

Project 2 A Simple Calculator

Name: 周思呈

SID: 12110644

Part 01 - Analysis

要求实现一个比project1优化一些的计算器，完成普通四则运算、带括号的四则运算、解方程、高精度计算等功能。主要实现思路如下：

1. 用 `cin` 从标准输入中读入一整个需要计算的表达式。如果输入的是 `q` 或者 `quit` 则退出程序，否则进行后续计算。
2. 从后往前遍历表达式的每个字符，有以下几种情况：赋值，即等号前面是未知数，等号后面是表达式；计算，即只有表达式。实际处理当中发现赋值过程中需要处理等号右边的表达式，包括将未知数替换成值然后赋给新的未知数（或者更新原有值）等等，所以用 `"="` 作为分隔，遇到等号即处理掉其之后的表达式。
3. 处理表达式大致分为以下几个板块：遇到括号，记录括号的位置；遇到变量，判断之前是否已经储存过，如有则将变量替换成其对应值；遇到等号，接着往前遍历；遍历到头，处理整个表达式。最后输出处理结果。

Part 02 - Code

数据储存方式

为了实现高精度快速计算，用自定义结构体 `Big_number` 储存数据。储存的值分别为数字个数 `length`，小数位数 `scale`，所有数字 `digits` 数组，以及符号 `sign`。这样的存储格式能大幅提高数据范围，也能为后续实现高精度四则运算方法服务。

```
struct Big_number
{
    int length;
    int scale;
    int digits[256];
    bool sign;
};
```

数据读写方式

运算方法实现过程中常需要进行其他数据类型与 `Big_number` 数据类型之间的转化，于是写了 `input` 和 `output` 两个函数来专门进行转化，前者将字符串转化为 `Big_number`，后者相反。需要特别说明的是 `input` 函数中在输入时去掉了小数末尾的0，提高了储存效率。

```
Big_number input(string exp)
{
    Big_number result;
    int length = exp.length();
    int scale = 0;
    int l = 0;
    result.digits[exp.length() - 1] = 0;
    if (exp[0] == '-')
    {
        result.sign = 0;
    }
    else
    {
        result.sign = 1;
    }
    for (int i = 0; i < exp.length(); i++)
    {
        if (exp[i] == '.')
        {
```

```

        scale = exp.length() - i - 1;
        length--;
    }
    else
    {
        result.digits[l] = exp[i] - '0';
        l++;
    }
}

if (scale != 0)
{
    for (int i = l - 1; i > 0; i--)
    {
        if (result.digits[i] == 0)
        {
            length--;
            scale--;
        }
        else
        {
            break;
        }
    }
}

result.length = length;
result.scale = scale;

return result;
}

string output(Big_number big_number)
{
    string output;
    int head_pos = 0;
    if (big_number.sign) //正数
    {
        output = "";
    }
    else
    {
        output = "-";
    }

    for (int i = 0; i < big_number.length; i++) // 去掉开头的0
    {
        if (i == big_number.length - big_number.scale)
        {
            output = output + ".";
        }
        char c = big_number.digits[i] + '0';
        output = output + c;
    }
    return output;
}

```

将中缀表达式转化为后缀表达式

change 函数通过栈操作将中缀表达式字符串转化成后缀表达式字符串，数字之间用空格隔开。其中**level** 函数为每一个操作符返回一个代表优先级的数。大二小朋友还没学到栈操作、中缀表达式、后缀表达式和二者转化，所以这个板块以及下面计算后缀表达式的板块都参考了https://blog.csdn.net/qg_45661967/article/details/104907475 **C++ 实现科学计算器**。

```

bool change(string &from, string &to)
{

```

```

int state = OUT;
char c;

for (int i = 0; i < from.length(); i++)
{
    c = from[i];
    if (isdigit(c))
    { //是小数点前的数字
        to = to + c;
        state = IN; //状态改为在数字内
    }
    else
    {
        if (state == IN && c == '.')
        { //碰到小数点, 仍旧是数字
            to = to + '.';
            continue;
        }
        if (state == IN && c != '.')
        { //不是数字
            to += ' ';
        }
        if (c == '=')
            break;
        else if (c == '(')
            opt.push(c);
        else if (c == ')')
        {
            while (!opt.empty() && opt.top() != '(')
            { //括号匹配
                to += opt.top();
                to += ' ';
                opt.pop();
            }
            opt.pop();
        }
        else
        {
            while (true)
            {
                if (opt.empty() || opt.top() == '(')
                    opt.push(c);
                else if (level(c) > level(opt.top()))
                    opt.push(c);
                else
                {
                    to += opt.top();
                    to += ' ';
                    opt.pop();
                    continue;
                }
            }
            break;
        }
    }
    state = OUT; //状态为在数字外
}

while (!opt.empty())
{
    to += opt.top();
    to += ' ';
    opt.pop();
}
return true;
}

int level(char theOpt)
{

```

```

switch (theOpt)
{
case '(':
    return 0;
case '+':
case '-':
    return 1;
case '*':
case '/':
    return 2;
case ')':
    return 3;
}
return -1;
}

```

计算后缀表达式

得到`change` 之后的后缀表达式结果，用`compute` 方法进行计算。将后缀表达式中的数字转化成`Big_number`后压进`val`栈，操作符按照优先级压入`opt`栈，然后依次出栈进行四则运算。

```

bool compute(string &theExp)
{
    int state = OUT; //初始状态为在数字外
    char c;
    bool single = true;
    int last_pos = -1; //上一个运算符的位置
    bool dot = false;

    int pos = 0;
    int scale = 0;
    int length = 0;
    for (int i = 0; i < theExp.length(); i++)
    {
        c = theExp[i];
        if (isdigit(c) || c == '.') //是数字
        {
            if (isdigit(c))
            {
                length++;
                big_numbers[big_num_pos].digits[pos] = c - '0';
                pos++;
                state = IN; //状态为在数字内
                if (dot == true)
                    scale++;
            }
            if (c == '.')
            {
                dot = true;
                continue;
            }
        }
        else //不是数字（或者小数点）
        {
            single = false;
            dot = false;
            if (state == IN)
            {
                big_numbers[big_num_pos].length = length;
                big_numbers[big_num_pos].scale = scale;
                big_numbers[big_num_pos].sign = true; //待定
                val.push(big_numbers[big_num_pos]);
                big_num_pos++;
                pos = 0;
                length = 0;
            }
        }
    }
}

```

```

        scale = 0;
    }

    // 四则运算
    Big_number x, y;
    if (c != ' ')
    {
        x = val.top();
        val.pop();
        y = val.top();
        val.pop();

        switch (c)
        {
            case '+':
                val.push(add(x, y));
                single = false;
                break;
            case '-':
                val.push(minus(y, x));
                single = false;
                break;
            case '*':
                val.push(mul(x, y));
                single = false;
                break;
            case '/':
                val.push(divide(y, x));
                single = false;
                break;
            default:
                cout << "未知的错误!" << endl;
        }
    }

    state = OUT;
}

//没有表达式的情况
if (single)
{
    if (state == IN)
    {
        big_numbers[big_num_pos].length = length;
        big_numbers[big_num_pos].scale = scale;
        big_numbers[big_num_pos].sign = true; //待定
        val.push(big_numbers[big_num_pos]);
        big_num_pos++;
        pos = 0;
        length = 0;
        scale = 0;
    }
}

return true;
}

```

高精度加法

将两个Big_number根据length和scale大小将整数和小数位分别补齐，即不够长的部分填充0，防止之后计算时数组越界。然后按位相加，用整数carry模拟进位，最后转化成Big_number输出。

补齐时暂定整数部分长度和小数部分长度都取x和y整数/小数最大的长度，result.digits数组总长度为二者相加后再加一，多出来的这一位用来处理可能存在的进位。最后如果发现并没有进位，则去掉开头的0。

```

Big_number add(Big_number x, Big_number y) //正数加法
{
    Big_number result;
    int x_before = x.length - x.scale;
    int y_before = y.length - y.scale;
    int length = max(x.scale, y.scale) + max(x_before, y_before) + 1;
    int *cur_x = new int[length]();
    int *cur_y = new int[length]();
    int carry = 0;

    //整数部分补齐
    for (int i = max(x_before, y_before); i >= 0; i--) //填充前max(x_before, y_before)位
    {
        if (i <= max(x_before, y_before) - x_before)
        {
            cur_x[i] = 0;
        }
        else
        {
            int filled = max(x_before, y_before) - i;
            cur_x[i] = x.digits[x_before - filled - 1];
        }
        if (i <= max(x_before, y_before) - y_before)
        {
            cur_y[i] = 0;
        }
        else
        {
            int filled = max(x_before, y_before) - i;
            cur_y[i] = y.digits[y_before - filled - 1];
        }
    }

    //小数部分补齐
    for (int i = max(x.scale, y.scale) - 1; i >= 0; i--)
    {
        if (i + 1 > x.scale)
        {
            cur_x[max(x_before, y_before) + i + 1] = 0;
        }
        else
        {
            cur_x[max(x_before, y_before) + i + 1] = x.digits[x_before + i];
        }
        if (i + 1 > y.scale)
        {
            cur_y[max(x_before, y_before) + i + 1] = 0;
        }
        else
        {
            cur_y[max(x_before, y_before) + i + 1] = y.digits[y_before + i];
        }
    }

    for (int i = length - 1; i >= 0; i--)
    {
        result.digits[i] = (cur_x[i] + cur_y[i] + carry) % 10;
        carry = (cur_x[i] + cur_y[i] + carry) / 10;
    }

    if (result.digits[0] == 0) //处理最大位进位
    {
        for (int i = 0; i < length - 1; i++)
        {
            result.digits[i] = result.digits[i + 1];
        }
    }
}

```

```

        length--;
    }

    result.length = length;
    result.scale = max(x.scale, y.scale);
    result.sign = x.sign; //存疑

    delete[] cur_x;
    delete[] cur_y;

    return result;
}

```

高精度减法

和高精度加法原理类似，先对齐小数点把减数补齐，然后计算减数的complement，将complement与被减数相加，去掉最大位，再加一，得到最终结果。scale取x和y中小数点位数更大者。不要问我为什么方法名字叫miinus，问就是minus好像和库里面某个方法重名了。

```

Big_number miinus(Big_number x, Big_number y) // x-y
{
    Big_number result;
    Big_number complement;
    int length = max(x.length, y.length);
    int com_pos = y.length - 1;
    int *cur_y = new int[length]();
    int x_before = x.length - x.scale;
    int y_before = y.length - y.scale;

    //小数部分补齐
    for (int i = max(x.scale, y.scale) - 1; i >= 0; i--)
    {
        if (i + 1 > y.scale)
        {
            cur_y[max(x_before, y_before) + i] = 0;
        }
        else
        {
            cur_y[max(x_before, y_before) + i] = y.digits[y_before + i];
        }
    }

    //整数部分补齐
    for (int i = max(x_before, y_before); i > 0; i--) //填充前max(x_before, y_before)位
    {
        if (i <= max(x_before, y_before) - y_before)
        {
            cur_y[i - 1] = 0;
        }
        else
        {
            int filled = max(x_before, y_before) - i;
            cur_y[i - 1] = y.digits[y_before - filled - 1];
        }
    }

    //计算complement
    for (int i = length - 1; i >= 0; i--)
    {
        if (com_pos >= 0)
        {
            complement.digits[i] = 9 - cur_y[i];
        }
        else
        {

```

```

        complement.digits[i] = 9;
    }
}
delete[] cur_y;
complement.length = length;
complement.scale = max(x.scale, y.scale);
complement.sign = true;
result = add(x, complement);

//去头
for (int i = 0; i < length; i++)
{
    result.digits[i] = result.digits[i + 1];
}
result.length = length;
result.scale = 0;
result = add(result, input("1"));
result.scale = max(x.scale, y.scale);

return result;
}

```

高精度乘法

手动模拟乘法及其进位计算效率较低，于是采用FFT快速傅立叶变换，将两个乘数看成 $x=10$ 的多项式，计算两个多项式的乘积，再转化为Big_number输出。

FFT主要思路如下。 n 次多项式上 $n+1$ 个不同的点能唯一确定这个多项式，因此我们可以将多项式系数表示法转化为点值表示法。将相应点值纵坐标相乘得到结果点值，再利用IDFT离散傅立叶逆变换转化为系数。过程中需要用到复数，于是定义了Complex结构体及其对应运算符。

这一部分参考了 <https://zhuanlan.zhihu.com/p/347091298> 快速傅里叶变换（FFT）超详解 - 星夜的文章 - 知乎，以及 <http://t.csdn.cn/EONGE> 【学习笔记】超简单的快速傅里叶变换（FFT）（含全套证明）。

```

struct Complex
{
    double x, y;
    Complex(double x = 0, double y = 0) : x(x), y(y) {}
} a[N], b[N];

Complex operator*(Complex J, Complex Q)
{
    //模长相乘，幅度相加
    return Complex(J.x * Q.x - J.y * Q.y, J.x * Q.y + J.y * Q.x);
}

Complex operator-(Complex J, Complex Q)
{
    return Complex(J.x - Q.x, J.y - Q.y);
}

Complex operator+(Complex J, Complex Q)
{
    return Complex(J.x + Q.x, J.y + Q.y);
}

Big_number mul(Big_number x, Big_number y)
{
    m = x.length - 1;
    n = y.length - 1;
    Big_number result;
    int *temp = new int[m + n + 1]();
    for (int i = 0; i < x.length; i++)
    {
        a[i].x = x.digits[i];
    }
    for (int i = 0; i < y.length; i++)

```



```

{
    b[i].x = y.digits[i];
}
while (limit <= n + m)
    limit <<= 1, L++;

for (int i = 0; i < limit; i++)
{
    R[i] = (R[i] >> 1) >> 1 | ((i & 1) << (L - 1));
}
FFT(a, 1); // FFT 把a的系数表示转化为点值表示
FFT(b, 1); // FFT 把b的系数表示转化为点值表示

//计算两个系数表示法的多项式相乘后的点值表示
for (int i = 0; i <= limit; ++i)
    a[i] = a[i] * b[i];

FFT(a, -1);

// temp暂存多项式系数
for (int i = 0; i <= m + n; i++)
{
    temp[i] = (int)(a[i].x + 0.5);
}

//计算结果值
int carry = 0;
int temp_pos = m + n;
for (int i = m + n + 2; i >= 0; i--)
{
    if (temp_pos >= 0)
    {
        result.digits[i] = (carry + temp[temp_pos]) % 10;
        carry = (carry + temp[temp_pos]) / 10;
        temp_pos--;
    }
    else
    {
        result.digits[i] = carry % 10;
        carry = carry / 10;
    }
}

//去头
int count = 0;
while (result.digits[0] == 0)
{
    count++;
    for (int i = 0; i < m + n + 3; i++)
    {
        result.digits[i] = result.digits[i + 1];
    }
}

result.scale = x.scale + y.scale;
result.length = x.length + y.length + 1 - count;
result.sign = !(x.sign ^ y.sign);

return result;
}

void FFT(Complex *A, int type)
{
    for (int i = 0; i < limit; ++i)
        if (i < R[i])
            swap(A[i], A[R[i]]);
    // i小于R[i]时才交换,防止同一个元素交换两次,回到它原来的位置。

```

```

//从底层往上合并
for (int mid = 1; mid < limit; mid <= 1)
{
    //待合并区间长度的一半，最开始是两个长度为1的序列合并,mid = 1;
    Complex wn(cos(PI / mid), type * sin(PI / mid)); //单位根w_n^1;

    for (int len = mid < 1, pos = 0; pos < limit; pos += len)
    {
        // len是区间的长度，pos是当前的位置，也就是合并到了哪一位
        Complex w(1, 0); //幂，一直乘，得到平方，三次方...

        for (int k = 0; k < mid; ++k, w = w * wn)
        {
            //只扫左半部分，蝴蝶变换得到右半部分的答案,w 为 w_n^k
            Complex x = A[pos + k]; //左半部分
            Complex y = w * A[pos + mid + k]; //右半部分
            A[pos + k] = x + y; //左边加
            A[pos + mid + k] = x - y; //右边减
        }
    }
}
if (type == 1)
    return;
for (int i = 0; i <= limit; ++i)
    a[i].x /= limit;
//最后要除以limit也就是补成了2的整数幂的那个N，将点值转换为系数
//（前面推过了点值与系数之间相除是N）
}

```

高精度除法

这一部分原本打算用减法模拟，再用二分提高其效率和精度，但写到后面发现时间好像不够了...所以没实现高精度的版本，只有初级版。

```

Big_number divide(Big_number x, Big_number y)
{
    Big_number result;
    double x_num = 0;
    double y_num = 0;
    for (int i = 0; i < x.length; i++)
    {
        x_num = x_num * 10 + x.digits[i];
    }
    x_num = x_num / pow(10, x.scale);

    for (int i = 0; i < y.length; i++)
    {
        y_num = y_num * 10 + y.digits[i];
    }
    y_num = y_num / pow(10, y.scale);

    double temp = x_num / y_num;
    result = input(to_string(temp));

    return result;
}

```

方程处理

处理方程主要用到了下面两个额外的方法。

`have_var` 通过遍历判断变量是否已经储存过，如果有则返回其对应位置的下标，没有则返回-1。

```
int have_var(string name)
{
    for (int i = 0; i < 256; i++)
    {
        if (var[i].name == name)
        {
            return i;
        }
    }
    return -1;
}
```

`get_var_name` 获取输入的表达式的首个变量名称。

```
string get_var_name(string exp)
{
    for (int i = 0; i < exp.size(); i++)
    {
        if (exp[i] == '=' || exp[i] == '+' || exp[i] == '-' || exp[i] == '*' || exp[i] == '/')
        {
            string output = exp.substr(0, i);
            return output;
        }
    }
    return exp;
}
```

储存变量名及其数学信息的数据类型是自定义结构体。另外开了一个全局数组储存输入的所有变量。

```
struct variable
{
    string name;
    Big_number value;
};
```

主函数

开头模拟bc初始界面。然后提示使用者输入要计算的表达式，或者输入q/quit退出程序。

```
(base) zhousicheng@zhousichengdeMacBook-Pro ~ % bc
bc 1.06
```

```
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
```

```
● (base) zhousicheng@zhousichengdeMacBook-Pro project2 % g++ main.cpp&& ./a.out
welcome to calculator 2.0
This is free software with ABSOLUTELY NO WARRANTY.
-----
please input something you want to calculate:
or you can type "q" or "quit" to exit
-----
```

开根函数放在最前面处理，利用二分法得到一定精度的开根结果。实际上这个方法在最后加得很仓促，属于ddl之前最后的挣扎（bushi），观察bc的计算精度和输入数字的计算精度一致，但是确实没时间写了。project永远也写不完，摊手。

其他运行逻辑在Analysis中已经介绍，此处不再重复。

```

int main()
{
    cout << "welcome to calculator 2.0" << endl;
    cout << "This is free software with ABSOLUTELY NO WARRANTY." << endl;
    ans = input("0");
    while (true)
    {
        //输入表达式
        string init_exp;
        cout << "-----" << endl;
        cout << "please input something you want to calculate:" << endl;
        cout << "or you can type \"q\" or \"quit\" to exit" << endl;
        cout << "-----" << endl;
        cin >> init_exp;
        num_of_exp++;
        have_brac = 0;
        right_brac_pos = init_exp.length();
        big_num_pos = 0;

        //sqrt
        if (init_exp.substr(0,4) == string("sqrt"))
        {
            int temp_length = init_exp.length() - 6;
            double num = get_num(init_exp.substr(5, temp_length));
            double left = 1;
            double right = num;
            double mid = 0.0;
            double result = 0.0;
            while (abs(result - num) > 0.0000000001)
            {
                mid = (left + right) / 2;
                result = mid * mid;
                if (result < num)
                {
                    left = mid;
                }
                else
                {
                    right = mid;
                }
            }
            ans = input(to_string(mid));
        }

        //判断是否要退出
        else if (init_exp == string("q") || init_exp == string("quit"))
        {
            cout << "-----" << endl;
            cout << "BYE" << endl;
            break;
        }

        else
        {
            for (int i = init_exp.length() - 1; i >= 0;)
            //括号, 记录位置
            //变量, 替换
            //等号, 把括号里的东西计算掉
            //其他, 接着往前挪
            {
                if (init_exp[i] == ')')
                {
                    right_brac_pos = i;
                    i--;
                    have_brac = 1;
                }
                else if (init_exp[i] == '(')

```

量

```
{
    left_brac_pos = i;
    i--;
    if (i < 0)
    {
        string infix = init_exp;
        string postfix = "";
        change(infix, postfix);
        compute(postfix);
        ans = input(output(val.top()));
    }
}

else if (is_numeric(init_exp[i]))
{
    if (i == 0) //只有计算, 没有赋值
    {
        string infix = init_exp;
        string postfix = "";
        change(infix, postfix);
        compute(postfix);
        ans = input(output(val.top()));
    }
    i--;
}

else if (init_exp[i] == '=' && init_exp[i - 1] != '=') //单个等号, 此时等号后面已经没有变

{
    int length = (right_brac_pos - 1) - (i + 1) + 1; //表达式的长度
    string infix = init_exp.substr(i + 1, length);
    string postfix = "";
    change(infix, postfix);
    compute(postfix);

    //找左括号, 顺便获取变量的名字
    string name = "";
    left_brac_pos = -1;
    for (int j = i - 1; j >= 0; j--)
    {
        if (init_exp[j] == '(')
        {
            left_brac_pos = j;
            break;
        }
        else
        {
            name = init_exp[j] + name;
        }
    }

    //给变量赋值
    int position = have_var(name);
    if (position != -1) //原先有这个变量, 更新他的值
    {
        var[position].value = input(output(val.top()));
    }
    else
    {
        var[flag].name = name;
        var[flag].value = input(output(val.top()));
        flag++;
    }

    //把整个括号里的东西替换掉
    length = right_brac_pos - left_brac_pos + 1 - (1 - have_brac) * 2;
    Big_number temp = input(output(val.top()));
    init_exp.replace(left_brac_pos + (1 - have_brac), length, output(temp));
    left_brac_pos = 0;
}
```

```

        have_brac = 0;
        right_brac_pos = init_exp.length();

        //更新i
        i = init_exp.length() - 1;
    }

    else //变量
    {
        int head = i;
        int tail = i;
        for (int j = i; j >= 0; j--)
        {
            if (is_numeric(init_exp[j]))
            {
                head = j + 1;
                break;
            }
        }
        string name = get_var_name(init_exp.substr(head, tail - head + 1));
        int position = have_var(name);
        int length = i - head + 1;
        if (position != -1) //原先有这个变量，获取他的值，替换字符串
        {
            init_exp.replace(head, length, output(var[position].value));
        }
        else
        {
            init_exp.replace(head, length, "0");
        }

        //更新i
        i = head - 1;
        if (i < 0)
        {
            i = 0;
        }
    }
}

//非法输入
// else
// {
//     cout << "(standard_in) " << num_of_exp << " : parse error";
// }

cout << output(ans) << endl;
}
return 0;
}

```

处理sqrt函数的get_num方法如下。

```

//字符串转化为数字
double get_num(string input)
{
    double a1 = 0;    // a的整数位
    double a2 = 0;    // a的小数位
    int k = 0;
    double dec = 0.1;
    bool flag = false;

    while (input[k] != 0)
    {
        if (input[k] == '.')

```

```

    {
        flag = true; //来到小数点之后
        k++;
        continue;
    }
    if (!flag)
    {
        a1 = a1 * 10 + input[k] - '0';
    }
    else
    {
        a2 = a2 + (input[k] - '0') * dec;
        dec = dec * 0.1;
    }
    k++;
}

return a1 + a2;
}

```

Part 03 - Result & Verification

Test Case 1

When you run your program and input an express in a line as follows, it can output the correct results. The operator precedence (order of operations) should be correct.

```

○ (base) zhousicheng@zhousichengdeMacBook-Pro project2 % g++ main.cpp&& ./a.out
welcome to calculator 2.0
This is free software with ABSOLUTELY NO WARRANTY.
-----
please input something you want to calculate:
or you can type "q" or "quit" to exit
-----
2+3
5
-----
please input something you want to calculate:
or you can type "q" or "quit" to exit
-----
5+2*3
11
-----
please input something you want to calculate:
or you can type "q" or "quit" to exit
-----

```

Test Case 2

Use parentheses to enforce the priorities.

```
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
5+2*3
11
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
(5+2)*3
21
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
```

Test Case 3

Variables can be defined as follows.

```
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
x=3
3
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
y=6
6
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
x+2*y
15
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
```

Test Case 4

Some math functions can be supported.

```
○ (base) zhousicheng@zhousichengdeMacBook-Pro project2 % g++ main.cpp&& ./a.out
welcome to calculator 2.0
This is free software with ABSOLUTELY NO WARRANTY.
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
sqrt(3)
1.732051
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
```


Test Case 5

It can support arbitrary precision.

```
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
```

```
9999999999999999.2222222222222+1.0
10000000000000000.2222222222222
```

```
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
```

Test Case 6

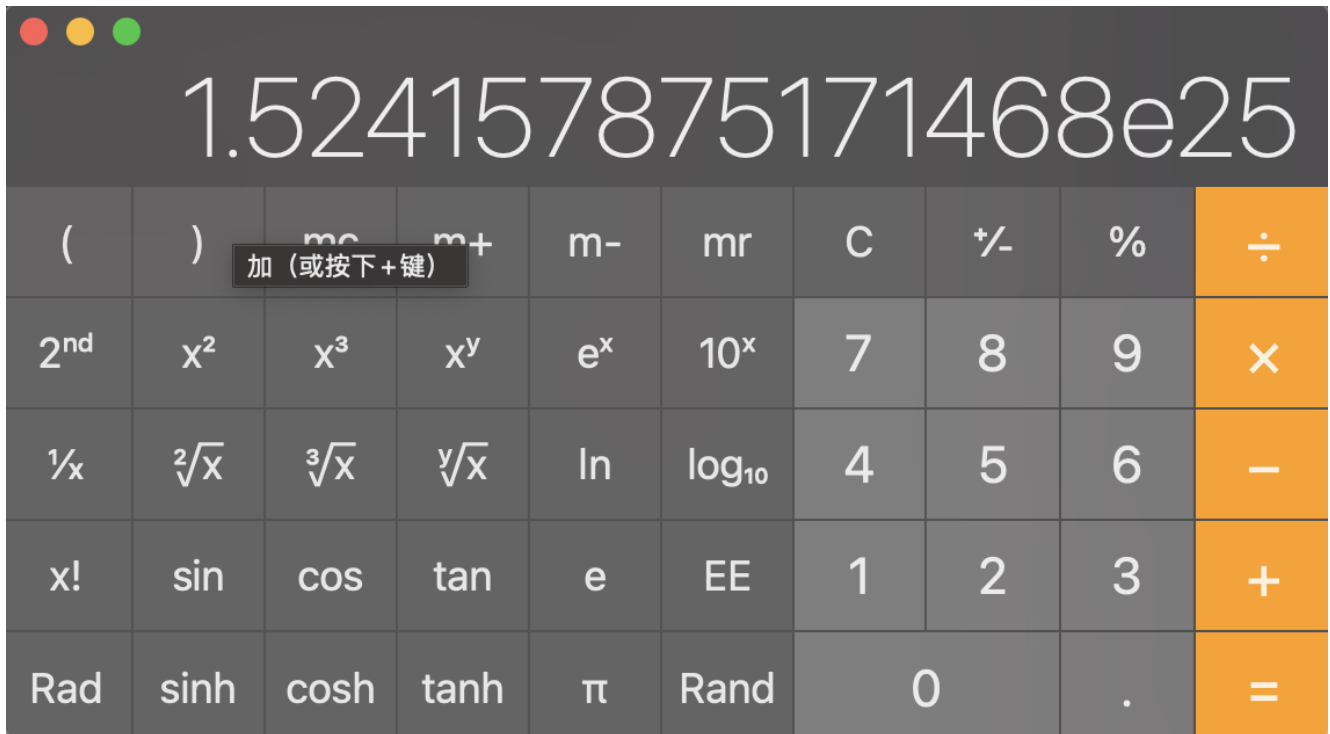
More features: 高效大数、方程四则运算。(even more precise than the calculator in mac /doge)

```
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
```

```
1234567890*12345678901234567
15241578751714677776253630
```

```
=====
please input something you want to calculate:
or you can type "q" or "quit" to exit
=====
```

```
zhousesicheng — bc — 80x24
Last login: Sat Oct 15 14:55:16 on ttys003
[(base) zhousesicheng@zhousesichengdeMacBook-Pro ~ % bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
[1234567890*12345678901234567
1524157875171467776253630
█
```



Part 04 - Difficulties & Solutions

指针乱飞

在上课讲内存管理之前，对于cpp中指针的理解还有所欠缺，于是出现了很多奇怪的bug。比如下面这两张图所示，在input函数当中打印出来x和y是1和2，但到了main函数，打印出来就变成了4098和1。

```

    result.length = length;
    result.scale = scale;
    result.digits = digits;
    cout << "digits inside are " << result.digits[0] << " " << result.digits[1] << endl;

    return result;
}

```

```

47     while (true)
48     {
49         //一些调试
50         Big_number big_num = input("1.2");
51         cout << "digits are " << big_num.digits[0] << " " << big_num.digits[1] << endl;
52         cout << "length is " << big_num.length << endl;
53         cout << "scale is " << big_num.scale << endl;
54         cout << "sign is " << big_num.sign << endl;
55         // string output = output(big_num);
56         // cout << output << endl;
57     }

```

问题 输出 调试控制台 终端 JUPYTER

```

bye
○ (base) zhousicheng@zhousichengdeMacBook-Pro project2 % g++ main.cpp&& ./a.out
welcome to calculator 2.0
This is free software with ABSOLUTELY NO WARRANTY.
number 0 = 1
number 0 = 1
number 1 = 2
digits inside are 1 2
digits are 4098 1
length is 2
scale is 1
sign is 1

```

后来通过老师上课讲解、查阅资料、和同学讨论终于稍微理解了一些指针，发现原来是两个数组指向了同一个地址。所以后面一个数组存了新的数字之后，原本的数字信息就丢失了，出现了野指针。为解决该问题，后来将原本Big_number结构体中的digits指针改成了定长数组。

```

welcome to calculator 2.0
This is free software with ABSOLUTELY NO WARRANTY.
number 0 = 1
number 0 = 1
number 1 = 2
digits inside are 1 2
address inside is 0x16f562f60
digits are 9091089 1
address outside is 0x16f562f60
length is 2
scale is 1
sign is 1
=====

```

```
18
19  struct Big_number
20  {
21      int length;
22      int scale;
23      int* digits;
24      bool sign;
25  };
26
```

```
27  struct Big_number
28  {
29      int length;
30      int scale;
31      int digits[256];
32      bool sign;
33  };
```

包括后面对Big_number进行压栈的时候也出现了指针相关问题，压栈时x和y还是不同的数字，出栈的时候就相同了。后来通过直接对Big_number中的定长数组赋值而不是指针指向临时数组，解决了这个问题。

```

366     }
367     else
368     {
369         dot = false;
370         if (state == IN)
371         {
372             Big_number ans;
373             ans.digits = cur;
374             ans.length = length;
375             ans.scale = scale;
376             ans.sign = true; //待定
377             cout << "ans = " << output(ans) << endl;
378             val.push(ans);
379             pos = 0;
380             length = 0;
381             scale = 0;
382         }
383

```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

JUPYTER

```

1+2
cur 0 = 1
ans = 1
cur 0 = 2
ans = 2
x = 2
y = 2
postfix is 1 2+
ans is 40
-----

```

```

384 // 四则运算
385 Big_number x, y;
386 if (c != ' ')
387 {
388     x = val.top();
389     cout << "x = " << output(x) << endl;
390     val.pop();
391     // val.pop();
392     y = val.top();
393     cout << "y = " << output(y) << endl;
394     val.pop();
395
396     switch (c)
397     {
398     case '+': //同号加法和异号减法 调用add
399         val.push(add(x, y));
400         single = false;
401         break;

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```

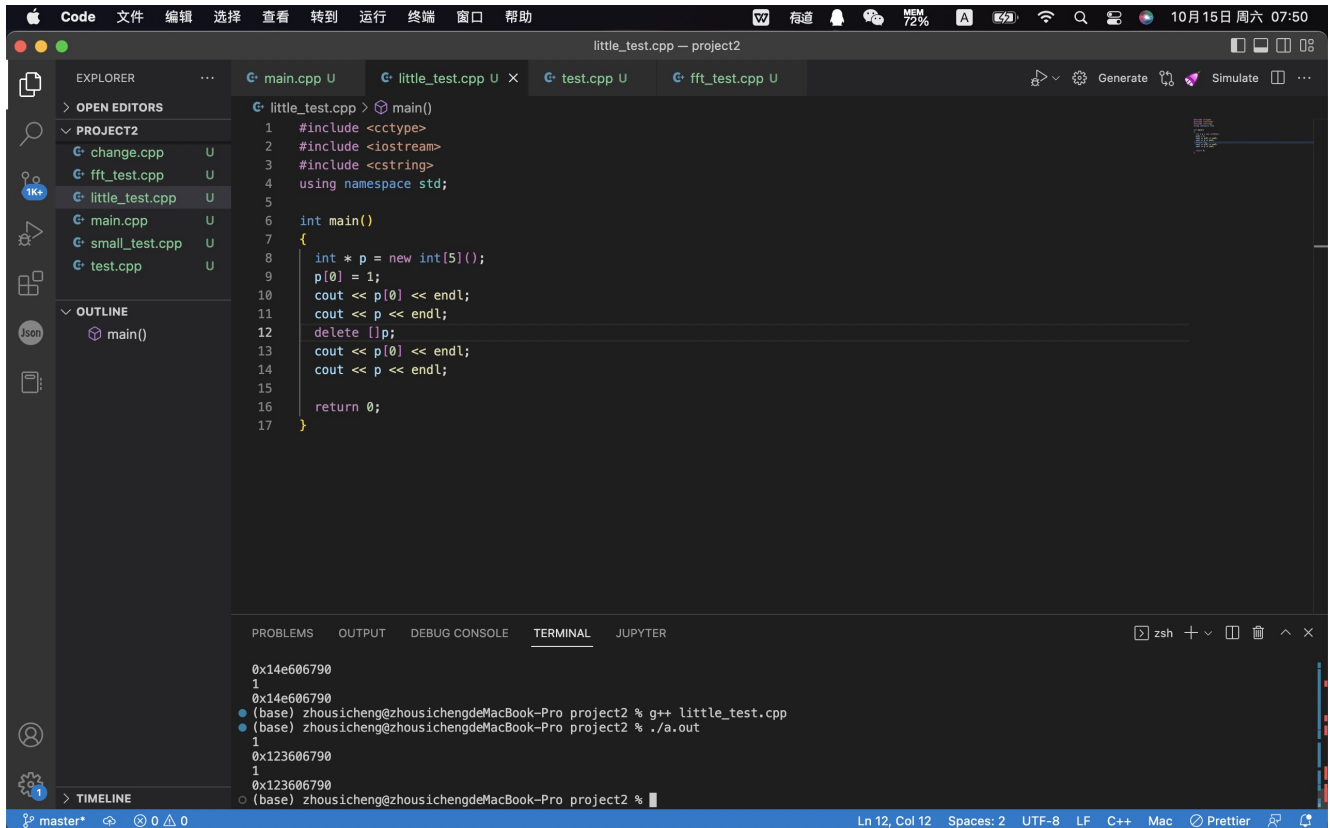
1+2
cur 0 = 1
ans = 1
cur 0 = 2
ans = 2
x = 2
y = 2
postfix is 1 2+
ans is 40

```

后面写加法减法的时候填充数组，实现的过程中越界了好多次也出现了奇怪的错误。想起来自己写java的时候也总是会报指针越界的问题，但是那里有报错，cpp就只有奇怪的输出结果。以后访问数组还是要好好检查边界。

并且，在一些测试中熟悉了释放内存的一些操作。如下图，发现delete这个指针之后，他的地址和原有的值并没有发生变化，后来通过查阅资料以及和同学讨论了解到，删除指针只是释放内存，也就是之后再申请内存时会申请到相同位置，但是原本的值并不会因为delete语句发生改变（至少在macos系统上是这个样子，看网上资料好像有的操作系统会把这个值变成一个随机数）。

如果想要删除指针，可以在delete之后将指针置成NULL。



```
1 #include <ctype>
2 #include <iostream>
3 #include <cstring>
4 using namespace std;
5
6 int main()
7 {
8     int * p = new int[5]();
9     p[0] = 1;
10    cout << p[0] << endl;
11    cout << p << endl;
12    delete []p;
13    cout << p[0] << endl;
14    cout << p << endl;
15
16    return 0;
17 }
```

```
0x14e606790
1
0x14e606790
(base) zhousicheng@zhousichengdeMacBook-Pro project2 % g++ little_test.cpp
(base) zhousicheng@zhousichengdeMacBook-Pro project2 % ./a.out
1
0x123606790
1
0x123606790
(base) zhousicheng@zhousichengdeMacBook-Pro project2 %
```

时间

写report到最后一部分difficulty，第一个想到的就是时间不够用，很多功能想实现但没来得及实现。一方面project确实在某些意义上是永远写不完的，另一方面加入功能需要动很多架构设定也说明代码本身扩展性不够好。害...且练且进步罢。下文简单列举一下想到了但是实在来不及实现的方法。

比如非法输入的报错，本来想的是在main函数里面设置成else，然后模仿bc的界面，在报错时带上错误的次数。但后来因为采用从后往前遍历，else用来处理变量赋值，就有点不知道去哪处理非法输入了。现在这部分在main函数中留着一个被注释掉的尸体。还有万一用户输入的时候喜欢敲空格怎么搞呢？去空格这一步也暂时没实现。

比如高精度除法，想到了可以通过减法进行模拟，想要加快效率的话可以对结果进行二分（就像那个仓促完成的sqrt函数那样）。但是写完方程、加法减法小数点对齐、乘法快速傅立叶变换、后缀表达式转化计算已经ddl烧眉毛，只能写了个最基本的除法。实际上在四则运算的实现过程中，因为方法内部已经足够复杂，所以打算把关于符号的判断和处理放在四则运算方法外部。但写到后来也发现没时间实现了。

刚开始研究计算器架构的时候还花了很长时间研究bc的架构，思考怎么实现自定义精度等等，后来也卷不动了。现在想想实现自定义精度也不算特别麻烦，只需要定义一个全局变量然后规范输出格式就行。但这好像比较bonus了，如果有多的时间还是好好处理一下上述其他问题。

开工第一天还花了半天研究git怎么使用，后来处于一种学会了但没完全学会的状态，还是选择了把所有方法实现全部放在同一个main.cpp文件当中。其实这并不是一个好的实现方式，因为方法实现过程中需要逐个调试，如果能在test.cpp当中调用调试会方便很多，但现在由于全部压在main.cpp当中，要调试需要注释很多东西改很多东西，然后到处cout，非常耗时。下次project一定好好文件管理。另外，因为所有代码写在一个main.cpp文件当中，所以没有使用makefile和cmake。下次一定。