****

**Tongji University**

**数据结构课程设计实验报告**

**004-表达式转换**

专 业： 软 件 工 程

指导教师： 张 颖

学 号： 2 1 5 3 0 6 1

姓 名： 谢 嘉 麒

1. **项目概述**
   1. **项目背景**

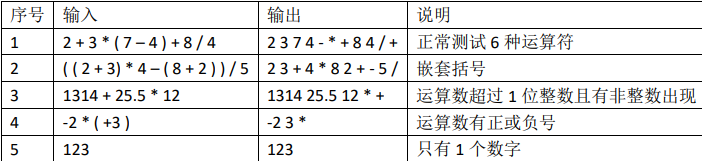
数据在一个计算式中，则称数据存在计算关系。而计算作为一个亘古不变的命题，对算力的开发，也成为当前需要发展的迫切问题。计算机的出现使计算能力有了质的飞越，而如何更进一步探索适合计算机语言的计算方法，优化算力，则需要不断探索与开发。本次项目要解决的就是基于计算需求而产生的表达式转换问题。

* 1. **项目要求**

本次项目为基于C++语言下的中缀表达式与后缀表达式之间的转换。主要需要实现用户手动键入中缀表达式，程序自动输出其对应的后缀表达式。表达式对象间以空格间隔作为标识，需要考虑到用户的使用需求与人性化的输入输出处理内容。

* + 1. **输入要求**

1. 输入为一行，其中给出以空格分隔不同对象的中缀表达式；
2. 运算符包括含+, -, \*, /, -, \*, /以及左右括号；
3. 表达式不超过 20 个字符（不包括空格）
   * 1. **输出要求**
4. 输出为一行，其中给出转换后的后缀表达式；
5. 要求不同对象（运算数，运算符号）之间以空格分隔；
6. 结尾不得有多余空格。
   * 1. **测试用例及输入输出示例**

****

* 1. **需求分析**

从中缀表达式和后缀表达式的特点入手，分析本次项目的需求：

1. 任何一个表达式都是由操作数（亦称运算对象）、操作符（亦称运算符）和分界符组成；
2. 中缀(infix)表示：

<操作数> <操作符> <操作数> 例如A + B

1. 后缀(postfix)表示：

<操作数> <操作数> <操作符> 例如A B +

* + 1. **完善项目功能**

项目要求以两行分别实现中缀表达式的键入和后缀表达式的生成输出，而考虑到用户的使用体验和需求优化，故可以在程序中增加适当的问候语句和是否进行多次试验的询问语句。具体可以用输入输出流结合while循环实现，方便用户使用。

* + 1. **保证执行效率**

保证用户体验的重要环节之一就是保证代码的执行效率。本次项目为了保证程序的运行效率以及避免不必要的空间浪费，未使用递归方法，而是采用了空间复杂度较低的循环进行实现。

* + 1. **注重代码健壮性**

程序需要在用户输入非法字符或者不合理数据的时候进行报错提示，确保正确性，体现代码健壮性。

* + 1. **代码复用的处理**

通过增加可复用的函数来提高代码的复用能力，使整个程序简洁美观，减少重复率，降低风险，便于修改。

1. **项目设计思路**
   1. **项目主要功能实现**
      1. **主程序内用户交互的设计**

本次项目主要将用户交互包括：

1. 程序开始时打印“欢迎使用表达式转换程序！”；
2. 程序运行时用cin.getline()将用户键入的表达式存储在expression静态数组中；
3. 一次转换结束后提示“是否继续运算?（y/n）”；
4. 如果用户输入y，则程序继续运行，while循环跳转至开头，重新等待用户键入中缀表达式；如果用户输入n，则程序结束；如果用户输入不合法或者不合理字符，则提示“输入错误,请重新输入!”，并跳转至输入语句等待重新输入；
5. 如果用户结束程序，则程序提示“欢迎下次再次使用!”；
6. 通过应用错误处理来提高用户的体验；
   * 1. **中缀表达式转化算法的设计**

根据中缀表达式和后缀表达式的特点，可以考虑利用栈来实现将中缀表达式转换为后缀表达式。

* 1. **数据结构设计思路**

根据相关算法的分析可知，本次项目需要建立栈的数据结构来实现中缀表达式转换成为后缀表达式。而出于对空间复杂度和算法运行效率的考虑，选择利用链表作为栈的底层存储设计。

* + 1. **链表类的设计与使用**

本次实验承用之前完成的链表类作为栈类的底层数据结构。且使用双向链表在牺牲空间复杂度的条件下降低时间复杂度，提高链表增删查改的速度，减少不必要的遍历和操作，优化用户体验。

本次链表类的设计共包含以下几个部分：

1. 设置linkNode结构存放结点内容，包括：

Type data;

linkNode<Type>\* next, \* prev;

1. 设置迭代器（游标）类方便实现链表类的自增，自减和运算符运算：
2. 默认构造函数

iterator()

1. //构造函数

iterator(\_iterator datas);

1. //读数据操作

Type operator\*()

1. //比较运算符

inline bool operator==(const iterator& v)const

inline bool operator<(const iterator& v)const

inline bool operator>(const iterator& v)const

inline bool operator<=(const iterator& v)const

inline bool operator>=(const iterator& v)const

inline bool operator!=(const iterator& v)const

1. //自增运算符

//前置自增

inline linkList<Type>::iterator& operator++()

//后置自增

linkList<Type>::iterator operator++(int)

1. //自减运算符

//前置自减

inline linkList<Type>::iterator& operator--()

//后置自减

linkList<Type>::iterator operator--(int)

1. //后面加上一个数

linkList<Type>::iterator operator+(int index)

1. //前面加上一个数

friend typename linkList<Type>::iterator operator+()

1. //后面减去一个数

linkList<Type>::iterator operator-(int index)

1. //自加

linkList<Type>::iterator& operator+=(int index)

1. //自减

linkList<Type>::iterator& operator-=(int index)

1. //计算两个迭代器之间的距离

int operator-(linkList<Type>::iterator i)

1. 设定链表类成员函数
2. //求解链表长度

inline int size()const;

1. //判断链表是否为空

inline bool Isempty()const;

1. //在尾部插入新数据

void push\_rear(Type data);

1. //在头部插入新数据

void push\_front(Type data);

1. //查找指定数据，返回迭代器

linkList<Type>::iterator find(const Type& data)const;

1. //删除指定迭代器对应的结点

Type remove(linkList<Type>::iterator index);

1. //删除尾结点

void erase\_rear();

1. //删除第一个结点

void erase\_front();

1. //清空链表

void clear();

1. //插入结点

bool insert(linkList<Type>::iterator index, Type i);

1. //返回指定位置的一系列函数

//起始位置迭代器

inline linkList<Type>::iterator begin();

inline const linkList<Type>::iterator begin()const;

//最后位置迭代器

inline linkList<Type>::iterator end();

inline const linkList<Type>::iterator end()const;

//重载等号运算符

linkList<Type>& operator=(const linkList<Type>& i);

考虑到栈LIFO(Last In First Out)的特点，链表类中设计了头指针linkNode<Type>\* head、尾指针linkNode<Type>\* rear和链表长度int \_size三个类成员，使得各种操作的时间复杂度进一步降低，经计算该设计下链表的增删查改操作时间复杂度均为O(n);

* + 1. **栈类的设计与使用**

栈（Stack）作为一种运算受限的线性表，规定只能在尾部（栈顶）进行插入和删除，而相应的栈的头部则被称为栈底；

若想向栈中添加新元素，则需要（也仅能）将新元素置于原栈顶元素之上，使其成为新的栈顶元素，这一过程被称为入栈或压栈（Push）；

而若想从栈中删除元素，则需要找到栈顶元素并将其删除，使其下一个元素成为新的栈顶，这一过程被称为出栈或退栈（Pop）；

根据以上性质，可以分析得到栈是一种具有LIFO（Last In First Out）性质的数据结构；

本次课程设计中简要定义并实现了栈类；

1. 定义链式栈存储数据，包括：

linkList<Type> data;

1. 定义类成员函数来进行各种栈操作

可以计算得到以下函数的时间复杂度均为O（1）

1. //默认构造函数

Stack();

1. //判断是否栈空

inline bool Isempty()const;

1. //返回栈的大小

inline int size()const;

1. //入栈

void push(const Type& i);

1. //出栈

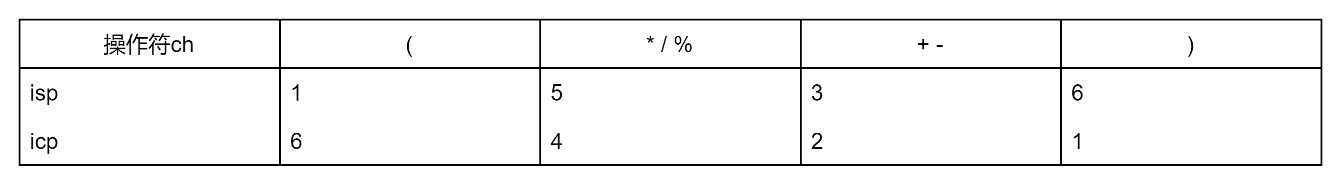
void pop();

1. //获取栈顶元素

const Type& top()const;

1. **算法实现和框架**
   1. **表达式转换算法translation()的实现**

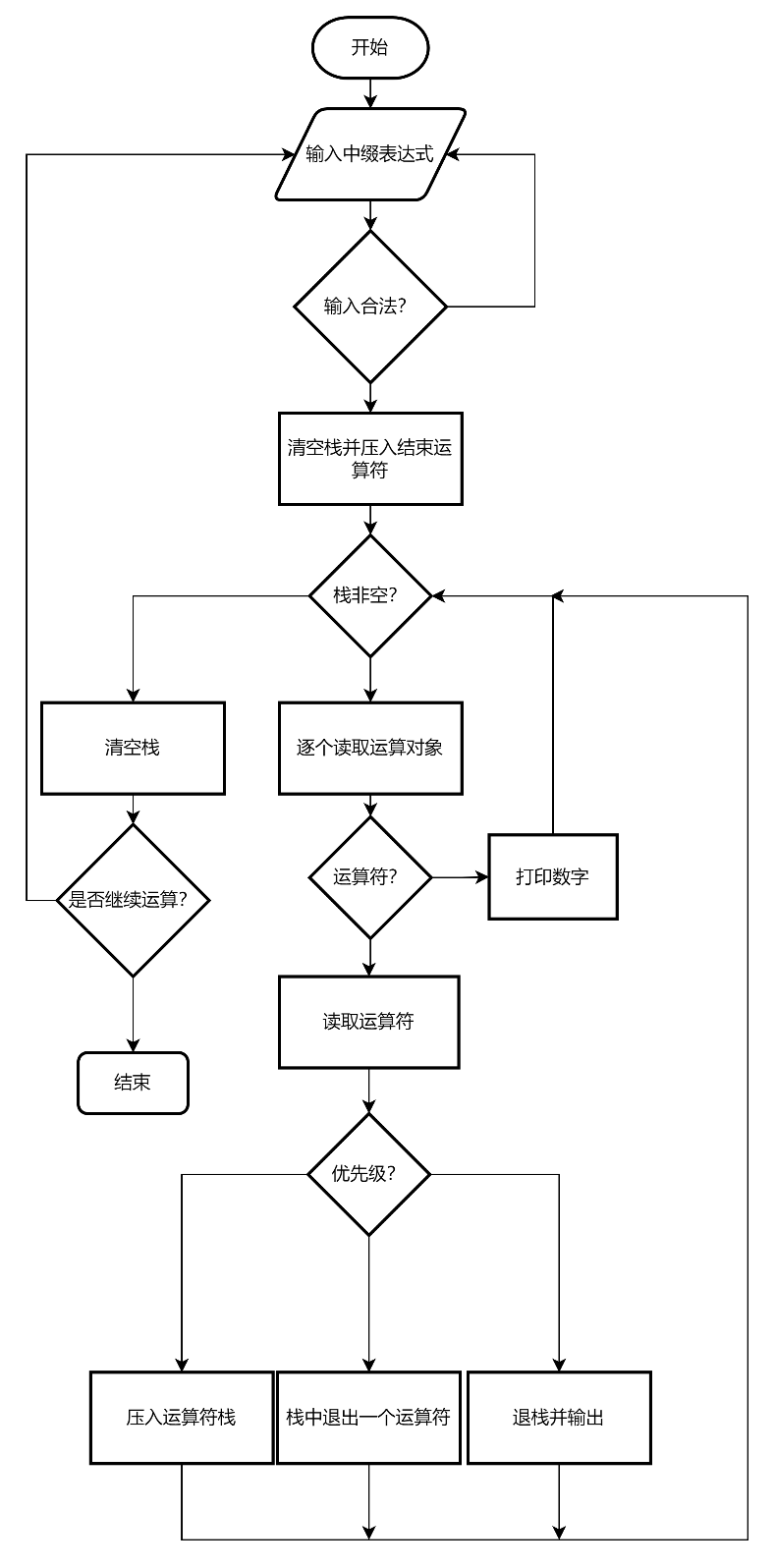
中缀表达式中操作符的优先级和括号使得求值过程复杂化，因此在转换时需要考虑各个操作符的优先级，整理如下：

****

Isp为栈内（in stack priority）优先数，icp为栈外（in coming priority）优先数。左括号栈外优先数最高，进栈后优先数最低，其他操作符进入栈中后优先数加一。

具体函数实现如下：

1. 定义栈ope，并每次查找运算对象（遇到空格停止）
2. 如果读入的是运算数则直接输出，继续读入下一个字符ch
3. 如果读入ch为操作符，则判断ch的优先级icp与当前栈顶操作符op的优先级isp
4. 若icp(ch) > isp(op)，令ch进栈，读入下一个字符ch
5. 若icp(ch) < isp(op)，退栈并输出
6. 若icp(ch)==isp(op)，退栈但不输出，若退出的是’(‘则读入下一个字符
7. 重复操作直到中缀表达式结束，若栈不为空则将栈中内容输出；
   1. **程序整体流程图**

****

较大

相等

较小

否

是

是

否

否

是

是

否

* 1. **表达式输入与读取处理**

本次项目的表达式可以通过字符型数组或者字符串（string）类存储，而考虑到程序需要将用户输入的空格也在相应位置存储起来，故选择采用字符型数组来作为存储类型。

定义char expression[]后，需要考虑如何读取空格的问题，此处采用cin函数集中的cin.getline（）函数进行读取：

cin.getline 允许读取包含空格的字符串。它将继续读取，直到它读取至最大指定的字符数，或直到按下了回车键。

1. **心得体会**
   1. **对于前缀、中缀和后缀表达式的学习**

本次项目加深了笔者对于前缀、中缀和后缀表达式的理解，人工计算一般更适应于中缀表达式，而程序计算则更多地需要后缀表达式。通过这一项目，可以将两种表达式之间建立转换关系，使二者相互联系，更加具体。

* 1. **对于带空格的字符串的输入输出处理**

通过本次项目，笔者学习了处理的带空格的字符串的方法，自主学习了字符串集和cin函数集，提高了自主学习能力，也在实践项目中多次尝试不同的方法，收获新知识。

1. **程序测试示例**

由于表达式较为复杂，所以测试主要包括正常表达式，多层括号嵌套表达式，运算数超过一位且有非整数运算数表达式，运算数有正负号的表达式，只有一个数字的表达式：

（1） 测试用例1：2 + 3 \* ( 7 – 4 ) + 8 / 4

预期结果：2 3 7 4 - \* + 8 4 / +

（2） 测试用例2：( ( 2 + 3 ) \* 4 – ( 8 + 2 ) ) / 5

预期结果：2 3 + 4 \* 8 2 + - 5 /

（3） 测试用例3：1314 + 25.5 \* 12

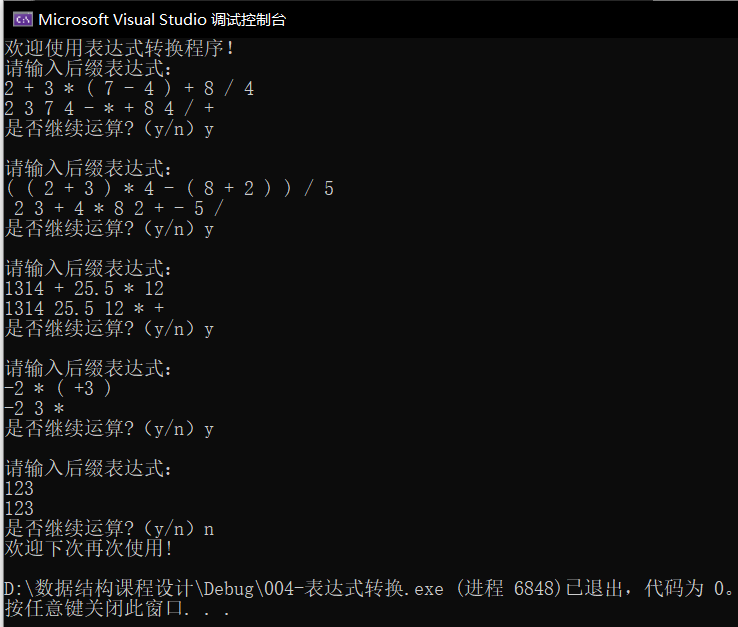
预期结果：1314 25.5 12 \* +

（4） 测试用例4：-2 \* ( +3 )

预期结果：-2 3 \*

（5） 测试用例5：123

预期结果：123

****

经过测试验证得程序可以得到预期结果，中缀表达式转换为后缀表达式成功。