 된다. (예: 날짜는 2019/12/30 형식)
- 메뉴에서 번호를 선택할 때, 2개이상의 character를 입력하면 안 된다.

1. GamePurchaseSystem 흐름도



2. 검색 방법과 그 방법이 효율적인 이유

① 방법 설명

- Member와 Game에 대한 검색

  Member와 Game은 각각 ID에를 기준으로 검색을 한다. 우선 ID를 입력 받고 그 ID를 가진 Member 또는 Game 객체를 생성한다. 다음 미리 overloading 한 ==연산자를 사용하여 ".dat"파일을 처음부터 Read하면서 같은 Record가 있으면 그 주소를 반환한 뒤, 그 주소를 기반으로 Record전체를 다시 읽어온다. 이때 삭제된 공간은 Read에서 -2를 반환하게 수정하였고 더 이상 읽을 곳이 없거나 잘못 읽었을 경우 -1을 반환하게 하였다. 따라서 address가 -1의 값을 가지게 되면 없는 ID라고 인식하게 된다. 이를 클래스 내 함수 Search를 통해 구현하였다.

- Purchase에 대한 검색

  Purchase는 앞의 형태와 같은 흐름을 가진다. 한가지 차이점은 overloading 한 ==연산자가 아닌 새로 정의한 Compare()이라는 함수를 사용한다는 것이다. Purchase의 경우 Member ID와 Game ID에 대한 검색도 지원이 가능해야 하기 때문에 단순히 ==연산자를 사용할 경우 Member ID가 같은 Record를 색출할 수 없다. 따라서 Record를 하나씩 read하면서 비교할 때 Member ID, Game ID, Purchase ID까지 다 비교해주는 Compare함수를 만들었고, 이를 Record File에서 사용할 수 있기 위해서 Search_For_Purchase 라는 클래스 내 함수를 또 작성해 주었다.

② 사용(수정) 함수

- int RecordFile<RecType>::Search(const RecType& record)

  BufferFile class에 미리 정의된 Read함수를 사용하여 File을 끝까지 읽는다. Read함수에 인자로 -1을 넣으면 이전에 읽던 주소부터 읽게 된다. Read가-1을 반환 받을 경우 일기 오류 또는 파일을 끝까지 읽은 것임으로 break를 하고 -1을 return한다. Read가-2를 return한 경우 삭제된 공간을 읽은 것으로 continue를 하여 다음 Record를 읽는다. 만약 그 외에 정상적인 record를 읽었을 시 Unpack을 하여 temp객체에 저장하고 그 객체를 record와 비교한다. 같다면 그 주소를 반환한다.

- int RecordFile<RecType>::Search_For_Purchase(const RecType& record)

  Purchase를 위한 Searching이다. 이 Search는 검색, 삽입, 삭제, 수정 중에서 오로지 검색에서만 사용된다. 이후에 삭제, 수정, 삽입의 경우 존재하는 Purchase ID가 있는지 확인할 때는 위의 Search함수를 사용하게 된다. Search_For_Purchase는 Search와 모든 것이 동일하지만 overloading 한 ==연산자로 비교하는 것이 아니라 compare라는 새로운 비교 함수를 사용한다.

③ 효율적인 이유

가변길이 레코드로 이루어진 파일 안에서 순차 검색 외에 다른 검색은 쉽지 않다. 순차적으로 접근하지 않기 위해서는 모든 레코드의 시작 위치를 알아야 되는데 이는 고정길이 레코드에서는 RNN을 통해서 쉽게 접근할 수 있지만 가변길이 레코드는 이를 유지하기 위해서는 수정, 삽입, 삭제 등에서 모든 record의 시작 위치를 수정해야 되기 때문에 유지하기 힘들다. 따라서 아직 학습 진도 내에서 유일하게 가능했던 검색은 순차 검색이었다. 또한 n이 기본적으로 1000언저리이기 때문에 시간 복잡도도 그렇게 크지 않았다.

3. 삭제 방법과 그 방법이 효율적인 이유

① 방법 설명

Delete는 검색을 이용하여 쉽게 구현할 수 있었다. Delete하기 위해서 우선 Table의 ID를 입력 받고 그에 맞는 Record를 검색한다. 이때, Purchase는 Search_For_Purchase가 아닌 Search 함수를 사용해야 된다. 입력한 ID에 대응하는 Record가 파일에 존재하는 경우 해당 Record의 활동상태인지 삭제되었는지 마크하도록 추가한 필드에 삭제되었다고 표시를 한다. 다음, 가용리스트를 유지하기 위해서 Header에 있던 가용리스트의 시작 주소를 삭제된 Record위에 덮어씌우고 Header에 삭제된 record의 주소를 적는다.

이 방법을 구현하기 위해서 추가로 Read, Write, writeheader(), readheader()함수들을 수정해야 됐다. 현재 프로젝트에서 사용하는 레코드는 variable length임으로 "varlen.h"와 "varlen.cpp"의 Read, Write등을 수정하였다. Read, Write는 앞에 활동상태인지 삭제되었는지 마크하도록 추가한 필드를 읽고 쓰는 부분을 추가하였고, Read는 만약 삭제되었다면 -2를 반환하도록 하였다. Write는 항상 활동상태인 Record를 추가하기 때문에 활동상태로 기록하게 하였다. Writeheader(), readheader()에는 int형 변수의 포인터를 인자로 넘겨주어 header을 읽을 때 가용리스트의 처음 시작 주소를 읽고 쓸 수 있게 하였고 (삭제된 record가 없다면 -1, "FFFFFFFF"를 저장하고 있다), 또한 buffer file에 private 변수로 Delete_first라는 변수를 설정하여 그 시작 주소를 저장하였다. writeheader를 호출하면 header에 적히는 가용리스트의 처음 시작 주소를 Delete_first로 덮어씌우게 수정하였다. 또한 Delete한번당 writeheader를 한 번씩 꼭 호출하여 언제 파일이 닫혀도 가용리스트의 처음 주소를 파일에 저장되게 하였다.

Delete를 하고 나면 file의 현재 쓰거나 읽는 포인터의 위치가 Read, Write를 하기에 애매한 위치로 남게 된다. 즉, Delete를 하고 난 뒤 열린 buffer File을 read, write를 하면 오류가 생길 수 있기 때문에 Delete를 하였으면 무조건 파일을 닫고 다시 열어서 read, write를 해야 된다. 이 점은 writeheader를 delete마다 호출하지 않고 close를 호출할 때 한 번만 수행하면 고칠 수 있다. (보고서를 쓰면서 발견한 사실이다...) 하지만 갑자기 프로그램이 종료될 경우를 생각하면, 매번 호출해주는 것도 괜찮은 것 같다.

② 사용(수정) 함수

- int RecordFile<RecType>::Delete(int recaddr)

  이 함수는 아래의 Delete를 호출한다.

- int BufferFile::Delete(int recaddr)

  이 함수는 아래의 Delete를 호출한다. 삭제된 레코드의 주소를 Delete_first에 넣는다. (주의, Delete함수를 호출 한 뒤 Delete_first를 수정해야 된다)

- int VariableLengthBuffer::Delete(ostream& stream, int recref, int next_addr)

  실질적으로 파일과 상호작용하는 함수이다. Recref는 삭제될 record의 시작 주소이다. Next_addr은 삭제될 record에 덮어씌울 가용리스트의 다음 주소이다. 보통 Delete_first변수를 넘겨주어 저장하게 된다. Record의 시작주소에 삭제 표시 '*'를 하고 2byte떨어진 주소에 (그 사이에는 레코드의 길이가 유지되어야 된다) next_addr을 저장한다.

③ 효율적인 이유

원하는 ID에 대응하는 record를 delete하기 위해서는 search를 해야 된다. 순차 탐색이 지금 사용할 수 있는 가장 효율적인 방법이라고 앞에서 설명했기 때문에 이 방법이 삭제에 있어서도 가장 효율적인 방법이다. 삭제에 따른 파일 compaction전략은 뒤에서 추가로 설명하겠다.

4. 삽입 방법과 그 방법이 효율적인 이유

① 방법 설명

우선 삽입하고자 하는 Record의 정보들을 입력 받는다. 다음 ID를 통해서 이미 존재하는 Record인지 확인한다. (Purchase의 경우 Game ID, Member ID가 존재하는 ID인지도 확인해야 된다.) 이때 buffer file에 정의한 get_packed_length함수를 사용하여 현재 삽입하고자 하는 Record의 pack한 이후 길이를 저장한다. 만약 해당하는 Record ID가 사용 가능하다면 이제 삽입을 시작한다. 앞에서 구한 pack한 이후의 길이를 기준으로 가용리스트를 돌면서 삭제된 공간의 크기가 pack한 이후의 길이보다 크다면 그 공간에 삽입하게 되고, 가용리스트를 모두 다 돌았는데도 저장할 만한 크기의 공간이 없다면 제일 뒤에 append한다. 만약 삽입하는 record보다 더 큰 공간에 삽입을 하게 되면 남는 공간만큼 공백을 채워 넣어 다음에 삽입한 record를 read할 때 오류가 발생하지 않도록 한다.

만약 가용리스트 중간에 삽입을 하게 되면, 삭제된 공간에 저장된 다음 주소를 가용리스트 이전의 record에 저장하는 과정도 필요하다. 만약 가용리스트의 head에 삽입된 경우 header의 첫 주소를 수정해야 된다.

② 사용(수정) 함수

- unsigned short BufferFile::Get_Packed_Length() → 아래의 함수를 호출한다.
- unsigned short VariableLengthBuffer::Get_Packed_Length()

  삽입할 레코드를 pack했을 시 byte수로 표현한 총 길이. 이는 가용리스트의 삭제된 레코드의 길이와 비교하여 삽입할 수 있는지 없는지를 판단할 때 사용된다.

- unsigned short BufferFile::Get_Buffer_Length(int recaddr) → 아래의 함수를 호출한다
- unsigned short VariableLengthBuffer::Get_Buffer_Length(istream & stream, int recref)

  해당 recref주소의 record 길이를 반환한다. 가용리스트를 돌면서 삽입할 레코드의 크기와 비교를 해야 되는데, 이때 삭제된 record의 길이를 알기 위해서 사용된다.

- int BufferFile::Get_Next_Addr(int recaddr) → 아래의 함수를 호출한다
- int VariableLengthBuffer::Get_Next_Addr(istream& stream, int recref)

  해당 recref주소의 record에 저장된 다음 가용리스트의 주소를 반환한다. 삭제 시에 가용리스트를 만들기 위해서 삭제된 record내에 다음 삭제된 record의 주소를 저장한다. 이 함수를 통해서 다음을 읽어올 수 있다.

- int BufferFile::Update(int recaddr, unsigned short length) → 아래의 함수를 호출한다
- int VariableLengthBuffer::Update(ostream& stream, int recaddr, unsigned short length)

  recaddr이 가리키는 주소에 buffer의 내용을 쓴다. Write와 역할이 같으며 다른 점은 length라는 인자를 가지고 있다는 것이다. 이 unsigned short length는 삽입하는 자리에 있던 원래 record의 길이를 담고 있다. 기존 삭제된 record의 길이와 새로 삽입하는 record의 길이의 차만큼 공백으로 공간을 채워야 되기 때문이다. 만약 삭제된 record의 길이가 20인 공간에 10인 record를 넣는다면, 10byte의 공간을 공백문자로 채워 넣어야 된다. 그래야 다음 Read에서 읽어올 때 오류가 발생하지 않는다.

- int BufferFile::update_Delete_first(int recaddr)

  Delete_first에 현재 주소를 저장하고, WriteHeader를 통해서 header에 저장된 가용리스트의 첫 주소를 수정한다. 이는 삽입이 이루어진 이후에 사용된다. 가용리스트의 첫 번째 주소에 새로운 record가 삽입이 되면 더 이상 파일의 header에 기존 첫 번째 주소가 저장되면 안 되고 기존의 두번째 주소가 저장되어야 된다. 이때 이 함수가 사용된다.

- int BufferFile::update_Deleted(int recaddr, int next) → 아래의 함수를 호출한다
- int VariableLengthBuffer::update_Deleted(ostream& stream, int recaddr, int next)

  위에서는 가용리스트의 첫번째 record에 삽입이 이루어졌을 경우를 위한 함수였다면, update_Deleted는 가용리스트 중간에 삽입이 이루어진 경우 그 삽입이 이루어진 주소를 담고 있던 리스트의 이전 record에 저장된 주소를 갱신해주는 역할을 한다. Recaddr은 삽입이 이루어진 record의 가용리스트 상 이전 주소를 담고 있고, next는 삽입이 이루어진 record에 저장되어 있던 다음 가용리스트 상 주소를 담고 있다. 즉 가용리스트가 다음과 같다면, (실제 주소) 가리키는 주소,

  (header) 16 → (16) 98 → (98) 34 → (34) 7

  98번 주소의 레코드에 삽입이 이루어진 경우, 두번째(주소가 16인) 레코드가 더 이상 98을 가리키면 안 되고 34를 가리켜야 된다. 이때 recaddr이 16이고 next가 34이 된다.

③ 효율적인 이유

사실 이 방법은 최고의 방법이라고 할 수 없다. 공간의 낭비가 많이 생기기 때문이다. 만약 20byte크기의 레코드가 삭제된 자리에 5byte의 레코드가 들어가게 된다면 남은 15byte는 사용하지 못하는 공간으로 남게 된다. 이 사용하지 못하는 공간은 삭제되었다고 표시도 되어있지 않기 때문에, 5byte의 레코드가 삭제되기 전까지는 아무런 정보를 추가로 저장하지 못한다. 하지만 무작정 새로운 record를 뒤에 append하는 것보다 그나마 공간의 재사용이 이루어지기 때문에 기존의 append보다는 효율적이라고 할 수 있다.

어떻게 보면 이 방법은 처음에 모든 레코드의 길이를 가변적으로 고정한 레코드에 대한 파일이라고 할 수 있다. 아직 배우지 않은 강의자료를 혼자 공부한 결과, 이 뒤에 남는 15byte의 공간도 표시를 해서 사용할 수 있는 공간으로 만든다고 한다. 하지만, 이는 매우 복잡하고 최소한의 레코드를 저장하기 위해 공간이 존재하는지, 존재한다면 그 자리에 구분자, 길이, 삭제 표시 등 작업이 많이 추가된다. 또한 일정 시간마다 중간중간의 모든 공간을 없애 주는 프로그램을 실행한다고 한다. 이는 프로그램이 종료될 때 dat파일들을 모두 열어서 삭제된 record들은 기록하지 않고 정상적인 record들만 다 다시 새로 기록하는 방법으로 구현할 수 있겠다. 이는 뒤에서 더 자세하게 설명한다.

5. 수정 방법과 그 방법이 효율적인 이유

① 방법 설명

수정은 삽입을 사용하여 쉽게 구현할 수 있다. 수정할 record를 찾기 위해 ID를 입력 받고 존재하는 record인지 search를 한다. 존재한다면 그 record를 가지고 온다. 수정할 field를 선택하고 수정해서 다시 record에 저장한다. (이때 purchase를 수정하는 경우

Game ID와 Member ID는 이미 존재하는 ID인지 확인한다). 다음 record가 pack했을시에 길이를 저장한다. 수정하려고 하는 record의 기존 기록은 삭제를 하고 새로 작성한 record를 새로 삽입하 것처럼 실행한다. 나머지 조건은 삽입과 동일하다.

② 사용(수정) 함수

수정은 삭제, 삽입, 검색을 사용하여 구현할 수 있어서 앞의 함수들을 적절히 사용하게 된다.

③ 효율적인 이유

앞의 삭제가 효율적인 이유와 동일한 내용이다.

6. 삭제에 따른 파일 compaction전략

앞에서 삭제에 대한 설명을 할 때 삭제가 어떤 식으로 진행되는지 설명하였다. 삭제되었다는 표시의 기호를 정하고, 가용리스트를 사용하여 삭제된 레코드에 빠르게 접근할 수 있게 하였다. 이를 이용하여 새로 추가되는 record가 추가적인 공간을 차지하지 않고, 즉 파일의 크기를 더 크게 만들지 않고 추가할 수 있게 되었다. 하지만 이는 내부단편화를 발생시킨다. 내부단편화는 원래 가변길이 레코드의 삭제에서는 발생하지 않는다. 뒤에 남는 공간을 또 다른 레코드가 들어갈 수 있게 작업이 추가되기 때문이다. 하지만 그렇게 남는 공간은 너무 작아 다른 레코드를 포함하기 힘들게 된다. 거기에 따른 외부단편화가 발생할 수 있다. 이를 프로그램이 특정 시간마다 작은 조각들을 모아서 새롭게 사용할 수 있는 공간으로 만들어줘야 되는데 우리 프로젝트는 그것까지 요구하고 있지는 않다. 따라서 오히려 원래의 가변길이 레코드처럼 내부단편화를 없애고 외부단편화를 선택하게 되면 사용할 수 있는 공간이 더 줄어들 것이다. 이를 고려하여 프로그램을 내부단편화가 이루어지도록 만들었다. 어떻게 보면 청음에 가변길이로 레코드를 정해주고 그 길이를 고정레코드처럼 사용하는 것이다.

7. 파일과 자료구조 설명

① Main.h, main.cpp

→ 대화형 프로그램이 돌아가는 main함수가 있는 파일.

② Member.h, member.cpp

→ member class와 함수들이 존재하는 파일

③ Game.h, game.cpp

→ game class와 함수들이 존재하는 파일

④ Purchase.h, purchase.cpp

→ purchase class와 함수들이 존재하는 파일

⑤ Recfile.h

→ RecordFile의 class가 존재한다. RecordFile은 BufferFile을 상속받은 class로 member, game, purchase class의 다양한 record type을 통일된 방식으로 사용할 수 있게 해준다. BufferFile의 함수를 사용하기 위해서는 여기서 함수를 새로 정의해 호출하는 형식을 취해야 된다.

⑥ Buffile.h, bufflie.cpp

→ file과 buffer을 하나의 class에 묶어서 보관하는 Bufferfile의 class와 그에 관한 함수가 저장된 파일들이다. 여기에 정의된 함수를 통해서 IOBuffer의 함수들을 호출하여 file과 buffer을 가지고 이것저것 수행할 수 있다.

⑦ Iobuffer.h

→ IObuffer의 class와 그에 관한 함수들이 포함된 header파일이다. 함수를 정의할 때 virtual을 사용하는 이유는 VariableLengthBuffer, FixedLengthBuffer등의 IOBuffer class를 상속하는 class들에서 함수들을 각각의 class에 맞게 재정의하여 사용하게 하기 위해서이다. 즉, 실질적으로 우리가 사용하는 함수는 varlen.h, varlen.cpp에서 정의하는 함수들이다.

⑧ Varlen.h varlen.cpp

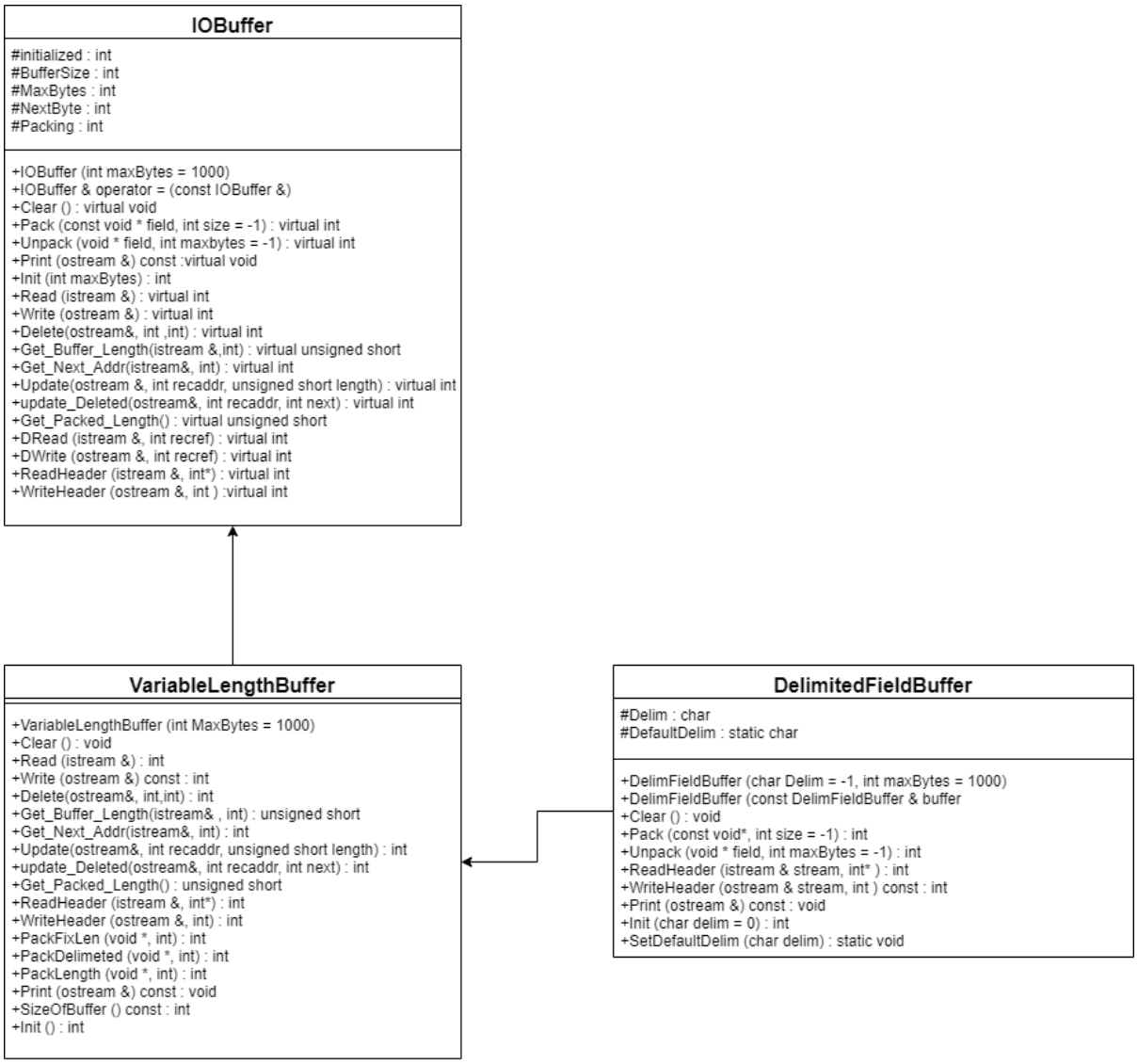
→ 가변길이 레코드에 대한 함수들이 선언, 정의되어 있다. Ostream, istream에서 직접 byte단위로 buffer의 내용을 쓰기도 하고 읽기도 한다. 결국 main에서 대화형 프로그램을 통해서 전달받은 명령은 이 부분에서 처리가 된다.

⑨ ".dat" file

→ varlen.h에 있는 writeheader을 거치면 IObuffer.h에 있는 writeheader를 실행하게 된다. 이때 "IOBuffer"가 찍히고, varlen.h의 writeheader에서 "Variable"이 찍힌다. 만약 VariableLengthBuffer가 아니라 FixedLengthBuffer이었다면 "Fixed"가 찍혔을 것이다. 앞에서 varlen.h의 writeheader을 수정하여 그 뒤에는 가용리스트의 시작 주소를 저장한다. (삭제된 record가 없다면 FFFFFFFF가 저장) 다음은 차례대로 record가 저장된다. 각 record의 길이 직전에 1byte크기의 00 또는 '*'를 입력하는 공간이 있다. 이 부분이 00이면 사용 중인 record, '*'이면 삭제된 record를 나타낸다.

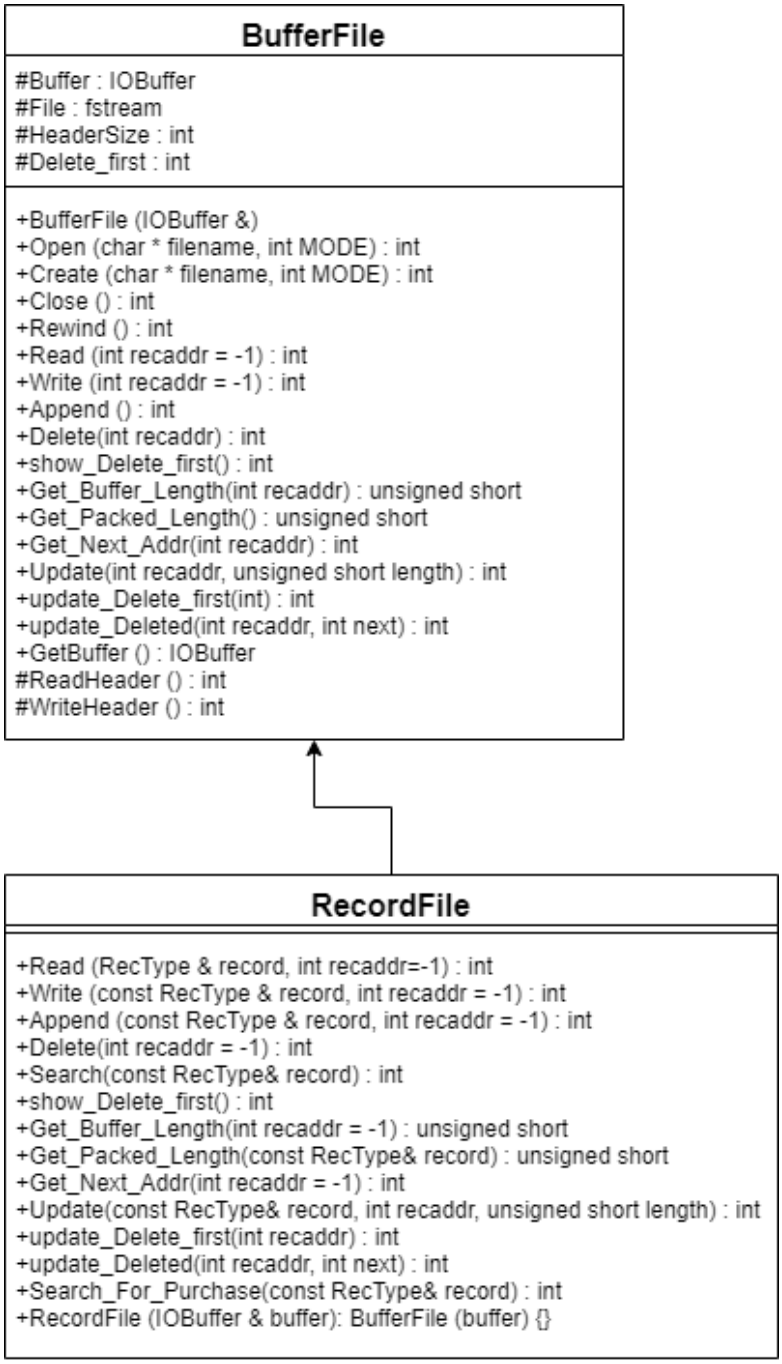
8. 구현한 클래스의 다이어그램

- **IOBuffer – VariableLengthBuffer – DelimitedFieldBuffer**



프로젝트에서 사용하는 레코드는 '|'의 구분자로 구분되는 Delimited Field를 사용하고 field의 내용이 string이 들어갈 수 있는 것으로 유추하면 variable length record이다. 따라서 위의 class들만을 이용해서 구현할 수 있었다. '+'는 public을, '#'은 protected를 의미한다며 화살표의 방향이 상속을 한다는 의미이다.

- **RecordFile – BufferFile**



Buffer와 file을 한 번에 관리하게 해주는 bufferfile class를 member, game, purchase모든 class에서 동일한 format으로 사용하기 위해 만든 것이 바로 RecordFile이다.

9. 6장 연습문제

※6장의 연습문제는 고정길이 레코드와 가변 길이 레코드 파일에 대한 프로그램 구현을 요구하고 있다. 이번 프로젝트가 가변길이 레코드를 기반으로 진행이 되어 가변길이 레코드를 지원하는 코드를 작성하였지만, 사실상 한가지만 변경하면 고정길이 레코드에 그대로 적용할 수 있다. 그 한가지가 바로 가변길이 레코드의 길이 기록이다. 고정길이 레코드에서는 레코드의 길이를 이미 알고 있기 때문에 오히려 편리하다. 밑에서는 Fixlen.cpp, fixlen.h의 어떤 부분을 수정하였는지에 대한 설명이 들어있다.

① 21번, 23번

21, 23번에서의 요구사항을 정리해보면

➔ Delete method의 구현

➔ 각 레코드마다 삭제 확인 필드 구현

➔ 그에 맞게 다른 Method 수정

앞에서 가변길이 레코드에 대해서는 write할 때 앞에 1byte를 할당하여 삭제 여부를 표시하고, read할 때 그 1byte를 읽어서 삭제 여부를 판단한다. 고정길이의 경우도 마찬가지다.

고정길이에서 달라지는 점은 read할 때 처음 2byte(unsigned short)만큼 읽어오지 않고 미리 저장되어 있는 레코드의 크기만큼 읽어오면 되는 것이며, write할때도 레코드의 길이를 써주지 않아도 된다는 것이다. Fixlen.cpp에 이미 구현되어 있는 read와 write에 삭제여부 필드만 추가하였다. 또한 delete만을 위한 함수도 구현하였다. 원하는 주소를 받아서 주소의 삭제여부 필드에 '*'표시를 통해서 삭제를 진행한다.

② 22번, 24번

22번, 24번의 요구사항은 다음과 같다

➔ 공간의 재사용이 될 수 있게 해라

➔ 파일의 헤더에 헤드 주소를 저장할 수 있는 공간을 추가하여라

➔ 삭제된 레코드에 있는 공간을 사용하여 다음 삭제된 레코드 주소를 저장하여라

앞에서 설명한 것과 같이 writehead, readhead 함수를 사용하여 처음 4byte(int의 크기)만큼의 공간을 할당하여 가용리스트의 처음 레코드 주소를 저장하였고, 레코드를 삭제할 때 레코드 공간에 가용리스트의 처음 레코드 주소를 저장하고 파일 헤더에 삭제한 레코드의 주소를 저장한다고 하였다. 이는 위에서 정의한 함수들로 가능하다. 공간의 재사용은 삽입과 수정에서 설명하였는데, 가용리스트를 돌면서 삽입하려고 하는 레코드보다 공

간이 더 넓은 삭제된 레코드가 있다면 삽입하고 없다면 끝에 append를 한다.

이를 고정길이 리스트에 적용하려면 어떤 수정사항을 거쳐야 할까? 우선 파일의 헤더에 헤드 주소를 저장하는 것은 똑같을 것이다. 하지만 공간의 재사용을 위해 모든 가용리스트를 돌 필요가 없고 처음 레코드를 사용해도 될 것이다. 고정길이기 때문에 크기는 다 같기 때문이다. 삭제된 레코드에 있는 공간의 재사용은 가변길이와 동일하게 진행하면 될 것이다. 따라서 이미 구현되어 있는 fixlen.cpp의 writeheader와 readheader에 가용리스트의 첫 주소를 저장하는 과정을 포함하였다. 또한 재사용이 이루어졌을 때 파일의 header에 있는 첫번째 가용리스트의 주소 수정이 필요한데 writeheader와 Get_Next_Addr함수로 구현하였다.

③ 25번

25번의 요구사항을 정리하면 다음과 같다

➔ 새 레코드의 크기가 대체할 레코드의 크기와 다른 경우 적절히 처리

한마디로 요약하자면 위와 같다. 이 경우가 이제 또 2가지로 나뉘는데, 기존의 레코드가 더 큰 경우와 더 작은 경우이다. 더 큰 경우에는 문제가 발생하지 않는다. 크기 차이만큼 공백으로 채워 넣으면 된다. 하지만 기존의 크기가 더 작은 경우에는 기존의 record를 삭제하고 대체할 레코드를 가용리스트상에 삽입 가능한 지점에 삽입해야 된다.

고정길이의 경우 레코드를 update하는 경우 길이 차이의 걱정을 할 필요가 없기 때문에 그냥 write method로 대체가 가능하다.

-감사합니다-