# Project 2 二次元n次多项式的运算

**项目分工：**

陈谋祥：代码编写

孙文欣：代码注释

张戈：文档编写

**GitHub地址：** [**https://github.com/Keytoyze/Polynomial-Calculator**](https://github.com/Keytoyze/Polynomial-Calculator)

1. **详细设计**

这个project主要由三个部分构成：①多项式的输入②多项式的运算③按照次幂顺序的多项式输出。于是我们就将项目分成一个主体程序和四个头文件分别解决小问题。

首先是主体程序Main.c，在主体程序中首先编写了输入命令的代码，提示使用者如何使用这个程序，然后需要的几个子操作的函数的编写，输入和输出等。但是函数的具体内容并不在主体程序中，这样使得主体程序更加简洁。

主体程序Main.c：

1. // 主体程序
2. #include "Header.h"
3. #include "Polynomial.h"
4. #include "IO.h"
5. #include "Calculate.h"
7. **void** printMenu();
8. **void** Input();
9. **void** Add();
10. **void** Sub();
11. **void** Mul();
12. **void** Output();
13. **void** printInputHint();
14. **void** wait();
16. **int** N = 0;
17. List lists[1024];
19. **int** main(**void**)
20. {
22. **char** str[60];
23. **int** op, i;
24. **do**
25. {
26. printf("==============================\n       二元多项式计算器\n==============================\n\n");
27. printf(" \* 当前已储存的多项式数量：%d\n\n", N);
28. printMenu();
29. printf("\n请输入所要进行操作的序号：\n");
30. printInputHint();
31. scanf("%s", str);
32. op = str[0] - '0';
33. **switch** (op)
34. {
35. **case** 1:
36. Input(); **break**;
37. **case** 2:
38. Add(); **break**;
39. **case** 3:
40. Sub(); **break**;
41. **case** 4:
42. Mul();**break**;
43. **case** 5:
44. Output(); **break**;
45. **case** 0:
46. **for** (i = 0; i < N; i++)
47. {
48. freeAll(lists[i]);
49. }
50. exit(0);
51. **default**:
52. printf("输入错误。");
53. **break**;
54. }
55. wait();
56. system("cls");
57. } **while** (TRUE);
58. }
60. //输出操作界面主菜单\*
61. **void** printMenu()
62. {
63. printf("    [1] 多项式输入\n");
64. printf("    [2] 多项式加法\n");
65. printf("    [3] 多项式减法\n");
66. printf("    [4] 多项式乘法\n");
67. printf("    [5] 输出所有已储存的多项式\n");
68. printf("    [0] 退出\n");
69. }
71. //操作[1] 多项式的输入
72. **void** Input()
73. {
74. printf("请输入一个多项式，每一项格式为(+/-)a\*x^p\*y^q。其中a, ^p, ^q, \*, x, y均可省略。\n");
75. printInputHint();
76. **char** m[60];
77. scanf("%s", m);
78. List l = newPolynomial();
79. processPolynomial(&l, m);
80. lists[N] = l;
81. printf("已成功输入多项式！该多项式编号为%d。\n", N);
82. printf(" \* P[%d] = %s\n", N, printResult(lists[N]));
83. N++;
84. }
86. //操作[2] 多项式加法 让两个多项式相加但可以自己选择是哪两个多项式
87. **void** Add()
88. {
89. **int** a, b;
90. printf("请输入要相加的两个多项式编号。两个编号之间用空格隔开。\n");
91. printInputHint();
92. List l = newPolynomial();
93. scanf("%d%d", &a, &b);
94. **if** (a >= N || a < 0 || b >= N || b < 0)
95. {
96. printf("无效输入。\n");
97. **return**;
98. }
99. l = add(l, lists[a]);
100. l = add(l, lists[b]);
101. lists[N] = l;
102. printf("操作成功！相加得到的多项式编号为%d。\n", N);
103. printf(" \* P[%d] = %s\n", N, printResult(lists[N]));
104. N++;
106. }
108. //操作[3] 多项式减法 前一个减后一个
109. **void** Sub()
110. {
111. **int** a, b;
112. printf("请输入要相减的两个多项式编号。两个编号之间用空格隔开。\n");
113. printInputHint();
114. List l = newPolynomial();
115. scanf("%d%d", &a, &b);
116. **if** (a >= N || a < 0 || b >= N || b < 0)
117. {
118. printf("无效输入。\n");
119. **return**;
120. }
121. l = add(l, lists[a]);
122. l = sub(l, lists[b]);
123. lists[N] = l;
124. printf("操作成功！相减得到的多项式编号为%d。\n", N);
125. printf(" \* P[%d] = %s\n", N, printResult(lists[N]));
126. N++;
127. }
129. //输出已储存的多项式
130. **void** Output()
131. {
132. **int** a, i;
133. **if** (N == 0)
134. {
135. printf("暂未输入。\n");
136. **return**;
137. }
138. **for** (i = 0; i < N; i++)
139. {
140. printf(" \* P[%d] = %s\n", i, printResult(lists[i]));
141. }
142. }
144. **void** printInputHint()
145. {
146. printf(">>>> ");
147. }
149. //每一次操作结束的提示
150. **void** wait()
151. {
152. printf("\n");
153. system("pause");
154. }
156. //操作[4] 多项式乘法 将选定的两个多项式相乘
157. **void** Mul()
158. {
159. **int** a, b;
160. printf("请输入要相乘的两个多项式编号。两个编号之间用空格隔开。\n");
161. printInputHint();
162. List l = newPolynomial();
163. scanf("%d%d", &a, &b);
164. **if** (a >= N || a < 0 || b >= N || b < 0)
165. {
166. printf("无效输入。\n");
167. **return**;
168. }
169. l = add(l, lists[a]);
170. l = multiply(l, lists[b]);
171. lists[N] = l;
172. printf("操作成功！相乘得到的多项式编号为%d。\n", N);
173. printf(" \* P[%d] = %s\n", N, printResult(lists[N]));
174. N++;
175. }

然后是第一个头文件Header.c。我们将一些函数的定义和结构的定义都放在这个头文件中，可以很清楚的知道每个函数是干什么的。

头文件Header.h：

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
5. #define TRUE 1
6. #define FALSE 0
8. /\* For Polynomial.h \*/
10. #define SIZE\_POLYNOMIAL (sizeof(Polynomial))
11. #define SIZE\_MONOMIAL (sizeof(Monomial))
12. #define SIZE\_RESULT (sizeof(IteratorResult))
14. **typedef** **struct** Monomial//定义多项式中的每个单项式结构  用于表达具体单项式节点
15. {
16. **double** data; //储存数据
17. **int** x\_degree; //储存x次数
18. **struct** Monomial\* next;
19. } Monomial;
21. **typedef** **struct** Polynomial//定义多项式结构 用于表达相同幂指数和单项式组成的多项式
22. {
23. **int** totalDegree; //储存x和y的总次数
24. Monomial\* header; //单项式链表的首地址
25. **struct** Polynomial\* next;
26. } Polynomial;
28. **typedef** **struct** IteratorResult //迭代器结构 用于将链表中的内容表达成可以单独分离的模块
29. {
30. /\*private\*/ Monomial\* monomial;
31. /\*private\*/ Polynomial\* polynomial;
32. **int** x\_degree;
33. **int** y\_degree;
34. **double** data;
35. } IteratorResult;
37. **typedef** Polynomial\* List;
39. Monomial\* findLocation(List\* header, **int** x\_degree, **int** y\_degree);//寻找header多项式中含有的单项式xy幂指数和皆为x\_degree+y\_degree的多项式节点
40. **double** getData(List header, **int** x\_degree, **int** y\_degree); //获得header多项式中x指数为x\_degree,y指数为y\_degree的项的系数
41. List setData(List header, **double** data, **int** x\_degree, **int** y\_degree);//将系数为data，x指数为x\_degree,y指数为y\_degree的项存入多项式header中
42. Monomial\* findLocationInMonomial(**int** x\_degree, Polynomial\* father);//在已经找到的多项式节点中寻找 x指数为x\_degree的单项式节点
43. Polynomial\* createPolynomial(**int** totalDegree);//创建totaldegree为totaldegree的多项式节点
44. Polynomial\* insertPolynomial(Polynomial\* last, Polynomial\* next, **int** totalDegree);//在last节点和next节点中插入totaldegree为totaldegree的多项式节点
45. Monomial\* insertMonomial(Monomial\* last, Monomial\* next, **int** x\_degree, **double** data);//在last节点和next节点中插入x指数为x\_degree,y指数为y\_degree的单项式节点
46. Monomial\* createMonomial(**int** x\_degree, **double** data);//创建系数为data的 x指数为x\_degree的单项式结构
47. IteratorResult\* createEmptyIteratorResult();//创建空的迭代结构
48. **int** iteratorResult(**int** init, List header, IteratorResult\* last);//从last迭代单元开始对多项式header进行继续迭代
49. **void** freeAll(List header);//释放以header为头部的多项式
50. **void** freeMonomial(Monomial \*p);//释放p指向的单项式节点
51. **void** freePolynomial(Polynomial \*p);//释放p指向的多项式节点
52. List newPolynomial();//创建新的多项式单元
54. /\* For Calculate.h \*/
56. List add(List L1, List L2);//将多项式L1与多项式L2相加
57. List sub(List L1, List L2);//将多项式L1与多项式L2相减
58. List multiply(List L1, List L2);//将多项式L1与多项式L2相乘
59. List copy(List L);//复制多项式L
61. /\* For IO.h \*/
63. **char**\* processMonomial(List\* L, **char**\* str);//将str数组中的字符转化后存入单项式L中
64. **int** isNumber(**char** c);//判断c是不是数字
65. **int** getNum(**char**\* p, **double**\* result);//将P字符串转化为数字result，并且获得result的长度
66. **char**\* deleteEnd0(**double** num);//删除小数点后多余的0
67. **char**\* printResult(List header);//输出多项式header
68. **void** processPolynomial(List\*L, **char**\* str);//将str数组中的字符转化后存入多项式L中

然后就是三个起主要作用的头文件。

Polynomial.h作用是操作得到的多项式，即进行操作后怎样得到新的多项式，多项式的储存是通过函数使用链表来进行的。

头文件Polynomial.h：

1. /\*\*
2. \* 创建一个空的多项式
3. \* @return 多项式
4. \*/
5. List newPolynomial()
6. {
7. List L1 = createPolynomial(0);
8. **return** setData(L1, 0, 0, 0);
9. }
11. /\*\*
12. \* 取出多项式中某一项的系数。
13. \* 如果这一项对应的链表节点还没有创建，则自动创建，并且返回系数0。
14. \* @param x\_degree 这一项的x次数
15. \* @param y\_degree 这一项的y次数
16. \* @return 这一项的系数
17. \*/
18. **double** getData(List header, **int** x\_degree, **int** y\_degree)
19. {
20. **return** findLocation(&header, x\_degree, y\_degree) -> data;
21. }
23. /\*\*
24. \* 设置多项式中某一项的数据。
25. \* 如果这一项对应的链表节点还没有创建，则自动创建。
26. \* @param data 这一项的系数
27. \* @param x\_degree 这一项的x次数
28. \* @param y\_degree 这一项的y次数
29. \* @return 得到的新多项式
30. \*/
31. List setData(List header, **double** data, **int** x\_degree, **int** y\_degree)
32. {
33. List header1 = header;
34. findLocation(&header1, x\_degree, y\_degree) -> data = data;
35. **return** header1;
36. }
38. /\*\*
39. \* 创建一个空的迭代结果
40. \* @return 新的迭代结果
41. \*/
42. IteratorResult\* createEmptyIteratorResult()
43. {
44. **return** (IteratorResult\*) malloc(SIZE\_RESULT);
45. }
47. /\*\*
48. \* 迭代器，用于遍历整个多项式。
49. \* 第一次使用时，init参数设置为TRUE，迭代器将从第一项开始遍历。
50. \* 之后使用时，init参数设置为FALSE，迭代器将从last以下调用
51. \* @param init 如果是TRUE则从第一项开始遍历，否则从上次返回的那一项之后开始遍历。
52. \* @param header 多项式
53. \* @param last 上一次迭代的结果
54. \* @return 迭代到底或者参数错误返回假，否则返回真
55. \*/
56. **int** iteratorResult(**int** init, List header, IteratorResult\* last)
57. {
58. Polynomial\* p1;//p1指向多项式节点
59. Monomial\* p2;//p2指向具体单项式节点
61. **if** (init == TRUE)
62. {
63. p1 = header;
64. p2 = header -> header;
65. } **else** {
66. **if** (last == NULL)
67. {
68. **return** FALSE;
69. }
70. p1 = last -> polynomial;
71. p2 = last -> monomial;
72. p2 = p2 -> next;
73. }
75. **if** (p2 == NULL)
76. {
77. p1 = p1 -> next;
78. **if** (p1 == NULL)
79. {
80. free(last);
81. **return** FALSE;
82. }
83. p2 = p1 -> header;
84. }
86. **if** (last == NULL)
87. {
88. **return** FALSE;
89. }
91. last -> polynomial = p1;//定位单项式所处的多项式节点
92. last -> monomial = p2;//定位该单项式所处的具体位置
93. last -> x\_degree = p2 -> x\_degree;
94. last -> y\_degree = p1 -> totalDegree - p2 -> x\_degree;
95. last -> data = p2 -> data;
97. **if** (p2 -> data == 0)
98. {
99. **return** iteratorResult(FALSE, header, last);
100. }
102. **return** TRUE;
104. }
106. /\*\*
107. \* 用递归的方式释放链表所占用的内存空间。结束程序前调用。
108. \*/
109. **void** freeAll(List header)
110. {
111. freePolynomial(header);
112. }
114. /\*private\*/ **void** freePolynomial(Polynomial \*p)
115. {
116. **if** (p == NULL)
117. {
118. **return**;
119. }
120. freeMonomial(p -> header);
121. freePolynomial(p -> next);
122. free(p);
123. }
125. /\*private\*/ **void** freeMonomial(Monomial \*p)
126. {
127. **if** (p == NULL)
128. {
129. **return**;
130. }
131. freeMonomial(p -> next);
132. free(p);
133. }
135. /\*private\*/ Monomial\* findLocation(List\* headerAddress, **int** x\_degree, **int** y\_degree)
136. {
137. List header = \*headerAddress;
138. **if** (header == NULL)
139. {
140. **return** NULL;
141. }
143. Polynomial\* p = header;//用于遍历现有多项式节点
144. Polynomial\* p2 = NULL;//存储历史节点位置
146. **while** (TRUE) {
148. **if** (p -> totalDegree == x\_degree + y\_degree) //在多项式中寻找相同幂指数和的节点
149. {
150. **if** (p -> header == NULL)
151. {
152. **return** p -> header = createMonomial(x\_degree, 0);//如果存在该多项式节点却没有内容则创建具体单项式节点
153. }
154. **return** findLocationInMonomial(x\_degree, p);//在原有多项式节点指向的单项式节点集合中寻找插入位置，返回找到的位置
155. }
157. **if** (p -> totalDegree < (x\_degree + y\_degree)) {//由于现有链表已经按照降序排列，一旦检测到幂指数和小于输入幂指数和的节点，则代表不存在幂指数和与输入幂指数和相同的节点，此时新建插入
158. **if** (p2 == NULL)//如果输入项的幂指数是现有幂指数和中最大的时，p2则未发生移动
159. {
160. header = insertPolynomial(NULL, p, x\_degree + y\_degree)；//用插入函数开辟一个多项式节点，同时该节点成为整个多项式的头部
161. \*headerAddress = header;
162. header -> header = createMonomial(x\_degree, 0);//在该多项式节点中链接具体的单项式节点
163. **return** header -> header;//返回该单项式位置，找到位置
164. }
165. p = insertPolynomial(p2, p, x\_degree + y\_degree);//若不是最大幂指数和的项，则向p2和p之间插入信的多项式节点
166. p -> header = createMonomial(x\_degree, 0);//同时也在该多项式节点中填充具体单项式内容
167. **return** p -> header; //返回单项式位置
168. }
170. p2 = p;//存储上一个节点位置
171. **if** (p -> next == NULL)//若p指针指向的节点是尾部，则创建新的末尾，与创建新的头部相似
172. {
173. p = insertPolynomial(p, NULL, x\_degree + y\_degree);
174. p -> header = createMonomial(x\_degree, 0);
175. **return** p -> header;
176. }
177. p = p -> next;//移动p指针
179. }
180. }
182. /\*private\*/ Monomial\* findLocationInMonomial(**int** x\_degree, Polynomial\* father)
183. {
184. Monomial\* header = father -> header;
185. Monomial\* p = header;//遍历相同幂指数和单项式节点
186. Monomial\* p2 = NULL;//存储历史节点位置
187. **while** (TRUE) {
188. **if** (p -> x\_degree == x\_degree)//发现有x的幂指数相同的项，由于在同一个多项式节点中，代表xy幂指数和相等，所以y的幂指数相同，找到同类项
189. {
190. **return** p;//找到位置，返回位置
191. }
192. **if** (p -> x\_degree < x\_degree)//具体单项式的排列是按照x降幂排列的，若发现有x的幂指数小于输入幂指数的项，则代表后续节点中不存在输入项的同类项
193. {
194. **if** (p2 == NULL)//若该项是x幂指数最大的项
195. {
196. **return** father -> header = insertMonomial(NULL, p, x\_degree, 0);//创建新头部
197. }
198. **return** insertMonomial(p2, p, x\_degree, 0);//找到的位置在上一个单项式与刚好小于输入单项式的单项式节点之间
199. }
200. p2 = p;//存储上一个节点位置
201. **if** (p -> next == NULL)
202. {
203. **return** insertMonomial(p, NULL, x\_degree, 0);//创建新尾部
204. }
205. p = p -> next;//若上面条件届不符合则继续遍历
206. }
207. }
209. /\*private\*/ Polynomial\* createPolynomial(**int** totalDegree)//创建一个多项式节点空间
210. {
211. Polynomial\* p = (Polynomial\*)malloc(SIZE\_POLYNOMIAL);
212. p -> totalDegree = totalDegree;
213. p -> next = NULL;
214. p -> header = NULL;
215. **return** p;
216. }
218. /\*private\*/ Polynomial\* insertPolynomial(Polynomial\* last, Polynomial\* next, **int** totalDegree)//在last与next之间插入一个新的多项式节点
219. {
220. Polynomial\* p = createPolynomial(totalDegree);
221. **if** (next == NULL)
222. {
223. **return** last -> next = p;
224. }
225. **if** (last == NULL)
226. {
227. p -> next = next;
228. **return** p;
229. }
230. p -> next = last -> next;
231. last -> next = p;
232. **return** p;
233. }
235. /\*private\*/ Monomial\* insertMonomial(Monomial\* last, Monomial\* next, **int** x\_degree, **double** data)//在last和next之间插入一个单项式节点
236. {
237. Monomial\* p = createMonomial(x\_degree, data);
238. **if** (next == NULL)
239. {
240. **return** last -> next = p;
241. }
242. **if** (last == NULL)
243. {
244. p -> next = next;
245. **return** p;
246. }
247. p -> next = last -> next;
248. last -> next = p;
249. **return** p;
250. }
252. /\*private\*/ Monomial\* createMonomial(**int** x\_degree, **double** data)//创建单项式节点空间
253. {
254. Monomial\* p = (Monomial\*)malloc(SIZE\_MONOMIAL);
255. p -> x\_degree = x\_degree;
256. p -> data = data;
257. p -> next = NULL;
258. **return** p;
259. }

IO.h的作用是处理多项式的输入与输出，其中我们将多项式分解为多个单项式来处理，每个单项式就是多项式链表的一个节点。

头文件IO.h：

1. **void** processPolynomial(List\* L, **char**\* str)//数组中的字符串输入
2. {
3. **int** i, j = 0;
4. **char** monomial[1024] = "";
5. **char**\* p = str;
6. List temp;
7. **for** (i = 0; str[i]; i++)
8. {
9. monomial[j++] = str[i];
10. **if** ((str[i] == '+' || str[i] == '-') && i != 0)//当有有效输入并且单项式输入完成时
11. {
12. monomial[j-1] = '\0';//完结上一个单项式
13. temp = newPolynomial();//新建一个多项式空间
14. processMonomial(&temp, monomial);//向新开辟的多项式空间里存储单项式
15. \*L = add(\*L, temp);//将生成的多项式与现有以存储多项式相加
16. freeAll(temp);//释放临时地址
17. monomial[0] = str[i]; //创建下一个单项式
18. j = 1;//从第二项开始，monomial【0】用于存储加减符号
19. }
20. }
21. **if** (j >= 1) {//完结最后一个单项式
22. monomial[j] = '\0';
23. temp = newPolynomial();
24. processMonomial(&temp, monomial);
25. \*L = add(\*L, temp);
26. freeAll(temp);
27. }
28. /\*
29. i = 0;
30. List temp;
31. while (\*p)
32. {
33. temp = newPolynomial();
34. p = processMonomial(&temp, p);
35. \*L = add(\*L, temp);
36. freeAll(temp);
37. }\*/
38. }
40. **char**\* processMonomial(List\* L, **char**\* str)//得到一个单项式
41. {
42. **char**\* p = str;
43. **char** tmp[60];
44. **char** c;
45. **int** n = -1, len;
46. **double** data, x = 0, y = 0;
48. **if** (str[0] == '-' && (str[1] == 'x' || str[1] == 'y')) //-x or -y
49. {
50. strcpy(tmp, "-1");
51. strcat(tmp, str + 1);
52. p = tmp;
53. }
55. **if** (str[0] == '+' && (str[1] == 'x' || str[1] == 'y')) // +x or +y
56. {
57. p++;
58. }
60. len = getNum(p, &data);//得到系数位数，与系数的值data
61. **if** (len == 0) //没有输入系数
62. {
63. data = 1;
64. }
65. p += len;
67. **if** (\*p == '\*')
68. {
69. p++;
70. }
72. **if** (\*p == 'x') // 输入了 x
73. {
74. p++;
75. **if** (\*p == '^')
76. {
77. p++;
78. p += getNum(p, &x);//得到指数占位，与指数的值x
79. } **else** {
80. x = 1;
81. }
82. }
84. **if** (\*p == '\*')
85. {
86. p++;
87. }
89. **if** (\*p == 'y') // 输入了 y
90. {
91. p++;
92. **if** (\*p == '^')
93. {
94. p++;
95. p += getNum(p, &y);//得到指数占位，与指数的值y
96. } **else** {
97. y = 1;
98. }
99. }
101. \*L = setData(\*L, data, (**int**)(x+0.5), (**int**)(y+0.5));//四舍五入
102. **return** p;
103. }
105. **int** isNumber(**char** c)//判断是否是数字
106. {
107. **return** c >= '0' && c <= '9';
108. }
110. **int** getNum(**char**\* p, **double**\* result)//返回系数占位的同时，改变result的值
111. {
112. **char** c;
113. **int** n = -1;
114. **double** re = 0;
115. **int** integer = TRUE;
116. **int** bit = 0;
117. **int** flag = 1;
118. **int** i;
119. **while** (++n, (c = \*(p++)))
120. {
121. **if** (n == 0)
122. {
123. **if** (c == '-')
124. {
125. flag = -1;
126. **continue**;
127. } **else** **if** (c == '+')
128. {
129. **continue**;
130. }
131. }
133. **if** (isNumber(c))
134. {
135. re = re \* 10 + c - '0';
136. **if** (!integer)//遇到小数 时记录小数位数
137. {
138. bit++;
139. }
140. } **else** **if** (c == '.')
141. {
142. integer = FALSE;
143. } **else** {
144. **break**;
145. }
146. }
147. **for** (i = 0; i < bit; i++)//将小数点左移从而恢复原值
148. {
149. re = re / 10;
150. }
151. \*result = flag \* re;//按照加减对数字进行处理
152. **return** n;
153. }
155. **char**\* deleteEnd0(**double** num) //去掉num小数点后的0
156. {
157. **static** **char** re[60];
158. re[0] = '\0';
159. **int** i, integer = TRUE;
160. **if** ((**int**)num == num)
161. {
162. sprintf(re, "%d", (**int**)num);
163. **return** re;
164. }
165. sprintf(re, "%lf", num);
166. **for** (i = 0; re[i]; i++)
167. {
168. **if** (re[i] == '.')
169. {
170. integer = FALSE;
171. } **else** **if** (re[i] == '0' && !integer)
172. {
173. re[i] = '\0';
174. **break**;
175. }
176. }
177. **return** re;
178. }
180. **char**\* printResult(List l)//输出已储存的多项式
181. {
182. IteratorResult\* p = createEmptyIteratorResult();
183. iteratorResult(TRUE, l, p);//打开新的迭代器
184. **int** boolean = TRUE;
185. **char** m[60], n[60];
186. **static** **char** re[1024];
187. re[0] = '\0';
188. **int** flag = 0;
189. **while**(boolean)
190. {
191. m[0] = '\0';
192. n[0] = '\0';
193. **if** (p -> data != 1)
194. {
195. **if** (p -> data == -1)
196. {
197. strcat(m, "-");
198. } **else** {
199. strcat(m, deleteEnd0(p -> data));
200. }
201. }
203. n[0] = '\0';
204. **if** (p -> x\_degree != 0)
205. {
206. **if** (p -> x\_degree != 1)
207. {
208. sprintf(n, "x^%d", p -> x\_degree);
209. } **else** {
210. strcpy(n, "x");
211. }
212. strcat(m, n);
213. }
215. n[0] = '\0';
216. **if** (p -> y\_degree != 0)
217. {
218. **if** (p -> x\_degree != 0)
219. {
220. strcat(m, "\*");
221. }
222. **if** (p -> y\_degree != 1)
223. {
224. sprintf(n, "y^%d", p -> y\_degree);
225. } **else** {
226. strcpy(n, "y");
227. }
228. strcat(m, n);
229. }
231. **if** (p -> x\_degree == 0 && p -> y\_degree == 0 && (p -> data == 1 || p -> data == -1))
232. {
233. strcat(m, "1");
234. }
235. **if** (flag && m[0] != '-')
236. {
237. strcat(re, "+");
238. }
239. strcat(re, m);
240. flag = 1;
241. boolean = iteratorResult(FALSE, l, p);
242. }
243. **return** re;
244. }

最后一个Calculate.h就是对多项式进行加减乘操作的函数。

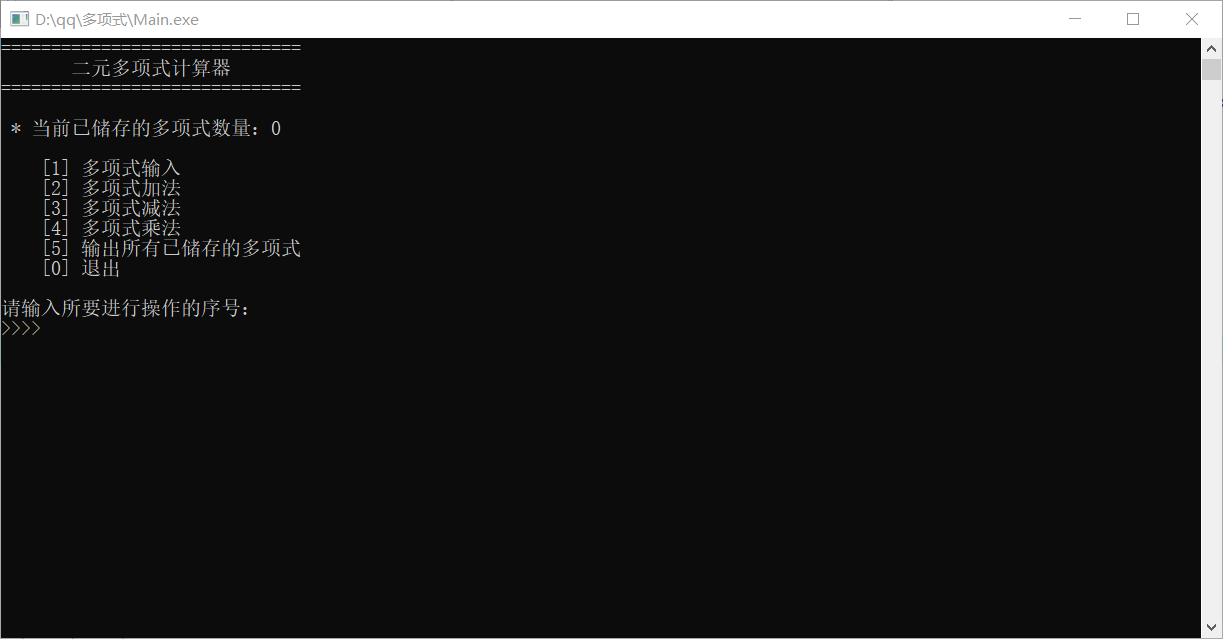
头文件Calculate.h：

1. List add(List L1, List L2)
2. {
3. IteratorResult\* p2 = createEmptyIteratorResult();//创建新的迭代器单元用于输出
4. iteratorResult(TRUE, L2, p2);//开启新的迭代器进行迭代
6. List result = L1;
7. **do**
8. result = setData(result, getData(result, p2 -> x\_degree, p2 -> y\_degree) + p2 -> data, p2 -> x\_degree, p2 -> y\_degree);  //对L1多项式中存在的与L2同类的项进行相加并将计算结果直接覆盖该项
9. **while** (iteratorResult(FALSE, L2, p2));//遍历L2
10. **return** result;
11. }
13. List sub(List L1, List L2)
14. {
15. IteratorResult\* p2 = createEmptyIteratorResult();
16. iteratorResult(TRUE, L2, p2);
17. List result = L1;
18. **do**
19. result = setData(result, getData(result, p2 -> x\_degree, p2 -> y\_degree) - p2 -> data, p2 -> x\_degree, p2 -> y\_degree);
20. **while** (iteratorResult(FALSE, L2, p2));
21. **return** result;
22. }
24. List multiply(List L1, List L2)
25. {
26. IteratorResult\* p1 = createEmptyIteratorResult();
27. iteratorResult(TRUE, L1, p1);//开启新的迭代器遍历L1
29. List result = newPolynomial();//创建一个新的多项式空间
30. **do** {
32. IteratorResult\* p2 = createEmptyIteratorResult();
33. iteratorResult(TRUE, L2, p2);//开启新的迭代器遍历L2
35. **do** {
36. **int** x = p1 -> x\_degree + p2 -> x\_degree;
37. **int** y = p1 -> y\_degree + p2 -> y\_degree;
38. **double** data = p1 -> data \* p2 -> data;
39. result = setData(result, getData(result, x, y) + data, x, y);
40. }
41. **while** (iteratorResult(FALSE, L2, p2));//将一个L1中的项与L2中的项分别相乘后存入result链表中
43. }
44. **while** (iteratorResult(FALSE, L1, p1));//对L1中的所有项进行如上操作
46. **return** result;
47. }
49. List copy(List L)//复制即将一个以有的多项式加在一个空的多项式空间内
50. {
51. **return** add(newPolynomial(), L);
52. }
54. /\*
55. int Main(void)
56. {
57. List l1 = newPolynomial();
58. List l2 = newPolynomial();
60. l1 = setData(l1, -2.3, 2, 5);
61. l1 = setData(l1, -1, 0, 0);
62. l1 = setData(l1, -3, 0, 0);
63. l1 = setData(l1, -1.1, 7, 0);
64. printResult(l1);
66. l2 = setData(l2, 3, 2, 5);
67. l2 = setData(l2, 7, 2, 8);
68. l2 = setData(l2, -1.1, 7, 0);
69. l2 = setData(l2, 1, 0, 0);
70. printResult(l2);
72. List re;
74. printf("\nadd:\n");
75. re = add(copy(l1), l2);
76. printResult(re);
77. freeAll(re);
79. printf("\nsub:\n");
80. re = sub(copy(l1), l2);
81. printResult(re);
82. freeAll(re);
84. printf("\nmultiply:\n");
85. re = multiply(copy(l1), l2);
86. printResult(re);
88. freeAll(re);
89. freeAll(l1);
90. freeAll(l2);
92. }\*/
93. **算法思路**

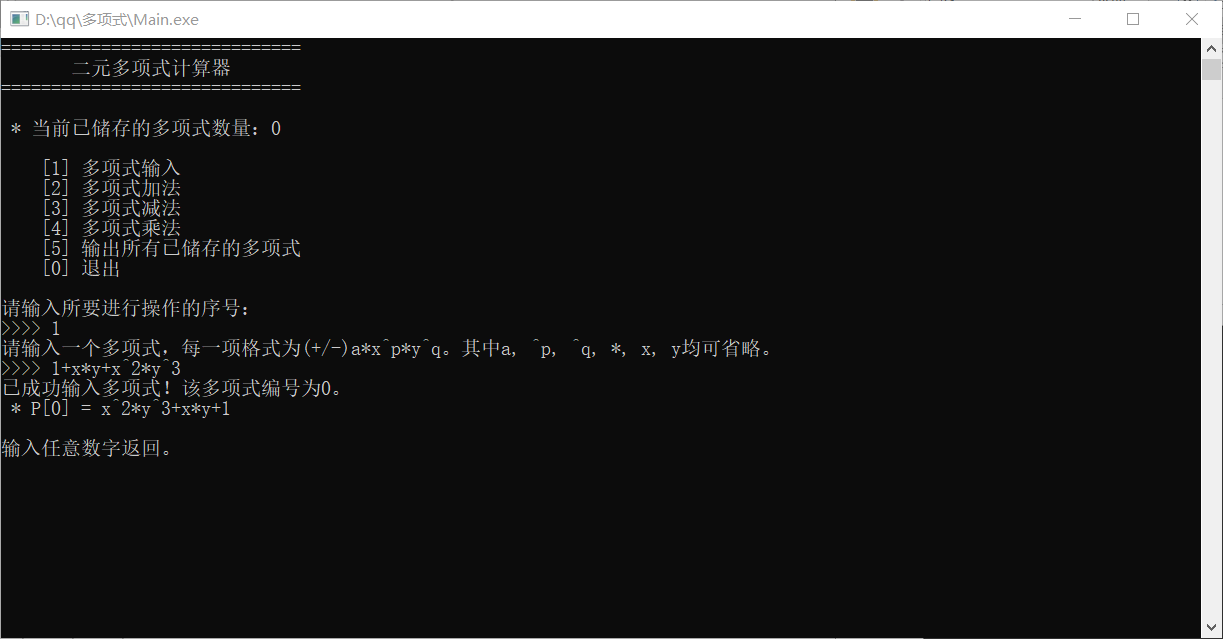
这个project的主要思路就是用链表来储存多项式，用链表的每一个节点来表示每一个单项式，这样对多项式进行加减乘操作就可以利用链表中添加删除节点等操作来进行。

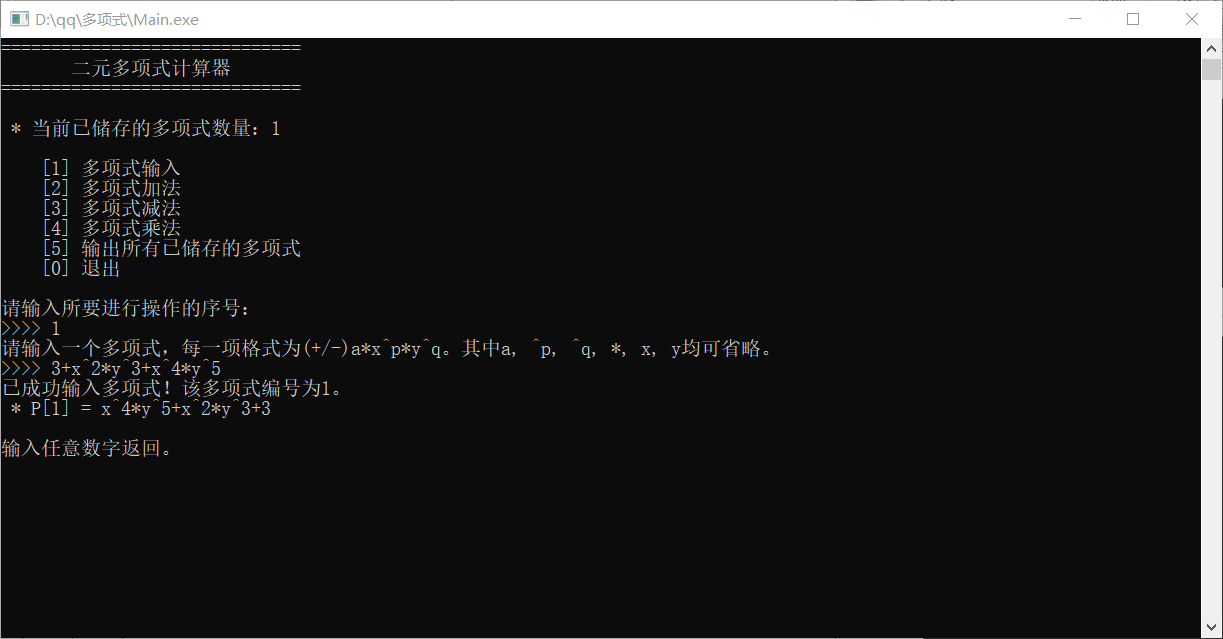
1. **流程图**

初始界面：

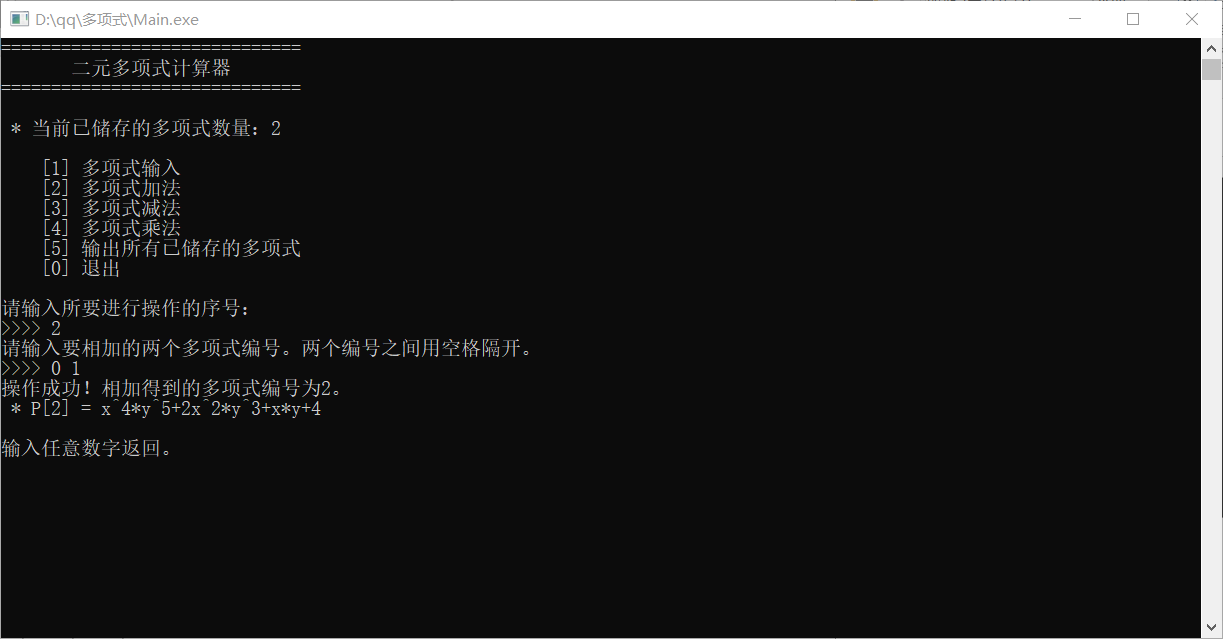


操作[1]多项输入：

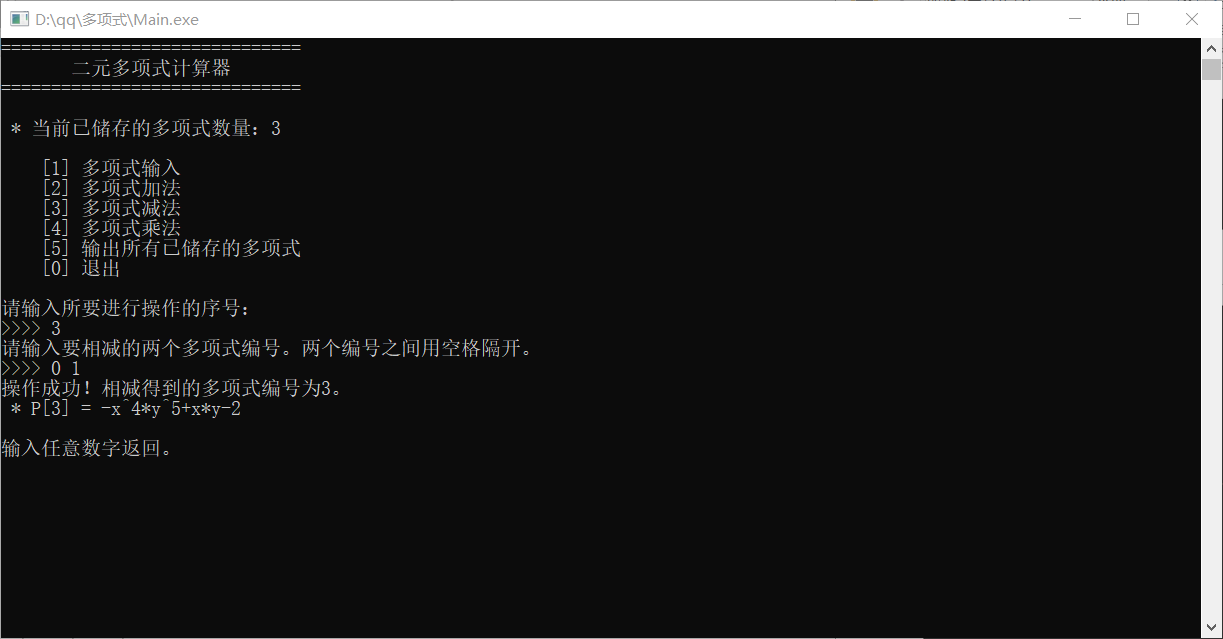




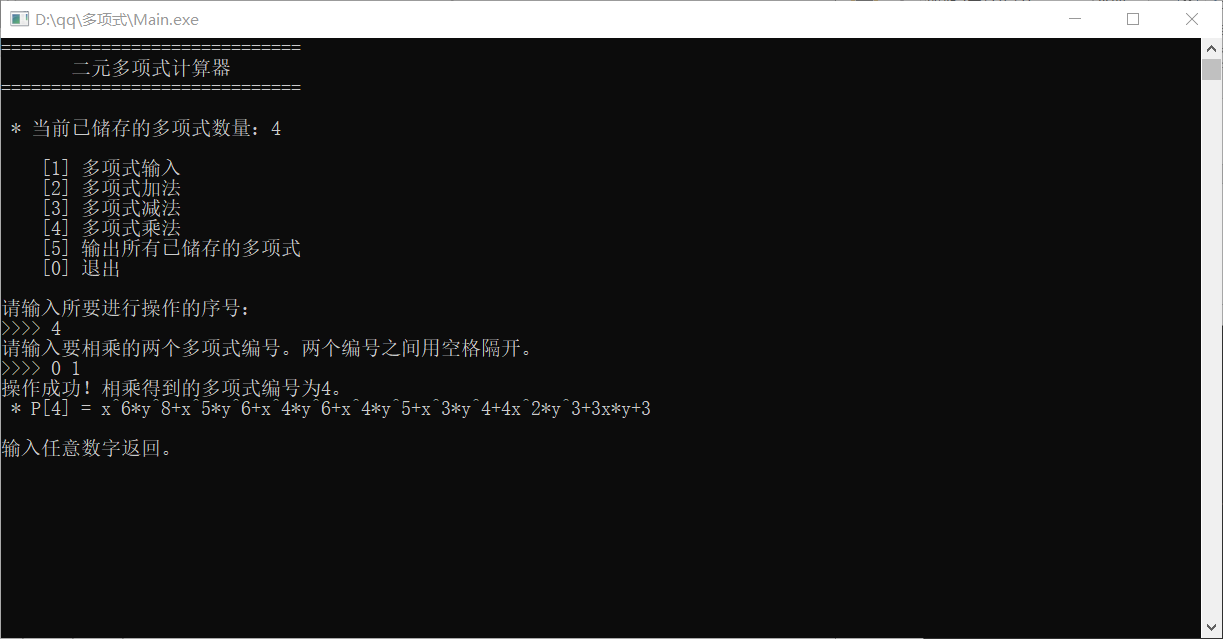
操作[2]多项式加法：



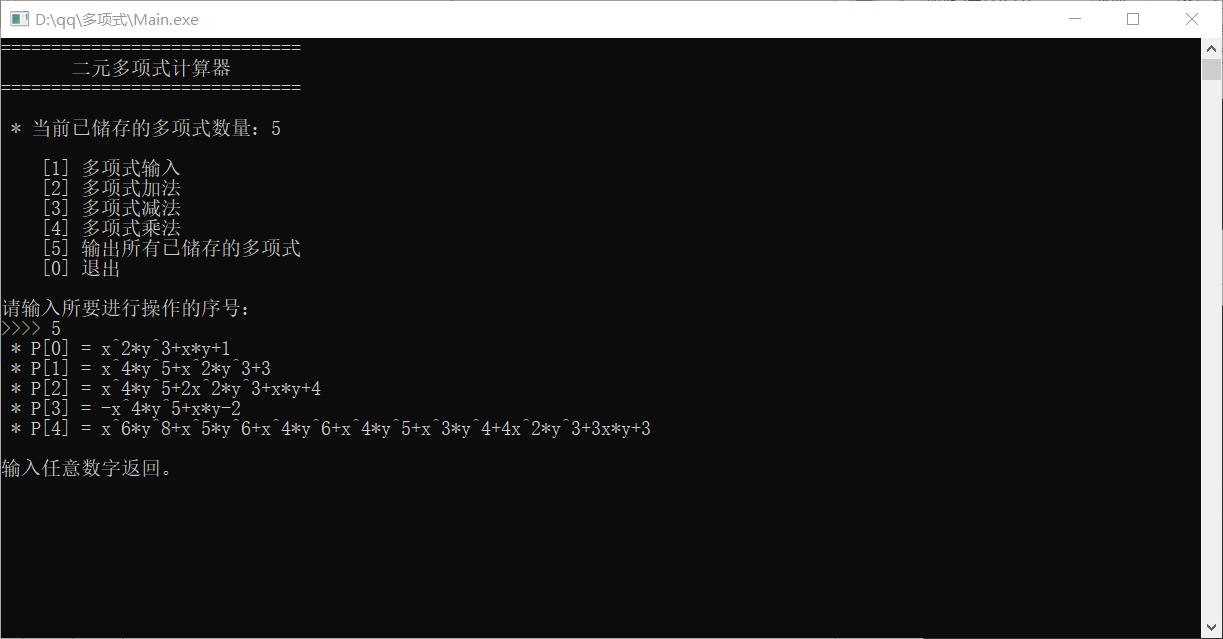
操作[3]多项式减法：



操作[4]多项式乘法：



操作[5]输出所有多项式：



操作[0]退出：

