栈的应用

提前写个栈的头文件

```cpp
#include<iostream>
using namespace std;
template <class T>
class Stack {
public:
	Stack() {
		size = 20;
		stack = new T[size];
		top = -1;
	}
	Stack(int s) {
		if (s <= 0)cerr << "创建失败";
		size = s;
		stack = new T[size];
		top = -1;
	}
	~Stack() { delete stack; }
public:
	int getTop() { return top; }
	T getTopData() { return stack[top]; }
	bool isEmpty() {
		if (top == -1)return true;
		else return false;
	}
	void makeEmpty() {
		top = -1;
	}
	void expand() {
		T* temp = new T[size * 2];
		for (int i = 0; i <= top; i++) {
			temp[i] = stack[i];
		}
		stack = temp;
		return;
	}
	void push(T input) {
		if (top == size - 1) this->expand();
		stack[++top] = input;
		return;
	}
	void pop(T& savePop) {
		savePop = stack[top];
		top--;
		return;
	}
	void show() {
		for (int i = 0; i <= top; i++) {
			cout << stack[i] << endl;
		}
		return;
	}
```

```
private:
        T* stack;
        int top;
        int size;
};
```

表达式求值

前缀转后缀的过程中计算

- 双栈：一个放操作数，一个放运算符
- 操作数栈：
  - 遇到操作数就入栈
  - 弹出两个操作数，先弹出的是右操作数，后弹出的是左操作数
- 运算符的优先级：分为栈内优先级和入栈优先级
- 操作符栈：
  - 入栈优先级>栈顶元素的栈内优先级,直接入栈
  - 入栈优先级<栈顶元素的栈内优先级,栈顶元素出栈并和操作数栈弹出的两个元素进行运算，结果压入操作数栈，一直重复这样的操作直到栈外的那个操作符能入栈

代码

还需优化的地方：输入的操作数只能是个位数，局限性大，不过这并不是这个算法关注的主要问题，所以就暂时不管它了

```
#include<iostream>
#include"Stack.h"
using namespace std;
class Calculator {
public:
	Calculator() {
		operands = new Stack<double>;
		operators = new Stack<char>;
		operators->push('#');
		result = 0;
	}
	~Calculator() {
		delete operands;
		delete operators;
	}
public:
	void Clear() {
		result = 0;
		operands->makeEmpty();
	}
	double getResult() { return result; }
	bool SuffixAndCalculate(char* expression) {
		int i = 0;
		char ope = '#';
		while (expression[i] != '\0') {
			if (expression[i] >= '0' && expression[i] <= '9') {
				operands->push((double)expression[i] - 48);
			}
			else if (expression[i] == '+' || expression[i] == '-' || expression[i] =
				expression[i] == '/' || expression[i] == '(' || expression[i] ==
				//<=中的等于是因为右括号的栈外优先级等于左括号的栈内优先级
				while (OutStackPriority(expression[i]) <= InStackPriority(operat
					//若左右括号相遇，弹出左括号且不进行运算，比较结束，读取下一个
					if (expression[i] == ')' && operators->getTopData() == '
						operators->pop(ope);
						break;
					}
					//弹出栈顶，并运算（若运算失败，则返回false），字符再与新的栈
					operators->pop(ope);
					if (!DoOperator(ope))return false;
				}
				//直至优先级比栈顶高，才入栈(右括号不入栈)
				if (expression[i] != ')')operators->push(expression[i]);
			}//end if
			else {
				cerr << "请输入正确的算数表达式！" << endl;
				return false;
			}
			i++;
		}//end while
		//将操作符栈剩下的运算符弹出并运算
```

```
                while (operators->getTopData() != '#') {
                        operators->pop(ope);
                        if (!DoOperator(ope))return false;
                }
                operands->pop(result);
                return true;
        }
private:
        bool GetTwoOperands(double& left, double& right) {
                if (!(operands->isEmpty()))operands->pop(right);
                else {
                        cerr << "缺少右操作数,计算失败" << endl;
                        return false;
                }
                if (!(operands->isEmpty()))operands->pop(left);
                else {
                        cerr << "缺少左操作数，计算失败" << endl;
                        return false;
                }
                return true;
        }
        bool DoOperator(char ope) {
                double left, right;
                if (this->GetTwoOperands(left, right)) {
                        switch (ope) {
                        case '+':
                                (this->operands)->push(left + right);
                                break;
                        case '-':
                                (this->operands)->push(left - right);
                                break;
                        case '*':
                                (this->operands)->push(left * right);
                                break;
                        case '/':
                                (this->operands)->push(left / right);
                                break;
                        default:
                                break;
                        }//end switch
                        return true;
                }//end if
                else return false;
        }
        int OutStackPriority(char ope) {
                if (ope == '#')return 0;
                if (ope == '+' || ope == '-')return 2;
                if (ope == '*' || ope == '/')return 4;
                if (ope == '(')return 6;
                if (ope == ')')return 1;
        }
```

```
        int InStackPriority(char ope) {
                if (ope == '#')return 0;
                if (ope == '+' || ope == '-')return 3;
                if (ope == '*' || ope == '/')return 5;
                if (ope == '(')return 1;
                if (ope == ')')return 6;
        }
private:
        Stack<double>* operands;//存放操作数的栈
        Stack<char>* operators;//存放操作符的栈
        double result;
};
int main() {
        Calculator calculator;
        char* expression = new char[20];
        while (1) {
                cout << "输入表达式: ";
                cin >> expression;
                if (calculator.SuffixAndCalculate(expression))
                        cout << "结果是: " << calculator.getResult() << endl;
                calculator.Clear();
        }
        return 0;
}
```

二叉树的非递归遍历

注：大循环指判断遍历是否结束的循环

前序遍历

每一次大循环的逻辑：栈顶弹栈并访问，然后先将栈顶元素的右子树入栈，再将栈顶元素的左子树入栈

```
void PrePrintNoRe(TreeNode<T>* root) {
                Stack<TreeNode<T>*> s;
                TreeNode<T>* temp = root;
                s.push(root);
                while (!s.isEmpty()) {
                        s.pop(temp);
                        cout << temp->data;
                        if (temp->rightChild != nullptr)s.push(temp->rightChild);
                        if (temp->leftChild != nullptr)s.push(temp->leftChild);
                }
                return;
}
```

中序遍历

每一次大循环的逻辑：一路将左子树入栈，然后弹出栈顶并访问，最后将结点赋给当前结点的右孩子

```
void MidPrintNoRe(TreeNode<T>* root) {
                Stack<TreeNode<T>*> s;
                TreeNode<T>* temp = root;
                while (temp!=nullptr||!s.isEmpty()) {//temp!=nullptr是为了在栈空时能进入循环压入第-
                        while (temp != nullptr) {
                                s.push(temp);
                                temp = temp->leftChild;
                        }
                        s.pop(temp);
                        cout << temp->data;
                        temp = temp->rightChild;
                }
}
```

后序遍历

需要两个栈，一个放结点，一个放对应结点的标识（如何对应：两个栈同时进，出栈）
标识为0，未进行遍历；为1，左子树已经遍历；为2，左右子树均已遍历。只有标识为2的结点才能出栈并访问
每一次大循环的逻辑：若结点标识为0，则一路将左子树入栈，并一路给标识，到头了就判断有无右孩子，若有则给标识，右孩子进栈，其对应的标识进栈；若无，则结点栈，标识栈均弹栈，并访问

```
void PostPrintNoRe(TreeNode<T>* root) {
                Stack<TreeNode<T>*> s;
                Stack<int> flag;
                int popFlag;
                TreeNode<T>* temp = root;
                s.push(root);
                flag.push(0);
                while (!s.isEmpty()) {
                        if (flag.getTopData()==0) {
                                while (temp->leftChild != nullptr) {
                                        s.push(temp->leftChild);
                                        temp = temp->leftChild;
                                        flag.pop(popFlag);
                                        flag.push(1);
                                        flag.push(0);
                                }
                                flag.pop(popFlag);
                                flag.push(1);
                        }
                        if (flag.getTopData()!=2&&temp->rightChild != nullptr) {
                                s.push(temp->rightChild);
                                flag.pop(popFlag);
                                flag.push(2);
                                temp = temp->rightChild;
                                flag.push(0);
                        }
                        else {
                                s.pop(temp);
                                cout << temp->data;
                                temp = s.getTopData();
                                flag.pop(popFlag);
                        }
                }
}
```