資料結構
作業二

41243152 劉康申

解題說明

題目要求：

```
class Polynomial {
// p(x) = a_0 x^{e_0} + ··· + a_n x^{e_n}; a set of ordered pairs of <e_i, a_i>,
// where a_i is a nonzero float coefficient and e_i is a non-negative integer exponent.
public:
    Polynomial();
    // Construct the polynomial p(x) = 0.

    Polynomial Add(Polynomial poly);
    // Return the sum of the polynomials *this and poly.

    Polynomial Mult(Polynomial poly);
    // Return the product of the polynomials *this and poly.

    float Eval(float f);
    // Evaluate the polynomial *this at f and return the result.
};
```

```
class Polynomial ; // forward declaration

class Term {
friend Polynomial;
private:
    float coef;   // coefficient
    int exp;      // exponent
};
```

The private data members of *Polynomial* are defined as follows:

```
private:
    Term *termArray;   // array of nonzero terms
    int capacity;      // size of termArray
    int terms;         // number of nonzero terms
```

1.實作以上兩類別，以實現多項式運算。

2.實現多載輸入輸出。

解題思路：

為實現題目要求的多項式運算，及多載輸入輸出，我分別在以下兩類別中實作了這些功能：

-函式(operator>>)：輸入的多載函式。

-函式(operator<<)：輸出的多載函式。

Term：

-浮點數成員Coef：作為多項式中的係數。

-整數成員exp：作為多項式中的指數。

-定義朋友類別(Polynomial)：使類別Polynomial可以完整取用私有成員。

-定義朋友函式(operator>>)：使函式可取用完整私有成員。

-定義朋友函式(operator<<)：使函式可取用完整私有成員。

Polynomial：

-物件成員termArray：可存入Term類別中Coef及exp成員的動態陣列。

-整數成員capacity：動態陣列大小。

-整數成員terms：用來表示多項式項數。

-函式(resize)：初始化及擴充動態陣列大小。

-建構子：初始化termArray為空、capacity為0、terms為0。

-解構子：釋放動態陣列termArray。

-複製建構子：以現有物件的內容來初始化一個新的物件。

-函式(Add)：做多項式相加。

-函式(Mult)：做多項式相乘。

-函式(Eval)：帶值進多項式。

程式實作

Term：

```
class Term {
    friend class Polynomial;
    friend istream& operator>>(istream& is, Polynomial& poly);
    friend ostream& operator<<(ostream& os, Polynomial poly);
private:
    float coef; //coefficient.
    int exp;    //exponent.
};
```

Polynomial：

```
class Polynomial {
private:
    void resize() {
        if(termArray!=nullptr){
            int newcapacity = capacity * 2;
            Term* newarray = new Term[newcapacity];
            copy(termArray, termArray + capacity, newarray);
            delete[] termArray;
            termArray = newarray;
            capacity = newcapacity;
        }
        else {
            capacity = 2;
            termArray = new Term[capacity];
        }
    }
    Term* termArray; //array of nonzero terms.
    int capacity;    //size of termArray.
    int terms;       //number of nonzero terms.
public:
    Polynomial():termArray(nullptr),capacity(0),terms(0){}
    Polynomial(const Polynomial& poly) {
        terms = poly.terms;
        capacity = poly.capacity;
        termArray = new Term[capacity];
        for (int i = 0; i < terms; i++) {
            termArray[i] = poly.termArray[i];
        }
    }
    ~Polynomial() { delete[] termArray; }
    Polynomial Add(Polynomial poly);
    Polynomial Mult(Polynomial poly);
    float Eval(float f);
    friend istream& operator>>(istream& is, Polynomial& poly);
    friend ostream& operator<<(ostream& os, Polynomial poly);
};
```

函式 (Add) ：

```
Polynomial Polynomial::Add(Polynomial poly) {
    Polynomial total;
    total.resize();
    int i = 0, j = 0, k = 0;
    while (i < terms || j < poly.terms) {
        if (i < terms && j < poly.terms) {
            if (termArray[i].exp == poly.termArray[j].exp) {
                total.termArray[k].exp = termArray[i].exp;
                total.termArray[k].coef = termArray[i].coef + poly.termArray[j].coef;
                i++;
                j++;
            }
            else if (termArray[i].exp > poly.termArray[j].exp) {
                total.termArray[k].exp = termArray[i].exp;
                total.termArray[k].coef = termArray[i].coef;
                i++;
            }
            else if (termArray[i].exp < poly.termArray[j].exp) {
                total.termArray[k].exp = poly.termArray[j].exp;
                total.termArray[k].coef = poly.termArray[j].coef;
                j++;
            }
        }
        else if (i < terms && j >= poly.terms) {
            total.termArray[k].exp = termArray[i].exp;
            total.termArray[k].coef = termArray[i].coef;
            i++;
        }
        else if (i >= terms && j < poly.terms) {
            total.termArray[k].exp = poly.termArray[j].exp;
            total.termArray[k].coef = poly.termArray[j].coef;
            j++;
        }
        k++;
        total.terms = k;
        if (k >= total.capacity) {
            total.resize();
        }
    }
    return total;
}
```

函式(Mult)：

```
Polynomial Polynomial::Mult(Polynomial poly) {
    Polynomial total;
    total.resize();
    int k = 0;
    for (int i = 0; i < terms; i++) {
        for (int j = 0; j < poly.terms; j++) {
            total.termArray[k].exp = termArray[i].exp + poly.termArray[j].exp;
            total.termArray[k].coef = termArray[i].coef * poly.termArray[j].coef;
            k++;
            total.terms = k;
            if (k >= total.capacity) {
                total.resize();
            }
        }
    }
    int g = 0;
    Polynomial result;
    result.resize();
    for (int i = 0; i < total.terms; i++) {
        result.termArray[g].exp = total.termArray[i].exp;
        result.termArray[g].coef = total.termArray[i].coef;
        for (int j = i+1; j < total.terms; j++) {
            if (total.termArray[i].exp == total.termArray[j].exp) {
                result.termArray[g].coef += total.termArray[j].coef;
                total.termArray[j].exp = 0;
                total.termArray[j].coef = 0;
            }
        }
        g++;
        result.terms = g;
        if (g >= result.capacity) {
            result.resize();
        }
    }
    for (int i = 0; i < result.terms; i++) {
        for (int n = 0; n < result.terms - 1; n++) {
            if (result.termArray[n].exp < result.termArray[n + 1].exp) {
                swap(result.termArray[n].exp, result.termArray[n + 1].exp);
                swap(result.termArray[n].coef, result.termArray[n + 1].coef);
            }
        }
    }
    return result;
}
```

函式(Eval)：

```
float Polynomial::Eval(float f) {
    float total = 0;
    for (int i = 0; i < terms; i++) {
        total += termArray[i].coef * pow(f, termArray[i].exp);
    }
    return total;
}
```

函式(operator>>)：

```
istream& operator>>(istream& is, Polynomial& poly) {
    poly.resize();
    cout << "輸入項數：";
    is >> poly.terms;
    while (poly.capacity < poly.terms) {
        poly.capacity = poly.capacity * 2;
    }
    poly.resize();
    for (int i = 0; i < poly.terms; i++) {
        cout << "輸入(係數 指數)：";
        is >> poly.termArray[i].coef >> poly.termArray[i].exp;
    }
    return is;
}
```

函式(operator<<)：

```
ostream& operator<<(ostream& os, Polynomial poly) {
    for (int i = 0; i < poly.terms; i++) {
        if (poly.termArray[i].exp == 0 && poly.termArray[i].coef == 0) continue;
        if (i > 0 && i <= poly.terms && poly.termArray[i].coef >= 0) os << "+";
        if (poly.termArray[i].coef == 1) {
            if (poly.termArray[i].exp == 0) {
                os << poly.termArray[i].coef;
            }
            else {
                os << "x^" << poly.termArray[i].exp;
            }
        }
        else {
            if (poly.termArray[i].exp == 0) {
                os << poly.termArray[i].coef;
            }
            else {
                os << poly.termArray[i].coef << "x^" << poly.termArray[i].exp;
            }
        }
    }
    return os;
}
```

主程式：

```
int main() {
    Polynomial p1, p2;
    cout << "輸入第一份多項式：" << endl;
    cin >> p1;
    cout << "輸入第二份多項式：" << endl;
    cin >> p2;
    cout << "p1:" << p1 << endl;
    cout << "p2:" << p2 << endl;
    cout << "Sum=" << p1.Add(p2) << endl;
    cout << "Mult=" << p1.Mult(p2) << endl;
    cout << "帶值：";
    float f;
    cin >> f;
    cout << "p1(" << f <<") =" << p1.Eval(f) << endl;
    return 0;
}
```

效能分析

函式(Add)：

-時間複雜度(time complexity)：

$$O(n+m)$$

-主迴圈執行時，**i** 和 **j** 分別遍歷 **this** 和 **poly** 的非零項數。

-每次操作需要比較 **termArray[i].exp** 和 **poly.termArray[j].exp**，且僅操作一次後移動索引。

-最壞情況下需要處理 **this.terms + poly.terms** 項。

-空間複雜度(space complexity)：

$$O(n+m)$$

-一個額外的 **Polynomial** 結果變數，空間需求為兩個多項式的總非零項數（因為可能沒有指數相同的項相加）。

-最壞情況下，結果項數是 **n + m**。

函式(Mult)：

-時間複雜度(time complexity)：

$$O(n\times m+k^2)$$

-外層雙重迴圈遍歷 **this** 和 **poly** 的所有項，總操作次數為 **n × m**。

-在結果整理階段（合併相同指數項），每次檢查都需要最多 **k^2** 次比較（其中 **k** 是中間結果的項數）。

-若假設 **k ≈ n \times m**，則最壞情況下為：

$$O((n\times m)^2)$$

-空間複雜度(space complexity)：

$$O(n\times m)$$

-中間結果需要 **n×m** 項（所有可能的乘積項）。

-合併過程需要一個新結果陣列（同樣大小）。

函式(Eval)：

-時間複雜度(time complexity)：

-最壞情況為：

$$O(n \times \log(e))$$

-單一迴圈遍歷所有項，每次計算 termArray[i].coef×pow(f,termArray[i].exp)。

-指數運算 pow 的時間複雜度為 O(log(e))（視實作方式），其中 e 是最高指數。

-空間複雜度(space complexity)：

$$O(1)$$

-僅需儲存單一變數 total，空間複雜度為常數。

函式(resize)：

-時間複雜度(time complexity)：

$$O(capacity)$$

-分配新陣列的操作為 O(capacity)，其中 capacity 是目前的陣列大小。

-將原陣列的內容複製到新陣列的時間也為 O(capacity)。

-空間複雜度(space complexity)：

$$O(capacity)$$

-每次會分配新的陣列，其大小為當前容量的兩倍。

函式(operator>>)：

-時間複雜度(time complexity)：

O(n)

-遍歷輸入的每一項並進行初始化，每次輸入需要常數時間。

-空間複雜度(space complexity)：

O(n)

-呼叫 resize 時會動態分配記憶體。最終需要額外的儲存空間為 O(n)。

函式(operator<<)：

-時間複雜度(time complexity)：

O(n)

-遍歷所有項，逐一輸出。

-空間複雜度(space complexity)：

O(1)

-不需要額外空間，為常數空間複雜度。

解構子：

-時間複雜度(time complexity)：

O(1)

-單次釋放動態記憶體。

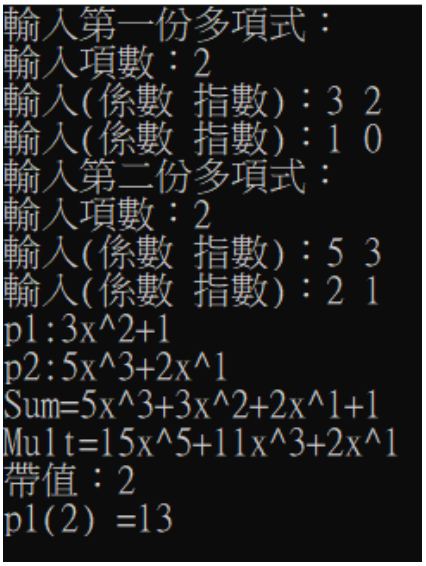
-空間複雜度(space complexity)：

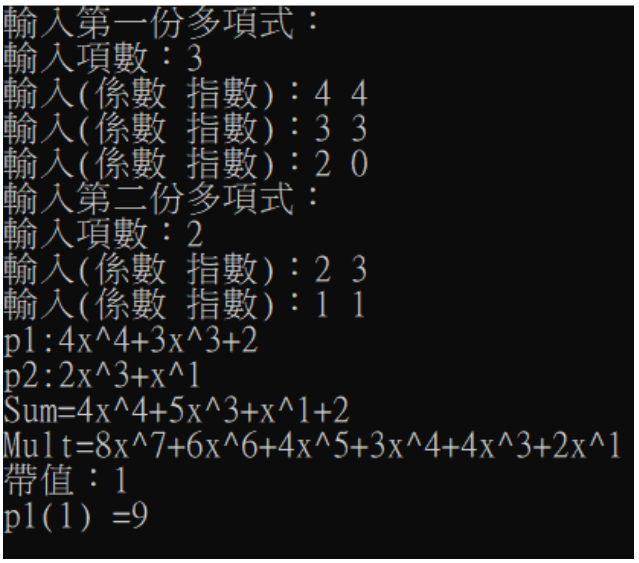
O(1)

-無額外空間需求，為常數。

測試與驗證

執行結果：





測試驗證：

驗證一：

```
p1:3x^2+1 p2:5x^3+2x^1
Sum=5x^3+3x^2+2x^1+1
Mult=15x^5+11x^3+2x
p1(2) =13
```

驗證二：

```
p1:4x^4+3x^3+2
p2:2x^3+1x^1
Sum=4x^4+5x^3+1x^1+2
Mult=8x^7+6x^6+4x^5+3x^4+4x^3+2x^1
p1(1) =9
```

申論及開發報告

程式實作思路：

-函式(Add)：

以三個指針分別指向三個不同陣列：多項式1、多項式2、相加總合。

-程式實作：

1.宣告Polynomial物件total，並且呼叫函式(resize)以初始化陣列。

2.宣告整數i、j、k分別指向total、this、poly的陣列。

3.設定while迴圈(i或j小於各自項數)。

4.以if來判斷情況，分別為：

-i與j皆小於項數：

-this陣列第i項與poly陣列第j項指數相同：

total第k項指數=this第i項=poly第j項。

total第k項項數=this第i項+poly第j項。

i及j皆指向下一個。

-this陣列第i項指數較大：

total第k項指數=this第i項。

total第k項項數=this第i項。

i指向下一個。

-poly陣列第j項指數較大：

total第k項指數=poly第j項。

total第k項項數=poly第j項。

j指向下一個。

-僅i小於項數：

total第k項指數=this第i項。

total第k項項數=this第i項。

i指向下一個。

-僅j小於項數：

total第k項指數=poly第j項。

total第k項項數=poly第j項。

j指向下一個。

5.k指向下一個(若陣列大小不夠，則呼叫resize)

-函式(resize)：

考量到建構初始狀態的陣列termArray是nullptr，"給定初始陣列大小"及"擴增陣列大小"。

-給定初始陣列大小：

設定陣列大小為2。

-擴增陣列大小：

1.宣告新陣列大小為原陣列2被。

2.將原陣列元素複製到新陣列。

3.釋放原陣列。

4.原陣列指向新陣列。

-函式(Mult)：

先將所有項相乘結果一一存入一個新的陣列，再整理指數大小、合併相同指數，並存入另一個陣列。

-程式實作：

1.宣告Polynomial物件total，並宣告k指向total陣列，並且呼叫函式(resize)以初始化陣列。

2.使用巢狀迴圈運算this和poly相乘結果(指數相加，項數相乘)

3.k指向下一個(若陣列大小不夠，則呼叫resize)

4.宣告Polynomial物件result，並宣告g指向result陣列，並且呼叫函式(resize)以初始化陣列。

5.使用巢狀迴圈合併total陣列相同指數的項，並存入result。

6.g指向下一個(若陣列大小不夠，則呼叫resize)

7.使用巢狀迴圈排序指數大小。

-函式(Eval)：

-程式實作：

1.宣告浮點數total計算求值結果。

2.以迴圈計算求值(係數*帶值^指數)

心得：

這次的作業相較於以往複雜上許多，令我感受到我對C++的理解還遠遠不足，不論是使用到的技巧、程式的規模都進階了不少，讓我在設計程式和除錯的過程中都吃上了不少苦頭，但這同時也讓我學會很多：朋友類別、動態記憶體、多載功能、複製建構子等等。當然我也明白我所撰寫出的程式還不是最佳解，還有更好且更有效率的處理辦法，我會反覆精進自己，不斷改善程式，不斷進步！