解題說明

Polynomial.cpp

`Polynomail()`:

- 建構子，構造函數會初始化 `capacity` 為 10，`terms` 為 0，並為 `termArray` 分配內存。

`Resize()`:

- 這個函數會在當前的 `termArray` 容量不足時，將容量加倍，並將舊的項目移到新的陣列中。
- 這樣的處理使得我們可以動態擴充 `termArray`，防止溢出。

`Add()`:

- 這個函數的目的是將兩個多項式相加。
- 我們逐個比對兩個多項式的項目：
  - 如果次方數相同，則將係數相加。
  - 如果次方數不同，則直接將項目加入結果多項式。
- 若相加的結果為零，則跳過該項目。

`Mult()`:

- 這個函數實現了多項式的乘法。
- 我們遍歷兩個多項式的所有項目，對每一對項目進行相乘，並將結果項目加入 `result`。

- 如果結果的次方數已經存在於 `result` 中，就將係數累加，否則就新增項目。

`Eval()`:

- 這個函數用來計算多項式在某個特定 `x` 值下的結果。
- 我們使用了 `pow(f, exp)` 計算每個項目的 `x` 次方，並將每項的結果累加起來。

Operator.cpp

`operator<<`:

- 這個函數實現了輸出多項式的功能。
- 我們遍歷 `Polynomial` 中的所有項目，將它們按照 `係數x^次方數` 的格式輸出。
- 如果有多個項目，我們會在每個項目後加上 `+`，除非是最後一個項目。

`operator>>`:

- 這個函數實現了從標準輸入讀取多項式的功能。
- 我們使用一個 `while` 迴圈來反覆讀取用戶輸入的係數和次方數。
- 若讀取的係數或次方數無效(例如空格或換行)，則退出循環。
- 在每次讀取新項目之前，我們檢查 `termArray` 是否已滿，若滿了則調用 `Resize()` 增加容量。

程式實作

Polynomial.h

```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#include <iostream>
using namespace std;

class Term {
public:
    float coef;  // 係數
    int exp;     // 次方數
};

class Polynomial {
public:
    Polynomial();  // 預設建構子
    Polynomial Add(const Polynomial& poly);  // 多項式相加
```

```
    Polynomial Mult(const Polynomial& poly);  // 多項式相乘
    float Eval(float f);  // 代入特定值計算多項式的結果

    // 運算子重載
    friend ostream& operator<<(ostream& out, const Polynomial& poly);
    friend istream& operator>>(istream& in, Polynomial& poly);

private:
    Term* termArray;  // 儲存項目陣列
    int capacity;     // 陣列容量
    int terms;        // 多項式中非零項的數量

    void Resize();  // 調整容量大小
};

#endif // POLYNOMIAL_H
```

main.cpp

```
#include <iostream>

#include "Polynomial.h"


using namespace std;


int main() {

    Polynomial p1, p2, result;

    cout << "請依序輸入多項式的係數與次方數(結束輸入以換行):";

    cin >> p1;

    cout << "第一個多項式為:" << p1 << endl;


    cout << "請依序輸入第二個多項式的係數與次方數(結束輸入以換行):";

    cin >> p2;

    cout << "第二個多項式為:" << p2 << endl;


    result = p1.Add(p2);

    cout << "兩多項式相加結果為:" << result << endl;
```

```
    result = p1.Mult(p2);
    cout << "兩多項式相乘結果為:" << result << endl;

    float x;
    cout << "請輸入一個x值代入第一個多項式進行計算:";
    cin >> x;
    cout << "計算結果為:" << p1.Eval(x) << endl;

    return 0;
}
```

Polynomial.cpp

```
#include "Polynomial.h"
#include <iostream>
#include <cmath>
using namespace std;

Polynomial::Polynomial() {
    capacity = 10;
    terms = 0;
    termArray = new Term[capacity];
}

void Polynomial::Resize() {
    capacity *= 2;
    Term* newArray = new Term[capacity];
    for (int i = 0; i < terms; i++) {
        newArray[i] = termArray[i];
    }
    delete[] termArray;
```

```
    termArray = newArray;
}

// 多項式相加
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) {
            float sumCoef = termArray[i].coef +
poly.termArray[j].coef;
            if (sumCoef != 0) {
                if (result.terms == result.capacity) {
                    result.Resize();
                }
                result.termArray[result.terms].coef = sumCoef;
                result.termArray[result.terms].exp =
termArray[i].exp;
                result.terms++;
```

```
    }
    i++;
    j++;
}
else if (termArray[i].exp > poly.termArray[j].exp) {
    if (result.terms == result.capacity) {
        result.Resize();
    }
    result.termArray[result.terms] = termArray[i];
    result.terms++;
    i++;
}
else {
    if (result.terms == result.capacity) {
        result.Resize();
    }
    result.termArray[result.terms] = poly.termArray[j];
    result.terms++;
    j++;
```

```
    }
}

while (i < terms) {
    if (result.terms == result.capacity) {
        result.Resize();
    }
    result.termArray[result.terms] = termArray[i];
    result.terms++;
    i++;
}

while (j < poly.terms) {
    if (result.terms == result.capacity) {
        result.Resize();
    }
    result.termArray[result.terms] = poly.termArray[j];
    result.terms++;
    j++;
```

```
    }

    return result;
}

// 多項式相乘
Polynomial Polynomial::Mult(const Polynomial& poly) {
    Polynomial result;

    for (int i = 0; i < terms; i++) {
        for (int j = 0; j < poly.terms; j++) {
            float newCoef = termArray[i].coef *
poly.termArray[j].coef;
            int newExp = termArray[i].exp +
poly.termArray[j].exp;

            bool added = false;
            for (int k = 0; k < result.terms; k++) {
                if (result.termArray[k].exp == newExp) {
                    result.termArray[k].coef += newCoef;
```

```
                added = true;
                break;
            }
        }

        if (!added && newCoef != 0) {
            if (result.terms == result.capacity) {
                result.Resize();
            }
            result.termArray[result.terms].coef = newCoef;
            result.termArray[result.terms].exp = newExp;
            result.terms++;
        }
    }
  }

  return result;
}
```

```
// 計算多項式在特定 x 值下的結果
float Polynomial::Eval(float f) {
    float result = 0;
    for (int i = 0; i < terms; i++) {
        result += termArray[i].coef * pow(f,
termArray[i].exp);
    }
    return result;
}
```

Operator.cpp

```
#include "Polynomial.h"

#include <iostream>

using namespace std;


// 輸出運算子 <<

ostream& operator<<(ostream& out, const Polynomial&
poly) {

    for (int i = 0; i < poly.terms; i++) {

        out << poly.termArray[i].coef << "x^" <<
poly.termArray[i].exp;

        if (i < poly.terms - 1) {

            out << " + ";

        }

    }

    return out;

}


// 輸入運算子 >>

istream& operator>>(istream& in, Polynomial& poly) {
```

```
poly.terms = 0;
while (true) {
    float coef;
    int exp;

    // 讀取係數
    in >> coef;
    if (in.fail()) {
        break;  // 如果無法讀取係數，跳出循環
    }

    // 讀取次方數
    in >> exp;
    if (in.fail()) {
        break;  // 如果無法讀取次方數，跳出循環
    }

    // 儲存多項式項目
```

```
        if (poly.terms == poly.capacity) {
            poly.Resize();
        }

        poly.termArray[poly.terms].coef = coef;
        poly.termArray[poly.terms].exp = exp;
        poly.terms++;

        // 嘗試讀取換行符號並處理結束輸入
        char ch = in.get();
        if (ch == '\n') {
            break;  // 若讀到換行符號，結束循環
        }
        else {
            in.putback(ch);  // 如果讀到其他字符，將其放回緩衝區，繼續讀取
        }
    }
    return in;
}
```

效能分析

時間複雜度

- 加法 (**`Add`**):
  - 每個多項式都需要遍歷其所有項目，因此最壞情況下，這個操作的時間複雜度是 O(n + m)，其中 `n` 和 `m` 分別是兩個多項式的項目數。這是因為我們需要逐項比對並處理兩個多項式的所有項目。
  - 實際情況可能會更好，尤其是當次方數相同或項目數量較少時。
- 乘法 (**`Mult`**):
  - 多項式乘法是將每個項目與另一個多項式的所有項目相乘，因此最壞情況下的時間複雜度是 O(n * m)，其中 `n` 和 `m` 分別是兩個多項式的項目數。這是因為我們需要進行雙層迴圈遍歷兩個多項式。
  - 在多項式相乘後，我們還要遍歷結果項目來合併同次方項，這需要 O(r)，其中 `r` 是結果中項目的數量。
- 評估 (**`Eval`**):
  - 評估多項式的時間複雜度是 O(n)，其中 `n` 是多項式的項目數。每個項目需要計算一次 `coef * pow(x, exp)`，並將所有結果相加。

空間複雜度

- 每個多項式都使用了一個動態陣列來存儲項目。`capacity` 變數表示陣列的最大容量，而 `terms` 表示當前儲存的項目數量。

- 在最壞情況下，空間複雜度是 O(n)，其中 `n` 是多項式的項目數量。這是因為每個項目佔據一個 `Term` 結構體的空間，並且 `termArray` 需要儲存所有項目。
- 隨著 `Resize` 函數的調用，內存容量會動態增長，但整體空間需求不會超過多項式項目數量的兩倍（因為容量是成倍增長的）。

測試與驗證

測試與驗證是確保程式正確性和效能的重要步驟。以下是對該程式的測試與驗證方法：

單元測試

- 多項式輸入測試：測試用戶輸入各種不同形式的多項式，確保程序能正確解析輸入的係數和次方數。
  - 測試案例：
    - `1 2 2 1 3 0`（正常輸入）
    - `0 1 2 2 3 1`（係數為零的項）
    - 非標準輸入（例如空格、錯誤格式）應該能正確處理。
- 多項式加法測試：
  - 測試兩個多項式相加的正確性。
  - 測試：
    - `1x^2 + 2x^1 + 3x^0` 與 `3x^2 + 2x^1 + 1x^0` 相加，預期結果應為 `4x^2 + 4x^1 + 4x^0`。
    - 加法結果中若有零項（如 `1x^2 + -1x^2`），應該消除零項。
- 多項式乘法測試：
  - 測試兩個多項式相乘的正確性。
  - 測試：
    - `1x^2 + 2x^1 + 3x^0` 與 `3x^2 + 2x^1 + 1x^0` 相乘，預期結果應為 `3x^4 + 8x^3 + 10x^2 + 7x^1 + 3x^0`。
- 多項式代入測試：
  - 測試不同的 `x` 值代入多項式計算結果。
  - 測試：

        - `1x^2 + 2x^1 + 3x^0` 在 `x = 2` 時，預期結果是 `1 * 2^2 + 2 * 2 + 3 = 11`。

集成測試

- 測試多個功能協同工作的情況。
    - 測試完整的流程：用戶輸入兩個多項式，進行加法、乘法運算，再代入 `x` 值並計算結果。確保各個步驟之間沒有錯誤。

邊界情況測試

- 測試極端情況，確保程式能處理特殊情況。
    - 測試多項式只有一項（如 `5x^2` 或 `3x^0`）。
    - 測試多項式有相同次方的項（如 `3x^2 + 4x^2`）。
    - 測試兩個多項式的項數非常大，檢查內存分配和程式效能。

申論及開發報告

開發報告

- 目標與範圍：
  - 目的是實現一個多項式處理類別，支持加法、乘法和代入計算功能。
  - 需要處理多項式的項目表示、存儲、運算及輸出，並支持用戶輸入與輸出。
- 設計思路：
  - 使用 `Polynomial` 類別來表示多項式，並使用 `Term` 類別來表示每個項目。通過動態數組儲存多項式項目，並利用 `Resize` 函數進行內存擴展。
  - 設計了 `Add`、`Mult` 和 `Eval` 等函數來實現多項式的運算。
  - 實現了 `operator<<` 和 `operator>>` 來支持多項式的輸入與輸出。
- 測試結果：
  - 經過多輪測試，程式能夠正確處理多項式的加法、乘法、代入計算等功能。
  - 測試結果顯示程式對不同輸入能夠給出正確結果，並且能夠處理多項式的邊界情況。
- 開發過程中的挑戰：
  - 內存管理：動態數組的擴展和項目的存儲是開發過程中的一大挑戰，尤其是在處理較大多項式時需要特別注意內存的釋放與管理。
  - 輸入解析：用戶輸入的格式可能會出現錯誤，對此我們進行了錯誤處理與檢查，確保程式能夠健壯地運行。
- 未來改進與擴展：

- 優化多項式加法與乘法的實現，考慮排序或使用哈希表等數據結構來提升效率。
- 添加更多的數學運算，如多項式除法、求導等。
- 提高程式的用戶體驗，增強錯誤提示和數據驗證功能。