一、主要工作

在本次小组合作中，本人主要负责完成积分模块的设计、实现以及后期的测试，并在最后撰写部分设计报告。——胡亮泽

二、积分模块设计思路

下面的靠前部分已经在总体的报告中提及过，这里重述一遍。

1. 本模块主要包括 calculus.h 和 calc.cpp 文件。

Calculus.h 头文件定义如下：

```
#pragma once
#ifndef CALCULUS_H_
#define CALCULUS_H_
#include "Function.h"

struct stack{        //利用堆栈实现函数表达式运算优先级
    char func[100];  //原函数表达式字符串
    double output[50]; //用以存储输出的数字
    char stack_out[10]; //用以存储堆栈中的运算符
    double Func_Calc(const double& x);

};


class Calculus : public Function{
private:
    const double high = 2000000000;  //运算的上限
    const double low = -2000000000;  //运算的下限
    const double distance = 0.0001;  //离散点之间的x向距离
    double x_from, x_to;   //积分的x范围

    stack pre; //堆栈

    double Calc(double x); //计算某个x值下的函数值，即某个点的函数值
    void GetFunction();    //读取函数表达式，需要输入，输入关于x的函数体，中间不能有空格等符号
    void GetRange();    //读取积分范围
public:
    Calculus(double from = 0, double to = 1000){
        x_from = from;
        x_to = to;

    }

    double GetResult();
    void function();
};



#endif
```

设计思路：头文件中的大多数定义已经有相应的注释了，不再赘述。主要涉及思路参照了 matlab 中积分的方法。即对于任意曲线，在曲线上取大量的点，并通过计算这些点的面积并相加得到曲线面积，所以本模块最后的计算结果实际上是一个估计值，误差大约在 0.00002 左右。

实现思路：

定义一个 calculus 的类和 stack 的结构。Calculus 类中有包括积分的区域 x_from, x_to 等变量，主要是描述积分的时候的一些内容。还有一个关于堆栈的结构变量，堆栈结构存在的主要意义就是对输入的字符串函数体进行处理分析，包括优先级和计算等，其中的成员函数 Func_Calc 就是用来计算某个点的横坐标为 x 时该点的纵坐标的。

执行过程主要包括：

调用 calculus 类中的 GetFunction 函数，主要作用是得到函数体字符串，该字符串会通过 pre 变量的调用输入到 pre 中。

调用 GetRange 函数，主要作用是获得积分范围，可以反向积分，即 x_from 的输入大于 x_to 的值。

上面两个函数都在调用 GetResult 函数的时候自动调用，主要是获得积分结果，该函数通过不断循环调用 Calc 函数（主要功能为计算某个点的面积值），获得许多点的函数值并求和，获得最后的积分结果。

2. 下面重点叙述各个函数的实现方法和工作原理

以下是 cpp 文件的源代码：

```
   double Calculus::GetResult(){
    GetFunction();
    GetRange();
    double result = 0;
    double calc = 0;
    if (x_from <= x_to)
    for (double i = x_from; i < x_to; i += distance){  //在积分
范围内求面积
        calc = Calc(i);
        if (calc<low || calc>high) throw 3; //超出范围抛出 3
        else result += calc * distance;
    }
    else
    for (double i = x_to; i < x_from; i += distance){  //在积分
范围内求面积
        calc = Calc(i);
        if (calc<low || calc>high) throw 3; //超出范围抛出 3
        else result += calc * distance;
    }
    if (x_from <= x_to)
        return result;
    else return -result;
    }
```

以上部分是 GetResult 函数源代码。在实际运行中，只要运行 GetResult 函数就可以完成所有的积分工作。

在这部分，我们可以发现，通过分别调用 GetFunction 函数和 GetRange 函数可以先得到字符串的函数体和积分范围，接下来，利用循环语句

```
for (double i = x_from; i < x_to; i += distance)
```

不断地调用 Calc 函数计算并将结果累加得到 result 返回。当超出积分范围时抛出 3。为了支持反向积分，利用两个 if——else 语句，在对 x_from 和 x_to 的大小进行比较后作出不同的循环计算（循环起点是 x_from 或者 x_to）并返回不同的值（正或负）。
而在这个过程中调用的三个函数都是极简单的。

```
    double Calculus::Calc(double x){

        return pre.Func_Calc(x);  //利用堆栈结构中的函数计算结果
}
    void Calculus::GetFunction(){
        cout << "请按照格式输入函数" << endl;
        cin.sync();
        cin.getline(pre.func,100);   //输入函数体
    }

    void Calculus::GetRange(){  //输入范围
        cout << "请输入 x 的积分起点" << endl;
        cin.sync();
        cin >> x_from;
        cout << "请输入 x 的积分终点" << endl;
        cin.sync();
        cin >> x_to;
}
```

Calc 函数只需要调用堆栈类中的 Func_Calc 函数并返回其值就可以。而 GetFunction 和 GetRange 只是简单的输入而已，需要注意的是，在输入前必须先清除掉输入流中的内容，否则会出现很多问题。比如因为空格和多次错误引起后面的连锁错误等。
最后是最为关键的部分，堆栈 Func_Calc 函数的实现：

```
    double stack::Func_Calc(const double& x){  //带有优先级的运算函数，以 x 的值作为自变量的值
    string digit = "";  //用以读取数字
    int i = 0, j = 0, k = 0; //i 指向当前函数字符串中读取的位数，j 指向输出数字中当前空白的一位，k 指向运算符输出当前空白的一位
    bool flag = 0;
    while (func[i]){
        if ((func[i] >= '0' && func[i] <= '9') || func[i] == '.'){ //读取一连串的数字，包括小数点
            for (; (func[i] >= '0' && func[i] <= '9') || func[i] == '.'; i++){

                char d[20];
                d[0] = func[i];
                d[1] = 0;
                digit += d;  //结果以字符串的形式记录

            }
            if (!flag)
            output[j++] = atof(digit.c_str());  //将字符串转换成 double 存储
```

```
            else {
                output[j++] = -atof(digit.c_str());
                flag = 0;
            }
            digit = "";
        }

        else if (func[i] == 'x' || func[i] == 'X'){  //读取到 X
时以自变量的值存储
            if (!flag)
            output[j++] = x;
            else{
                output[j++] = -x;
                flag = 0;
            }
            i++;
        }
        else if (func[i] == 'e' || func[i] == 'E'){ //读取到 e
时以 e 的值存储
            if (!flag)
            output[j++] = 2.718281828459;
            else{
                output[j++] = -2.718281828459;
                flag = 0;
            }
            i++;
        }

        else {  //读取到运算符时
            switch (func[i]){
            case '+':  //加法时，在堆栈不空或者没有遇到‘(’括号时，所
有运算符优先级都比+高，前面的计算要全部完成
                while ((k - 1 >= 0) && (stack_out[k - 1] != '(')){
                    switch (stack_out[k - 1]){
                    case '+':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] + output[j];
                        break;
                    case '-':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] - output[j];
                        break;
                    case '*':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] * output[j];
                        break;
                    case '/':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] / output[j];
                        break;
                    case '^':
                        k--;
                        j--;
                        output[j  -  1]  =  pow(output[j  -  1],
output[j]);
                        break;
```

```
                    case ')':
                        throw 1; //括号输入错误，抛出 1
                        break;
                    default: throw 0; //有任何规定范围内的输入，抛出
0
                    }

                }
                stack_out[k++] = '+'; //运算结束后将+推入堆栈
                break;
            case '-': //与加号类似
                if ((stack_out[k - 1] == '(')) //负号，需加括号
                {
                    flag = 1;
                }
                else{
                    while ((k - 1 >= 0) && (stack_out[k - 1] !=
'('){
                        switch (stack_out[k - 1]){
                        case '+':
                            k--;
                            j--;
                            output[j - 1] = output[j - 1] +
output[j];
                            break;
                        case '-':
                            k--;
                            j--;
                            output[j - 1] = output[j - 1] -
output[j];
                            break;
                        case '*':
                            k--;
                            j--;
                            output[j - 1] = output[j - 1] *
output[j];
                            break;
                        case '/':
                            k--;
                            j--;
                            output[j - 1] = output[j - 1] /
output[j];
                            break;
                        case '^':
                            k--;
                            j--;
                            output[j - 1] = pow(output[j - 1],
output[j]);
                            break;
                        case ')':
                            throw 1;
                            break;
                        default: throw 0;
                        }

                    }

                    stack_out[k++] = '-';
                }
                    break;
```

```
            case '*': //在推入堆栈前遇到减号和加号时，加减不需运算，其他运算符要运算
                while ((k - 1 >= 0) && (stack_out[k - 1] != '(') && (stack_out[k - 1] != '+') && (stack_out[k - 1] != '-')){
                    switch (stack_out[k - 1]){
                    case '*':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] * output[j];
                        break;
                    case '/':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] / output[j];
                        break;
                    case '^':
                        k--;
                        j--;
                        output[j  -  1]  =  pow(output[j  -  1], output[j]);
                        break;
                    case ')':
                        throw 1;
                        break;
                    default: throw 0;
                    }

                }
                stack_out[k++] = '*';

                break;
            case '/':  //与乘号类似
                while ((k - 1 >= 0) && (stack_out[k - 1] != '(') && (stack_out[k - 1] != '+') && (stack_out[k - 1] != '-')){
                    switch (stack_out[k - 1]){
                    case '*':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] * output[j];
                        break;
                    case '/':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] / output[j];
                        break;
                    case '^':
                        k--;
                        j--;
                        output[j  -  1]  =  pow(output[j  -  1], output[j]);
                        break;
                    case ')':
                        throw 1;
                        break;
                    default: throw 0;
                    }

                }
                stack_out[k++] = '/';
                break;
```

```
            case '^'://在推入堆栈前，除了遇到指数运算符‘^’,其他运算符
都不需要计算
                while ((k - 1 >= 0) && (stack_out[k - 1] != '(')
&& (stack_out[k - 1] != '+') && (stack_out[k - 1] != '-') &&
(stack_out[k - 1] != '*') && (stack_out[k - 1] != '/')){
                    switch (stack_out[k - 1]){
                    case '^':
                        k--;
                        j--;
                        output[j  -  1]  =  pow(output[j  -  1],
output[j]);
                        break;
                    case ')':
                        throw 1;
                        break;
                    default: throw 0;
                    }

                }
                stack_out[k++] = '^';
                break;
            case '(': //直接推入堆栈
                stack_out[k++] = '(';
                break;
            case ')': //在没有遇到左括号'('时，将所有运算符的计算全部
完成
                while ((k - 1 >= 0) && (stack_out[k - 1] != '(')){
                    switch (stack_out[k - 1]){
                    case '+':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] + output[j];
                        break;
                    case '-':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] - output[j];
                        break;
                    case '*':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] * output[j];
                        break;
                    case '/':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] / output[j];
                        break;

                    case ')':
                        throw 1;
                        break;
                    default: throw 0;
                    }
                }
                if (stack_out[k - 1] != '(') throw 1;
                else k--;
                break;
            default: throw 0;
            }
```

```
            i++;
        }
    }
    while (k - 1 >= 0){  //函数表达式字符串读取完成后，按顺序将堆栈中剩余的运算符计算全部完成
        switch (stack_out[k - 1]){
        case '+':
            k--;
            j--;
            output[j - 1] = output[j - 1] + output[j];
            break;
        case '-':
            k--;
            j--;
            output[j - 1] = output[j - 1] - output[j];
            break;
        case '*':
            k--;
            j--;
            output[j - 1] = output[j - 1] * output[j];
            break;
        case '/':
            k--;
            j--;
            output[j - 1] = output[j - 1] / output[j];
            break;
        case '^':
            k--;
            j--;
            output[j - 1] = pow(output[j - 1], output[j]);
            break;
        case ')':
            throw 1;
            break;
        default: throw 0;
        }
        k--;
    }
    return output[0]; //最后的计算结果保存在这里
    }
```

在数据结构基础这门课程中，曾经学习过利用的堆栈的方法确定一个输入表达式的计算优先级并进行正确的计算的方法。本次的堆栈结构通过输入字符串的表达式并将字符串转换成相应的信息进行计算的方法获得结果。由于运算符较多，所以整个函数中出现了非常多的 switch 语句，虽然看上去复杂，但事实上函数的结构还是比较简单的。

首先是在读取中遇到的字符类型的问题，可能涉及到数字，自变量 x，自然底数 e，操作符加减乘除指数还有负号等，包括括号，而数字还包括小数点。一旦出现非法输入，则会抛出 0 值并跳出。

通过循环，每次读取一个字符方式进行处理。

首先是对 x 和 e 值的处理。通过循环，

```
    else if (func[i] == 'x' || func[i] == 'X'){  //读取到 X 时以自变量的值存储
            if (!flag)
            output[j++] = x;
            else{
                output[j++] = -x;
                flag = 0;
```

```
            }
            i++;
        }
        else if (func[i] == 'e' || func[i] == 'E'){ //读取到 e
时以 e 的值存储
            if (!flag)
            output[j++] = 2.718281828459;
            else{
                output[j++] = -2.718281828459;
                flag = 0;
            }
            i++;
        }
```

通过 else if 语句将这两个值存储到 output 数组中，这个数组用以存储读取到的数字或者计算得到的数字结果，以此类推，在最后一次计算完成后，最终的结果会被存储在 output[0]中。

其中的 flag 是用来标记是否有读取到负号的，如果有，则将读取的到数值置为负数并重置 flag 的值为 0.

然后是对数字的处理

```
    if ((func[i] >= '0' && func[i] <= '9') || func[i] == '.'){ //
读取一连串的数字，包括小数点
            for (; (func[i] >= '0' && func[i] <= '9') || func[i]
== '.'; i++){

                char d[20];
                d[0] = func[i];
                d[1] = 0;
                digit += d;  //结果以字符串的形式记录

            }
            if (!flag)
            output[j++] = atof(digit.c_str());  //将字符串转换成
double 存储
            else {
                output[j++] = -atof(digit.c_str());
                flag = 0;
            }
            digit = "";
        }
```

一旦读取到一个数字的字符，则通过循环语句将接下来的所有连续的数字字符和小数点全部读取到一个字符串类型的 digit 中，在最后利用 atof 函数将字符串转换为数字存储到 output 中，同理，也需要利用 flag 变量来判断这个数值的正负。

最后是关于运算符的处理方法：

首先是加法运算符

```
    switch (func[i]){
            case '+':  //加法时，在堆栈不空或者没有遇到'('括号时，所
有运算符优先级都比+高，前面的计算要全部完成
                while ((k - 1 >= 0) && (stack_out[k - 1] != '(')){
                    switch (stack_out[k - 1]){
                    case '+':
                        k--;
                        j--;
```

```
                    output[j - 1] = output[j - 1] + output[j];
                    break;
                case '-':
                    k--;
                    j--;
                    output[j - 1] = output[j - 1] - output[j];
                    break;
                case '*':
                    k--;
                    j--;
                    output[j - 1] = output[j - 1] * output[j];
                    break;
                case '/':
                    k--;
                    j--;
                    output[j - 1] = output[j - 1] / output[j];
                    break;
                case '^':
                    k--;
                    j--;
                    output[j  -  1]  =  pow(output[j  -  1],
output[j]);
                    break;
                case ')':
                    throw 1; //括号输入错误，抛出 1
                    break;
                default: throw 0; //有任何规定范围内的输入，抛出
0
                }

            }
            stack_out[k++] = '+'; //运算结束后将+推入堆栈
            break;
```

加法的优先级是最低的，在遇到加法运算符时，先检查已经被推入堆栈的运算符，在遇到‘(’前将所有运算符（因为没有比加法的运算优先级更低的运算符）推出堆栈，这里直接将最外围的 output 中的两个数字进行运算并重新存储，如除法 `output[j - 1] = output[j - 1] / output[j];`

而所有与输入括号有关的错误抛出 1

```
            case ')':
            throw 1; //括号输入错误，抛出 1
            break;
```

最后运算完成后再将这个加号推入堆栈。

减号与加号基本相同，但是需要处理的情况多了一种，那就是作为负号出现。

```
   if ((stack_out[k - 1] == '(')) //负号，需加括号
            {
                flag = 1;
            }
```

由于本次输入要求所有负号必须与括号一起输入，所以在堆栈中前一个运算符为‘(’号时，将 flag 置为 1，表明下一个读取的数字要置为负值，负号是直接运算的，不需要推入堆栈。

乘除法的实现方法完全相同，这里以乘法为例：

```
   case '*': //在推入堆栈前遇到减号和加号时，加减不需运算，其他运算符
要运算
```

```
                while ((k - 1 >= 0) && (stack_out[k - 1] != '(')
&& (stack_out[k - 1] != '+') && (stack_out[k - 1] != '-')){
                    switch (stack_out[k - 1]){
                    case '*':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] * output[j];
                        break;
                    case '/':
                        k--;
                        j--;
                        output[j - 1] = output[j - 1] / output[j];
                        break;
                    case '^':
                        k--;
                        j--;
                        output[j  -  1]  =  pow(output[j  -  1],
output[j]);
                        break;
                    case ')':
                        throw 1;
                        break;
                    default: throw 0;
                    }
```

堆栈中，在遇到优先级低于本运算符的运算符之前，所有的运算符都会被输出并进行运算，所以在 while 语句中，增加了运算符不等于加号和减号的前提下进行循环，其余的和加减法完全相同。

指数运算的原理和以上的运算也完全相同，只不过除了遇到指数运算符之外，指数运算符都会被推入堆栈。

在所有运算符都被推入堆栈后

```
   while (k - 1 >= 0){  //函数表达式字符串读取完成后，按顺序将堆栈中
剩余的运算符计算全部完成
       switch (stack_out[k - 1]){
       case '+':
           k--;
           j--;
           output[j - 1] = output[j - 1] + output[j];
           break;
       case '-':
           k--;
           j--;
           output[j - 1] = output[j - 1] - output[j];
           break;
       case '*':
           k--;
           j--;
           output[j - 1] = output[j - 1] * output[j];
           break;
       case '/':
           k--;
           j--;
           output[j - 1] = output[j - 1] / output[j];
           break;
       case '^':
           k--;
           j--;
           output[j - 1] = pow(output[j - 1], output[j]);
```

```
            break;
        case ')':
            throw 1;
            break;
        default: throw 0;
        }
        k--;
    }
    return output[0]; //最后的计算结果保存在这里
}
```

利用一个 while 语句将所有的运算符都进行运算。在前面提到过，最后的运算结果会被存储在 output[0]中，返回即可得到结果。

三、个人心得

本次的大程合作是我第二次以小组合作的形式完成一个系统的设计，又是第一次基于 C++语言的合作编程经历。在整个过程中，我个人的收获还是比较大的。虽然在讨论和各人的代码编写和单独调试、测试环节没有出现非常大的问题，但事实上，在后期的整合过程中却会出现各种各样的问题。

这些问题除了在个人测试环境没有发现的问题以外，更多的是将所有人的模块整合后接口能否对接的问题。因为在设计过程中，虽然所有人都知道自己需要做的是什么，需要实现什么功能，而事实上大家也都做到了，但是却不知道其他人需要以怎样的形式利用自己编写的模块，或者说在利用形式上也会出现一些问题。本次的选题——数学软件，各模块在一定程度上来说已经算是比较独立的一款软件了，只需要在最后利用一种方法，分别对接 4 个不同功能的模块就可以了，而 4 个模块彼此确实相互独立的，这已经大大降低了后期对接的难度。然而即使如此，也出现了很多的问题，而个人设计中的一些问题也被无线放大，由此可以预见在未来较大型程序的设计中，这种合作能力的提升有多么重要。

因此，在这次合作中最大的收获反而不是程序本身，而是合作的意识和方法上的提升，这对将来的小组合作有着至关重要的作用。