组号: 6

上海大学计算机工程与科学学院

实 验 报 告

（数据结构 1）

学　　期：大二冬季学期

组　　长：林仪

学　　号：23121029

指导教师：朱频频

成绩评定：　　（教师填写）

二〇二三年四月六日

小组信息				
登记序号	姓名	学号	分工	签名
1	林仪	23121029	ppt 制作、汇报	
2	罗苗	23121791	ppt 制作、汇报	
3	彭欣然	23122809	代码实现	
4	顾宜凌	23121721	代码实现	
5	张静雯	23121790	代码实现	
6	饶思莹	23121617	代码实现	
7	杨源	23121651	报告	
8	李临宜	23122814	报告	

实验概述	
实验零	（熟悉上机环境、进度安排、评分制度；确定小组成员）
实验一	一元 n 次多项式的设计
实验二	

实验三	
实验四	

实验一

一、实验题目

一元 n 次多项式的设计

二、实验内容

1、任务目标

基于线性表的相关知识和代码实现，实现对一元 n 次多项式的求导、四则运算和求值等操作。

2、任务描述

1）可以在教材代码的基础上进行编码，也可以自行从头开始编码；

2）提供菜单或其他 UI，实现多项式输入（两个多项式）、多项式显示、加法、减法、乘法、求导、求值、综合运算等功能；

3）可以考虑使用 GUI 显示多项式；

4）选择运算功能的时候如果在没有输入两个多项式的情况下，提示输入；

5）后续会提供两个多项式作为测试，自行输入多项式后，将以下显示结果的截图作为测试结果在报告和 PPT 中进行展示：

多项式 A，多项式 B，A+B， A-B，A*B， d(A)，d(B)， A+B+d(A)+d(B)

x=1，综合运算结果

x=3，综合运算结果

3、功能列表

1）输入多项式 A、B；

2）显示多项式 A 和 B；

3） 计算 A + B;

4）计算 A - B;

5）计算 A * B;

6）计算 d(A) 和 d(B);

7）综合运算: A + B + d(A) + d(B);

8）计算综合运算在 x 的值;

9）退出操作系统;

三、解决方案

1、算法设计

在本实验中使用 C++语言进行编程，根据实验要求：

（1）我们首先设计了一个 通用异常类（Error），用于提示异常，该类位于头文件 Assistance.h，是后续头文件以及源文件运行的基础。

（2）其次，我们基于头文件 Assistance.h，编写了头文件 Node.h（节点类）、头文件 LinkList.h（单链表类），用于存储线性表中的数据元素。

（3）再次，创建头文件 PolyItem.h（多项式类），数据成员为 coef（系数）、expn（指数）。

（4）最后，创建头文件 Polynomial，并创建数据成员 LinkList<PolyItem> polyList（多项式组成的线性表）。

头文件 Assistance.h

```
#pragma once
#pragma once
#ifndef __ASSISTANCE_H__                // 如果没有定义__ASSISTANCE_H__
#define __ASSISTANCE_H__                // 那么定义__ASSISTANCE_H__

// 辅助软件包

// ANSI C++标准库头文件
#include <cstring>                      // 标准串操作
#include <iostream>                     // 标准流操作
#include <limits>                       // 极限
#include <cmath>                        // 数据函数
#include <fstream>                      // 文件输入输出
#include <cctype>                       // 字符处理
#include <ctime>                        // 日期和时间函数
#include <cstdlib>                      // 标准库
#include <cstdio>                     // 标准输入输出
#include <iomanip>                      // 输入输出流格式设置
#include <cstdarg>                      // 支持变长函数参数
#include <cassert>                      // 支持断言
using namespace std;                  // 标准库包含在命名空间 std 中

// 自定义类型
enum Status {
    SUCCESS, FAIL, UNDER_FLOW, OVER_FLOW, RANGE_ERROR, DUPLICATE_ERROR,
    NOT_PRESENT, ENTRY_INSERTED, ENTRY_FOUND, VISITED, UNVISITED
};

// 宏定义
#define DEFAULT_SIZE 1000             // 缺省元素个数
#define DEFAULT_INFINITY 1000000      // 缺省无穷大



// 辅助函数声明

char GetChar(istream& inStream = cin); // 从输入流 inStream 中跳过空格及制表符获取一字符

template <class ElemType >
void Swap(ElemType& e1, ElemType& e2); // 交换 e1, e2 之值

template<class ElemType>
```

```
void Display(ElemType elem[], int n);  // 显示数组 elem 的各数据元素值

template <class ElemType>
void Write(const ElemType& e);              // 显示数据元素

// 辅助类
class Error;            // 通用异常类

char GetChar(istream& inStream)
// 操作结果：从输入流 inStream 中跳过空格及制表符获取一字符
{
    char ch;                                // 临时变量
    while ((ch = (inStream).peek()) != EOF     // 文件结束符(peek()函数从输入流中接受 1
        // 字符,流的当前位置不变)
        && ((ch = (inStream).get()) == ' '// 空格(get()函数从输入流中接受 1 字符,流
            // 的当前位置向后移 1 个位置)
            || ch == '\t'));                    // 制表符

    return ch;                                  // 返回字符
}


// 通用异常类
#define MAX_ERROR_MESSAGE_LEN 100
class Error
{
private:
    // 数据成员
    char message[MAX_ERROR_MESSAGE_LEN];// 异常信息

public:
    //  方法声明
    Error(const char* mes = "一般性异常!");     // 构造函数
    ~Error(void) {};                    // 析构函数
    void Show() const;                      // 显示异常信息
};

// 通用异常类的实现部分
Error::Error(const char* mes) {
    strcpy_s(message, sizeof(message), mes);  // 安全复制异常信息
}
```

```
void Error::Show()const
// 操作结果：显示异常信息
{
    cout << message << endl;              // 显示异常信息
}


template <class ElemType >
void Swap(ElemType& e1, ElemType& e2)
// 操作结果：交换 e1, e2 之值
{
    ElemType temp;       // 临时变量
    // 循环赋值实现交换 e1, e2
    temp = e1;  e1 = e2;  e2 = temp;
}

template<class ElemType>
void Display(ElemType elem[], int n)
// 操作结果：显示数组 elem 的各数据元素值
{
    for (int i = 0; i < n; i++)
    {    // 显示数组 elem
        cout << elem[i] << "   ";
    }
    cout << endl;
}

template <class ElemType>
void Write(const ElemType& e)
// 操作结果：显示数据元素
{
    cout << e << "   ";
}

#endif
```

头文件 Node.h

```
#pragma once
#ifndef __NODE_H__
#define __NODE_H__
#include "Assistance.h"
```

```
// 结点类
template <class ElemType>
struct Node
{
    // 数据成员:
    ElemType data;                  // 数据域
    Node<ElemType>* next;           // 指针域

    // 构造函数:
    Node();                         // 无参数的构造函数
    Node(ElemType e, Node<ElemType>* link = NULL);// 已知数数据元素值和指针建立结构
};

// 结点类的实现部分
template<class ElemType>
Node<ElemType>::Node()
// 操作结果：构造指针域为空的结点
{
    next = NULL;
}

template<class ElemType>
Node<ElemType>::Node(ElemType e, Node<ElemType>* link)
// 操作结果：构造一个数据域为 e 和指针域为 link 的结点
{
    data = e;
    next = link;
}

#endif
```

头文件 LinkList.h

```
#pragma once
#pragma once
#ifndef __LK_LIST_H__
#define __LK_LIST_H__

#include "Node.h"                       // 结点类
#include "Assistance.h"
// 单链表类
```

```
template <class ElemType>
class LinkList
{
protected:
    //  单链表的数据成员
    Node<ElemType>* head;              // 头结点指针
    int length;                        // 单链表长度

public:
    //  单链表的函数成员
    LinkList();                        // 无参数的构造函数
    LinkList(ElemType v[], int n);     // 有参数的构造函数
    virtual ~LinkList();               // 析构函数
    int GetLength() const;             // 求单链表长度
    bool IsEmpty() const;              // 判断单链表是否为空
    void Clear();                      // 将单链表清空
    void Traverse(void (*Visit)(const ElemType&)) const;// 遍历单链表
    int LocateElem(const ElemType& e) const;          // 元素定位
    Status GetElem(int position, ElemType& e) const;   // 求指定位置的元素
    Status SetElem(int position, const ElemType& e);   // 设置指定位置的元素值
    Status DeleteElem(int position, ElemType& e);      // 删除元素
    Status InsertElem(int position, const ElemType& e); // 在制定位置插入元素
    Status InsertElem(const ElemType& e);              // 在表尾插入元素
    LinkList(const LinkList<ElemType>& copy);          // 复制构造函数
    LinkList<ElemType>& operator =(const LinkList<ElemType>& copy); // 重载赋值运算
};


// 单链表类的实现部分


template <class ElemType>
LinkList<ElemType>::LinkList()
// 操作结果：构造一个空链表
{
    head = new Node<ElemType>;         // 构造头结点
    assert(head != 0);                 // 构造头结点失败，终止程序运行
    length = 0;                        // 初始化单链表长度为 0
}

template <class ElemType>
LinkList<ElemType>::LinkList(ElemType v[], int n)
```

```
// 操作结果：根据数组 v 中的元素构造单链表
{
    Node<ElemType>* p;
    p = head = new Node<ElemType>;    // 构造头结点
    assert(head != 0);                // 构造头结点失败，终止程序运行
    for (int i = 0; i < n; i++) {
        p->next = new Node<ElemType>(v[i], NULL);
        assert(p->next != 0);         // 构造元素结点失败，终止程序运行
        p = p->next;
    }
    length = n;                       // 初始化单链表长度为 n
}

template <class ElemType>
LinkList<ElemType>::~LinkList()
// 操作结果：销毁单链表
{
    Clear();          // 清空单链表
    delete head;      // 释放头结点所指空间
}

template <class ElemType>
int LinkList<ElemType>::GetLength() const
// 操作结果：返回单链表的长度
{
    return length;
}

template <class ElemType>
bool LinkList<ElemType>::IsEmpty() const
// 操作结果：如单链表为空，则返回 true，否则返回 false
{
    return head->next == NULL;
}

template <class ElemType>
void LinkList<ElemType>::Clear()
// 操作结果：清空单链表,删除单链表中所有元素结点
{
    Node<ElemType>* p = head->next;
    while (p != NULL) {
        head->next = p->next;
```

```
        delete p;
        p = head->next;
    }
    length = 0;
}

template <class ElemType>
void LinkList<ElemType>::Traverse(void (*Visit)(const ElemType&)) const
// 操作结果：依次对单链表的每个元素调用函数(*visit)访问
{
    Node<ElemType>* p = head->next;
    while (p != NULL) {
        (*Visit)(p->data);     // 对单链表中每个元素调用函数(*visit)访问
        p = p->next;
    }
}

template <class ElemType>
int LinkList<ElemType>::LocateElem(const ElemType& e) const
// 元素定位
{
    Node<ElemType>* p = head->next;
    int count = 1;
    while (p != NULL && p->data != e) {
        count++;
        p = p->next;
    }
    return  (p != NULL) ? count : 0;
}

template <class ElemType>
Status LinkList<ElemType>::GetElem(int i, ElemType& e) const
// 操作结果：当单链表存在第 i 个元素时，用 e 返回其值，函数返回 ENTRY_FOUND,
//   否则函数返回 NOT_PRESENT
{
    if (i < 1 || i > length)
        return RANGE_ERROR;
    else {
        Node<ElemType>* p = head->next;
        int count;
        for (count = 1; count < i; count++)
            p = p->next;                    // p 指向第 i 个结点
```

```
            e = p->data;                    // 用 e 返回第 i 个元素的值
            return ENTRY_FOUND;
        }
}

template <class ElemType>
Status LinkList<ElemType>::SetElem(int i, const ElemType& e)
// 操作结果：将单链表的第 i 个位置的元素赋值为 e,
//    i 的取值范围为 1≤i≤length,
//    i 合法时函数返回 SUCCESS,否则函数返回 RANGE_ERROR
{
    if (i < 1 || i > length)
        return RANGE_ERROR;
    else {
        Node<ElemType>* p = head->next;
        int count;
        for (count = 1; count < i; count++)
            p = p->next;                    // 取出指向第 i 个结点的指针
        p->data = e;                    // 修改第 i 个元素的值为 e
        return SUCCESS;
    }
}

template <class ElemType>
Status LinkList<ElemType>::DeleteElem(int i, ElemType& e)
// 操作结果：删除单链表的第 i 个位置的元素，并用 e 返回其值,
//    i 的取值范围为 1≤i≤length,
//    i 合法时函数返回 SUCCESS,否则函数返回 RANGE_ERROR
{
    if (i < 1 || i > length)
        return RANGE_ERROR;    // i 范围错
    else {
        Node<ElemType>* p = head, * q;
        int count;
        for (count = 1; count < i; count++)
            p = p->next;            // p 指向第 i-1 个结点
        q = p->next;            // q 指向第 i 个结点
        p->next = q->next;     // 删除结点
        e = q->data;            // 用 e 返回被删结点元素值
        length--;                // 删除成功后元素个数减 1
        delete q;                // 释放被删结点
        return SUCCESS;
```

```
    }
}

template <class ElemType>
Status LinkList<ElemType>::InsertElem(int i, const ElemType& e)
// 操作结果：在单链表的第 i 个位置前插入元素 e
//   i 的取值范围为 1≤i≤length+1
//   i 合法时返回 SUCCESS，否则函数返回 RANGE_ERROR
{
    if (i < 1 || i > length + 1)
        return RANGE_ERROR;
    else {
        Node<ElemType>* p = head, * q;
        int count;
        for (count = 1; count < i; count++)
            p = p->next;                          // p 指向第 i-1 个结点
        q = new Node<ElemType>(e, p->next); // 生成新结点 q
        assert(q != 0);                       // 申请结点失败，终止程序运行
        p->next = q;                          // 将 q 插入到链表中
        length++;                             // 插入成功后，单链表长度加 1
        return SUCCESS;
    }
}

template <class ElemType>
Status LinkList<ElemType>::InsertElem(const ElemType& e)
// 操作结果：在单链表的表尾位置插入元素 e
{
    Node<ElemType>* p, * q;
    q = new Node<ElemType>(e, NULL);     // 生成新结点 q
    assert(q != 0);                      // 申请结点失败，终止程序运行
    for (p = head; p->next != NULL; p = p->next);// p 指向表尾结点
    p->next = q;                         // 在单链表的表尾位置插入新结点
    length++;                            // 插入成功后，单链表长度加 1
    return SUCCESS;
}

template <class ElemType>
LinkList<ElemType>::LinkList(const LinkList<ElemType>& copy)
// 操作结果：复制构造函数，由单链表 copy 构造新单链表
{
    int copyLength = copy.GetLength();    // 取被复制单链表的长度
```

```
    ElemType e;
    head = new Node<ElemType>;              // 构造头指针
    assert(head != 0);                      // 构造头指针失败，终止程序运行
    length = 0;                             // 初始化元素个数

    for (int i = 1; i <= copyLength; i++) { // 复制数据元素
        copy.GetElem(i, e);   // 取出第 i 个元素的值放在 e 中
        InsertElem(e);        // 将 e 插入到当前单链表的表尾
    }
}

template <class ElemType>
LinkList<ElemType>& LinkList<ElemType>::operator =(const LinkList<ElemType>& other)
// 操作结果：重载赋值运算符，将单链表 other 赋值给当前单链表
{
    if (&other != this) {
        int otherLength = other.GetLength();// 取被赋值单链表的长度
        ElemType e;
        Clear();                            // 清空当前单链表
        for (int i = 1; i <= otherLength; i++) {
            other.GetElem(i, e);            // 取出第 i 个元素的值放在 e 中
            InsertElem(e);                  // 将 e 插入到当前单链表的表尾
        }
    }
    return *this;
}

#endif
```

头文件 PolyItem.h

```
#pragma once
#ifndef __POLY_ITEM_H__
#define __POLY_ITEM_H__
#include "Assistance.h"

// 多项式项类
struct PolyItem
{
    // 数据成员:
    double coef;            // 系数
    int    expn;            // 指数
```

```
    // 构造函数:
    PolyItem();                 // 无数据的构造函数
    PolyItem(double cf, int en);// 已知系数域和指数域建立结构
};



// 多项式项类的实现部分

PolyItem::PolyItem()
// 操作结果：构造指数域为-1 的结点
{
    expn = -1;
}


PolyItem::PolyItem(double cf, int en)
// 操作结果：构造一个系数域为 cf 和指数域为 en 的结点
{
    coef = cf;
    expn = en;
}

#endif

#pragma once
```

头文件 Polynomial.h

```
#pragma once
#pragma once
#ifndef __POLYNOMIAL_H__
#define __POLYNOMIAL_H__

#include "LinkList.h" // 链表类
#include"PolyItem.h"
#include "Assistance.h"
#include "graphics.h"
#include <tchar.h>  // 引入 TCHAR 支持
#include <cwchar>   // 引入 wcstombs 转换
#include <iostream>  // 包含 cin 和 ignore
#include <limits>    // 包含 numeric_limits
using namespace std;
```

```
// 多项式类
class Polynomial {
protected:
    LinkList<PolyItem> polyList; // 多项式组成的线性表

public:
    Polynomial() {}                          // 无参构造函数
    ~Polynomial() {}                         // 析构函数
    Polynomial(const Polynomial& copy); // 复制构造函数
    Polynomial(const LinkList<PolyItem>& copyLinkList); // 转换构造函数
    int Length() const;                      // 求多项式的项数
    bool IsZero() const;                     // 判断多项式是否为 0
    void SetZero();                          // 将多项式置为 0
    void input();
    void Display();                          // 显示多项式
    void InsItem(const PolyItem& item); // 插入一项
    Polynomial operator +(const Polynomial& p) const; // 加法运算符重载
    Polynomial operator -(const Polynomial& p) const;
    Polynomial operator *(const Polynomial& p) const;
    Polynomial& operator =(const Polynomial& copy); // 赋值语句重载
    Polynomial& operator =(const LinkList<PolyItem>& copyLinkList); // 赋值语句重载
    Polynomial derivative();
    double evaluate(double x) const; // 求值
};

// 多项式类的实现部分
Polynomial::Polynomial(const Polynomial& copy) { // 复制构造函数
    polyList = copy.polyList;
}

Polynomial::Polynomial(const LinkList<PolyItem>& copyLinkList) { // 转换构造函数
    polyList = copyLinkList;
}

int Polynomial::Length() const { // 返回多项式的项数
    return polyList.GetLength();
}

bool Polynomial::IsZero() const { // 多项式为 0，则返回 true，否则返回 false
    return polyList.IsEmpty();
}
```

```
void Polynomial::SetZero() { // 将多项式置为 0
    polyList.Clear();
}

void Polynomial::input() {
    polyList.Clear();
    double coef;
    int expn;

    while (true) {
        cout << "请输入系数和指数 (系数 指数)，输入结束请输入 0 0：" << endl;

        if (!(cin >> coef >> expn)) {                          // 检查输入流是否存在错误
            cout << "无效输入，请输入有效的数字！" << endl;
            cin.clear();
            cin.ignore(100, '\n');
            continue;
        }
        if (cin.peek() != '\n') {        // 检查是否有额外输入
            cout << "警告：检测到额外输入，系统将忽略额外输入。" << endl;
            cin.ignore(100, '\n');
        }
        if (coef == 0 && expn == 0) {                          // 输入结束条件
            break;
        }
        if (expn < 0) {                                        // 校验指数是否为负数
            cout << "指数不能为负数，请重新输入！" << endl;
            continue;
        }
        // 插入合法项
        PolyItem newItem(coef, expn);
        InsItem(newItem);
        cout << "输入成功！" << endl;
    }
    cout << "输入完成。" << endl;
}

void Polynomial::Display() {
    // 初始化图形界面
    initgraph(640, 480);
    cleardevice();
```

```
    int pos = 1;
    PolyItem it;
    bool isFirst = true;
    Status status = polyList.GetElem(pos, it);
    int xPos = 50; // 起始位置
    int yPos = 200;
    const int maxWidth = 900; // 每行最大宽度
    // 循环遍历多项式的每一项，绘制
    while (status == ENTRY_FOUND) {
      wchar_t text[100] = L"";
      // 处理符号
      if (it.coef > 0 && pos > 1) {
          wcscat_s(text, sizeof(text) / sizeof(wchar_t), L"+");
      }
      else if (it.coef < 0) {
          wcscat_s(text, sizeof(text) / sizeof(wchar_t), L"-");
      }
    // 处理系数
    if (it.coef != 0) {
        if (it.coef != 1 && it.coef != -1) {
            swprintf_s(text + wcslen(text), sizeof(text) / sizeof(wchar_t) - wcslen(text), L"%.1f",
it.coef);
        }
        else if (it.coef == -1) {
            wcscat_s(text, sizeof(text) / sizeof(wchar_t), L"-");
        }
    }
    // 检测换行
        if (xPos + wcslen(text) * 8 > maxWidth) {
            xPos = 50;
            yPos += 50;
        }
    // 绘制系数部分
    outtextxy(xPos, yPos, text);
    xPos += wcslen(text) * 8;
    // 处理指数
    if (it.expn > 1) {
        // 绘制 x^n
        wchar_t xText[] = L"x";
        outtextxy(xPos, yPos, xText);
        xPos += wcslen(xText) * 8;
```

```
            wchar_t expnText[10];
            swprintf_s(expnText, sizeof(expnText) / sizeof(wchar_t), L"%d", it.expn);
            outtextxy(xPos, yPos-10, expnText);  // 上标显示
            xPos += wcslen(expnText) * 8;
        }
        else if (it.expn == 1) {
            // 处理 x^1
            wchar_t xText[] = L"x";
            outtextxy(xPos, yPos, xText);
            xPos += wcslen(xText) * 8;
        }
        // 获取下一项
        status = polyList.GetElem(++pos, it);
        isFirst = false;
    }
    // 如果多项式是零，显示 "0"
    if (isFirst) {
        wchar_t zeroText[] = L"0";
        outtextxy(50, 200, zeroText);
    }
    // 等待用户按任意键后关闭图形窗口
    system("pause");
    closegraph();  // 关闭图形窗口
}

void Polynomial::InsItem(const PolyItem& node) {
    if (node.coef == 0) return; // 如果系数为 0，则不插入

    int pos = 1;
    PolyItem it;
    Status status = polyList.GetElem(pos, it);

    // 查找插入位置
    while (status == ENTRY_FOUND && it.expn > node.expn) {
        pos++;
        status = polyList.GetElem(pos, it);
    }

    if (status == ENTRY_FOUND && it.expn == node.expn) {
        // 同指数项，合并系数
        PolyItem mergedNode(it.coef + node.coef, it.expn);
        if (mergedNode.coef != 0) {
```

```
                polyList.SetElem(pos, mergedNode); // 更新项
            }
            else {
                PolyItem dummy;
                polyList.DeleteElem(pos, dummy); // 如果系数合并后为 0，则删除该项
            }
        }
        else {
            // 插入新项
            polyList.InsertElem(pos, node);
        }
}

Polynomial& Polynomial::operator =(const Polynomial& copy) { // 赋值运算符重载
    if (this == &copy)
        return *this;
    polyList = copy.polyList;
    return *this;
}

Polynomial& Polynomial::operator =(const LinkList<PolyItem>& copyLinkList) { // 赋值运算符重载
    polyList = copyLinkList;
    return *this;
}
Polynomial Polynomial::operator +(const Polynomial& p) const { // 加法运算符重载
    LinkList<PolyItem> la = polyList; // 当前多项式对应的线性表
    LinkList<PolyItem> lb = p.polyList; // 多项式 p 对应的线性表
    LinkList<PolyItem> lc; // 和多项式对应的线性表
    int aPos = 1, bPos = 1;
    PolyItem aNode, bNode;
    Status aStatus, bStatus;

    aStatus = la.GetElem(aPos++, aNode); // 取出 la 的第 1 项
    bStatus = lb.GetElem(bPos++, bNode); // 取出 lb 的第 1 项

    while (aStatus == ENTRY_FOUND && bStatus == ENTRY_FOUND) {
        if (aNode.expn > bNode.expn) { // la 中的项 aNode 指数较大
            lc.InsertElem(aNode); // 将 aNode 追加到 lc 的表尾
            aStatus = la.GetElem(aPos++, aNode); // 取出 la 的下一项
        }
        else if (aNode.expn < bNode.expn) { // lb 中的项 bNode 指数较大
            lc.InsertElem(bNode); // 将 bNode 追加到 lc 的表尾
```

```
            bStatus = lb.GetElem(bPos++, bNode); // 取出 lb 的下一项
        }
        else { // la 中的项 aNode 和 lb 中的项 bNode 指数相等
            PolyItem sumItem(aNode.coef + bNode.coef, aNode.expn);
            if (sumItem.coef != 0)
                lc.InsertElem(sumItem); // 将两项的和追加到 lc 的表尾
            aStatus = la.GetElem(aPos++, aNode); // 取出 la 的下一项
            bStatus = lb.GetElem(bPos++, bNode); // 取出 lb 的下一项
        }
    }

    while (aStatus == ENTRY_FOUND) { // 将 la 的剩余项追加到 lc 的后面
        lc.InsertElem(aNode); // 将 aNode 追加到 lc 的后面
        aStatus = la.GetElem(aPos++, aNode); // 取出 la 的下一项
    }

    while (bStatus == ENTRY_FOUND) { // 将 lb 的剩余项追加到 lc 的后面
        lc.InsertElem(bNode); // 将 bNode 追加到 lc 的后面
        bStatus = lb.GetElem(bPos++, bNode); // 取出 lb 的下一项
    }

    Polynomial fc; // 和多项式
    fc.polyList = lc;

    return fc;
}

Polynomial Polynomial::operator -(const Polynomial& p) const { //减法运算符重载
    LinkList<PolyItem> la = polyList, lb = p.polyList, lc;
    int aPos = 1, bPos = 1;
    PolyItem aItem, bItem;
    Status aStatus = la.GetElem(aPos++, aItem);
    Status bStatus = lb.GetElem(bPos++, bItem);

    while (aStatus == ENTRY_FOUND || bStatus == ENTRY_FOUND) {
        if (aStatus == ENTRY_FOUND && (bStatus != ENTRY_FOUND || aItem.expn > bItem.expn)) {
            lc.InsertElem(aItem);
            aStatus = la.GetElem(aPos++, aItem);
        }
        else if (bStatus == ENTRY_FOUND && (aStatus != ENTRY_FOUND || aItem.expn <
bItem.expn)) {
            PolyItem subItem(-bItem.coef, bItem.expn);
```

```
            lc.InsertElem(subItem);
            bStatus = lb.GetElem(bPos++, bItem);
        }
        else if (aItem.expn == bItem.expn) {
            PolyItem diffItem(aItem.coef - bItem.coef, aItem.expn);
            if (diffItem.coef != 0) lc.InsertElem(diffItem);
            aStatus = la.GetElem(aPos++, aItem);
            bStatus = lb.GetElem(bPos++, bItem);
        }
    }
    Polynomial fc;
    fc.polyList = lc;
    return fc;
}


Polynomial Polynomial::operator *(const Polynomial& p) const {
    LinkList<PolyItem> la = polyList;
    LinkList<PolyItem> lb = p.polyList;
    LinkList<PolyItem> lc;
    int aPos = 1, bPos = 1;
    PolyItem aItem, bItem;

    // 遍历第一个多项式
    while (la.GetElem(aPos++, aItem) == ENTRY_FOUND) {
        bPos = 1;
        // 遍历第二个多项式
        while (lb.GetElem(bPos++, bItem) == ENTRY_FOUND) {
            PolyItem productItem(aItem.coef * bItem.coef, aItem.expn + bItem.expn);
            int pos = 1;
            bool found = false;
            PolyItem cItem;
            // 查找是否已有相同指数的项
            while (lc.GetElem(pos, cItem) == ENTRY_FOUND) {
                if (cItem.expn == productItem.expn) {
                    // 如果找到相同指数的项，合并系数
                    cItem.coef += productItem.coef;
                    if (cItem.coef != 0) {
                        lc.InsertElem(cItem);  // 插入合并后的项
                    }
                    else {
                        lc.DeleteElem(pos, cItem);  // 系数为 0 时删除该项
```

```
                    }
                    found = true;
                    break;
                }
                pos++;
            }
            // 如果没有找到相同指数的项，则插入新的乘积项
            if (!found && productItem.coef != 0) {
                lc.InsertElem(productItem);
            }
        }
    }
    Polynomial fc;
    fc.polyList = lc;  // 返回结果
    return fc;
}

Polynomial Polynomial::derivative() { // 求导
    LinkList<PolyItem> la = polyList, lc;
    int pos = 1;
    PolyItem item;
    Status astatus = la.GetElem(pos++, item);
    while (astatus == ENTRY_FOUND) {
        PolyItem productItem(item.coef * item.expn, item.expn - 1);
        lc.InsertElem(productItem);
        astatus = la.GetElem(pos++, item);  // 取出 la 的下一项
    }
    Polynomial fc;
    fc.polyList = lc;
    return fc;
}

double Polynomial::evaluate(double x) const { // 多项式求值
    LinkList<PolyItem> la = polyList;
    double sum = 0;
    int pos = 1;
    PolyItem item;
    Status status = la.GetElem(pos++, item);
    while (status == ENTRY_FOUND) {
        sum += item.coef * pow(x, item.expn);
        status = polyList.GetElem(pos++, item);
    }
```

```
    return sum;
}

#endif
```

Main.cpp

```
#include "Assistance.h"   // 实用程序软件包
#include "Polynomial.h"   // 多项式类
int main(void)
{
    char c = '1';
    Polynomial A, B, fc;
    bool hasA = false, hasB = false;
    PolyItem it;

    while (c != '0') {
        cout << "1. 输入多项式 A" << endl;
        cout << "2. 输入多项式 B" << endl;
        cout << "3. 显示多项式 A 和 B" << endl;
        cout << "4. 计算 A + B" << endl;
        cout << "5. 计算 A - B" << endl;
        cout << "6. 计算 A * B" << endl;
        cout << "7. 计算 d(A) 和 d(B)" << endl;
        cout << "8. 综合运算: A + B + d(A) + d(B)" << endl;
        cout << "9. 计算综合运算在 x 的值" << endl;
        cout << "0. 退出" << endl;
        cout << endl << "选择功能(0~9):" << endl;
        cin >> c;
        switch (c) {
        case '1':
            cout << "输入多项式 A:" << endl;
            A.input();
            hasA = true;
            break;
        case '2':
            cout << "输入多项式 B:" << endl;
            B.input();
            hasB = true;
            break;
        case '3':
            if (hasA) {
                cout << "多项式 A: ";
```

```
                A.Display();
                cout << endl;
            }
            else {
                cout << "多项式 A: 未输入！" << endl;
            }
            if (hasB) {
                cout << "多项式 B: ";
                B.Display();
                cout << endl;
            }
            else {
                cout << "多项式 B: 未输入！" << endl;
            }
            break;
        case '4':
            if (hasA && hasB) {
                cout << "A + B = ";
                (A + B).Display();
                cout << endl;
            }
            else {
                cout << "请先输入两个多项式！" << endl;
            }
            break;
        case '5':
            if (hasA && hasB) {
                cout << "A - B = ";
                (A - B).Display();
                cout << endl;

            }
            else {
                cout << "请先输入两个多项式！" << endl;
            }
            break;
        case '6':
            if (hasA && hasB) {
                cout << "A * B = ";
                (A * B).Display();
                cout << endl;
```

```
            }
            else {
                cout << "请先输入两个多项式！" << endl;
            }
            break;
        case '7':
            if (hasA) {
                cout << "d(A) = ";
                A.derivative().Display();
                cout << endl;

            }
            else {
                cout << "多项式 A 未输入！" << endl;
            }
            if (hasB) {
                cout << "d(B) = ";
                B.derivative().Display();
                cout << endl;

            }
            else {
                cout << "多项式 B 未输入！" << endl;
            }
            break;
        case '8':
            if (hasA && hasB) {
                Polynomial result = A + B + A.derivative() + B.derivative();
                cout << "A + B + d(A) + d(B) = ";
                result.Display();
                cout << endl;

            }
            else {
                cout << "请先输入两个多项式！" << endl;
            }
            break;
        case '9':
            if (hasA && hasB) {
                double x;
                cout << "输入 x 的值: ";
                cin >> x;
```

```
                double result = (A + B + A.derivative() + B.derivative()).evaluate(x);
                cout << "综合运算在 x = " << x << " 时的值: " << fixed << setprecision(2) << result
<< endl;
            }
            else {
                cout << "请先输入两个多项式！" << endl;
            }
            break;
        }
    }
    system("PAUSE");                    // 调用库函数 system()
    return 0;                           // 返回值 0，返回操作系统
}
```

主要采用了面向对象的编程思想，将一元 n 次多项式的求导、四则运算和求值等操作的各个功能封装成了不同的类和结构体，并通过这些类和结构体进行数据的存储和操作。下面对其算法思想进行详细展开叙述。

（1）头文件 Assistance.h 辅助软件包

该头文件实现了一个基础的辅助软件包，其中包含了一些常见的功能，主要涉及输入输出、数组操作、错误处理和数据类型的操作等。通过合理地使用 C++ 的标准库和一些自定义的函数与类，代码实现了若干实用的功能，可以方便地为其他项目提供支持。以下是对这段代码的总结，围绕功能的实现展开。

总体而言，该头文件实现了多个实用的功能模块，涵盖了输入输出、数组操作、交换、异常处理等方面的内容。这些功能模块被封装成了简单易用的函数，能够在实际开发中提高代码的复用性和可维护性。通过合理的命名和良好的结构设计，代码具有很高的可读性和扩展性，能够适应不同类型的需求。在实际应用中，可以方便地将这些辅助函数集成到更复杂的程序中，从而提高开发效率。

(2) 头文件 Node.h 节点类

该头文件定义了一个结点类 Node，用于实现链表结构中的基本单元。它利用模板（template）技术，使得该结点类能够处理不同类型的数据。代码提供了一个简单的结点结构，该结构包含数据域和指针域，并且提供了两个构造函数，用于灵活地创建结点。以下是对这段代码的功能描述。

该头文件通过定义一个模板类 Node，实现了链表结点的基本结构。该结点类不仅包含数据域和指针域，还提供了两个构造函数，分别用于创建空结点和指定数据及指针的结点。由于 Node 是模板类，它能够支持多种不同类型的数据，从而可以用于实现各种链表结构。该代码为链表的构建提供了一个基础框架，后续可以在此基础上进一步实现链表的插入、删除、查找等操作，构建更加复杂的数据结构。

(3) 头文件 LinkList.h 单链表类

该头文件实现了一个单链表类 LinkList，它支持基本的链表操作，包括插入、删除、查找和修改等功能。该类采用模板技术，支持不同数据类型的单链表操作。

该头文件实现充分利用了链表的灵活性，支持在任意位置插入和删除元素。通过模板化设计，链表可以处理任意类型的数据，具有较好的通用性。对于开发中需要频繁进行插入、删除操作的数据结构应用，这种实现方式提供了高效且易于扩展的解决方案。

(4) 头文件 PolyItem.h

该头文件定义了一个名为 PolyItem 的结构体，表示一个多项式的单项式项。它包含了多项式项的系数和指数，并且提供了两个构造函数以便创建该结构体的实例。

该代码段实现了一个表示多项式项的结构体，能够存储每个项的系数和指数，提供了两种方式来构造这些项：一种是创建一个空的无效项（指数为 -1），另一种是通过指定系数和指数来创建一个有效的多项式项。

(5) 头文件 Polynomial.h 多项式类

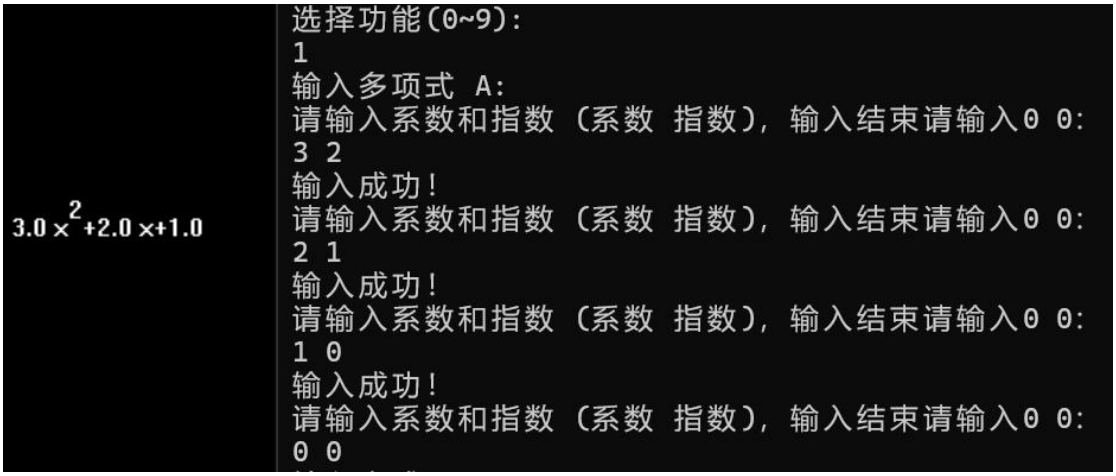
这段代码实现了一个多项式类 Polynomial 的一些常用操作，包括加法、减法、乘法、求导和求值。这些操作通过重载运算符和成员函数的方式进行。代码中的多项式使用 LinkList<PolyItem> 来表示，其中 PolyItem 结构体存储了多项式项的系数和指数。下面是对代码功能的详细总结。

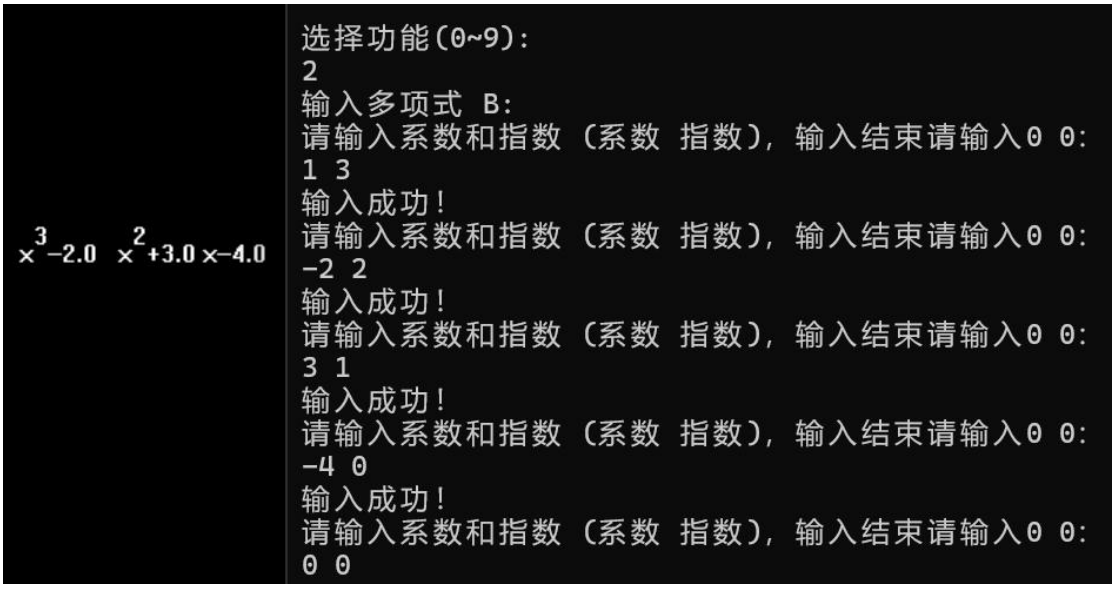
这些运算符重载和成员函数有效地实现了多项式的基本运算：加法、减法、乘法、求导和求值。使用 LinkList<PolyItem> 作为多项式项的存储结构，使得这些操作可以灵活地处理多项式的不同项（按指数排序）。这些功能对多项式的处理提供了极大的便利，能够支持多项式的常见运算，且考虑到了系数合并、零项删除等细节。总之，这段代码为多项式类提供了完备的功能，适合进行多项式的各种数学运算。

3、实验结果

(1)多项式输入与显示

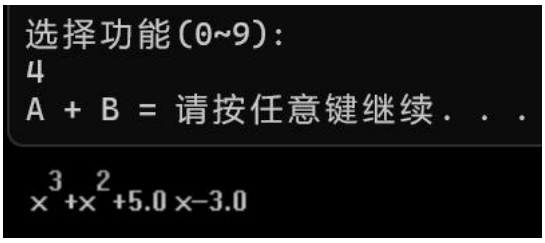
分别输入“1”和“2”，输入多项式 A 和 B。例如，输入多项式 A 为 “3x^2 + 2x + 1”（3，2；2，1；1，0），输入多项式 B 为 “x^3 - 2x^2 + 3x - 4”（1，3；-2，2；3，1；-4，0）。输入完成后，通过选择功能 “3”. 显示多项式 A 和 B”，在图形界面中正确显示出多项式 A 和 B 的表达式，与输入一致。如下图所示：



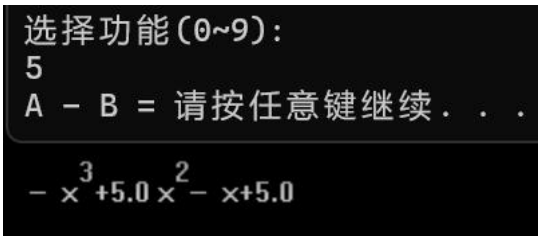


(2)多项式四则运算结果

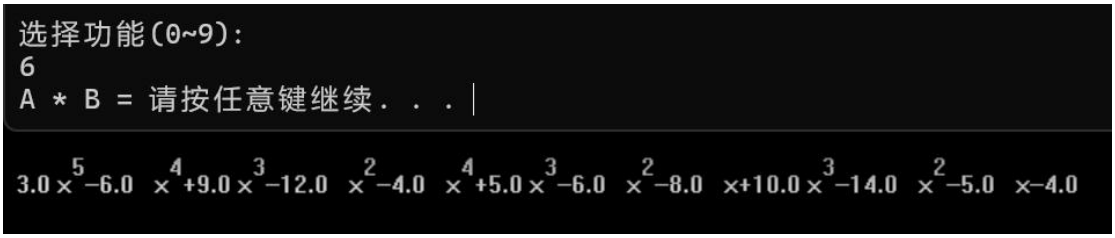
加法（A + B）：计算结果为 “x^3 + x^2 + 5x - 3”。图形界面显示多项式各项，符合预期。



减法（A - B）：结果为 “-x^3 + 5x^2 - x + 5”。图形界面显示多项式各项，符合预期。



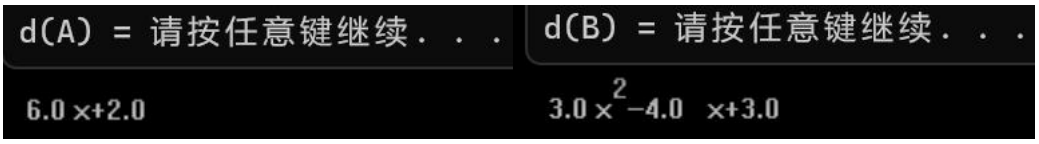
乘法（A * B）：得到 “3x^5 - 4x^4 + 5x^3 - 11x^2 + 2x - 4”。图形界面显示多项式各项，符合预期。



(3)多项式求导结果

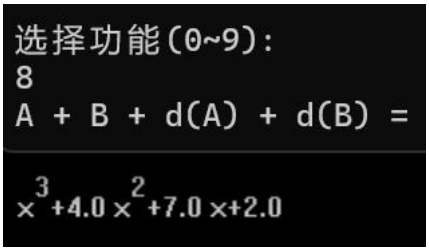
对于多项式 A，求导后为 “6x + 2”。图形界面显示求导后的多项式。

多项式 B 求导结果为 “3x^2 - 4x + 3”。图形界面中求导结果的表达式正确呈现。

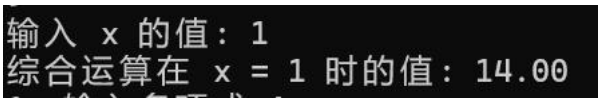


(4)综合运算及求值结果

综合运算（A + B + d (A) + d (B)）：计算结果为 “x^3 + 3x^2 + 11x - 1”。图形界面显示，该综合运算后的多项式准确无误。



当输入 x = 1 时，综合运算结果的值为 “1 + 3 + 11 - 1 = 14”。程序正确计算并输出该值，保留两位小数显示为 “14.00”。



输入 x = 3 时，综合运算结果的值为 “27 + 27 + 33 - 1 = 86”。程序输出 “86.00”，与计算结果一致。

4、算法分析

（1）时间复杂度

输入多项式：在 input 函数中，由于需要逐个输入多项式的每一项，并且可能存在输入错误时的处理（如重新输入、忽略额外输入等），在最坏情况下，假设输入的多项式项数为 n，每次输入检查和处理的时间复杂度为常数级，所以总的时间复杂度为 $O(n)$。

多项式加法、减法、乘法：对于加法和减法，在 operator +和 operator -函数中，需要遍历两个多项式的每一项，比较指数大小并进行合并或插入操作。假设两个多项式的项数分别为 m 和 n，在最坏情况下，需要遍历 $m+n$ 次，所以时间复杂度为 $O(m+n)$。对于乘法，在 operator *函数中，需要遍历两个多项式的每一项进行相乘，并在结果多项式中查找或插入新项，由于乘法结果可能产生最多 $m*n$ 项，所以时间复杂度为 $O(mn)$。

求导：在 derivative 函数中，只需遍历多项式的每一项，计算求导后的系数和指数，假设多项式项数为 n，时间复杂度为 $O(n)$。

求值：在 evaluate 函数中，遍历多项式的每一项计算的幂次并乘以系数后累加，同样假设多项式项数为 n，计算的幂次的时间复杂度为 $O(\log n$（使用快速幂算法），所以总的时间复杂度为 $O(n \log n)$。

（2）空间复杂度

程序中使用了多个类和结构体来表示多项式及其相关操作，其中主要的空间占用来自于多项式类 Polynomial 中 LinkList<PolyItem> 用于存储多项式的各项。在最坏情况下，假设多项式的项数为 n，则空间复杂度为 O(n)，用于存储多项式的各项系数和指数。其他辅助类和函数的空间占用相对较小，可忽略不计。

5、总结与心得

（1）知识巩固与提升

通过本次实验，深入理解了线性表的相关知识，尤其是链表在多项式表示和运算中的应用。掌握了如何使用链表存储多项式的各项，以及如何通过链表操作实现多项式的各种数学运算，包括加法、减法、乘法、求导和求值等。对 C++ 的面向对象编程思想有了更深刻的体会，学会了如何将复杂的问题分解为多个类和对象，提高代码的可维护性和可扩展性。

（2）编程技能提升

在代码实现过程中，提高了处理复杂逻辑和算法的能力。例如，在实现多项式乘法时，需要考虑如何高效地合并同类项，避免重复计算，这锻炼了自己的算法设计和优化能力。学会了使用 C++ 的模板技术，使代码能够处理不同类型的数据，增强了代码的通用性。同时，在处理输入输出和错误处理方面也积累了经验，如在输入多项式时对各种错误情况的判断和提示，使程序更加健壮。

（3）团队协作与问题解决

在实验过程中，小组成员分工明确，共同协作完成了实验任务。在遇到问题时，通过团队成员之间的讨论和交流，能够快速定位问题并找到解决方案。

（4）不足与改进方向

虽然完成了实验要求的功能，但代码的效率还有提升空间。在多项式乘法运算中，时间复杂度较高，可以进一步研究更高效的算法，如使用快速傅里叶变换（FFT）来优化乘法运算。代码的结构可以进一步优化，使各个类和函数的职责更加单一，提高代码的可读性。在异常处理方面，虽然已经考虑了一些常见的输入错误，但还可以增加更多的异常情况处理，使程序更加稳定和可靠。未来将继续学习数据结构和算法的知识，不断提高编程能力，以应对更复杂的编程任务。