壹、 解題說明

一、加法功能 (Add)

透過兩個指標 aCurrent 和 bCurrent 來遍歷兩個多項式。如果兩個指標指向的項目有相同的次方數，則將它們的係數相加並加到結果多項式中；如果 poly1 的該項次方數小於 poly2 的該項次方數，則將 poly2 的該項加到結果多項式中；如果 poly1 的該項次方數大於 poly2 的該項次方數，則將 poly1 的該項加到結果多項式中。整個過程中會依次比較並將非零係數的項目加入結果中，最終形成新的多項式。

二、乘法功能 (Mult)

使用兩層迴圈來實現，外層迴圈遍歷 poly1 的每一項，內層迴圈遍歷 poly2 的每一項。每當兩項相乘時，結果的係數是兩個多項式項目係數的乘積，次方數是兩者次方數的和。計算結果會加到新的多項式中，然後會進行合併相同次方數的項目，最終按照次方數降冪排序。

三、求值 (Eval)

將使用者提供的 x 值代入每一項的多項式公式中進行計算，然後將每一項計算結果加總，最終返回該多項式在 x 位置的值。這是通過遍歷多項式中的每一項，使用公式 係數 * x^次方數 來計算每項的貢獻，然後將所有項的結果相加。

四、輸入

使用重載的輸入運算子 >> 來讀取多項式。首先，讀入項數，接著對每一項，讀取其係數和次方數，並將每項的數據插入到圓形鏈結結構中。這樣，當所有項都輸入後，該多項式就會被保存為圓形鏈結結構，每一項都正確鏈接到下一項，直到回到表頭形成圓環。

五、輸出

使用重載的輸出運算子 << 來將多項式格式化並輸出。如果多項式的項數為零，則顯示 "0"；如果多項式非零，則遍歷多項式中的每一項，根據次方數將每項格式化為 "係數 x^次方數" 的格式。如果次方數為 0，則只顯示係數；如果該項是非最後一項且係數為正，則在項目後面加上 "＋" 來表示。

貳、程式實作

```
class Term {
public:
    int coef; // 系數
    int exp;  // 指數
    Term* link; // 指向下一項的指標

    Term(int c = 0, int e = 0) : coef(c), exp(e), link(nullptr) {} // 建構子，初始化系數、指數及鏈結指標
};
```

圖 1 定義 Term 物件

```
class Polynomial {
private:
    Term* header; // 圓形鏈結串列的表頭節點

    // 清除多項式的節點
    void clear();
    // 複製多項式的內容
    void copyFrom(const Polynomial& a);

public:
    // 建構子，初始化空的多項式
    Polynomial();
    // 複製建構子
    Polynomial(const Polynomial& a);
    // 解構子，釋放記憶體
    ~Polynomial();
    // 賦值運算子，深度複製多項式
    Polynomial& operator=(const Polynomial& a);
    // 輸入運算子，讀取多項式並轉換為圓形鏈結
    friend istream& operator>>(istream& is, Polynomial& x);
    // 輸出運算子，將多項式轉換為字串並輸出
    friend ostream& operator<<(ostream& os, const Polynomial& x);
    // 多項式加法
    Polynomial operator+(const Polynomial& b) const;
    // 多項式減法，透過加上另一多項式的負值實現
    Polynomial operator-(const Polynomial& b) const;
    // 多項式乘法
    Polynomial operator*(const Polynomial& b) const;
    // 計算多項式在x處的值
    double Evaluate(double x) const;
};
```

圖 2 定義 Polynomial 物件

```
// 建構子，初始化空的多項式
Polynomial::Polynomial() {
    header = new Term(); // 創建表頭節點
    header->link = header; // 指向自己，形成圓形鏈結
}
```

圖3 Polynomial預設建構子

```
// 複製建構子
Polynomial::Polynomial(const Polynomial& a) : header(nullptr) {
    copyFrom(a);
}
```

圖 4 複製建構子

```
// 賦值運算子
Polynomial& Polynomial::operator=(const Polynomial& a) {
    if (this != &a) {
        clear(); // 先清除當前多項式
        copyFrom(a); // 然後複製新多項式
    }
    return *this;
}
```

圖 5 賦值建構子

```
// 解構子，釋放記憶體
Polynomial::~Polynomial() {
    clear();
}
```

圖 6 Polynomial 解構子

```cpp
// 清除多項式的節點
void Polynomial::clear() {
    if (!header) return;
    Term* current = header->link;
    while (current != header) {
        Term* toDelete = current;
        current = current->link;
        delete toDelete;
    }
    delete header;
    header = nullptr;
}
```

圖 7 清除節點

```cpp
// 多項式加法
Polynomial Polynomial::operator+(const Polynomial& b) const {
    Polynomial result;
    Term* aCurrent = header->link;
    Term* bCurrent = b.header->link;
    Term* tail = result.header;

    while (aCurrent != header || bCurrent != b.header) {
        int coef, exp;
        if (aCurrent != header && (bCurrent == b.header || aCurrent->exp > bCurrent->exp)) {
            coef = aCurrent->coef;
            exp = aCurrent->exp;
            aCurrent = aCurrent->link;
        }
        else if (bCurrent != b.header && (aCurrent == header || bCurrent->exp > aCurrent->exp)) {
            coef = bCurrent->coef;
            exp = bCurrent->exp;
            bCurrent = bCurrent->link;
        }
        else {
            coef = aCurrent->coef + bCurrent->coef;
            exp = aCurrent->exp;
            aCurrent = aCurrent->link;
            bCurrent = bCurrent->link;
        }

        if (coef != 0) { // 只有當係數不為零時，才添加到結果中
            Term* newTerm = new Term(coef, exp);
            tail->link = newTerm;
            tail = newTerm;
        }
    }
    tail->link = result.header; // 完成圓形鏈結
    return result;
}
```

圖 8 多項式加法運算

```cpp
// 多項式減法，透過加上另一多項式的負值實現
Polynomial Polynomial::operator-(const Polynomial& b) const {
    Polynomial negB = b;
    Term* current = negB.header->link;
    while (current != negB.header) {
        current->coef = -current->coef; // 取負
        current = current->link;
    }
    return *this + negB; // 加上負的多項式
}
```

圖 9 多項式減法運算

```cpp
// 多項式乘法
Polynomial Polynomial::operator*(const Polynomial& b) const {
    Polynomial result;
    Term* aCurrent = header->link;

    while (aCurrent != header) {
        Polynomial temp;
        Term* bCurrent = b.header->link;
        Term* tail = temp.header;

        while (bCurrent != b.header) {
            int coef = aCurrent->coef * bCurrent->coef;
            int exp = aCurrent->exp + bCurrent->exp;
            Term* newTerm = new Term(coef, exp);
            tail->link = newTerm;
            tail = newTerm;
            bCurrent = bCurrent->link;
        }
        tail->link = temp.header;
        result = result + temp; // 將中間結果相加
        aCurrent = aCurrent->link;
    }
    return result;
}
```

圖 10 多項式乘法運算

```
// 計算多項式在x處的值
double Polynomial::Evaluate(double x) const {
    double result = 0;
    Term* current = header->link;
    while (current != header) {
        result += current->coef * pow(x, current->exp); // 計算每一項的值並累加
        current = current->link;
    }
    return result;
}
```

圖 11 多項式求值

```
// 輸入運算子，讀取多項式並轉換為圓形鏈結
istream& operator>>(istream& is, Polynomial& x) {
    x.clear(); // 先清除原來的多項式
    int n;
    is >> n; // 讀取項數
    if (n <= 0) return is;

    x.header = new Term(); // 創建空的表頭節點
    Term* tail = x.header;

    // 讀取每一項的係數和指數，並鏈結到多項式中
    for (int i = 0; i < n; ++i) {
        int c, e;
        is >> c >> e;
        Term* newTerm = new Term(c, e);
        tail->link = newTerm;
        tail = newTerm;
    }
    tail->link = x.header; // 完成圓形鏈結
    return is;
}
```

圖 12 多載>>

```cpp
// 輸出運算子，將多項式轉換為字串並輸出
ostream& operator<<(ostream& os, const Polynomial& x) {
    if (!x.header || x.header->link == x.header) { // 如果多項式為空
        os << "0";
        return os;
    }

    Term* current = x.header->link;
    bool first = true;
    while (current != x.header) { // 遍歷多項式的每一項
        if (!first && current->coef > 0) os << "+"; // 若不是第一項且係數為正，加上 "+"
        os << current->coef << "x^" << current->exp;
        current = current->link;
        if (current != x.header) os << " "; // 若不是最後一項，則加上空格
        first = false;
    }
    return os;
}
```

圖 13 多載<<

```cpp
int main() {
    Polynomial p1, p2;
    cout << "請輸入第一個多項式（格式: n c1 e1 c2 e2 ...): ";
    cin >> p1;
    cout << "請輸入第二個多項式（格式: n c1 e1 c2 e2 ...): ";
    cin >> p2;

    Polynomial sum = p1 + p2;
    Polynomial diff = p1 - p2;
    Polynomial prod = p1 * p2;

    cout << "第一個多項式 P1: " << p1 << endl;
    cout << "第二個多項式 P2: " << p2 << endl;
    cout << "多項式加法結果: " << sum << endl;
    cout << "多項式減法結果: " << diff << endl;
    cout << "多項式乘法結果: " << prod << endl;

    double x;
    cout << "請輸入 x 的值以計算各個多項式的值: ";
    cin >> x;

    // 計算並顯示所有多項式的值
    cout << "P1(" << x << ") = " << p1.Evaluate(x) << endl;
    cout << "P2(" << x << ") = " << p2.Evaluate(x) << endl;
    cout << "P1 + P2(" << x << ") = " << sum.Evaluate(x) << endl;
    cout << "P1 - P2(" << x << ") = " << diff.Evaluate(x) << endl;
    cout << "P1 * P2(" << x << ") = " << prod.Evaluate(x) << endl;

    return 0;
}
```

圖 13 主程式

參、效能分析

	時間複雜度	空間複雜度
多項式加法	O(n1 + n2)	O(n1 + n2)
多項式乘法	O(n1 + n2)	O(n1 + n2)
求值	O(n)	O(1)
多載<<	O(n)	O(n)
多載>>	O(n)	O(1)

表 1 效能分析

肆、測試與驗證

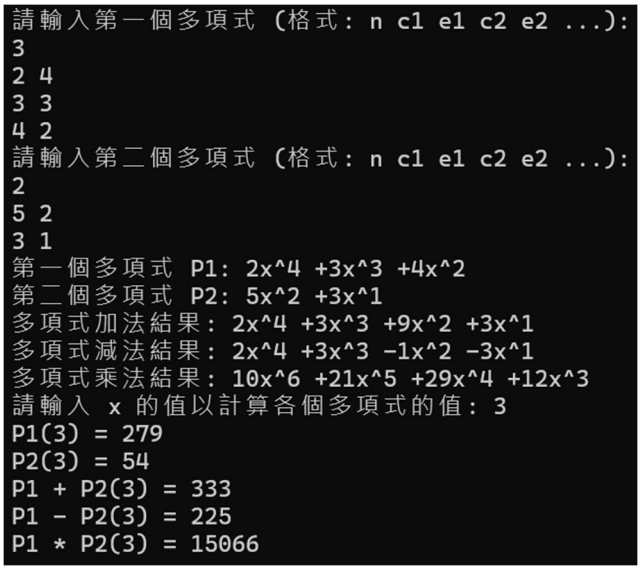

圖 14 測試與驗證

伍、申論及開發報告

在開發過程中，我遇到了一些與物件存取及記憶體管理有關的挑戰，這讓我更加理解 C++ 的內部機制以及如何有效管理資源。

首先，當我處理多項式類別 Polynomial 的運算時，發現有些函式在返回新物件時，會因為錯誤的物件存取或記憶體釋放問題，導致程式異常終止。這讓我意識到必須正確處理拷貝建構子與賦值建構子，才能確保物件在返回過程中能夠保持正確的記憶體管理與資料一致性。

為了解決這個問題，我修改了相關函式，加入適當的拷貝建構子來確保在創建新物件時進行深拷貝，並且對賦值操作進行了處理，避免了對原物件的錯

誤操作。這樣，當多項式運算結果被賦值給新物件時，所有資料都能正確保留，防止程式崩潰。

此外，我也加強了對類別內部資料成員的存取控制，確保只能透過公共接口來操作私有資料。這不僅提升了程式的穩定性，也增強了程式的封裝性與可維護性。

這些修改幫助我更加深入了解 C++ 中的記憶體管理、物件生命週期及封裝概念，也提高了我對程式設計的整體理解。