

# Homework 2

## 解題說明：

實作多項式存儲與運算。

## Algorithm Design & Programming

Term 類別 (Term)：

有係數和次方項兩個屬性。

Polynomial 類別 (Polynomial)：

動態陣列 termArray 存儲多項式的項。

有容量 (capacity) 和項數 (terms) 屬性。

提供新增項、加法、乘法和代入值的方法。

```
#include <iostream>
#include <math.h>
using namespace std;
class Term {
    friend class Polynomial;
    friend ostream& operator<<(ostream& os, const Polynomial& p);
    friend istream& operator>>(istream& is, const Polynomial& p);
private:
    float coef; // 係數
    int exp;    // 次方項

public:
    float GetCoef() const { return coef; }
    int GetExp() const { return exp; }
};

class Polynomial {
    friend ostream& operator<<(ostream& os, const Polynomial& p);
    friend istream& operator>>(istream& is, const Polynomial& p);
private:
    Term* termArray;
    int capacity; // 空間大小
    int terms;    // 非零項數量

public:
    Polynomial(); // 建構子
    Polynomial(const Polynomial& poly); // 複製建構子
    ~Polynomial(); // 解構子
    Polynomial Add(const Polynomial& poly);
    Polynomial Mult(const Polynomial& poly);
    float Eval(float f);
    void NewTerm(const float newCoef, const int newExp); // 新增非零項
};

Polynomial::Polynomial() : capacity(2), terms(0) {
    termArray = new Term[capacity];
}

Polynomial::Polynomial(const Polynomial& poly) {
    capacity = poly.capacity;
    terms = poly.terms;
```

```

    termArray = new Term[capacity];
    for (int i = 0; i < terms; ++i) {
        termArray[i] = poly.termArray[i];
    }
}

Polynomial::~Polynomial() {
    //刪除不需要的陣列
    delete[] termArray;
}

void Polynomial::NewTerm(const float newCoef, const int newExp) {
    // 空間不足時重新配置空間
    if (capacity == terms) {
        int newCapacity = capacity * 2;
        Term* tempArray = new Term[newCapacity];
        for (int i = 0; i < capacity; ++i) {
            tempArray[i] = termArray[i];
        }
        delete[] termArray;
        termArray = tempArray;
        capacity = newCapacity;
    }
    termArray[terms].coef = newCoef;
    termArray[terms].exp = newExp;
    terms++;
}

Polynomial Polynomial::Add(const Polynomial& poly) {
    Polynomial result;
    int i = 0, j = 0;
    while (i < terms && j < poly.terms) {
        if (termArray[i].exp == poly.termArray[j].exp) {
            result.NewTerm(termArray[i].coef + poly.termArray[j].coef, termArray[i].exp);
            i++;
            j++;
        }
        else if (termArray[i].exp > poly.termArray[j].exp) {
            result.NewTerm(termArray[i].coef, termArray[i].exp);
            i++;
        }
        else {
            result.NewTerm(poly.termArray[j].coef, poly.termArray[j].exp);
            j++;
        }
    }
    while (i < terms) {
        result.NewTerm(termArray[i].coef, termArray[i].exp);
        i++;
    }
    while (j < poly.terms) {
        result.NewTerm(poly.termArray[j].coef, poly.termArray[j].exp);
        j++;
    }
    return result;
}

Polynomial Polynomial::Mult(const Polynomial& poly) {
    Polynomial result;
    for (int i = 0; i < terms; ++i) {
        for (int j = 0; j < poly.terms; ++j) {

```

```

        int newExp = termArray[i].exp + poly.termArray[j].exp;
        float newCoef = termArray[i].coef * poly.termArray[j].coef;
        result.NewTerm(newCoef, newExp);
    }
}
return result;
}

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

ostream& operator<<(ostream& os, const Polynomial& p) {
    for (int i = 0; i < p.terms; ++i) {
        if (p.termArray[i].GetExp() == 0) {
            os << p.termArray[i].GetCoef();
        }
        else {
            os << p.termArray[i].GetCoef() << "X^" << p.termArray[i].GetExp();
        }
        if (i != p.terms - 1) {
            os << " + ";
        }
    }
    return os;
}

istream& operator>>(istream& is, Polynomial& p) {
    int terms;
    float coef, exp;

    is >> terms;
    while (cin >> coef >> exp) {
        if (coef == -1 && exp == -1) {
            break; // 結束輸入
        }
        p.NewTerm(coef, terms);
    }
    return is;
}

int main() {
    Polynomial poly1, poly2;

    // 多項式1的輸入
    cout << "輸入多項式1(係數、次方項) : " << endl;
    cin >> poly1;
    // 多項式2的輸入
    cout << "輸入多項式2(係數、次方項) : " << endl;
    cin >> poly2;

    //輸出多項式
    cout << "多項式 1: " << poly1 << endl;
    cout << "多項式 2: " << poly2 << endl;

    // 計算Add和Mult
    Polynomial additionResult = poly1.Add(poly2);
    Polynomial multiplicationResult = poly1.Mult(poly2);
}

```

```

// 輸出Add和Mult的計算結果
cout << "Add : " << additionResult << endl;
cout << "Mult : " << multiplicationResult << endl;

//計算X代入後多項式的值
float value;
cout << "輸出X : ";
cin >> value;

cout << "Eval 1 = " << value << ": " << poly1.Eval(value) << endl;
cout << "Eval 2 = " << value << ": " << poly2.Eval(value) << endl;

return 0;
}

```

## 效能分析:

建構子 `Polynomial::Polynomial()`: 時間複雜度為  $O(1)$ , 僅涉及初始化。

複製建構子 `Polynomial::Polynomial(const Polynomial& poly)`: 時間複雜度為  $O(n)$ , 其中  $n$  是多項式中的項數。

解構子 `Polynomial::~~Polynomial()`: 時間複雜度為  $O(1)$ , 僅涉及釋放資源。

`Polynomial::NewTerm(const float newCoef, const int newExp)`: 在空間不足時重新配置空間的時間複雜度為  $O(n)$ , 其中  $n$  是項數。

`Polynomial::Add(const Polynomial& poly)`: 對兩個多項式進行加法操作的時間複雜度為  $O(m + n)$ , 其中  $m$  和  $n$  是兩個多項式的項數。

`Polynomial::Mult(const Polynomial& poly)`: 對兩個多項式進行乘法操作的時間複雜度為  $O(m * n)$ , 其中  $m$  和  $n$  是兩個多項式的項數。

`Polynomial::Eval(float f)`: 對多項式進行評估的時間複雜度為  $O(n)$ , 其中  $n$  是多項式的項數。

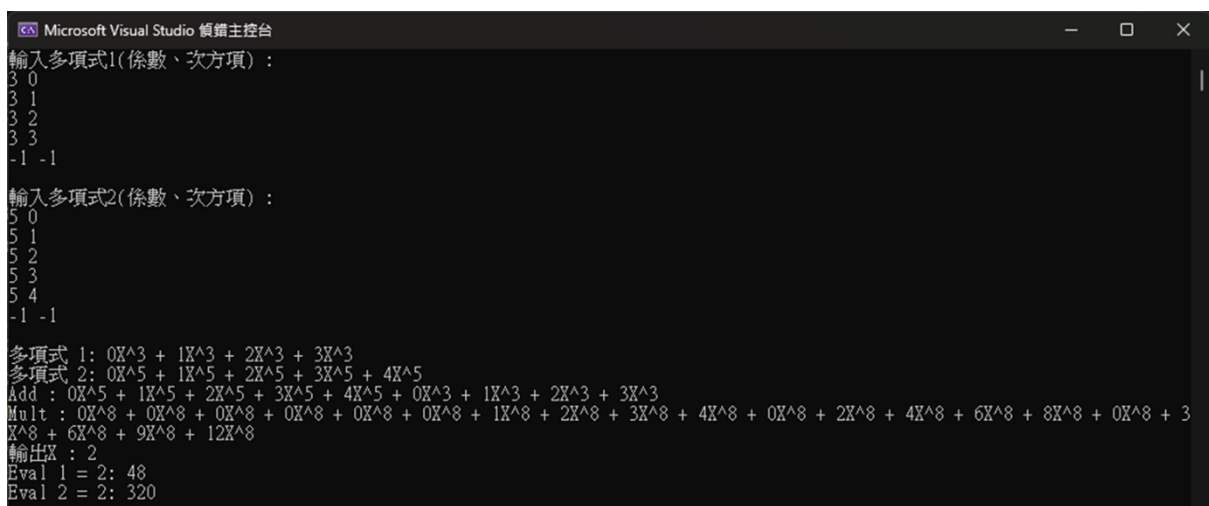
### 空間複雜度

`Polynomial` 物件的空間複雜度: 主要佔用空間的地方是 `Term` 陣列, 其大小隨著項數而變化。因此, 空間複雜度是  $O(n)$ , 其中  $n$  是多項式的項數。

這段程式碼的時間複雜度主要取決於操作的多項式的項數。在加法和乘法操作中, 需要對兩個多項式的每一項進行比較和操作, 因此它們的複雜度是  $m$  和  $n$  的線性或乘法關係。

另外, 效能分析也會受到實際系統的影響, 因為有些操作可能比預期更快或更慢, 具體取決於編譯器、硬體等因素。

## 測試與驗證:



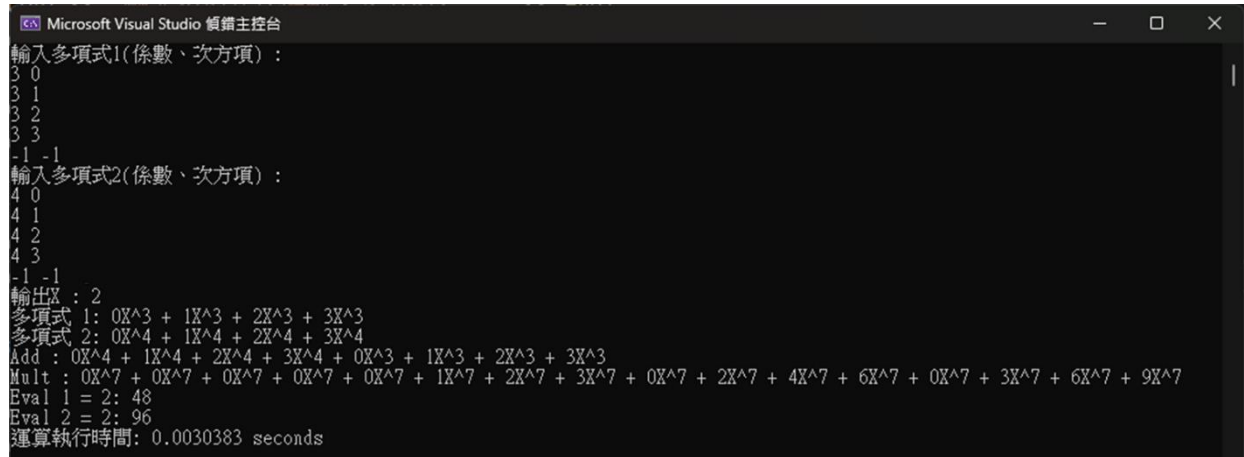
```

Microsoft Visual Studio 偵錯主控台
輸入多項式1(係數、次方項):
3 0
3 1
3 2
3 3
-1 -1
輸入多項式2(係數、次方項):
5 0
5 1
5 2
5 3
5 4
-1 -1
多項式 1: 0X^3 + 1X^3 + 2X^3 + 3X^3
多項式 2: 0X^5 + 1X^5 + 2X^5 + 3X^5 + 4X^5
Add: 0X^5 + 1X^5 + 2X^5 + 3X^5 + 4X^5 + 0X^3 + 1X^3 + 2X^3 + 3X^3
Mult: 0X^8 + 0X^8 + 0X^8 + 0X^8 + 0X^8 + 0X^8 + 1X^8 + 2X^8 + 3X^8 + 4X^8 + 0X^8 + 2X^8 + 4X^8 + 6X^8 + 8X^8 + 0X^8 + 3X^8 + 6X^8 + 9X^8 + 12X^8
輸出X: 2
Eval 1 = 2: 48
Eval 2 = 2: 320

```

## 效能測量:

使用<chrono>函式進行計時，在輸入多項式後開始計時，直到結果輸出結束，並且輸出時間，結果如下：



```
Microsoft Visual Studio 幀繪主控台
輸入多項式1(係數、次方項) :
3 0
3 1
3 2
3 3
-1 -1
輸入多項式2(係數、次方項) :
4 0
4 1
4 2
4 3
-1 -1
輸出X : 2
多項式 1: 0X^3 + 1X^3 + 2X^3 + 3X^3
多項式 2: 0X^4 + 1X^4 + 2X^4 + 3X^4
Add : 0X^4 + 1X^4 + 2X^4 + 3X^4 + 0X^3 + 1X^3 + 2X^3 + 3X^3
Mult : 0X^7 + 0X^7 + 0X^7 + 0X^7 + 0X^7 + 1X^7 + 2X^7 + 3X^7 + 0X^7 + 2X^7 + 4X^7 + 6X^7 + 0X^7 + 3X^7 + 6X^7 + 9X^7
Eval 1 = 2: 48
Eval 2 = 2: 96
運算執行時間: 0.0030383 seconds
```

## 心得:

這次作業讓我深入了解了運算子多載在 **C++** 中的應用。透過實作多項式類別和其相關操作，如加法、乘法、評估值等，我理解到如何運用運算子多載來提高程式碼的可讀性和易用性。尤其在輸入輸出部分，透過重載 << 和 >> 運算子，讓我可以更直觀地輸出多項式物件，提升了程式的易讀性。

此外，這份作業也讓我更加熟悉了動態記憶體配置和釋放的概念，在多項式類別中，我使用了動態配置 **Term** 陣列來儲存項，並且適時地釋放記憶體，以避免不必要的資源浪費。

運算子的多載讓程式碼更具有彈性和直觀性，能夠更自然地表達操作。透過這次作業，我不僅加深了對運算子多載的理解，也對 **C++** 中類別設計和動態記憶體管理有了更進一步的認識。這對我日後開發更複雜的程式和應用中會大有幫助。