CSED232 Object-Oriented Programming (Spring 2020)

Programming Assignment #4

Smart Pointer and Smart Matrix

-Templates & STL -

Due data: 2020년 6월 6일 토요일 23:59

담당조교: 남석현 (obiwan96@postech.ac.kr)

1. Objective

본 과제에서는 Standard Templates Library의 하나인 shared_ptr를 직접 구현해보고 이를 이용하여 행렬 클래스를 구현하여 봄으로서 templates 과 STL에 대한 이해를 높인다.

2. 과제 설명

2.1 개요

스마트 포인터(smart pointer)는 메모리 누수로부터 프로그램을 보호하기 위해 C++에서 제공하는 기능이다. 포인터처럼 동작하는 클래스 템플릿으로, 사용이 끝난 메모리를 자동으로 해제해주는 것이 특징이다. 일반적인 포인터를 사용할 때에는 new 명령어를 통해 실제 메모리를 가리키도록 선언하며 사용이 끝난 포인터에 대해 delete 명령어를 통해 메모리를 수동으로 해제해야 한다. 하지만 스마트 포인터는 다음 그림 1과 같이 기본 포인터(raw pointer)를 new 명령어로 선언하여 생성한 후 스마트 포인터에 대입하여 선언된다. 해당 스마트 포인터의 수명이 다하면 클래스에 정의된 소멸자를 통해 스마트 포인터에 할당된 포인터를 자동으로 해제한다. 스마트 포인터의 더 자세한 동작에 대해서는 따로 따로 자료를 찾아보는 것을 추천한다.

```
SmartPtr<int> ptr = new int(5);
```

그림 1 스마트 포인터 선언 예시

shared_ptr는 스마트 포인터의 한 종류로서 C++ STL에 정의되어 있다. memory 헤더파일을 include하면 사용가능하다. shared_ptr는 어떤 하나의 객체를 참조하는 스마트 포인터의 개수를 참조하는 참조 카운트(reference count) 기능을 포함하고 있다. 해당 메모리를 참조하는 포인터가 늘어나면 shared_ptr는 자동으로 참조 카운트 값을 증가시키며, 참조 카운트 값이 0이 되면 자동으

```
shared_ptr<int> ptr1 = new int(5);
cout << "현재 소유자 수 " << ptr1.use_count() << endl; // result : 1
auto ptr2=ptr1;
cout << "현재 소유자 수 " << ptr1.use_count() << endl; // result : 2
```

그림 2 shared_ptr 사용 예시

로 스마트 포인터를 해제하여 메모리를 해제하는 것이 주요한 기능이다. 참조 카운터의 구현 방법에 대해서는 4장 부록을 참고하기 바란다.

그림 2와 같이 shared_ptr을 선언한 후 use_count 메소드를 통해 포인터를 참조하고 있는 스마트 포인터의 수를 출력할 수 있다. 본 과제에서는 배포된 SmartClass.h를 수정하여 간단한 shared_ptr을 직접 구현해본다. 또한 이를 이용하여 간단한 matrix class 를 선언해본다. matrix class는 SmartMatrix로서 간단한 기능들을 포함하고 있으며, 스마트포인터를 이용하였기 때문에 메모리 해제를 해주지 않아도 된다.

2.2 상세 설명

배포된 skeleton 코드에는 두 클래스의 뼈대가 각각 정의되어 있다. 여러분이 작성해야할 기능에 대해서 설명하겠다. 조교가 주석을 통해 작성하라고 명시된 곳 이외에는 건들지 말 것. 또한 스켈레톤 코드를 꼼꼼히 읽고 전체 구조를 이해한 다음 시작하는 것을 추천한다. 여러분은 10개의 클래스 함수를 정의해야 한다. 함수에 따라서 여러 부분을 구현해야하는 것도 있으니 코드를 잘 읽어볼 것. 함수 구현의 난이도가 대체로 뒤로 갈수록 어려우며, 앞 함수를 구현하지 않으면 뒷 함수가 구현되지 않는 경우도 많기 때문에 1번부터 차근 차근 구현해보는 것을 추천한다.

- **SmartPtr**

SmartPtr 클래스는 포인터를 선언하고 참조 카운트를 관리하기 위한 *CountedObjectContainer* 클래스를 포함하고 있다. 여러분은 이를 이용하여 *SmartPtr* templates을 만들어야 하기 때문에 *CountedObjectContainer*의 코드를 우선 분석하여 이해하고 작업을 시작하는 것을 추천한다.

코딩을 완료한 후에는 스스로 SmartPtr을 이용하는 예제를 구현해보고 테스트 해볼 것. 컴파일 할 때 다음 그림 3과 같이 DeallocMessage를 선언하는 옵션을 넣어주면 dealloc message를 확인할 수 있으니 본인이 구현한 class가 참조 카운팅을 제대로 해서 delloc이 제 때 일어나는지 확인하여 보는 것을 추천한다. 그림 3은 하나의 예시일 뿐, 본인이 만든 예제에서는 다르게 나올 것이다.

```
$ g++ -o test test.cpp –D DeallocMessage
$ ./test
5
Deallocate an object
```

그림 3 Dealloc Message 컴파일 예시

(1) SmartPtr 생성자

본 클래스는 세 개의 생성자를 가지고 있다. NULL이 입력될 때의 생성자는 예외처리로서 이

미 작성되어있다. 여러분은 특정 포인터가 객체로서 전달될 때의 생성자와 *SmartPtr* 복제를 위한 생성자를 작성하여야 한다. *SmartPtr*은 templates로서 작성되기 때문에 어떤 형태의 포인터가 입력되도 생성이 될 수 있어야 한다. 입력된 포인터가 빈 포인터가 아닌지 확인하여 빈 포인터가 아닐 경우 *CountedObjectContainer*를 이용하여 *SmartPtr*의 *m_ref_object* 요소를 관리하여야 한다. 새로운 포인터를 참조하게 되었을 때도 참조 카운트 값을 더해주어야 한다.

(2) SmartPtr 소멸자

CountedObjectContiner 클래스를 이용하여 현재 스마트 포인터에서 관리하고 있는 포인터에 대한 참조를 해제한다.

(3) 대입연산자

두 곳을 작성해야한다. 생성자와 마찬가지로 *SmartPtr*형태가 대입될 때와 다른 포인터가 대입될 때를 구별하여서 작성하여야 한다. 기존에 참조하고 있던 포인터에 대해서는 참조 카운트 값을 감소시키고 새로 참조하게 된 포인터에 대해서는 참조 카운트 값을 증가시켜야 한다.

- **SmartMatrix**

*SmartMatrix*는 *SmartPtr* 을 이용하여 행렬을 선언한다. 따라서 *SmartPtr*을 완벽히 구현한 후에 작업을 하여야 한다. row 수와 column 수인 *m_rows*와 *m_cols*를 포함하고 있으며, *SmartArray*로 선언된 *m_values*에 행렬 값들을 저장한다. *SmartArray*는 *SmartPtr*을 이용하여 정의된 스마트 배열로서, 함께 배포된 예시 사용 코드에서 사용법을 확인할 수 있다.

더하기, 빼기, 곱하기의 기능은 연산자 오버로딩 (Operator Overloading)으로서 클래스 밖에 구현되어있다. Stream extraction 연산자는 이미 구현이 되어있으니 여러분은 행렬을 출력시켜볼 수 있을 것이다. 본인의 코드가 잘 작동하는지 직접 행렬을 출력하며 확인해보자.

(4) 생성자

세 개의 생성자를 구현해야한다. 우선 row 수와 column 수만 입력이 된 경우 행렬을 생성하고 행렬의 모든 값을 0으로 초기화 시키는 생성자를 구현한다. 두번째 생성자는 *SmartPtr*와 마찬가지로 클래스 복제를 위한 생성자로서, 생성자의 인자로 *SmartMatrix* 형이 들어올 때 내용을 복제한 새로운 *SmartMatrix*를 생성할 수 있어야 한다. 세번째는 행렬 값도 함께 인자로 줄 때이다. row 수, column 수와 함께 행렬의 값들이 *m_row * m_col* 개의 원소로 이루어진 1차원 배열로서 입력이 된다. 그러면 행렬을 생성하고 1차원 배열의 값을 행렬의 왼쪽 위부터 순서대로 넣으면 된다. 세번 째 경우의 입력 예시를 예시 사용 코드에서 확인할 수 있을 것이다.

*SmartMatrix*는 스마트 포인터를 이용하기 때문에 소멸자를 별도로 작성하지 않아도 메모리 해제가 일어날 것이다.

(5) Row 추가

행렬에 row 를 추가하는 함수이다. *m_column*개의 원소로 이루어진 1차원 배열을 인자로 입력받는다. 그러면 여러분은 메모리를 그만큼 더 할당한 다음 주어진 row를 추가하면 된다. 기존 메모리를 할당 해제하고 새로 메모리를 생성하여도 된다.

(6) Column 추가

(5)와 마찬가지로 *m_row*개의 원소로 이루어진 배열이 인자로 입력되면 column을 추가하는 함수이다. AddRow보다 약간 더 난이도가 있으니 AddRow부터 구현하는 것을 추천한다.

(7) 역행렬 계산

2x2 행렬에 관해서만 역행렬을 계산하여 반환하는 함수이다. 함수가 호출되면 Assert를 통해 현재 행렬의 행과 열이 2인지 확인을 하여 2x2 행렬이 아닐 경우 프로그램을 종료하는 기능을 포함시키도록 한다. Assert 사용법에 대해서는 'cassert'를 검색해볼 것. 또한 2x2 행렬이라도 역행렬이 없는 행렬이라면 마찬가지로 assert를 통해 프로그램을 종료하는 기능을 포함시킨다. 2x2 행렬의 역행렬 계산은 어렵지 않기 때문에 쉽게 구현할 수 있을 것이다.

(8) 더하기 연산자

더하기 연산자부터는 c++의 기본연산자를 대체하여 정의하여 우리가 만든 *SmartMatrix*도 + 연산을 이용할 수 있도록 한다. 연산자 오버로딩의 개념을 완전히 이해하고 작업을 시작하는 것을 추천한다. 더하기 연산자는 입력된 두 행렬의 열과 행 수가 같은지 확인하여야 할 것이다 (7)과 마찬가지로 assert를 통해 열과 행이 같지 않을 경우 프로그램을 종료하는 기능을 포함시킨다.

(9) 빼기 연산자

*SmartMatrix*가 – 연산을 사용할 수 있도록 한다. 입력된 두 행렬의 열과 행수가 같은지 확인하는 assert를 포함시킨다.

(10) 곱하기 연산자

*SmartMatrix*가 * 연산을 사용할 수 있도록 한다. 행렬과 스칼라의 곱, 행렬과 행렬의 곱을 모두 구현하여야 한다. 원래는 스칼라 * 행렬과 행렬 * 스칼라를 모두 구현하여야 하지만, 행렬 * 스칼라만 구현하면 두 기능 모두 사용할 수 있도록 이미 구현되어있다.

행렬간의 곱셈은 더하기와 빼기와 마찬가지로 연산이 가능한지 먼저 확인하여야 한다. 앞 행렬의 행 수와 뒷 행렬의 열 수가 같은지 assert를 통해 확인한 후 계산을 하도록 한다.

곱셈 계산 시 현재 행렬이 변하면 안되기 때문에 같은 크기의 행렬을 새로 선언하도록 설계되어있다.

3. 제출 방식 및 채점 방식

코드 작성이 완료된 후 각 함수 별로 본인의 알고리즘을 간략히 설명하는 보고서를 작성한다. 작성이 완료된 헤더 파일만 본인 학번 폴더 안에 보고서 파일과 함께 저장한다. 보고서 파일은 '학번.pdf'의 형태로 저장한다. 모든 파일은 압축하여 '학번.zip'파일로서 lms에 제출할 것. 즉, 학번.zip을 압축 해제 하면 SmartClass.h, 학번.pdf의 두 파일이 존재하여야 한다. 제출 기한을 넘길 시 조교에게 메일(obiwan96@postech.ac.kr)로 제출할 것. 하루(24시간) 늦을 때마다 20%씩

감점 한다. 1일 이내 지연: 20% 감점, 2일 이내 지연: 40% 감점, 5일 이상 지연: 0점.

제출 형식이 맞지 않으면 전체 점수의 10%의 감점이 있을 수 있다. **다른 사람의 프로그램이나 인터넷에 있는 프로그램을 복사(copy)하거나 간단히 수정해서 제출하면 학점은 무조건 'F'가 된다. 이러한 부정행위가 발견되면 학과에서 정한 기준에 따라 추가의 불이익이 있을 수 있다.**

컴파일러는 minGW를 사용하는 것을 원칙으로 한다. 제출된 코드를 이용하여 조교는 해당 클래스의 각 함수들이 잘 작동되는지 직접 사용해보며 채점을 진행한다. 본 과제는 10개의 함수를 구현하는 것으로 이루어져 있다. 각 함수 별로 해당 기능이 충실히 구현되었을 경우 9점씩 배점하여 총 90점의 프로그램 기능 점수를 부여하며, 보고서 점수 10점을 합하여 100점만점으로 구성된다. 앞 함수가 구현되지 않아서 작동하지 않는 함수에 대해서는 부분 점수가 존재하지 않으니 앞에서부터 차근차근 구현해야 한다.

코드 테스트를 위한 간단한 예제를 함께 첨부하였으나, 이는 스마트 포인터를 어떻게 사용해야 하는지에 대한 사용법을 알려주기 위함이다. 완벽한 점수를 위해서는 직접 해당 클래스를 이용해 보며 본인의 클래스가 잘 구현되었는지 확인해볼 것을 추천한다. 조교는 해당 코드로 채점하지 않고 더 복잡한 상황을 테스트해볼 것이다. 또한 과제를 시작하기 전 스마트 포인터와 행렬의 기본 연산에 대해 공부 후 과제를 시작할 것을 추천한다. 각종 예외 상황을 스스로 생각해보며 예외 처리도 하는 것을 추천한다.

4. 부록 – 참조 카운팅 (Reference Counting)

참조 카운팅은 스마트포인터에서 사용하는 소유권 관리 방법이다. 참조 카운팅 기법은 동일한 객체를 가리키는 스마트 포인터의 개수를 추적한다. 또한 그 숫자가 0이 되면 해당 객체를 소멸시키다.

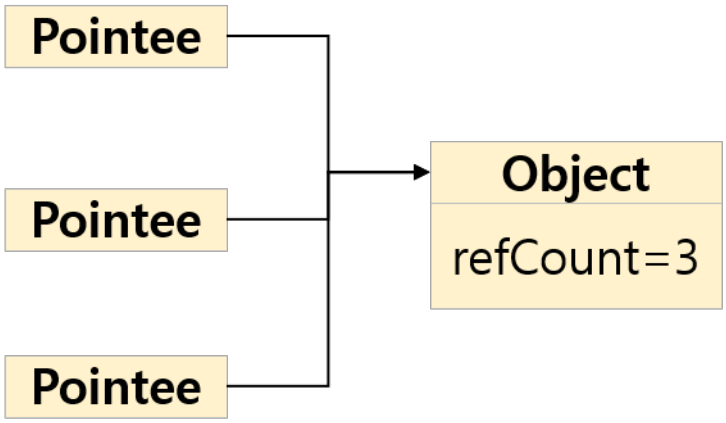

그림 4 참조 카운팅

참조 카운팅을 구현하는 방법에는 여러가지가 있을 수 있는데, 가장 효율적인 방법은 그림 4와 같이 포인팅 받는 객체 자신이 참조 카운팅 변수를 저장하는 방식이다. 이 방식은 본 과제에서도 사용하고 있다. 이를 위하여 본 과제에서는 *CountedObjectContainer* 클래스를 선언하여 포인팅 받는 객체에서 카운팅 변수를 관리하도록 구현되어있다.

이 외에도 스마트포인터에서 직접 참조카운팅을 할 수도 있지만, 이는 스마트 포인터끼리 카운터 변수가 공유되어야 하기 때문에 비효율적이다.