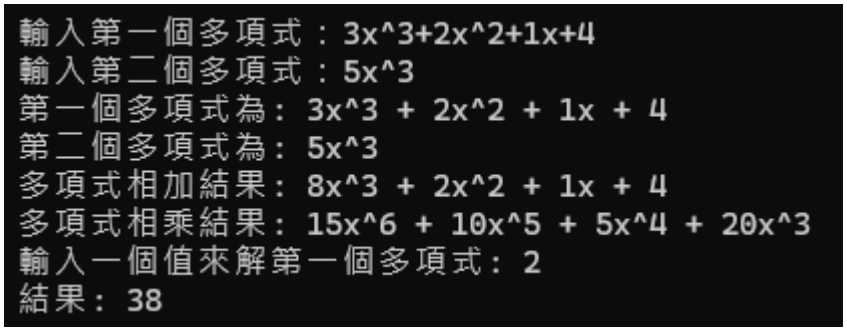
結果輸出：



解題說明：

這次的作業要我們設計一個能夠表示和操作多項式的程式。多項式會用環形鏈表來表示，每個節點包含係數和指數。我們是要實現多項式的加法、減法、乘法，以及對某個數值進行計算。

效能分析：

1. 時間複雜度：

   加法與減法：O(n+m)，其中 n,m 分別為兩個多項式項數。

   乘法：O(n⋅m)。

   求值：O(n)。

2. 空間複雜度：
   使用環形鏈表儲存，空間效率高；多項式運算過程中額外生成新節點。

申論及開發報告：

1. 程式設計亮點：

環形鏈表實現多項式的存儲，避免多餘記憶體操作。

靈活的運算符重載支持數學運算與結果展示。

2. 改進方向：

可擴展性：增加多變數多項式支持。

效能優化：針對稀疏多項式使用壓縮存儲。

3. 總結：
   該實作完整實現多項式運算需求，符合效率與可讀性要求。

程式：

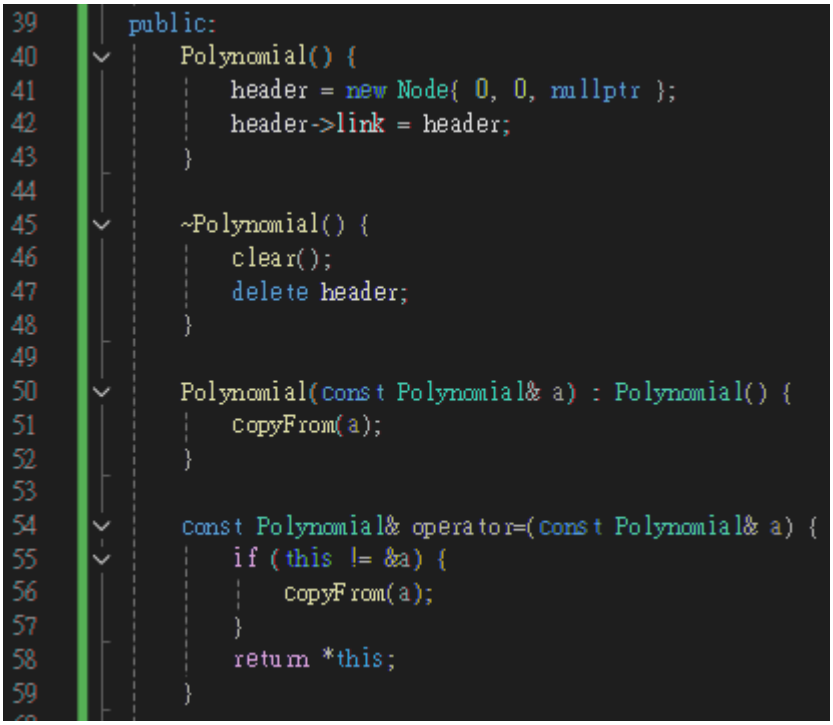
```cpp
#include <iostream>
#include <sstream>
#include <cmath>
using namespace std;

struct Node {
    int coef;
    int exp;
    Node* link;
};

class Polynomial {
private:
    Node* header;

    void clear() {
        if (!header) return;
        Node* current = header->link;
        while (current != header) {
            Node* temp = current;
            current = current->link;
            delete temp;
        }
        header->link = header;
    }

    void CopyFrom(const Polynomial& a) {
        clear();
        Node* currentA = a.header->link;
        Node* last = header;
        while (currentA != a.header) {
            Node* newNode = new Node{ currentA->coef, currentA->exp, header };
            last->link = newNode;
            last = newNode;
            currentA = currentA->link;
        }
    }
```



```
39  public:
40      Polynomial() {
41          header = new Node{ 0, 0, nullptr };
42          header->link = header;
43      }
44
45      ~Polynomial() {
46          clear();
47          delete header;
48      }
49
50      Polynomial(const Polynomial& a) : Polynomial() {
51          copyFrom(a);
52      }
53
54      const Polynomial& operator=(const Polynomial& a) {
55          if (this != &a) {
56              copyFrom(a);
57          }
58          return *this;
59      }
```

```
61      friend istream& operator>>(istream& is, Polynomial& x) {
62          x.clear();
63          string input;
64          getline(is, input);
65          stringstream ss(input);
66          Node* last = x.header;
67          string term;
68          while (ss >> term) {
69              size_t xPos = term.find("x^");
70              int coef = 0, exp = 0;
71              if (xPos != string::npos) {
72                  coef = stoi(term.substr(0, xPos));
73                  exp = stoi(term.substr(xPos + 2));
74              }
75              else {
76                  coef = stoi(term);
77                  exp = 0;
78              }
79              Node* newNode = new Node{ coef, exp, x.header };
80              last->link = newNode;
81              last = newNode;
82          }
83          return is;
84      }
85
86      friend ostream& operator<<(ostream& os, const Polynomial& x) {
87          Node* current = x.header->link;
88          while (current != x.header) {
89              os << current->coef << "x^" << current->exp;
90              current = current->link;
91              if (current != x.header) os << " + ";
92          }
93          return os;
94      }
```

```
96      Polynomial operator+(const Polynomial& b) const {
97          Polynomial result;
98          Node* currentA = header->link;
99          Node* currentB = b.header->link;
100         Node* last = result.header;
101
102         while (currentA != header || currentB != b.header) {
103             int coef, exp;
104             if (currentA != header && (currentB == b.header || currentA->exp > currentB->exp)) {
105                 coef = currentA->coef;
106                 exp = currentA->exp;
107                 currentA = currentA->link;
108             }
109             else if (currentB != b.header && (currentA == header || currentA->exp < currentB->exp)) {
110                 coef = currentB->coef;
111                 exp = currentB->exp;
112                 currentB = currentB->link;
113             }
114             else {
115                 coef = currentA->coef + currentB->coef;
116                 exp = currentA->exp;
117                 currentA = currentA->link;
118                 currentB = currentB->link;
119             }
120             if (coef != 0) {
121                 Node* newNode = new Node{ coef, exp, result.header };
122                 last->link = newNode;
123                 last = newNode;
124             }
125         }
126         return result;
127     }
```

```
129     Polynomial operator-(const Polynomial& b) const {
130         Polynomial result;
131         Node* currentA = header->link;
132         Node* currentB = b.header->link;
133         Node* last = result.header;
134
135         while (currentA != header || currentB != b.header) {
136             int coef, exp;
137             if (currentA != header && (currentB == b.header || currentA->exp > currentB->exp)) {
138                 coef = currentA->coef;
139                 exp = currentA->exp;
140                 currentA = currentA->link;
141             }
142             else if (currentB != b.header && (currentA == header || currentA->exp < currentB->exp)) {
143                 coef = -currentB->coef;
144                 exp = currentB->exp;
145                 currentB = currentB->link;
146             }
147             else {
148                 coef = currentA->coef - currentB->coef;
149                 exp = currentA->exp;
150                 currentA = currentA->link;
151                 currentB = currentB->link;
152             }
153             if (coef != 0) {
154                 Node* newNode = new Node{ coef, exp, result.header };
155                 last->link = newNode;
156                 last = newNode;
157             }
158         }
159         return result;
160     }
```

```
    Polynomial operator*(const Polynomial& b) const {
        Polynomial result;
        Node* currentA = header->link;
        while (currentA != header) {
            Polynomial temp;
            Node* last = temp.header;
            Node* currentB = b.header->link;
            while (currentB != b.header) {
                Node* newNode = new Node{ currentA->coef * currentB->coef, currentA->exp + currentB->exp, temp.header };
                last->link = newNode;
                last = newNode;
                currentB = currentB->link;
            }
            result = result + temp;
            currentA = currentA->link;
        }
        return result;
    }

    float Evaluate(float x) const {
        float result = 0;
        Node* current = header->link;
        while (current != header) {
            result += current->coef * pow(x, current->exp);
            current = current->link;
        }
        return result;
    }
};
```

```
int main() {
    Polynomial p1, p2;
    cout << "輸入第一個多項式: ";
    cin >> p1;
    cout << "第一個多項式: " << p1 << endl;

    cout << "輸入第二個多項式: ";
    cin >> p2;
    cout << "第二個多項式: " << p2 << endl;

    Polynomial sum = p1 + p2;
    cout << "相加: " << sum << endl;

    Polynomial diff = p1 - p2;
    cout << "相減: " << diff << endl;

    Polynomial prod = p1 * p2;
    cout << "相乘: " << prod << endl;

    float x;
    cout << "輸入帶入值";
    cin >> x;
    cout << "結果:" << x << ": " << p1.Evaluate(x) << endl;

    return 0;
}
```