자료구조와 c++ 프로그래밍

이름	학번	학과	레포트 번호
이인	2021125050	소프트웨어과	과제 2번

1. 문제 정의 및 분석

배열에 저장하는 원소를 객체 형태의 원소라고 생각한다면 쉽게 해결할 수 있다. 입력으로 받는 수들에 대해, 계수와 지수 부분이 하나의 항이라고 생각한다면, 다항식은 계수와 지수 부분으로 이루어진 객체를 여러개 가지고 있는 배열로 생각할 수 있다. 즉 계수와 지수에 대한 정보를 담는 TERM 객체와, TERM 객체를 배열로 가지고, 항의 개수와 전체 배열의 크기에 관한 정보를 저장하는 다항식 객체 POLYNOMIAL이 필요하다는 것을 알 수 있다. 입력받는 숫자를 다항식의 형태로 출력하고, 내부적으로는 계수와 지수부분에 올바르게 저장하기 위해 입력 연산자 오버로딩이 필요하다. 또한 계수와 지수 법칙에 따라 계수가 0일 때, 지수가 0일 때, 지수가 음수일 때, 지수가 정수가 아닐 때에 대한 예외를 생각하여 출력 연산자에 대한 오버로딩이 요구된다.

입력 연산자 오버로딩을 통해 받은 계수와 지수부에 대한 수를 저장하기 위해 POLYNOMIAL 객체에 TERM에 대한 배열을 생성하고, 입력값의 길이에 따라 배열의 크기를 가변적으로 조절해야 하기 때문에 TERM에 대한 포인터 변수를 생성한 후, NEW 키워드를 이용하여 배열 공간을 할당과 동시에 초기화 해준다. 그후 크기 변화가 필요할 때 마다 길이가 연장된 새로운 배열을 생성하고 기존 배열의 TERM들을 복사하여 내용물은 같지만 크기가 더 커진 배열을 계속 만들어내는 방식으로 입력을 무한정으로 받을 수 있도록 할 수 있다,

또한 입력받은 식이나 곱한 식에 대해 같은 지수항에 대하여 정리한 식을 출력해야하기 때문에 식을 정리해주는 메소드가 필요하다.
다항식 A와 B에 대한 곱연산을 구현하려면 A에 각항과 B의 각 항 모두에 접근하여 곱연산을 수행할 수 있는 곱연산 메소드가 필요하다.

다항식의 미지수 X에 대해 특정값을 대입하여 다항식의 최종 값을 도출해내는 메소드가 필요하다. 이는 X값을 인자로 받은 후 계수와 X의 지수제곱의 곱으로 구현이 가능하다.

2. 문제 해결

```
friend istream &operator>>(istream& _is, Polynomial& poly){ // 입력 연산자 오버로딩, poly 객체이 값을 변경할 것이니 참조로 parameter 선언
    float tmpCoef =0;
    int tmpExp = 0;
```

```
friend istream &operator>>(istream& _is, Polynomial& poly){ // 입력 연산자 오버로딩, poly 객체이 값을 변경할 것이니 참조로 parameter 선언
    float tmpCoef =0;//임시 계수
    int tmpExp = 0;//임시 지수
    int count = 1; // 지수 혹은 계수부의 입력에 비정상적으로 값이 들어가는 것을 막고 순서대로 입력이 계수-순서대로 할당되도록 하기 위해 설정
    while(true){// 엔터키 입력까지 계속 반복, 입력 마무리 이후, 엔터키 한 번 더 입력해줘야 함
        if(count%2 == 1) {//홀수 번째 수 일 때 계수를 받도록 함
            _is >> tmpCoef;
            count++;
            if (tmpCoef == 0) { // 입력 받은 계수가 0일 때
                if (cin.get() == '₩n') break; // 다음 수가 없다면 종료
                else {// 그 다음 수가 존재한다면
                    _is >> tmpExp;// 계수가 0, 항 성립 안됨 -> 입력 순서를 지키기 위해 입력만 받고 할당은 암함.
                    count++; // 입력 순서를 맞추어 다음번 루프에 계수를 입력받도록 함
                    if (cin.get() == '₩n') break;
                    else continue;
                }
            }
            else { //입력 받은 계수가 0이 아닐 때
                if (cin.get() == '₩n') { //다음 수가 없다면
                    poly.newTerm(tmpCoef, 0);// 지수부 입력이 없기때문에 상수항으로 생성
                    break; //입력 종료
                }
                else {// 다음 수가 존재한다면
```

```
            if (count % 2 == 0) {// 짝수 번째 수 일때 지수 입력받기
                _is >> tmpExp;
                count++;
                if(tmpExp >=0)poly.newTerm(tmpCoef,tmpExp); // 지수가 음수인 경우를 제외하고는 새로운 항 추가
                if (cin.get() == '₩n') break; // 지수 입력 뒤에 추가로 숫자가 없다면 입력 종료
                else continue;//추가로 숫자가 있다면 순환
            }
        }
    }
  }
  poly.sum(); // 해당 식을 정리해주는 메서드 호출
  return _is;
};
```

^ **입력 연산자 오버로딩** - 입력 연산자를 오버로딩 할 때 필요한 것은 연산자 클래스에 대한 객체와, 오버로딩을 할 대상 객체가 필요하다. 따라서 _is와 poly에 대한 참조를 인자로 받아 그 기능을 바꿀 것이다. cin에서는 입력을 받을 때 엔터키나 공백을 기준으로 입력을 끊는다.즉 입력의 마무리를 인식한다. 그러나 우리가 지수와 계수를 받을 때 홀 수개의 수 만 받게 되는 경우에는 지수부분에 대한 처리가 제대로 되지 않으며, 입력 개수를 한정할 수 없다. 따라서 cin.get()을 사용하여 입력 버퍼에 남아있는 값이 ₩n인 것이 확인 되는 경우, 즉 입력이 종료된 경우에 대해서 홀수 개를 받는 경우랑 짝수 개를 받는 경우를 구분하고, ₩n가 인식이 안된 경우에 대해 입력을 무한정으로 받도록 하여, 원하는 만큼의 수를 입력받도록 구현한다. 단 계수가 0인 항은 무조건 무시하도록 하며, 무시 이후 계수-지수 입력 순서를 맞춰주기 위해서는 계수 0 에 대한 입력을 무시했으니 마찬가지로 지수 부분의 입력 또한 한 번 건너뛰어야 입력순서를 맞출 수 있다. 따라서 _is >> tmpExp로 지수부에 대한 입력을 받지만 term생성은 하지 않도록 하여 계수-지수 에대한 입력값 받는 순서를 유지한다. 각 경우에 대해 입력받은 식을 newTerm함수로 배열에 할당하고, sum()함수를 이용해 같은 지수에 대해 계수를 정리하도록 한다. 최종적으로 입력 연산자 객체를 반환하여 해당 오버로딩이 식에 적용되게 한다.

```
friend ostream &operator<<(ostream& os, Polynomial& poly) { // 출력 연산자 오버로딩
  for (int i = 0; i < poly.terms; i++) {//모든 항들에 대해
    if (poly.termArray[i].exponent != 1 && poly.termArray[i].exponent != 0) { // 지수가 1이 아닐 때,
      if (poly.termArray[i].coefficient > 0) { //계수가 양수인 경우
        if (i == 0) os << poly.termArray[i].coefficient << "x^" << poly.termArray[i].exponent; // 첫 항
        else {
```

```
                os << "+" << poly.termArray[i].coefficient << "x^" <<
poly.termArray[i].exponent; // 첫 항이 아닐 때
            }
        }
        else if (poly.termArray[i].coefficient < 0) //계수가 음수인 경우
            os << poly.termArray[i].coefficient << "x^" << poly.termArray[i].exponent;
    }
    else if (poly.termArray[i].exponent == 1) { // 지수가 1인 경우
        if (poly.termArray[i].coefficient > 0) { //계수가 양수
            if (i == 0) os << poly.termArray[i].coefficient << "x"; // 첫 항, x^1은 x와 같음
            else {
                os << "+" << poly.termArray[i].coefficient<< "x"; //첫 항 아님
            }
        }
        else if (poly.termArray[i].coefficient < 0) { //계수가 음수인 경우
            os << poly.termArray[i].coefficient<< "x";
        }
    }
    else if (poly.termArray[i].exponent == 0){ // 지수가 0인 경우
        if (poly.termArray[i].coefficient > 0) { //계수가 양수인 경우
            if (i == 0) os << poly.termArray[i].coefficient; // 첫 항 x^0 은 1이기에 계수부만
표시
            else {
                os << "+" << poly.termArray[i].coefficient; // 첫 항 아님
            }
        }
        else if (poly.termArray[i].coefficient < 0) { //계수가 음수인 경우
            os << poly.termArray[i].coefficient;
        }
        else if(poly.termArray[i].coefficient==0){ // 계수가 0인 경우(다항식 곱이 0이 되는
경우)
            os << 0;
        }
    }
```

```
    }
    cout << endl;
    //cout << "항의 수:"<< poly.terms << endl;
    return os; //ostream 객체 리턴
};
```

^ **출력연산자 오버로딩** - 입력 연산자와 마찬가지로 입력 연산자 클래스에 대한 객체와, 피연산 객체에 대한 참조가 필요하기에 os와 poly의 참조를 받는다. 먼저 지수값을 기준으로 지수가 1일 때와 지수가 0인 경우, 1과 0 둘다 아닌 경우로 구분하여 각각의 경우에 대해 계수가 양수일 때와 음수일 때를 구별하여 출력할 수 있다. 이때 양수인 계수의 항이 맨 처음 출력되는 경우는 앞에 +가 오면 조건에 맞지 않기 때문에 에외처리를 해준다. 계수가 0인 경우에 대해서는 입력 연산자 오버로딩에서 예외처리를 했기 때문에 따로 처리를 하지 않는다.단 다항식의 곱 연산에서 도출된 지수0-계수0 에 대한 식은 따로 출력을 해주기로 한다.

```
void sum(){ //해당 다항식 객체에 대해 같은 지수에 대하여 식을 정리하는 함수
    for(int i = 0; i<terms; i++){ // 기준 index

        for(int j = i+1; j<terms; j++) { // 비교 인덱스, i보다 한칸 뒤여야 함

            if(termArray[i].exponent == termArray[j].exponent){// 두 항의 지수가 같다면
                float tmpCoef = termArray[i].coefficient + termArray[j].coefficient; // 계수를 합하고
                termArray[i] = Term(tmpCoef,termArray[i].exponent); // 합한 계수 및 지수에 대한 term을 생성하여 기준 인덱스에 삽입

                for(int k = j; k<terms;k++){ // 비교인덱스에 대해
                    termArray[k] = termArray[k+1]; //두 항을 합해주었으니 비교 인덱스에 있는 항을 제거하고, 그 뒤 항들을 앞으로 당겨온다.
                }
                terms--;// 항이 하나 줄었으니 총 항의 개수도 감소
            }
        }
    }
    for(int i = 0; i<terms; i++){
        if(termArray[i].coefficient == 0){// 항을 합친 이후 계수가 0인 항에 대해서는
            for(int k = i; k<terms;k++){
```

```
            termArray[k] = termArray[k+1];//0인 항들은 지우고, 각 항들을 앞으로 당긴다.
        }
        terms--; //항 개수 감소
      }
    }
    for(int i = 0; i<terms; i++) { // 버블 소트를 활용하여 정리된 다항식을 지수의 오름차순으로 다시 정리한다.
      for (int j = i + 1; j < terms; j++) {
        if(termArray[i].exponent < termArray[j].exponent){
          Term tmpTerm = termArray[i];
          termArray[i] = termArray[j];
          termArray[j] =tmpTerm;
        }
      }
    }
};
```

^ sum 함수 - 각 다항식에 대해 같은 지수를 가진 항들의 계수를 합하여 식을 정리해주는 함수이다. 객체에 대한 메소드로써 호출할 것이기 때문에 따로 인자값을 받지 않고 호출자 객체값에 대한 정보만으로 연산을 진행한다. 선택정렬 알고리즘을 기반으로 for문 두 개를 이용하여 기준 항과 비교 항으로 나누어 각 항들을 비교하고 지수가 같다면, 그 두 항의 계수를 더한 새로운 term을 생성해 기존 배열의 기준 인덱스에 할당한다. 두 개의 항을 합치고 기준 인덱스에 넣어주었으므로 비교인덱스에 있는 항은 사라져야한다. 따라서 비교 인덱스 뒤에 있는 항들을 한칸 씩 앞당기고, 총 항의 갯수가 줄었으니 terms-- 를 해준다. 이후 계수가 0인 항의 경우는 배열에서 제외해야 하므로 같은 방법을 이용해 제거 해 준다.같은 차수에 대한 항들을 정리해 준 다음, 차수가 높은 순서대로 항들을 오름차순으로 정리하기 위해 이중 반복문을 사용하여 버블정렬 알고리즘을 이용해 지수가 높은 순으로 식을 정렬했다.

```
static Polynomial mul(const Polynomial& A, const Polynomial& B){ // 두 다항식을 곱하는 함수, static키워드로 해당 객체 없이도 호출 가능하게 선언 및 정의, 두 polynomial 객체의 값을 직접적으로 변경하는 것이 아니기에 const 키워드 선언
    Polynomial tempP;
    // 임시 polynomial 객체 생성
    if(A.terms !=0 and B.terms !=0){
      for(int i = 0; i<A.terms; i++){ // 기준 인덱스
        for(int j = 0; j<B.terms; j++){ // 비교 인덱스
          float tmpCoef = A.termArray[i].coefficient * B.termArray[j].coefficient; // 계수는 곱하기
          int tmpExp = A.termArray[i].exponent + B.termArray[j].exponent; // 지수는 더하기
```

```
            tempP.newTerm(tmpCoef,tmpExp); // 해당 지수와 계수를 가지는 새로운 Term을 tempP의 term배열에 할당
        }
    }
}
else{// 다항식을 sum으로 정리한 식에 대해 아무런 항도 존재 하지 않으면 다항식 끼리의 곱셈은 0을 곱한 것으로 간주함
        tempP.newTerm(0,0);
        return tempP;
}
tempP.sum(); //식 정리 함수 호출
return tempP;
}
```

^ 두 다항식의 곱 함수 mul - 객체 없이 단독으로 함수를 호출하여 비교할 다항식 객체 2개를 인자로 받을 것이기 때문에 static 형태의 함수를 만들어 전역 함수로 만들어준다. 두 다항식 객체의 값을 참고하여 새로운 다항식 객체를 생성해 반환할 것이기 때문에 두 인자 모드 const 키워드로 받는다. 이중 배열을 통해 모든 서로항에 대한 곱을 구현하고 임시 계수와 지수 변수에 값을 할당해 임시 계수와 임시 지수를 가지는 term을 새 다항식 객체의 term 배열에 newterm을 통해 넣어준다. 단 입력받은 두 다항식에 대해 하나 이상의 식이 항 자체가 존재하지 않는 경우(다항식 정리로 합이 0이 되버린 경우)는 다항식 과 0의 곱으로 생각하여 이후 x값 대입시 0 이 도출되도록 계수 0 지수 0인 term 하나가 있는 객체를 반환한다.이후 sum 메소드를 호출하여 식을 정리한 후 반환한다.

```
void newTerm(const float theCoeff, const int theExp) // 계수와 지수의 값 변동을 할 것이 아니기에 const 키워드 선언
{
    if (terms == capacity) //capacity의 초기값이 1이니, 항의 갯수가 term배열의 크기와 같을때마다 배열의 크기를 늘려주어야 함
    {
        capacity *= 2; // 배열 크기 두배로 늘리기
        Term *temp = new Term[capacity]; //new array // 늘어난 capacity만큼의 크기를 가지는 새로운 term배열에 대한 주소를 temp에 할당
```

```
        copy(termArray, termArray + terms, temp); //
termarray의 첫번째 원소 주소에서, 항이 저장되어 있는 공간까지를 temp에
복사
        delete[]termArray; //기존 term 배열 삭제
        termArray = temp; // 기존 배열에 복사한 배열 할당
    }
    termArray[terms].coefficient = theCoeff; //terms 단위는
1부터 시작하고, 배열은 0부터 시작하기 때문에 term 삽입이후 terms는 항상
삽입된 term 뒤의 위치를 가르키게 된다.
    termArray[terms++].exponent = theExp;
};
```

^ newTerm함수 - term객체를 받을 배열은 크기의 가변성을 위해 포인터로 선언하여 new로 동적 할당 및 초기화 하였다. 배열의 크기를 늘리기 위해서는 기존 배열의 크기보다 큰 크기를 가진 배열을 동적할당 해주고 임시 포인터 변수에 할당한다. 그리고 기존 배열의 원소에 대한 정보를 가지고 와야 하기 때문에 copy라이브러리 함수를 이용하여 기존 배열의 시작지점의 주소값과 항이 존재하는 끝 값의 주소값을 넣어 temp가 바인딩하는 새로운 term배열에 복사한다. 기존 termarray는 역할을 다했기 때문에 delete [] 를 이용해 바인딩하는 배열의 메모리를 지워주고 temp의 주소값을 기존 termarray에 할당하여 기존의 원소는 유지하고 크기가 늘어난 새로운 배열을 만들게 된다. terms는 항의 총 갯수이면서 term배열 안에선 마지막으로 항이 있는 원소의 바로 뒤 index를 가리키기에 terms를 인덱스로 하여 인자로 받은 계수와 지수에 대한 term 객체를 할당한다.

```
Polynomial& operator=(const Polynomial &A){ // 할당 연산자
오버로딩
    this->termArray = A.termArray; // 피 연산 객체에 인자로 들
어온 객체의 termarray, terms, capacity를 할당해 인자로 들어온 객체
를 바인딩하도록 함.
    this->terms = A.terms;
    this->capacity = A.capacity;

    return *this;
}
```

^ =연산자 오버로딩 - polynomial 객체에 대한 = 연산자가 구현되지 않았기 때문에, mul static 함수에 대한 반환값을 할당하기 위해서 구현한다. 여기서 this는 = 기준 앞에 있는 객체이고, 인자로 받은 polynomial 객체에 대한 정보를 여기에 각각 넣어준다.결국 대입이 완료된 연산자 호출 객체인 = 앞의 객체가 반환되어야 연산 수행에 대한 정보가 =기준 앞의 객체에 저장되기 때문에 *this를 반환한다.

```
include <iostream>
#include "polynomial.h" //polynomial 객체 삽입

using namespace std; //std namespace를 사용하여 관련 메소드를 std:: 명시 없이 사용한다.

int main(){
    Polynomial A,B,C; // 다항식 객체 A B C 생성
    int x; // 다항식의 미지수에 대입할 값을 저장하는 변수

    cout << "A(x) 입력:";
    cin >> A;

    cout << "B(x) 입력:";
    cin >> B;

    cout << "A(x)=" << A;
    cout << "B(x)=" << B;

    C = Polynomial::mul(A, B); // mul 연산을 통해 A와 B다항식을 곱한 다항식을 C에 바인딩
    cout << "A(x) * B(x) =" << C;
    cout << "대입 값 x 입력:";
    cin >> x;

    cout <<"A("<<x<<")="<<A.eval(x)<<endl<< "B(" << x << ")=" << B.eval(x) << " C(" << x
<< ")=" << C.eval(x) << endl;
    //출력 형식에 맞게 각 식에 x=? 를 대입한 값 출력
    return 0;
}
```

^ Main.cpp - polynomial 객체 A,B,C를 constructor의 기본 형태를 이용하여 생성한 후, A와 B에 대해 지수 계수 값을 입력받아 해당 식을 출력한 다음, static함수 mul을 이용해 두 A,B 다항

식 객체를 곱한 값을 오버로딩된 = 연산자를 통해 C에 할당한다. 이후 대입할 값 x를 입력받고 eval메소드를 이용해 각 식에 대해 x를 대입한 값을 출력한다.

3-1. 결과 해설

A(x) 입력:3 -1 0 2 4 3 -1

B(x) 입력:-2 4 5 1 0
A(x)=4x^3-1
B(x)=-2x^4+5x
A(x) * B(x) =-8x^7+22x^4-5x
대입 값 x 입력:1
A(1)=3
B(1)=3 C(1)=9

A입력을 마무리 할 때 엔터를 두 번 누르면 B입력으로 넘어간다. 입력과 출력값을 비교하면 A다항식은 3 -1 에서 지수가 -1 즉 음수인 경우는 항에 포함하지 않았고 0-2 에서 계수가 0인 항도 포함하지 않았다. 4-3에서 4x^3을 항에 포함하고, -1이 단독으로 계수로서 주어졌기 때문에 뒤의 지수는 0이라고 자동으로 판단하고 상수항 -1로 식에 포함되었다.
B입력의 경우 2-4 => 2x^4의 식으로 항에 포함 되었고 5-1 => 5x^1식으로 항에 포함되었다. 이후 0이 단독 계수로 주어지고 식이 마무리 되었기에 항에 포함되지 않도록 하여, 지수로 음수가 오는 경우, 계수가 0인 경우, 입력이 홀수개로 마무리 되어 계수만 주어진 경우, 음수와 0으로 입력이 끝나는 경우에 대한 모든 예외처리가 적용된 것을 알 수 있다.
이후 bruteforce 알고리즘을 이용해 A와 B 다항식의 모든항이 서로 곱해져서 출력되는 A*B식의 값은 정상적으로 출력된다. 이때 같은 지수 항 끼리 계수 정리가 이루어지고, 지수 기준으로 오름차순 정렬이 적용되는 것을 알 수 있다. x에 1을 대입했을 때의 각 식의 값과 AXB식의 값을 비교하면 정상적으로 다항식의 곱이 이루어지고, x입력이 정상적으로 이루어 진 것을 알 수 있다.

3-2. 배운점, 어려웠던 점 등

constructor 도 다른 메서드 처럼 오버로딩이 가능하다는 것을 알게되어 기본적인 상황에서의 constructor에 대한 정의와, 인자가 들어오는 특수한 상황에 대해 멤버변수 값을 임의로 지정할 수 있는 것을 알 수 있었다. int float 처럼 임이의 클래스 객체도 결국 type 의 하나가 되기 때문에 class type에 해당하는 배열을 만들 수 있다. 배열은 결국 type만큼의 메모리 공간이 연속적으로 붙어있는 형태이기 때문에 포인터 변수를 이용한다면 배열과 같은 기능을 하도록 할 수 있다. 이런 방식으로 배열을 선언할 때 해당 포인터 변수에 시작 주소는 같지만 총 길이가 늘어나거나 줄어드는 배열을 할당하여 크기의 가변을 구현할 수 있는데, 이때 메모리 할당과 각 원소에 대한 초기화가 필요하다. 이 상황에 사용하는 키워드가 new와 delete 이다. 메모리 동적할당과 동시에 초기화를 시켜주는 new와 할당 메모리를 삭제해주는 delete인데, new를 통해 타입[size] 형태로 배열을 만들면 해당 공간을 생성하고, 바인딩 포인터 주소에 type에 해당하는 메모리를 size만큼 연결한 배열의 첫번째 주소를 전달한다. 이렇게 new와 delete가 어떻게 작동하는지에 대해 알 수 있었다.

클래스 메서드의 경우 static 키워드가 붙어 전역 함수화가 되는 것이 아니라면, 객체.멤버함수()

의 형태로 호출하게 되는데 이 때 멤버함수는 해당 객체에 의해 호출되는 것이기 때문에 객체가 가지고 있는 모든 정보를 사용할 수 있다. 만약 함수 내에서 해당 객체의 멤버변수를 사용하는 경우 아무런 명시없이 멤버 변수의 이름을 사용하면 해당 객체의 멤버변수인 것을 인식하지만 실제로는 this->멤버변수 와 같이 작동된다.
이와 상응되는 것이 static 멤버 함수인데, 엄밀이 말해서 static멤버 함수는 멤버 함수가 아니라 전역화된 함수이다. 객체 메서드의 경우에는 호출을 위해선 반드시 호출객체가 필요한데 static 키워드가 붙는 순간 전역화가 되면서, 오직 인자로 받는 변수에 관해 참조할 수 있다. 또한 전역화가 되었기 때문에 어떤 클래스에서 선언 및 정의되었는지와는 상관없이 함수 단독으로 호출할 수 있게 된다.

call by value 나 call by address는 인자로 받은 변수의 값이나 주소값을 parameter에 복사해 주는 것이기에 const 명시의 중요성이 크게 중요하지 않지만, call by reference는 참조 인자에 대해 메모리 자체를 공유하게 되기 때문에 value의 변경 유무에 따라 const 명시를 명확히 해야 한다. 만약 레퍼런스로 가지고 온 인자의 값을 참조만 하고 값을 변경하지 않는다면, const 키워드를 사용하는 것이 바람직하다.

cin.get()과 cin의 차이점을 구별하는 것이 어려웠는데, cin의 경우는 공백이나 엔터등을 입력의 마무리로 받아들이고 입력버퍼에 해당 기호(스페이스나 엔터)가 남게 되지만, cin.get()은 입력버퍼에 값이 남아 있다면 그 값을 가져오게 된다.

지수없이 계수만 주어지는 경우를 처리하기가 까다로웠는데, 입력단에서 계수만 주어지는 경우(입력 숫자가 홀수개인 경우)는 지수를 0으로 인식하고 상수항으로 입력 받도록 함으로써 문제를 해결할 수 있었다.