資料結構(一) HW3

一、 題目 2
二、 解題思維 3
三、 所有資料類別 3
1. ChainNode 類別 3
2. CircularListWithHeader 類別 3
3. ChainIterator 類別 4
4. Term 類別 4
5. Polynomial 類別 4
四、 ChainIterator の類別 實作 5
五、 CircularListWithHeader の類別 實作 6
六、 Polynomial の類別 實作 7
七、 心得討論 13
八、 所有測試一覽 14

圖 1：HW3 の題目 2
圖 2：ChainNode の class 3
圖 3：CircularListWithHeader の class 3
圖 4：CricularListWithHeader 中 ChainIterator の class 4
圖 5：Term の class 4
圖 6：Polynomial の class 4
圖 7：CircularListWithHeader 迭代器 class の操作 5
圖 8：CircularListWithHeader の建構子 6
圖 9：CircularListWithHeader の插入節點 6
圖 10：CircularListWithHeader の回傳頭或尾結點 7
圖 11：Polynomial の建構子 7
圖 12：Polynomial 輸入的 Code 8
圖 13：Polynomial 輸出的 Code 9
圖 14：Polynomial の operator= 9
圖 15：Polynomial の operator+ 10
圖 16：Polynomial の operator 11
圖 17：Polynomial の operator* 12
圖 18：Polynomial の Evaluate 的 Code 13
圖 19：Polynomial の測試驗證 14
圖 20：Polynomial の測試驗證（與上次報告同參數） 14

一、題目

Homework 3

[***Programming Project***] Develop a C++ class *Polynomial* to represent and manipulate univariate polynomials with integer coefficients (use circular linked lists with header nodes). Each term of the polynomial will be represented as a node. Thus, a node in this system will have three data members as below:

coef	exp	link

Each polynomial is to be represented as a circular list with header node. To delete polynomials efficiently, we need to use an available-space list and associated functions as described in Section 4.5. The external (i.e., for input or output) representation of a univariate polynomial will be assumed to be a sequence of integers of the form: $n, c_1, e_1, c_2, e_2, c_3, e_3, \ldots, c_n, e_n$, where e_i represents an exponent and c_i a coefficient; n gives the number of terms in the polynomial. The exponents are in decreasing order—$e_1 > e_2 > \cdots > e_n$.

Write and test the following functions:

(a) *istream*& **operator>>**(*istream*& *is*, *Polynomial*& *x*): Read in an input polynomial and convert it to its circular list representation using a header node.

(b) *ostream*& **operator<<**(*ostream*& *os*, *Polynomial*& *x*): Convert *x* from its linked list representation to its external representation and output it.

(c) *Polynomial*::*Polynomial*(**const** *Polynomial*& *a*) [Copy Constructor]: Initialize the polynomial ∗**this** to the polynomial *a*.

(d) **const** *Polynomial*& *Polynomial*::**operator=**(**const** *Polynomial*& *a*) **const** [Assignment Operator]: Assign polynomial *a* to ∗**this**.

(e) *Polynomial*::~*Polynomial*() [Destructor]: Return all nodes of the polynomial ∗**this** to the available-space list.

(f) *Polynomial* **operator+** (**const** *Polynomial*& *b*) **const** [Addition]: Create and return the polynomial ∗**this** + *b*.

(g) *Polynomial* **operator**− (**const** *Polynomial*& *b*) **const** [Subtraction] : Create and return the polynomial ∗**this** − *b*.

(h) *Polynomial* **operator**∗(**const** *Polynomial*& *b*) **const** [Multiplication]: Create and return the polynomial ∗**this** ∗ *b*.

(i) **float** *Polynomial*::*Evaluate*(**float** *x*) **const**: Evaluate the polynomial ∗**this** at *x* and return the result.

圖 1：HW3 の題目

二、解題思維

本次作業是實作「Polynomial（多項式）」，要求 Polynomial 內部是要用「Circular Linked Lists with Header Nodes」，也就是說要用環形鏈結串列作為底層儲存資料結構，因此在實作上總共有五個類別分別是 ChainNode、CircularListWithHeader、ChainIterator、Term 和 Polynommial。ChainNode 和 Term 是最基底的資料類別，CircularListWithHeader 是實作在 ChainNode 之上，CircularListWithHeader 內部有 ChainIterator 類別用於環形串列可以有迭代器的功能，Polynomial 是實作在 CircularListWithHeader 之上，CircularListWithHeader 內部資料是使用 Term 類別。

三、所有資料類別

1. ChainNode 類別

```
template <class T>
class ChainNode {
    friend class CircularListWithHeader<T>;
private:
    T data;             // 儲存節點的資料
    ChainNode* next;    // 指向下一個節點的指標
public:
    ChainNode() : next(nullptr) {}
    ChainNode(T data) : data(data), next(nullptr) {};
    ChainNode(T data, ChainNode* next) : data(data), next(nullptr) {};
};
```

圖 2：ChainNode の class

2. CircularListWithHeader 類別

```
template <class T>
class CircularListWithHeader {
private:
    ChainNode<T>* head;    // 環形串列的頭節點指標
public:
    CircularListWithHeader();    // 預設建構函式
    ~CircularListWithHeader();  // 解構函式
    void InsertFront(const T& e);    // 在串列前端插入一個元素
    void InsertBack(const T& e);     // 在串列後端插入一個元素

    class ChainIterator;             // 環形串列迭代器類別的宣告
    ChainIterator begin() const;     // 返回迭代器指向第一個元素
    ChainIterator end() const;       // 返回迭代器指向最後一個元素的下一個位置
};
```

圖 3：CircularListWithHeader の class

3. ChainIterator 類別

```
template <class T>
class CircularListWithHeader<T>::ChainIterator {
private:
    ChainNode<T>* current;   // 迭代器指向的當前節點
public:
    ChainIterator(ChainNode<T>* startNode = nullptr);   // 建構函式，初始化迭代器，預設起始節點為空
    T& operator *() const;   // 重載取值運算子(*)，返回當前節點的資料指標
    T* operator->() const;   // 重載箭頭運算子(->)，返回指向當前節點資料的指標
    ChainIterator& operator++();   // 重載前置遞增運算子(++i)，使迭代器指向下一個節點
    ChainIterator operator++(int);   // 重載後置遞增運算子(i++)，使迭代器指向下一個節點，返回原來的迭代器
    bool operator!=(const ChainIterator right) const;   // 重載不等於運算子(!=)，比較兩個迭代器是否不相等
    bool operator==(const ChainIterator right) const;   // 重載等於運算子(==)，比較兩個迭代器是否相等
};
```

圖 4：CricularListWithHeader 中 ChainIterator の class

4. Term 類別

```
class Term {
    friend Polynomial;
    friend ostream& operator<<(ostream& output, const Polynomial& p);
private:
    float coef;    // 係數，表示多項式中項的係數
    int exp;       // 指數，表示多項式中項的指數
public:
    Term Set(float c, int e) {    // 設定 Term 的係數和指數的成員函式
        this->coef = c; this->exp = e;
        return *this;    // 返回當前 Term 物件的指標
    };
};
```

圖 5：Term の class

5. Polynomial 類別

```
class Polynomial {
    friend istream& operator>>(istream& input, Polynomial& p);
    friend ostream& operator<<(ostream& output, const Polynomial& p);
private:
    CircularListWithHeader<Term> poly;
public:
    Polynomial(); // Construct the polynomial p(x) = 0.
    ~Polynomial();
    const Polynomial& operator=(const Polynomial& a);
    Polynomial operator+(const Polynomial& b) const;
    Polynomial operator-(const Polynomial& b) const;
    Polynomial operator*(const Polynomial& b) const;
    float Evaluate(float x) const;
};
```

圖 6：Polynomial の class

四、ChainIterator の類別 實作

```
//ChainIterator 實作↓
template<class T>
CircularListWithHeader<T>::ChainIterator::ChainIterator(ChainNode<T>* startNode) {
    current = startNode;
}

template<class T>
T& CircularListWithHeader<T>::ChainIterator::operator* () const {
    return current->data;
}

template<class T>
T* CircularListWithHeader<T>::ChainIterator::operator->() const {
    return &current->data;
}

template<class T>
typename CircularListWithHeader<T>::ChainIterator& CircularListWithHeader<T>::ChainIterator::operator++() {
    current = current->next;
    return *this;
}

template<class T>
typename CircularListWithHeader<T>::ChainIterator CircularListWithHeader<T>::ChainIterator::operator++(int) {
    ChainIterator old = *this;
    current = current->next;
    return old;
}

template<class T>
bool CircularListWithHeader<T>::ChainIterator::operator!=(const ChainIterator right) const {
    return current != right.current;
}

template<class T>
bool CircularListWithHeader<T>::ChainIterator::operator==(const ChainIterator right) const {
    return current == right.current;
}
//ChainIterator 實作↑
```

圖 7：CircularListWithHeader 迭代器 class の操作

ChainIterator 是 foucs 在環形串列（CircularList）的迭代器，作為對環形串列當前的內容迭代的運算，包含了建構串列迭代器、指標(*)、箭頭運算子(->)、前後置遞增和等於與不等於，都是一些迭代器的基本操作。

五、CircularListWithHeader の類別 實作

```
template <class T>
CircularListWithHeader<T>::CircularListWithHeader() {
    head = new ChainNode<T>();  // 創建一個新的頭節點，並將 head 指向它
    head->next = head;          // 將頭節點的 next 指向自己，形成循環
}
```

圖 8：CircularListWithHeader の建構子

```
template <class T>
void CircularListWithHeader<T>::InsertFront(const T& e) {
    ChainNode<T>* newNode = new ChainNode<T>(e);  // 創建包含新元素的節點
    if (head) {  // nonempty list
        newNode->next = head->next;  // 新節點的下一個指向原先第一個節點
        head->next = newNode;  // 頭節點的下一個指向新節點
    }
    else {  // empty list
        head = newNode;  // 頭節點指向新節點
        newNode->next = newNode;  // 新節點的下一個指向自己，形成循環
    }
}

template <class T>
void CircularListWithHeader<T>::InsertBack(const T& e) {
    ChainNode<T>* newNode = new ChainNode<T>(e);  // 創建包含新元素的節點
    if (head) {  // nonempty list
        newNode->next = head;  // 新節點的下一個指向頭節點
        ChainNode<T>* last = head;
        while (last->next != head)
            last = last->next;  // 找到目前最後一個節點
        last->next = newNode;  // 目前最後一個節點的下一個指向新節點，形成循環
    }
    else {  // empty list
        head = newNode;  // 頭節點指向新節點
        newNode->next = newNode;  // 新節點的下一個指向自己，形成循環
    }
}
```

圖 9：CircularListWithHeader の插入節點

這兩個函式分別處理了非空鏈串列和空鏈列的情況，確保在插入新節點後指標仍然是循環的。InsertFront 將新節點插入到頭節點之後，而 InsertBack 則將新節點插入到尾節點之後，同時保持循環結構。

```
template<class T>
typename CircularListWithHeader<T>::ChainIterator CircularListWithHeader<T>::begin() const {
    // 返回從頭節點的下一個節點開始的迭代器
    return CircularListWithHeader<T>::ChainIterator(head->next);
}

template<class T>
typename CircularListWithHeader<T>::ChainIterator CircularListWithHeader<T>::end() const {
    ChainNode<T>* last = head;
    while (last->next != head)
        last = last->next;  // 找到目前最後一個節點
    // 返回指向目前最後一個節點的下一個節點的迭代
    return CircularListWithHeader<T>::ChainIterator(last->next);
}
```

圖 10：CircularListWithHeader の回傳頭或尾結點

透過迭代器可以讓我們在程式撰寫上面，可以快速的找到我們要的資訊，因此實作了環形串列中回傳頭和尾的函式，begin 是回傳頭；end 是回傳尾。

六、Polynomial の類別 實作

```
Polynomial::Polynomial() {
    poly = CircularListWithHeader<Term>();
}
```

圖 11：Polynomial の建構子

多項式（Ploynomial）是由多組係數和指數所構成，因此我們可以用 Term 類別去作為多項是每一組的資料（如圖 5 所示），多項式類別只用環形串列 ploy 去儲存資料。這支多項式的程式碼，提供了讓使用者輸入多項式和輸出多項式，再來是透過運算子重載方式，時做了有等於（=）、加法（+）、減法（-）、乘法（*）和求值（Evaluate）的操作。

1. operator>>

```
istream& operator>>(istream& input, Polynomial& p) {
    Term tmp; // 暫存每一項的係數和指數
    float t_coef; // 暫存係數的變數
    int t_exp; // 暫存指數的變數
    string s;
    getline(input, s); // 從輸入流讀取整行多項式字串
    istringstream sin(s); // 將整行字串轉換成輸入流，以便逐項讀取
    p.poly = CircularListWithHeader<Term>(); // 初始化多項式的循環鏈表
    while (1) {
        if (sin.eof()) break; // 若已到達輸入流結尾，則跳出迴圈
        else if (sin.peek() == '+') sin.ignore(1); // 忽略正號
        if (sin.peek() == 'x' || sin.peek() == 'X') {
            sin.ignore(1); // 忽略 'x' 或 'X'
            if (sin.peek() == '^') {
                sin.ignore(1); // 忽略 '^'
                sin >> t_exp; // 讀取指數
                p.poly.InsertBack(tmp.Set(1, t_exp)); // 將項插入到多項式
            }
            else if (sin.peek() == '+' || sin.peek() == '-' || sin.eof())
                p.poly.InsertBack(tmp.Set(1, 1)); // 若無指數，預設為 1
        }
        else { // 標準項
            sin >> t_coef; // 讀取係數
            if (sin.eof())
                p.poly.InsertBack(tmp.Set(t_coef, 0)); // 若已到達輸入流結尾，則插入常數項
            else if (sin.peek() == 'x' || sin.peek() == 'X') {
                sin.ignore(1); // 忽略 'x' 或 'X
                if (sin.peek() == '+' || sin.peek() == '-' || sin.eof())
                    p.poly.InsertBack(tmp.Set(t_coef, 1)); // 若無指數，預設為 1
                else if (sin.peek() == '^') {
                    sin.ignore(1); // 忽略 '^'
                    sin >> t_exp; // 讀取指數
                    p.poly.InsertBack(tmp.Set(t_coef, t_exp)); // 將項插入到多項式
                }
            }
        }
    }
    return input;
}
```

圖 12：Polynomial 輸入的 Code

多項式（polynomial）運算子重載>>，根據上份報告，提到的三個問題，本次作業將上份作業沒有克服的第三個問題解決了，但在測試的時候發現，若有沒按順序輸入的話，可能也會有錯誤，雖然解決一般在寫多項式時，會有的省略寫法，但還不夠全面。本次的寫法加入了 istringstream 的寫法，可以利用 peek 偷看，去預測現在或是下一個是什麼樣的項目，已針對不同的項目給予不同的流程。

2. operator<<

```
ostream& operator<<(ostream& output, const Polynomial& p) {
    for (CircularListWithHeader<Term>::ChainIterator i = p.poly.begin(); i != p.poly.end(); i++) {
        if (i != p.poly.begin()) {
            if (i->coef > 0) output << "+"; // 若係數為正，輸出正號
        }
        if (i->exp == 0)
            output << i->coef; // 若指數為0，只輸出係數
        else if (i->coef == 1 && i->exp == 1)
            output << "x"; // 若係數為1，且指數為1，只輸出 'x'
        else if (i->exp == 1)
            output << i->coef << "x"; // 若指數為1，輸出係數和 'x'
        else if (i->coef == 1)
            output << "x^" << i->exp; // 若係數為1，輸出 'x' 的指數
        else
            output << i->coef << "x^" << i->exp; // 一般情況，輸出係數、'x' 和指數
    }
    output << endl; // 換行
    return output;
}
```

圖 13：Polynomial 輸出的 Code

多項式（polynomial）運算子重載<<，由於要比上份報告更加進步，所以在輸出的部分就會比較多樣性，所以本次報告的流程比較注重於判斷，又加上使用環形串列，又有迭代器的功能，因此撰寫上都是判斷的結構，較容易於撰寫，該運算子多載主要是輸出，因此沒有在輸出前對多項式以指數作為排序。

3. operator=

```
const Polynomial& Polynomial::operator=(const Polynomial& a) {
    Polynomial np; // 創建一個新的多項式
    np.poly = CircularListWithHeader<Term>(a.poly); // 複製其他多項式的環形串列
    return np; // 回傳新的多項式
}
```

圖 14：Polynomial の operator=

運算子重載實行 operator=，該函式較簡單就是將自己等於對方，讓所有的資料讓對方指向自己，就完成了。

4. operator+

```
Polynomial Polynomial::operator+(const Polynomial& b) const {
    //Polynomials *this(a) an b are added and the sum returned.
    Term temp;
    CircularListWithHeader<Term>::ChainIterator ai = poly.begin(),
                                                bi = b.poly.begin();
    Polynomial c; //assume constructor sets head->exp = -1
    while (ai != poly.end() && bi != b.poly.end()) {
        if (ai->exp == bi->exp) {
            float sum = ai->coef + bi->coef;
            if (sum) c.poly.InsertBack(temp.Set(sum, ai->exp));
            ai++; bi++; //advance to next term
        }
        else if (ai->exp < bi->exp) {
            c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
            bi++; //next term of b;
        }
        else {
            c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
            ai++; //next term of a
        }
    }
    while (ai != poly.end()) {
        c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
        ai++;
    }
    while (bi != b.poly.end()) {
        c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
        bi++;
    }
    return c;
}
```

圖 15：Polynomial の operator+

Polynomial 的加法運算，本次的流程跟上次的不一樣，上次是根據多項式，直接開一個陣列，一一存入，以陣列索引當過指數位置，係數是陣列索引的內容，本次是較注重判斷，當沒有重複的就接續插入環形串列尾端，但有重複的就做加法運算再存入，所以對於順序比較在意，因此整個多項式在插入和刪除時，要有序的概念，整個操作是要規劃好的。

時間複雜度是 O(n+m)，空間複雜度是 2(n+m)+4。

5. operator-

```
Polynomial Polynomial::operator-(const Polynomial& b) const {
    Polynomial c; // 創建一個新的多項式，用於存儲相減結果
    Term temp; // 用於暫存新生成的項
    CircularListWithHeader<Term>::ChainIterator ai = poly.begin(),
                                                bi = b.poly.begin();
    // 走訪兩個多項式的環形串列，進行相減操作
    while (ai != poly.end() && bi != b.poly.end()) {
        if (ai->exp > bi->exp) { // 若第一個多項式的指數大於第二個多項式的指數
            c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
            ai++;
        }
        else if (ai->exp < bi->exp) { // 若第一個多項式的指數小於第二個多項式的指數
            c.poly.InsertBack(temp.Set(bi->coef * -1, bi->exp));
            bi++;
        }
        else { // 若兩個多項式的指數相等
            if (ai->coef != bi->coef)
                c.poly.InsertBack(temp.Set(ai->coef - bi->coef, ai->exp));
            ai++;  bi++;
        }
    }
    while (ai != poly.end()) { // 將第一個多項式剩餘的項直接加入結果多項式
        c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
        ai++;
    }
    while (bi != b.poly.end()) { // 將第二個多項式剩餘的項直接加入結果多項式
        c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
        bi++;
    }

    return c; // 返回相減結果
}
```

圖 16：Polynomial の operator-

Polynomial 多項式的減法運算，因為上次沒有該運算，所以不加以比較，該減法運算跟加法運算是類似的只是差再指數相同時，不作加法改做減法，該流程對於多項式一開始的進入有序的概念，但在運算過程中，若原始式有序的，出去也會是有序的，不然會有錯誤的可能性。

時間複雜度是 O(n+m)，空間複雜度是 2(n+m)+4。

6. operator*

```
Polynomial Polynomial::operator*(const Polynomial& b) const {
    Polynomial c; // 創建一個新的多項式對象，用於存儲相乘結果
    Term temp; // 用於暫存新生成的項
    CircularListWithHeader<Term>::ChainIterator ai, bi, ci;
    map<int, float> data; // 用於存儲相乘後的項的係數和指數的對應關係

    // 走訪兩個多項式的環形串列，進行相乘操作
    for (ai = poly.begin(); ai != poly.end(); ai++) {
        for (bi = b.poly.begin(); bi != b.poly.end(); bi++) {
            float t_coef = ai->coef * bi->coef; // 計算兩個項的係數相乘
            int t_exp = ai->exp + bi->exp; // 計算兩個項的指數相加

            // 將計算得到的項的係數和指數存入map中
            if (data.find(t_exp) == data.end()) // 若指數尚未存在於map中
                data[t_exp] = t_coef; // 直接添加新的項
            else
                data[t_exp] += t_coef; // 若指數已存在於map中，則將係數相加
        }
    }

    // 走訪map，將其中的項按指數遞減的順序加入結果多項式
    for (auto id = data.begin(); id != data.end(); id++)
        if (id->second != 0)
            c.poly.InsertFront(temp.Set(id->second, id->first));

    return c; // 返回相乘結果
}
```

圖 17：Polynomial の operator*

Polynomial 的乘法運算，上次是利用雙層 for 迴圈模擬乘法運算，也是直接一開始就開一定大小的陣列，很消耗記憶體空間，但本次加入了字典的結構 map，對於記憶體上就會比較節省，用多少開多少，在索引上也較方便，不必像上次作業的方式，要額外開一條陣列儲存位置，流程上，依舊是雙層迴圈模擬，運算出係數和指數後，對字典做查詢，有的話直接加上去，否則，增加字典內容，最後，直接走訪 map，插入到新的多項式中，回傳結果。

時間複雜度是 O(n*m)，空間複雜度是 3(n+m)+7。

7. Polynomial の Evaluate(int)

```
float Polynomial::Evaluate(float x) const {
    float res = 0.0f; // 初始結果為0
    CircularListWithHeader<Term>::ChainIterator ai = poly.begin();

    // 走訪每一項，計算其在指定x值下的值，並將結果加總
    while (ai != poly.end()) {
        float ans = ai->coef; // 初始化ans為該項的係數
        for (int i = 0; i < ai->exp; i++)
            ans *= x; // 將該項的指數次方加入計算
        res += ans; // 將計算結果加總到總結果中
        ai++; // 移動到下一項
    }

    return res; // 返回多項式在指定x值下的計算結果
}
```

圖 18：Polynomial の Evaluate 的 Code

Polynomial 的 Evaluate()是給 x 求出多項式的值，我是先透過 res 去儲存結果值，再去走訪多項式，先用 ans 變數儲存係數，在根據項次的指數去乘上輸入的 x 值，以模擬計算，最後再加到 res，跑完就會知道值是多少，最後回傳結果。

時間複雜度是 O(n)，空間複雜度是 n+6。

七、心得討論

本次的作業是續上次實作多項式類別，這次也是多項式，只是內部儲存方式不同，在寫程式的過程中，讓我學到了如何去規劃程式碼專案，原先是想將程式碼，可以分開到不同的檔案裡，用時 include 就好，debug 應該也較方便，不會同一個檔案程式碼超長，最後，因為好複雜，因此放棄分開，問題太多了，這次的程式碼相對上次，其實有用個是更好寫得，完完全全是讓我 coding 技能大大 up 的作業，也試玩了一些不同的結構搭配撰寫。

八、所有測試一覽

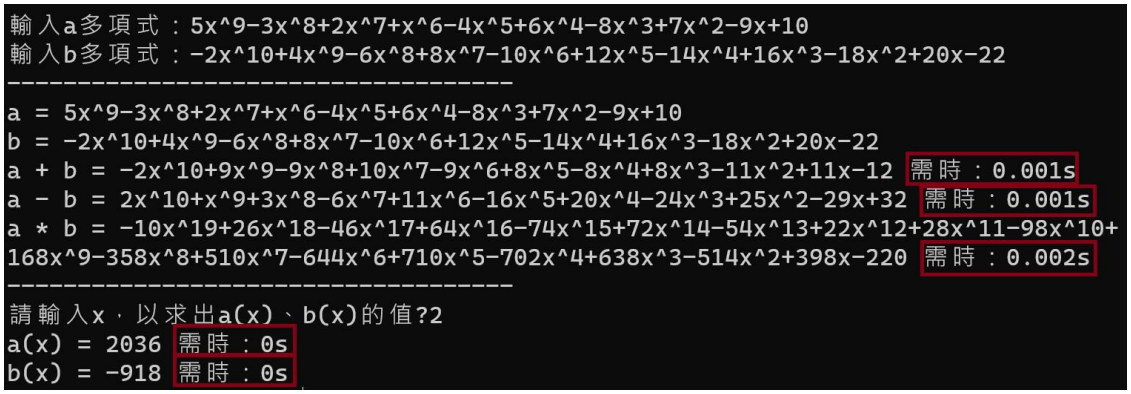

圖 19：Polynomial の測試驗證

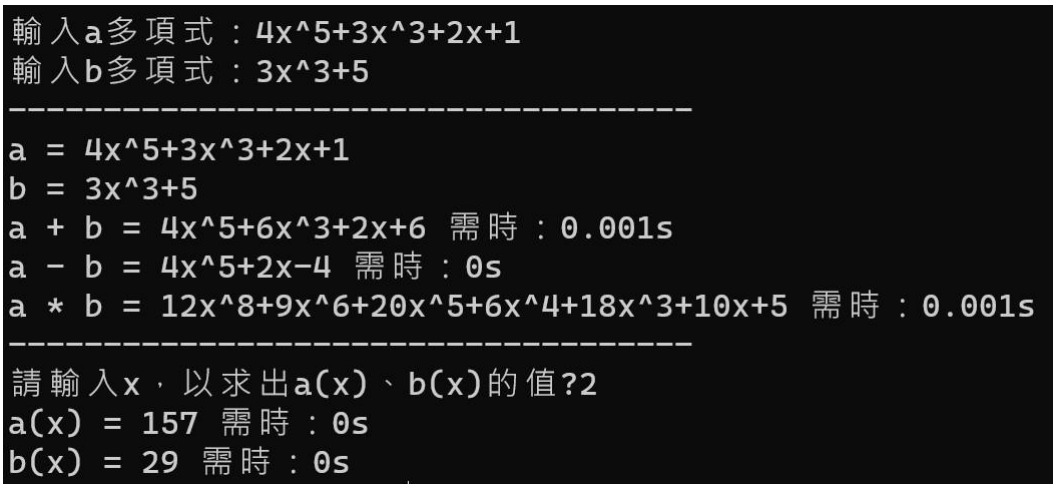

圖 20：Polynomial の測試驗證（與上次報告同參數）

※完整的程式碼檢附於報告的最後（下一頁）。

```cpp
#include <iostream>
#include <sstream>
#include <map>
#include <ctime>
using namespace std;

template <class T> class CircularListWithHeader; //forward declaration

template <class T>
class ChainNode {
    friend class CircularListWithHeader<T>; // 將 CircularListWithHeader 類別設定為 ChainNode 的友元類別，以便訪問私有成員
private:
    T data;             // 儲存節點的資料
    ChainNode* next;    // 指向下一個節點的指標
public:
    ChainNode() : next(nullptr) {}    // 預設建構函式，初始化下一個指標為空
    ChainNode(T data) : data(data), next(nullptr) {};    // 帶有資料的建構函式，初始化下一個節點指針為空
    ChainNode(T data, ChainNode* next) : data(data), next(nullptr) {};    // 帶有資料和下一個節點指針的建構函式，初始化下一個節點指針為空
};

template <class T>
class CircularListWithHeader {
private:
    ChainNode<T>* head;    // 環形串列的頭節點指標
public:
    CircularListWithHeader();    // 預設建構函式
    ~CircularListWithHeader();  // 解構函式
    void InsertFront(const T& e);    // 在串列前端插入一個元素
    void InsertBack(const T& e);     // 在串列後端插入一個元素

    class ChainIterator;             // 環形串列迭代器類別的宣告
    ChainIterator begin() const;     // 返回迭代器指向第一個元素
    ChainIterator end() const;       // 返回迭代器指向最後一個元素的下一個位置
};


template <class T>
class CircularListWithHeader<T>::ChainIterator {
private:
    ChainNode<T>* current;    // 迭代器指向的當前節點
public:
    ChainIterator(ChainNode<T>* startNode = nullptr);    // 建構函式，初始化迭代器，預設起始節點為空
    T& operator *() const;    // 重載取值運算子(*)，返回當前節點的資料指標
    T* operator->() const;    //  重載箭頭運算子(->)，返回指向當前節點資料的指標
    ChainIterator& operator++();    //  重載前置遞增運算子(++i)，使迭代器指向下一個節點
    ChainIterator operator++(int);    // 重載後置遞增運算子(i++)，使迭代器指向下一個節點，返回原來的迭代器
    bool operator!=(const ChainIterator right) const;    // 重載不等於運算子(!=)，比較兩個迭代器是否不相等
```

```
48      bool operator==(const ChainIterator right) const;     // 重載等於運算子(==)，比較兩
          個迭代器是否相等
49  };
50
51  //ChainIterator 實作↓
52  template<class T>
53  CircularListWithHeader<T>::ChainIterator::ChainIterator(ChainNode<T>*  startNode)  {
54      current = startNode;
55  }
56
57  template<class T>
58  T& CircularListWithHeader<T>::ChainIterator::operator* () const {
59      return current->data;
60  }
61
62  template<class T>
63  T* CircularListWithHeader<T>::ChainIterator::operator->() const {
64      return &current->data;
65  }
66
67  template<class T>
68  typename CircularListWithHeader<T>::ChainIterator&
      CircularListWithHeader<T>::ChainIterator::operator++() {
69      current = current->next;
70      return *this;
71  }
72
73  template<class T>
74  typename CircularListWithHeader<T>::ChainIterator
      CircularListWithHeader<T>::ChainIterator::operator++(int) {
75      ChainIterator old = *this;
76      current = current->next;
77      return old;
78  }
79
80  template<class T>
81  bool CircularListWithHeader<T>::ChainIterator::operator!=(const ChainIterator right)
      const {
82      return current != right.current;
83  }
84
85  template<class T>
86  bool CircularListWithHeader<T>::ChainIterator::operator==(const ChainIterator right)
      const {
87      return current == right.current;
88  }
89  //ChainIterator 實作↑
90
91  //CircularListWithHeader 實作↓
92  template <class T>
93  CircularListWithHeader<T>::CircularListWithHeader()  {
94      head = new ChainNode<T>();  // 創建一個新的頭節點，並將 head 指向它
95      head->next = head;          // 將頭節點的 next 指向自己，形成循環
```

```
96  }
97
98
99  template <class T>
100 CircularListWithHeader<T>::~CircularListWithHeader() {
101
102 }
103
104 template <class T>
105 void CircularListWithHeader<T>::InsertFront(const T& e) {
106     ChainNode<T>* newNode = new ChainNode<T>(e);  // 創建包含新元素的節點
107     if (head) {  // nonempty list
108         newNode->next = head->next;  // 新節點的下一個指向原先第一個節點
109         head->next = newNode;  // 頭節點的下一個指向新節點
110     }
111     else {  // empty list
112         head = newNode;  // 頭節點指向新節點
113         newNode->next = newNode;  // 新節點的下一個指向自己，形成循環
114     }
115 }
116
117 template <class T>
118 void CircularListWithHeader<T>::InsertBack(const T& e) {
119     ChainNode<T>* newNode = new ChainNode<T>(e);  // 創建包含新元素的節點
120     if (head) {  // nonempty list
121         newNode->next = head;  // 新節點的下一個指向頭節點
122         ChainNode<T>* last = head;
123         while (last->next != head)
124             last = last->next;  // 找到目前最後一個節點
125         last->next = newNode;  // 目前最後一個節點的下一個指向新節點，形成循環
126     }
127     else {  // empty list
128         head = newNode;  // 頭節點指向新節點
129         newNode->next = newNode;  // 新節點的下一個指向自己，形成循環
130     }
131 }
132
133
134 template<class T>
135 typename CircularListWithHeader<T>::ChainIterator CircularListWithHeader<T>::begin() ↵
      const {
136     // 返回從頭節點的下一個節點開始的迭代器
137     return  CircularListWithHeader<T>::ChainIterator(head->next);
138 }
139
140 template<class T>
141 typename CircularListWithHeader<T>::ChainIterator CircularListWithHeader<T>::end()  ↵
      const {
142     ChainNode<T>* last = head;
143     while (last->next != head)
144         last = last->next;  // 找到目前最後一個節點
145     //  返回指向目前最後一個節點的下一個節點的迭代
146     return  CircularListWithHeader<T>::ChainIterator(last->next);
```

```
147 }
148
149 //CircularListWithHeader 實作↑
150
151 class Polynomial;
152
153 class Term {
154     friend Polynomial;
155     friend ostream& operator<<(ostream& output, const Polynomial& p);
156 private:
157     float coef;    // 係數，表示多項式中項的係數
158     int exp;       // 指數，表示多項式中項的指數
159 public:
160     Term Set(float c, int e) {    // 設定 Term 的係數和指數的成員函式
161         this->coef = c; this->exp = e;
162         return *this;    // 返回當前 Term 物件的指標
163     };
164 };
165
166
167 class Polynomial {
168     friend istream& operator>>(istream& input, Polynomial& p);
169     friend ostream& operator<<(ostream& output, const Polynomial& p);
170 private:
171     CircularListWithHeader<Term>  poly;
172 public:
173     Polynomial(); // Construct the polynomial p(x) = 0.
174     ~Polynomial();
175     const Polynomial& operator=(const Polynomial& a);
176     Polynomial operator+(const Polynomial& b) const;
177     Polynomial operator-(const Polynomial& b) const;
178     Polynomial operator*(const Polynomial& b) const;
179     float Evaluate(float x) const;
180 };
181
182 //Polynomial 實作↓
183 Polynomial::Polynomial()  {
184     poly = CircularListWithHeader<Term>();
185 }
186
187 Polynomial::~Polynomial()  {
188
189 }
190
191 istream& operator>>(istream& input, Polynomial& p)  {
192     Term tmp; // 暫存每一項的係數和指數
193     float t_coef; // 暫存係數的變數
194     int t_exp; // 暫存指數的變數
195     string s;
196     getline(input, s); // 從輸入流讀取整行多項式字串
197     istringstream sin(s); // 將整行字串轉換成輸入流，以便逐項讀取
198     p.poly = CircularListWithHeader<Term>(); // 初始化多項式的循環鏈表
199     while (1) {
```

```
200         if (sin.eof()) break; // 若已到達輸入流結尾，則跳出迴圈
201         else if (sin.peek() == '+') sin.ignore(1); // 忽略正號
202         if (sin.peek() == 'x' || sin.peek() == 'X') {
203             sin.ignore(1); // 忽略 'x' 或 'X'
204             if (sin.peek() == '^') {
205                 sin.ignore(1); // 忽略 '^'
206                 sin >> t_exp; // 讀取指數
207                 p.poly.InsertBack(tmp.Set(1, t_exp)); // 將項插入到多項式
208             }
209             else if (sin.peek() == '+' || sin.peek() == '-' || sin.eof())
210                 p.poly.InsertBack(tmp.Set(1, 1)); // 若無指數，預設為 1
211         }
212         else { // 標準項
213             sin >> t_coef; // 讀取係數
214             if (sin.eof())
215                 p.poly.InsertBack(tmp.Set(t_coef, 0)); // 若已到達輸入流結尾，則插入常數項
216             else if (sin.peek() == 'x' || sin.peek() == 'X') {
217                 sin.ignore(1); // 忽略 'x' 或 'X
218                 if (sin.peek() == '+' || sin.peek() == '-' || sin.eof())
219                     p.poly.InsertBack(tmp.Set(t_coef, 1)); // 若無指數，預設為 1
220                 else if (sin.peek() == '^') {
221                     sin.ignore(1); // 忽略 '^'
222                     sin >> t_exp; // 讀取指數
223                     p.poly.InsertBack(tmp.Set(t_coef, t_exp)); // 將項插入到多項式
224                 }
225             }
226         }
227     }
228     return input;
229 }
230
231 ostream& operator<<(ostream& output, const Polynomial& p) {
232     for (CircularListWithHeader<Term>::ChainIterator i = p.poly.begin(); i != p.poly.end(); i++) {
233         if (i != p.poly.begin()) {
234             if (i->coef > 0) output << "+"; // 若係數為正，輸出正號
235         }
236         if (i->exp == 0)
237             output << i->coef; // 若指數為0，只輸出係數
238         else if (i->coef == 1 && i->exp == 1)
239             output << "x"; // 若係數為1，且指數為1，只輸出 'x'
240         else if (i->exp == 1)
241             output << i->coef << "x"; // 若指數為1，輸出係數和 'x'
242         else if (i->coef == 1)
243             output << "x^" << i->exp; // 若係數為1，輸出 'x' 的指數
244         else
245             output << i->coef << "x^" << i->exp; // 一般情況，輸出係數、'x' 和指數
246     }
247     return output;
248 }
249
250 const Polynomial& Polynomial::operator=(const Polynomial& a) {
```

```
251     Polynomial np; // 創建一個新的多項式
252     np.poly = CircularListWithHeader<Term>(a.poly); // 複製其他多項式的環形串列
253     return np; // 回傳新的多項式
254 }
255
256 Polynomial Polynomial::operator+(const Polynomial& b) const {
257     //Polynomials *this(a) an b are added and the sum returned.
258     Term temp;
259     CircularListWithHeader<Term>::ChainIterator ai = poly.begin(),
260                                                 bi = b.poly.begin();
261     Polynomial c; //assume constructor sets head->exp = -1
262     while (ai != poly.end() && bi != b.poly.end()) {
263         if (ai->exp == bi->exp) {
264             float sum = ai->coef + bi->coef;
265             if (sum) c.poly.InsertBack(temp.Set(sum, ai->exp));
266             ai++; bi++; //advance to next term
267         }
268         else if (ai->exp < bi->exp) {
269             c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
270             bi++; //next term of b;
271         }
272         else {
273             c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
274             ai++; //next term of a
275         }
276     }
277     while (ai != poly.end()) {
278         c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
279         ai++;
280     }
281     while (bi != b.poly.end()) {
282         c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
283         bi++;
284     }
285     return c;
286 }
287
288 Polynomial Polynomial::operator-(const Polynomial& b) const {
289     Polynomial c; // 創建一個新的多項式，用於存儲相減結果
290     Term temp; // 用於暫存新生成的項
291     CircularListWithHeader<Term>::ChainIterator ai = poly.begin(),
292                                                 bi = b.poly.begin();
293     // 走訪兩個多項式的環形串列，進行相減操作
294     while (ai != poly.end() && bi != b.poly.end()) {
295         if (ai->exp > bi->exp) { // 若第一個多項式的指數大於第二個多項式的指數
296             c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
297             ai++;
298         }
299         else if (ai->exp < bi->exp) { // 若第一個多項式的指數小於第二個多項式的指數
300             c.poly.InsertBack(temp.Set(bi->coef * -1, bi->exp));
301             bi++;
302         }
303         else { // 若兩個多項式的指數相等
```

```
304             if (ai->coef != bi->coef)
305                 c.poly.InsertBack(temp.Set(ai->coef - bi->coef, ai->exp));
306             ai++;  bi++;
307         }
308     }
309     while (ai != poly.end()) { // 將第一個多項式剩餘的項直接加入結果多項式
310         c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
311         ai++;
312     }
313     while (bi != b.poly.end()) { // 將第二個多項式剩餘的項直接加入結果多項式
314         c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
315         bi++;
316     }
317 
318     return c; // 返回相減結果
319 }
320 
321 Polynomial Polynomial::operator*(const Polynomial& b) const {
322     Polynomial c; // 創建一個新的多項式對象，用於存儲相乘結果
323     Term temp; // 用於暫存新生成的項
324     CircularListWithHeader<Term>::ChainIterator ai, bi, ci;
325     map<int, float> data; // 用於存儲相乘後的項的係數和指數的對應關係
326 
327     // 走訪兩個多項式的環形串列，進行相乘操作
328     for (ai = poly.begin(); ai != poly.end(); ai++) {
329         for (bi = b.poly.begin(); bi != b.poly.end(); bi++) {
330             float t_coef = ai->coef * bi->coef; // 計算兩個項的係數相乘
331             int t_exp = ai->exp + bi->exp; // 計算兩個項的指數相加
332 
333             // 將計算得到的項的係數和指數存入map中
334             if (data.find(t_exp) == data.end()) // 若指數尚未存在於map中
335                 data[t_exp] = t_coef; // 直接添加新的項
336             else
337                 data[t_exp] += t_coef; // 若指數已存在於map中，則將係數相加
338         }
339     }
340 
341     //  走訪map，將其中的項按指數遞減的順序加入結果多項式
342     for (auto id = data.begin(); id != data.end(); id++)
343         if (id->second != 0)
344             c.poly.InsertFront(temp.Set(id->second,  id->first));
345 
346     return c; // 返回相乘結果
347 }
348 
349 float Polynomial::Evaluate(float x) const {
350     float res = 0.0f; // 初始結果為0
351     CircularListWithHeader<Term>::ChainIterator ai = poly.begin();
352 
353     //  走訪每一項，計算其在指定x值下的值，並將結果加總
354     while (ai != poly.end()) {
355         float ans = ai->coef; // 初始化ans為該項的係數
356         for (int i = 0; i < ai->exp; i++)
```

```
357             ans *= x; // 將該項的指數次方加入計算
358         res += ans; // 將計算結果加總到總結果中
359         ai++; // 移動到下一項
360     }
361
362     return res; // 返回多項式在指定x值下的計算結果
363 }
364 //Polynomial 實作↑
365
366 int main() {
367     clock_t start, finish;
368     Polynomial a, b;
369     cout << "輸入a多項式：";
370     cin >> a;
371     cout << "輸入b多項式：";
372     cin >> b;
373     cout << "--------------------------------------" << endl;
374     cout << "a = " << a<<endl;
375     cout << "b = " << b<<endl;
376     start = clock();
377     cout << "a + b = " << a + b;
378     finish = clock();
379     cout << " 需時：" << (double)(finish - start) / CLOCKS_PER_SEC << "s" << endl;
380     start = clock();
381     cout << "a - b = " << a - b;
382     finish = clock();
383     cout << " 需時：" << (double)(finish - start) / CLOCKS_PER_SEC << "s" << endl;
384     start = clock();
385     cout << "a * b = " << a * b;
386     finish = clock();
387     cout << " 需時：" << (double)(finish - start) / CLOCKS_PER_SEC << "s" << endl;
388     cout << "--------------------------------------" << endl;
389     cout << "請輸入x，以求出a(x)、b(x)的值?";
390     int x;
391     cin >> x;
392     start = clock();
393     cout << "a(x) = " << a.Evaluate(x);
394     finish = clock();
395     cout << " 需時：" << (double)(finish - start) / CLOCKS_PER_SEC << "s" << endl;
396     start = clock();
397     cout << "b(x) = " << b.Evaluate(x);
398     finish = clock();
399     cout << " 需時：" << (double)(finish - start) / CLOCKS_PER_SEC << "s" << endl;
400     system("pause");
401     return 0;
402 }
403 /*
404  5x^9-3x^8+2x^7+x^6-4x^5+6x^4-8x^3+7x^2-9x+10
405   -2x^10+4x^9-6x^8+8x^7-10x^6+12x^5-14x^4+16x^3-18x^2+20x-22
406 2
407
408 */
```