计算表达式

计算表达式分三部分

> 重要一点是以下代码调用了用户自己实现的栈类（ImprovedStack.h）

1.提取

提取表达式中的运算符（operator)、运算对象（operand)和括号(parenthesis)。
具体如下

```
vector<string> split(const string &expression)
{
    vector<string> v;    //A vector to store split items as strings
    string numberString; //A numeric string

    for (unsigned i = 0; i < expression.size(); ++i)
    {
        if (isdigit(expression[i]))
            numberString.append(1, expression[i]); //Append a digit
        else
        {
            if (numberString.size() > 0)
            {
                v.push_back(numberString); //Store the numeric string
                numberString.erase();      //Empty the numeric string
            }

            if (!isspace(expression[i]))
            {
                string s;
                s.append(1, expression[i]);
                v.push_back(s); //Store an operator and parenthesis
            }
        }
    }
    //Store the last numeric string
    if (numberString.size() > 0)
    {
        v.push_back(numberString);
    }
    return v;
}
```

2.计算表达式

根据条件计算表达式
定义两个栈（operandStack、operatorStack）
程序从左向右扫描表达式，抽取运算数，操作符和括号

步骤1（phase1）扫描表达式

1.）如果抽取的是操作数，则压进栈operandStack。

```
            operandStack.push(atoi(tokens[i].c_str()));
```

2.)如果抽取的是" + "或" - "运算符，则处理operatorStack栈顶比" + "或者" - "更高级的或相同级别的所有运算符（即" + "，" - "," * "," / ")最后把抽到的运算度压进栈operatorStack。

```
/*
下面注意一点是，vector容器保存的是string，类似于"A"，"+"，"B",······
所以下面的token[i][0]的表示是一个字符 如：'A','+','B'，······
*/
if (tokens[i][0] == '+' || tokens[i][0] == '-')
        {
            //process all + , - , * , / in the top of the operator stack
            while (!operatorStack.empty() && (operatorStack.peek() == '+') || (operatorStack.pee
            {
                processAnOperator(operandStack, operatorStack);
            }

            // Push the + or - operator into operator stack
            operatorStack.push(tokens[i][0]);
        }
```

3.)如果抽到的是" * "或" / "，则处理operatorStack栈顶比" * "或者" / "更高级的或相同级别的所有运算符（即" * "," / ")最后把抽到的运算度压进栈
operatorStack。

> 具体代码类似上面，不重复了

4.)如果抽取的是" (",则将其压进operatorStack栈。

```
else if (tokens[i][0] == '(')
        {
            operatorStack.push('('); //Push '(' to stack
        }
```

5.)如果抽到的是") ",重复处理operatorStack栈顶的所有运算符直到遇到" ("。

```
else if (tokens[i][0] == ')')
        {
            //process all operators in the stack until seeing '('
            while (operatorStack.peek() != '(')
            {
                processAnOperator(operandStack, operatorStack);
            }

            operatorStack.pop();
        }
```

步骤2（phase2）清空栈

重复处理operatorStack栈顶所有运算符，直到operatorStack栈为空。

```
 //Phase 2: process all the remaining operators in the stack
    while (!operatorStack.empty())
    {
        processAnOperator(operandStack, operatorStack);
    }
```

3.根据运算符计算运算对象

1.）分别提取两个栈的元素
2.）从operatorStack栈顶提取运算符
3.）从operandStack栈顶提取两个运算对象（注意后提取出的是操作数，先提取出的是被操作数，详喜操作如下）
4.）将计算结果压回operandStack栈中。
具体操作如下

```
void processAnOperator(
    Stack<int> &operandStack, Stack<char> &operatorStack)
{
    char op = operatorStack.pop();
    int op1 = operandStack.pop();
    int op2 = operandStack.pop();
    if (op == '+')
        operandStack.push(op2 + op1);
    else if (op == '-')
        operandStack.push(op2 - op1);
    else if (op == '*')
        operandStack.push(op2 * op1);
    else if (op == '/')
        operandStack.push(op2 / op1);
}
```

附上 ImprovedStack.h

```
#ifndef IMPROVEDSTACK_H
#define IMPROVEDSTACK_H

template <typename T>
class Stack
{
private:
    T *elements;
    int size;
    int capacity;
    void ensureCapacity();

public:
    Stack();
    Stack(const Stack &);
    ~Stack();
    bool empty() const;
    T peek() const;
    void push(T value);
    T pop();
    int getSize() const;
};

template <typename T>
Stack<T>::Stack() : size(0), capacity(16)
{
    elements = new T[capacity];
}

template <typename T>
Stack<T>::Stack(const Stack &Stack)
{
    elements = new T[Stack.capacity];
    size = Stack.size;
    capacity = Stack.capacity;
    for (int i = 0; i < size; ++i)
    {
        elements[i] = Stack.elements[i];
    }
}

template <typename T>
Stack<T>::~Stack()
{
    delete[] elements;
}

template <typename T>
bool Stack<T>::empty() const
{
    return size == 0;
}

template <typename T>
T Stack<T>::peek() const
{
    return elements[size - 1];
}

template <typename T>
void Stack<T>::push(T value)
{
    ensureCapacity();
    elements[size++] = value;
}

template <typename T>
void Stack<T>::ensureCapacity()
{
    if (size >= capacity)
    {
        T *old = elements;
        capacity = 2 * size;
        elements = new T[size * 2];
        for (int i = 0; i < size; ++i)
        {
            elements[i] = old[i];
        }

        delete[] old;
    }
}

template <typename T>
T Stack<T>::pop()
{
    return elements[--size];
}

template <typename T>
int Stack<T>::getSize() const
{
    return size;
}

#endif
```