

Tongji University

Data Structure Course Curriculum Design Document

Task4: Arithmetic expression solving

Author:

嘉杰 李

Jiajie Li

Supervisor:

颖 张

Prof. Ying Zhang

A companion use documentation for a program submitted in fulfillment
of the requirements for the Data Structure curriculum design

in the

School of Software Engineering, Tongji



December 27, 2019

“Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism.”

Dave Barry

Contents

1	Analysis	1
1.1	Background Analysis	1
1.2	Functional Analysis	1
2	Design	3
2.1	Data structure design	3
2.2	Member function design	3
2.3	System design	5
3	Implementation	7
3.1	Infix to suffix Function	7
3.1.1	Infix to suffix core code	7
3.1.2	Infix to suffix flowchart	8
4	Test	9
4.1	Function Tests	9
4.1.1	6 operands	9
4.1.2	Nested parentheses	9
4.1.3	Operator has more than one integer and non-integer occurrences	9
4.1.4	Operands have positive or negative signs	10
4.1.5	Only one number	10
A	Frequently Asked Questions	11
A.1	Why this document looks so sparse?	11
A.2	Why is there header files in this project?	11

Chapter 1

Analysis

1.1 Background Analysis

Arithmetic expressions have the form of prefix notation, infix notation, and suffix notation. The arithmetic expressions used daily are infix notation, that is, the binary operator is located between two operands. Because of the problem of the precedence of operators in infix expressions and the problem of changing the operation order by adding parentheses, for compilers, infix expressions are generally not used. The solution is to use suffixes.

1.2 Functional Analysis

The infix expression needs to be converted into a suffix representation. For example, the infix represents $A + B * (C - D) - E / F$, and the suffix corresponding to it is $ABCD-* + EF /-$.

Chapter 2

Design

2.1 Data structure design

As described in the functional analysis above, an array-based sequential stack can be used to convert the infix representation of an expression to a suffix representation.

2.2 Member function design

Stack Class:

```
template <class T, class Sequence=list<T> >
class stack {
protected:
    Sequence c;
public:
    bool empty() const {return c.isEmpty();}
    int size() const {return c.len;}
    T top() {return c.isEmpty()? -1: c.hnode->prev->data;}
    void push(const T& x) {c.push_back(x);}
    void pop() {c.del(c.len-1);}
};
```

ListNode Class:

```
template <class T> struct __listNode {
    __listNode<T>* prev;
    __listNode<T>* next;
    T data;
};
```

Linked List Class:

```
template <class T> class list{
private:
    typedef __listNode<T> list_node;
    typedef list_node* link_type;
    link_type get_node(){
        link_type p=(link_type)malloc(sizeof(list_node));
```

```

        p->next=NULL;
        p->prev=NULL;
        return p;
    }
    void put_node(link_type p){
        p->prev=NULL;
        p->next=NULL;
        p=NULL;
        free(p);
    }
    link_type create_node(const T& x){
        link_type p = get_node();
        p->data = x;
        return p;
    }
    void destroy_node(link_type p){
        put_node(p);
    }
    void __insert(link_type p, const T &x){
        len++;
        link_type temp = create_node(x);
        p->next->prev = temp;
        temp->next = p->next;
        temp->prev = p;
        p->next = temp;
    }
    void __del(link_type p){
        len--;
        p->prev->next=p->next;
        p->next->prev=p->prev;
        destroy_node(p);
    }
    link_type loc(int pos){
        pos+=1;
        if (pos>len) return NULL;
        link_type temp = hnode;
        while (pos--) temp=temp->next;
        return temp;
    }

public:
    list();
    ~list();
    int Class_test(T temp);
    int push_back(const T& x) {__insert(hnode->prev, x); return 0;}
    int empty();
    int plist();
    bool isEmpty() const {return hnode==hnode->next;}

```



```
int del(int pos);
int insert(int pos, const T &x);
int modify(int pos, const T &x);

int len;
link_type hnode; //head node
};
```

Queue class:

```
template <class T, class Sequence=list<T> >
class queue {
protected:
    Sequence c;
public:
    bool empty() const {return c.isEmpty();}
    int size() const {return c.len;}
    T front() {return c.isEmpty()?-1:c.hnode->next->data;}
    void push(const T& x) {c.push_back(x);}
    void pop() {c.del(0);}
};
```

2.3 System design

The system gives infix expressions that separate different objects with spaces in a line according to user input. It can include +, -, *, /, -, *, /, and left and right parentheses. The expression does not exceed 20 characters (excluding spaces), Which is converted into a suffix expression and output. The operand and each operator are separated by a space, and there is no extra space at the end.

Chapter 3

Implementation

3.1 Infix to suffix Function

3.1.1 Infix to suffix core code

```

for (int i = 0; i < s.length(); ++i) {
    if (isNum(s[i])) {
        double temp = s[i] - '0';
        while ((i + 1) < s.length() && isNum(s[i + 1])) {
            ++i;
            temp = temp * 10 + s[i] - '0';
        }
        if (s[i + 1] == '.') {
            ++i;
            double temp2 = 0;
            while ((i + 1) < s.length() && isNum(s[i + 1])) {
                ++i;
                temp2 = temp2 * 10 + s[i] - '0';
            }
            while (temp2 > 1) temp2 = temp2 / 10;
            temp += temp2;
        }
        data.push(temp);
    } else {
        if (s[i] == ')') {
            while (op.top() != '(') calculate();
            op.pop();
        } else if (s[i] == '(') op.push(s[i]);
        else {
            while (!op.empty() && lev[op.top()] >= lev[s[i]])
                calculate();
            op.push(s[i]);
        }
    }
}
//op.c.plist();
//data.c.plist();
//cout << endl;

```

3.1.2 Infix to suffix flowchart

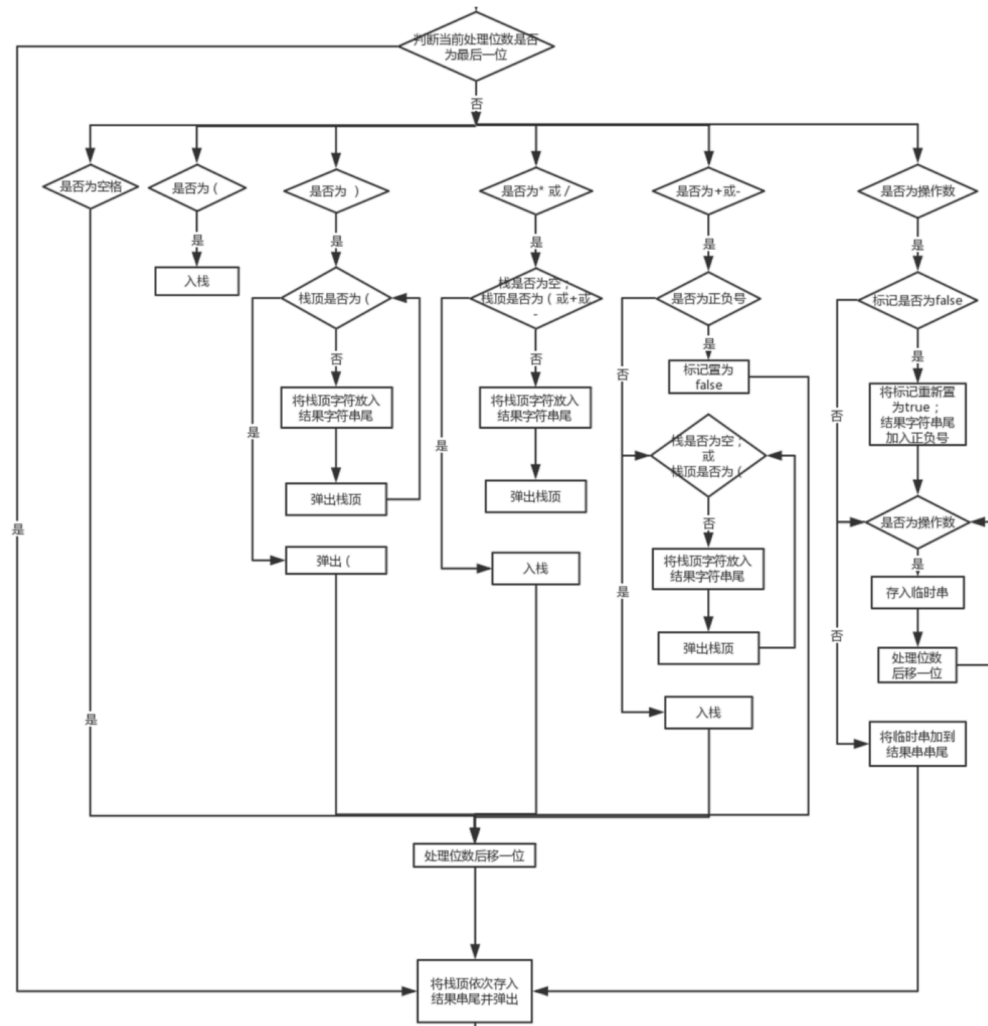


Figure 3.1: Infix to suffix flowchart

Chapter 4

Test

4.1 Function Tests

4.1.1 6 operands

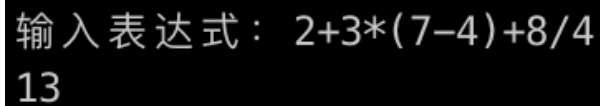
Test Case:

$2+3*(7-4)+8/4$

Expected Result:

13

Program Result:



输入表达式: $2+3*(7-4)+8/4$
13

Figure 4.1: Screenshots

4.1.2 Nested parentheses

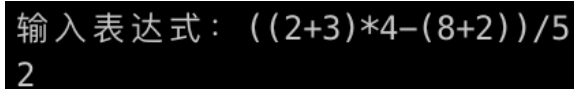
Test Case:

$((2+3)*4-(8+2))/5$

Expected Result:

2

Program Result:



输入表达式: $((2+3)*4-(8+2))/5$
2

Figure 4.2: Screenshots

4.1.3 Operator has more than one integer and non-integer occurrences

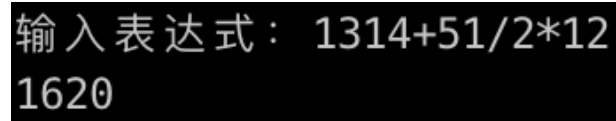
Test Case:

$1314+51/2*12$

Expected Result:

1620

Program Result:



```
输入表达式： 1314+51/2*12
1620
```

Figure 4.3: Screenshots

4.1.4 Operands have positive or negative signs

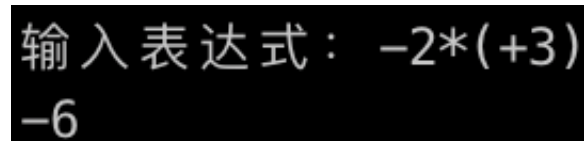
Test Case:

$-2*(+3)$

Expected Result:

-6

Program Result:



```
输入表达式： -2*(+3)
-6
```

Figure 4.4: Screenshots

4.1.5 Only one number

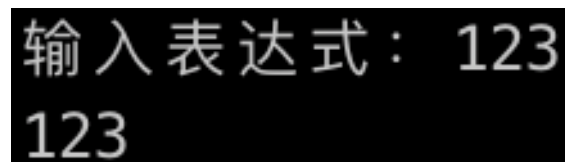
Test Case:

123

Expected Result:

123

Program Result:



```
输入表达式： 123
123
```

Figure 4.5: Screenshots

Appendix A

Frequently Asked Questions

A.1 Why this document looks so sparse?

This document is built using \LaTeX and is arranged in book format. There may be blank pages before and after each chapter.

A.2 Why is there header files in this project?

I wrote basic stl header files, such as `vector.h`, etc. Except for the header files I wrote, the only header files used in all projects are `iostream` and `string.h`