同济大学
TONGJI UNIVERSITY

数据结构课程设计

项目说明文档

表达式转换系统

姓　　名：　　马小龙

学　　号：　　2353814

学　　院：计算机科学与技术学院（软件学院）

专　　业：　　软件工程

指导教师：　　张颖

二〇二三年十一月十九日

目录

第 1 章 项目分析

1.1 项目背景分析

在计算机科学中，表达式的求值是编译器设计中的一个基础问题。尤其是在计算机对算术表达式进行计算时，表达式的表示方式对其处理效率具有重要影响。人类通常使用中缀表达式，这种表示方式在书写和阅读上非常直观，但它不适合计算机的高效处理。相比之下，后缀表达式（也称为逆波兰表示法）不依赖于括号来表示操作符的优先级，且无需考虑操作符之间的顺序问题，因而更适合计算机的处理。通过将中缀表达式转换成后缀表达式，可以简化表达式求值过程，提高计算机处理表达式的效率。此转换过程在编译器、表达式求值器等程序中广泛应用，因此设计一个能够将中缀表达式转换为后缀表达式的程序，能够为后续的算术计算、表达式求值等任务打下基础。

1.2 项目需求分析

本项目的目标是设计并实现一个程序，能够接收一个中缀算术表达式，验证其合法性，并将其转换为对应的后缀表达式。根据背景分析，本项目的需求如下：

输入要求：用户通过命令行界面输入一个整数算术表达式，表达式由数字、运算符（+、-、*、/）和括号组成，要求不同对象（运算数，运算符号）之间以空格分隔，且表达式长度(不含空格)不超过 20 个字符。

解析与验证：程序需要能够正确解析并验证输入的中缀表达式，确保括号匹配，运算符正确使用，并处理表达式中的数字、运算符和括号。

转换功能：程序将输入的中缀表达式转换为后缀表达式，并输出该后缀表达式，运算符与操作数之间用空格分隔。

计算功能：程序能够根据后缀表达式计算其结果。

异常处理：程序应实现健全的异常处理机制，防止用户输入错误时导致系统崩溃或计算结果错误。

技术需求：程序应支持 Windows 和 Linux 等操作系统，采用 C++等主流编程语言，采用合适的数据结构。

1.3 项目功能分析

为满足上述需求，本项目将涉及多个核心功能模块，分别处理表达式的解析、转换、计算以及异常处理等任务。以下是对项目功能的详细分析：

1.3.1 构建表达式二叉树功能

本功能的目的是将输入的中缀表达式转换为二叉树形式。表达式二叉树是将算术表达式中的每个运算符作为树的结点，操作数作为叶子结点，从而体现出操作符的优先级及括号的层级关系。中缀表达式经过转换后，可以通过二叉树结构清晰地表示操作符之间的关系，为后续的遍历提供便利。

1.3.2 后缀表达式获取与输出功能

该功能是项目的核心之一，主要实现将中缀表达式转换为后缀表达式（逆波兰表示法）。通过后序遍历已经建立的二叉树，可以实现将中缀表达式改为后缀表达式。

1.3.3 异常处理功能

在实现过程中，需要考虑到用户输入的各种异常情况。例如，用户可能输入错误的表达式，如缺失括号、运算符位置不合法等，程序应能够捕获这些异常并给出提示，而不是让程序崩溃。异常处理还包括对无效字符的检查，以及对运算符优先级的管理，确保程序在任何情况下都能稳定运行并返回准确的结果。同时，还要考虑各种内存分配问题。

第 2 章 项目设计

2.1 数据结构设计

本项目主要采用三种数据结构，分别为二叉树、栈、以及 String（string 类型自己实现）。

（一）二叉树是一种常见的数据结构，特别适合表示数学表达式，尤其是中缀表达式转换为后缀表达式后的结构。通过二叉树，可以直观地表示运算符的优先级和操作数之间的关系。

1.表达式优先级：在二叉树中，根结点通常是运算符，而左右子树分别表示运算符的操作数。二叉树的层级结构自然地反映了操作符的优先级和括号的层次结构，能有效地组织中缀表达式中的运算顺序。

2.递归遍历：二叉树的遍历方式（前序、中序、后序遍历）能够方便地转换为不同形式的表达式（中缀、前缀、后缀表达式）。通过中序遍历，可以恢复原始的中缀表达式；通过前序或后序遍历，则可以输出相应的前缀和后缀表达式。

3.结构清晰：二叉树为表达式提供了一个清晰的结构，易于理解和操作，尤其是在处理复杂的嵌套表达式时。

（二）栈是一种非常适合用于表达式求值和转换的线性数据结构，尤其在处理中缀表达式转换为后缀表达式时，栈的作用至关重要。栈的先进后出特性能够帮助管理操作符的优先级以及括号匹配。

1.运算符优先级管理：在中缀表达式转换为后缀表达式的过程中，栈用来暂时存放运算符。当遇到新的运算符时，栈会根据运算符的优先级来判断是否需要将栈中的运算符弹出并输出。这种“推送和弹出”的方式恰好符合运算符优先级的管理。

2.括号匹配：栈也能有效处理括号。遇到左括号时，将其压栈；遇到右括号时，则将栈中的运算符弹出，直到遇到左括号，从而实现括号内的运算优先进行、或检查表达式中括号数目位置是否合理。

3.后缀表达式生成：栈的使用使得将操作数和运算符按正确顺序输出成为可能。当栈中只剩下一个运算符时，该表达式的后缀形式就被成功地生成。

（三）使用 string 类，主要时因为无法使用字符数组将模板示例化，将字符数组转化为 string 类，可以实现模板化二叉树结点结构体。

2.2 结构体与类设计

2.2.1 链式栈结点（Node）设计

2.2.1.1 概述

Node 用于储存信息，包括结点储存的具体信息内容和下一结点的地址。

2.2.1.2 结构体定义

```
template <typename T>
struct Node {
    T data;
    Node<T>* next;
    Node(Node<T>* ptr = nullptr);
    Node(const T& item, Node<T>* ptr = nullptr);
};
```

2.2.1.3 数据成员

T data: 数据域，存储结点的数据

Node<T>* next: 指针域，指向下一个结点的指针

2.2.1.4 构造函数

Node(Node<T>* ptr = nullptr);

构造函数，初始化指针域。

Node(const T& item, Node<T>* ptr = nullptr);

构造函数，初始化数据域和指针域。

2.2.2 链式栈（Stack）设计

2.2.2 链式栈（Stack）设计

2.2.2.1 概述

该通用模板类 Stack 用于表示链式栈。能够实现进栈、弹出、取栈顶元素、获得栈的大小等基本操作；采用链式栈而不采用顺序栈，可以动态扩展、缩小栈的大小，提高内存的利用率。

2.2.2.2 Stack 类定义

```
template <typename T>
class Stack {
public:
    Stack();
    ~Stack();
    void Push(const T& x);
    T Pop();
    T getTop();
    bool IsEmpty();
    int getSize();
    void makeEmpty();
private:
    Node<T>* top;
    int size;
};
```

2.2.2.3 私有数据成员

Node<T>* top: 指向栈顶结点的指针

int size：栈的大小

2.2.2.4 构造函数与析构函数

Stack();

默认构造函数，将栈顶指针指向 nullptr。

~Stack();

析构函数，释放链式栈的内存资源，包括所有结点的内存。

2.2.2.5 公有成员函数

void Push(const T& x);

向栈中添加元素。

T Pop();

弹出栈顶元素，并返回其值。

T getTop();

获取栈顶元素的值。

bool IsEmpty();

检查栈是否为空。

int getSize();

获取栈的大小。

void makeEmpty();

通过遍历堆栈的结点，释放其内存，并将 top 置为 nullptr 来实现将栈清空。

2.2.3 二叉树结点(BinTreeNode)设计

2.2.3.1 概述

BinTreeNode 用来作为二叉树结点，其主要存储当前结点左孩子、右孩子地指针，以及当前的结点数据。

2.2.3.2 BinTreeNode 结构体定义

```
struct BinTreeNode {
    T data;
    BinTreeNode<T>* leftChild, * rightChild;
    BinTreeNode();
    BinTreeNode(T x,  BinTreeNode<T>* left =nullptr, BinTreeNode<T>*
right = nullptr);
};
```

2.2.3.3 数据成员

T data:当前结点数据；

BinTreeNode<T>* leftChild: 当前结点左孩子的地址；

BinTreeNode<T>* rightChild: 当前结点右孩子的地址；

2.2.3.4 构造函数

BinTreeNode();

默认构造函数，将左孩子与右孩子设为 nullptr

BinTreeNode(T x, BinTreeNode<T>* left =nullptr, BinTreeNode<T>* right = nullptr);

含参构造函数，将结点数据、左孩子、右孩子置为传入的参数，当没有传入左孩子、右孩子值时，将其置为默认参数 nullptr；

2.2.4 String 类设计

2.2.4.1 概述

String 为自定义数据类型，可以用来储存及处理字符串。

2.2.4.2 String 类定义

```
class String {
private:
    char* _str;
    size_t _str_len;
    size_t _str_cap;
public:
    static constexpr size_t npos = -1;
    typedef char* iterator;
    typedef const char* const_iterator;
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
    String(const char* str = "");
    String(const String& s);
   ~String();
    const char* C_str ()const;
    size_t Size()const;
    size_t Capacity()const;
    void Swap(String& s);
    void Reserve(size_t n);
    void Resize(size_t n, char ch = '\0');
    void PushBack(char ch);
    void Append(const char* str);
    void Insert(size_t pos, char ch, size_t n=1);
    void Insert(size_t pos, const char* str);
    void Erase(size_t pos, size_t len = npos);
    size_t Find(char ch, size_t pos = 0) const;
    size_t Find(const char* str, size_t pos = 0) const;
    String Substr(size_t pos = 0, size_t len = npos) const;
    void Clear();
    static int Memcmp(const void *dst,const void *src,size_t n) ;
```

```
    static int Strlen(const char* src);
    static char* Strstr(const char* str1, const char* str2);
    String& operator+=(char ch);
    String& operator+=(const char* str);
    bool operator<(const String& s) const;
    bool operator==(const String& s) const;
    bool operator<=(const String& s) const;
    bool operator>(const String& s) const;
    bool operator>=(const String& s) const;
    bool operator!=(const String& s) const;
    char& operator[](size_t pos);
    const char& operator[](size_t pos) const;
};
```

2.2.4.3 私有数据成员

char* _str: 存储字符串的字符数组

size_t _str_len: 当前长度

size_t _str_cap: 当前容量

2.2.4.4 公有数据成员

static constexpr size_t npos = -1: npos 常量表示无效位

2.2.4.5 构造函数与析构函数

String(const char* str = "");

构造函数，默认空字符串

String(const String& s);

拷贝构造函数

~String();

析构函数，释放字符串所占内存

2.2.4.6 公有成员函数

iterator begin();

获取迭代器开始位置。

iterator end();

获取迭代器结束位置。

const_iterator begin() const;

获取常量迭代器开始位置。

const_iterator end() const;

获取常量迭代器结束位置。

const char* C_str ()const;

获取当前字符串。

size_t Size()const;

获取字符串长度。

size_t Capacity()const;

获取当前容量。

void Swap(String& s);

交换字符串内容。

void Reserve(size_t n);

预留容量，扩展字符串容量。

void Resize(size_t n, char ch = '\0');

调整当前字符串大小。

void PushBack(char ch);

在字符串末尾添加字符。

void Append(const char* str);

在字符串末尾添加字符串。

void Insert(size_t pos, char ch, size_t n=1);

在指定位置插入 n 个字符。

void Insert(size_t pos, const char* str);

在指定位置插入字符串。

void Erase(size_t pos, size_t len = npos);

删除指定位置指定长度的字符串。

size_t Find(char ch, size_t pos = 0) const;

在字符串中查找指定字符位置。

size_t Find(const char* str, size_t pos = 0) const;

在字符串查找指定字符串位置。

String Substr(size_t pos = 0, size_t len = npos) const;

提取子字符串。

void Clear();

清空字符串。

static int Memcmp(const void *dst,const void *src,size_t n) ;

检查字符串长度是否相等，根据返回结果进行判断 1 为 dst 大于 src，-1 为 dst 小于 srcdst ，0 为等于 src。

static int Strlen(const char* src);

获取字符串长度。

static char* Strstr(const char* str1, const char* str2);

查找子字符串。

String& operator+=(char ch);

+=运算符重载，将新的字符放在字符串末尾。

String& operator+=(const char* str);

+=运算符重载，将新的字符串放在字符串末尾。

bool operator<(const String& s) const;

小于运算符重载，用于字符串大小比较。

bool operator==(const String& s) const;

等于运算符重载，用于字符串大小比较。

bool operator<=(const String& s) const;

小于等于运算符重载，用于字符串大小比较。

bool operator>(const String& s) const;

大于运算符重载，用于字符串大小比较。

bool operator>=(const String& s) const;

大于等于运算符重载，用于字符串大小比较。

bool operator!=(const String& s) const;

不等于运算符重载，用于字符串大小比较。

char& operator[](size_t pos);

下标运算符重载，用于查找指定位置字符。

const char& operator[](size_t pos) const;

下标运算符重载，用于查找指定位置字符。

String& operator=(const String& s);

赋值运算符重载，用于 String 类的赋值。

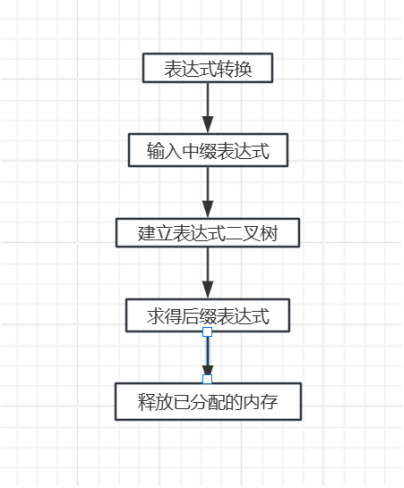
2.2.4.7 输入重载

inline std::istream& operator>>(std::istream& input, String& s)；

输入流操作符重载，规范 String 数据类型的输入

2.3 项目框架设计

2.3.1 项目框架流程图



2.3.2 项目框架流程

1. 进入表达式转换系统，输出提示信息；
2. 用户输入正确的表达式，输入错误需要重新输入；
3. 根据中缀表达式建立表达式二叉树；
4. 通过后序遍二叉树求出后缀表达式并输出；
5. 删除表达式二叉树，释放内存空间；
6. 退出表达式转换系统。

第 3 章 项目功能实现

3.1 项目主体架构

3.1.1 实现思路

1.进入 ExpressionConversion 函数以进入表达式转换系统,输出提示信息。

2.调用 Input 函数，用户开始输入中缀表达式，输入的表达式暂存在一维字符数组中，需要经过 isLegal 函数进行合法性检测，不合法需要重新输入，合法则将一位字符数组转换为二维字符数组（每个一维数组存储一个运算符、运算数或括号），进入下一步。

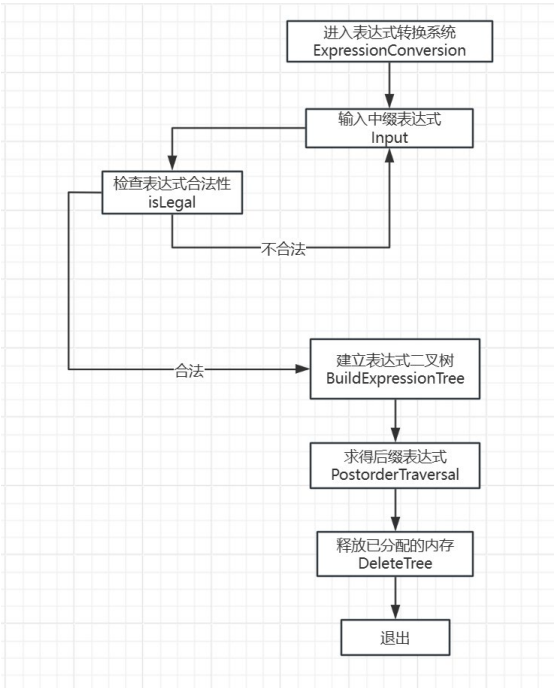
3.调用 BuildExpressionTree 函数，根据中缀表达式，借助链式栈，根据运算符优先级，建立表达式二叉树。

4.调用 PostorderTraversal 函数，后序遍历二叉树，得到后缀表达式并输出。

5.调用 DeleteTree 函数释放给二叉树分配的内存，防止内存泄漏；

6.退出程序。

3.1.2 流程图



3.1.3 核心代码

```
void ExpressionConversion()
{
    std::cout << "+-----------------------------------------+\n";
    std::cout << "|                表达式转换               |\n";
    std::cout << "|          Expression Conversion          |\n";
    std::cout << "+-----------------------------------------+\n";
    char expression[MAX_NUM +1][MAX_NUM + 1] = { '\0' };
    std::cout << "*输入说明：\n"
        << "1.中缀表达式可以包含+，-，*，/以及左右括号；\n"
        << "2.用空格分隔不同对象，正负号与数字不用空格分隔（若分隔则视为运算符）；\n"
        << "3.表达式不能超过 20 个字符（不包含空格）；\n"
        << "4.输入长度不能超过 100，否则会截断；\n"
        << "5.示例：2 + 3 * ( 7 - 4 ) + 8 / 4 * ( -3 + 2 ).\n";
    Input(expression);
    BinTreeNode<mine::String> *root = BuildExpressionTree(expression);
    PostorderTraversal(root);
    DeleteTree(root);
}
```

3.1.4 示例

```
+-----------------------------------------+
|                表达式转换               |
|          Expression Conversion          |
+-----------------------------------------+
*输入说明：
1.中缀表达式可以包含+，-，*，/以及左右括号；
2.用空格分隔不同对象，正负号与数字不用空格分隔（若分隔则视为运算符）；
3.表达式不能超过20个字符（不包含空格）；
4.输入长度不能超过100，否则会截断；
5.示例：2 + 3 * ( 7 - 4 ) + 8 / 4 * ( -3 + 2 ).
请输入中缀表达式
2 + 3 * ( 7 - 4 ) + 8 / 4 * ( -3 + 2 )
后缀表达式为：
2 3 7 4 - * + 8 4 / -3 2 + * +
```

3.2 构建表达式二叉树功能

3.2.1 实现思路

1.初始化栈结构，创建两个新的栈 operandStack 用于存储操作数（包括数字或子表达式）对应的二叉树结点指针；operatorStack 用于存储运算符；

2.进入一个循环，当当前字符数组不为空时，进行下一步处理，否则退出循环；

3.创建规范化字符串，将当前字符数组转化为一个 String 类，若当前字符串含有正号，去掉正好号，若当前字符串小数点前没有数字，则在小数点前补零，若当前字符串小数点后没有数字，则去掉小数点；

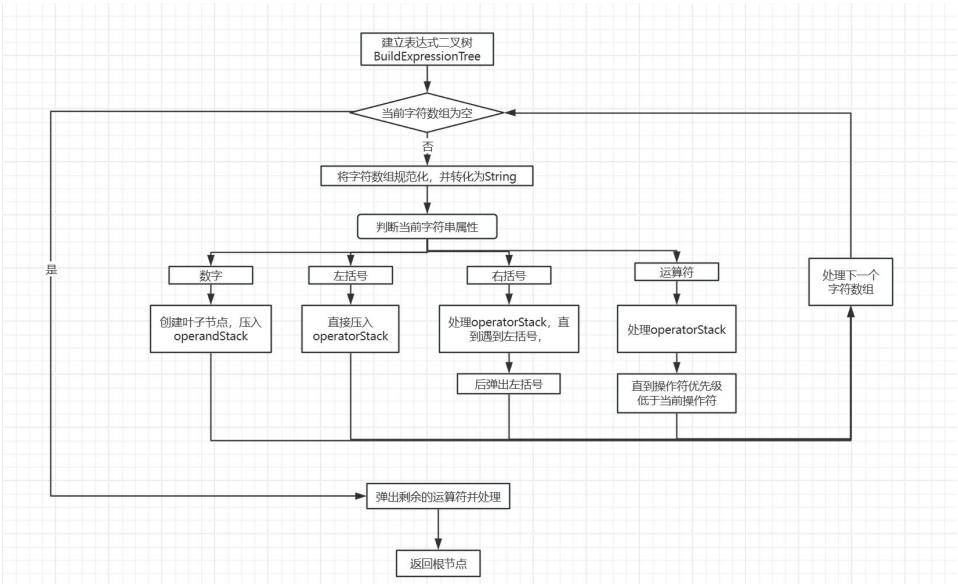
4.分类处理当前字符串：若字符串为操作数，创建叶子结点，则将结点指针压入 operandStack；若字符串为左括号，则将左括号压入 operatorStack；若字符串为右括号，则开始处理 operatorStack 中的操作符，直到遇到左括号；若字符串为运算符，则从栈顶开始处理操作符，直到栈顶操作符优先级低于当前操作符，将当前操作符压入 operatorStack；

5.循环结束后，继续处理 operatorStack 操作符，直到 operatorStack 为空；

6.正常情况下（输入符合要求），此时 operandStack 只剩一个结点指针，及为根结点指针，返回根结点指针。

操作符处理逻辑：从 operatorStack 取出需要处理的操作符，创建一个新的结点，储存当前操作符，在 operandStack 取出两个元素，依次作为该结点的右孩子和左孩子，将当前结点地址压入 operandStack。

3.2.2 流程图



3.2.3 核心代码

```
BinTreeNode<mine::String>* BuildExpressionTree(const char
expression[MAX_NUM + 1][MAX_NUM + 1])
{
    Stack<BinTreeNode<mine::String>*> operandStack;
    Stack<char> operatorStack;
    for (int i = 0; expression[i][0]!='\0'; i++) {
        mine::String expr;
        for (int j = 0; expression[i][j] != '\0'; j++)
            expr.PushBack(expression[i][j]);
        if (expr.Size() < 1)
            continue;
        if (expr.Size() > 1 && expr[0] == '+')
            expr.Erase(0, 1);
        if (expr[0] == '.')
            expr.Insert(0, '0');
        if(expr.Size() > 1&&expr[0] == '-'&& expr[1]=='.')
            expr.Insert(1, '0');
        if (expr[expr.Size()-1] == '.')
            expr.Erase(expr.Size() - 1, 1);
        char ch = expr[0];
        if (expr.Size() > 1 && (expr[0] == '+' || expr[0] == '-'))
            ch = expr[1];
        if (LegalFigure(ch)||ch=='.') {
            auto* newNode = new(std::nothrow)
BinTreeNode<mine::String>(expr);
            assert(newNode!=nullptr);
            operandStack.Push(newNode);
        }
        else if (ch == '(') {
            operatorStack.Push(ch);
        }
        else if (ch == ')') {
            while (!operatorStack.IsEmpty() && operatorStack.getTop() !=
'(') {
                char op = operatorStack.Pop();
```

```
                    BinTreeNode<mine::String>* right = operandStack.Pop();
                    BinTreeNode<mine::String>* left = operandStack.Pop();
                    mine::String temp;
                    temp.PushBack(op);
                    auto* newNode = new(std::nothrow)
BinTreeNode<mine::String>(temp, left, right);
                    assert(newNode!=nullptr);
                    operandStack.Push(newNode);
                }
                if (!operatorStack.IsEmpty()) operatorStack.Pop();
            }
            else if (LegalCharacter(ch)) {
                while (!operatorStack.IsEmpty() &&
getPriority(operatorStack.getTop()) >= getPriority(ch)) {
                    char op = operatorStack.Pop();
                    BinTreeNode<mine::String>* right = operandStack.Pop();
                    BinTreeNode<mine::String>* left = operandStack.Pop();
                    mine::String temp;
                    temp.PushBack(op);
                    auto* newNode = new(std::nothrow)
BinTreeNode<mine::String>(temp,    left, right);
                    assert(newNode!=nullptr);
                    operandStack.Push(newNode);
                }
                operatorStack.Push(ch);
            }
        }
        while (!operatorStack.IsEmpty()) {
            char op = operatorStack.Pop();
            BinTreeNode<mine::String>* right = operandStack.Pop();
            BinTreeNode<mine::String>* left = operandStack.Pop();
            mine::String temp;
            temp.PushBack(op);
```

```
                auto* newNode = new(std::nothrow)
BinTreeNode<mine::String>(temp, left, right);
                assert(newNode!=nullptr);
                operandStack.Push(newNode);
            }
            BinTreeNode<mine::String>* subTree = operandStack.Pop();
            return subTree;
        }
```

3.3 后缀表达式获取与输出功能

3.3.1 实现思路

后缀表达式获取与输出功能实现的思路为：

1.创建两个栈 s1、s2，s1 用于控制结点的遍历顺序，s2 用于存储调整后的顺序，使得最终的输出符合后序遍历规则；

2.将树的根结点指针插入到 s1 中；

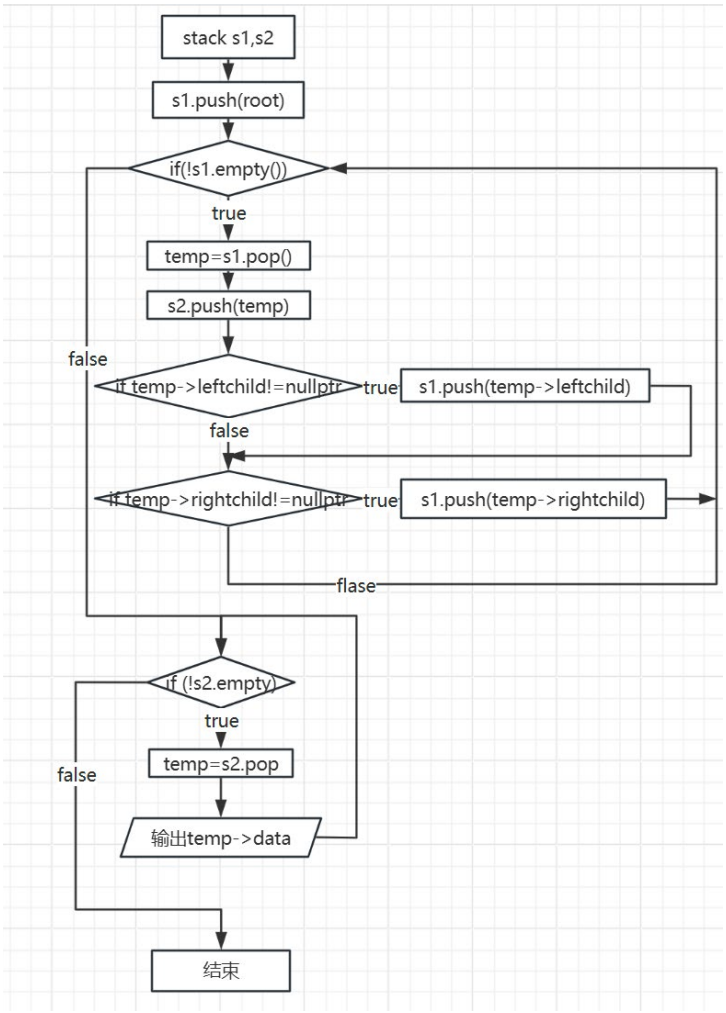
3.进入一个循环，循环条件为当 s1 不为空；

4.当满足循环条件时，从 s1 中弹出栈顶元素，插入 s2 中；判断取出指针对应的结点是否有左孩子，若有，将左孩子指针插入到 s1 中，在判断取出的结点是否有右孩子，若有，将右孩子指针插入到 s1 中；

5.当循环条件不满足时，开始进行输出，进入一个新循环，循环条件为当 s2 不为空；

6.若符合循环条件，则弹出栈顶元素，输出其对应的值；不符合循环条件则退出循环并退出当前函数。

3.3.2 流程图

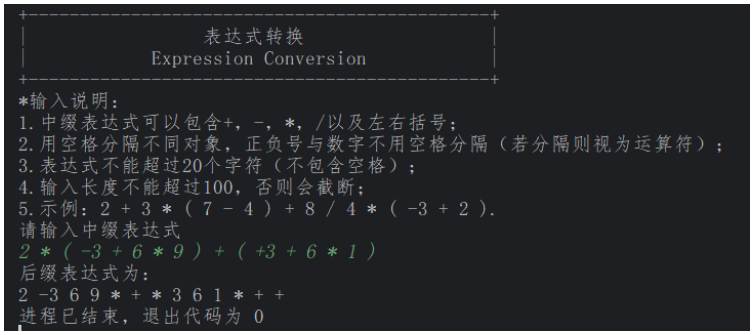


3.3.3 核心代码

```
void PostorderTraversal(BinTreeNode<mine::String> *const root)
{
    if (root == nullptr)
        return;
    Stack<BinTreeNode<mine::String>*> s1, s2;
    s1.Push(root);
    while (!s1.IsEmpty()) {
        BinTreeNode<mine::String>* node = s1.getTop();
        s2.Push(node);
        s1.Pop();
```

```
            if (node->leftChild!=nullptr) {
                s1.Push(node->leftChild);
            }
            if (node->rightChild!= nullptr) {
                s1.Push(node->rightChild);
            }
        }
        bool first = true;
        std::cout << "后缀表达式为：\n";
        while (!s2.IsEmpty()) {
            if (!first)
                std::cout << " ";
            std::cout << s2.getTop()->data ;
            s2.Pop();
            first = false;
        }
        std::cout<<"\n";
    }
```

3.3.4 示例



3.4 异常处理功能

3.4.1 输入非法的异常处理

在将中缀表达式转换为后缀表达式时，错误处理十分关键，一个错误的中缀表达式无论如何都无法成功转化为一个正确的后缀表达式。

在表达式输入中，常常会遇到以下输入错误：

1.没有输入任何值；

2.中缀表达式字符个数（不含空格）超过 20；

3.输入非法的字符（不是 0~9，+，-，*，/，(，),.）；

4.输入过程中未使用空格分隔不同的对象；

5.除第一项外，带正负号的数前没有括号；

6.小数点单独出现或者小数点过多；

7.二原运算符前后缺少变量，包括以二元运算符作为开头或结尾、二元运算符前为左括号或后为右括号；

8.表达式中存在空括号；

9.表达式中括号不匹配，即左括号和右括号数目不等；

10.存在连续的运算数而不用运算符连接。

在输入完成后，应当对于以上可能存在的问题进行验证，以保证表达式正确；在这一方面，我才用 do-while 循环，每次输入完成后，调用 isLegal 函数验证，不符合要求需要重新输入；对于验证逻辑，具体代码如下：

```
bool isLegal(const char expr[])
{
    if (expr[0] == '\0') {
        std::cout << "计算表达式为空，请重新输入！\n";
        return false;
    }
    int len = 0;
    for (int i = 0; expr[i] != '\0'; i++)
        if (expr[i] != ' ')
            len++;
    if (len > MAX_NUM) {
        std::cout << "计算表达式中的字符个数大于"<< MAX_NUM <<"，
请重新输入！\n";
        return false;
    }
    char temp[MAX_NUM +1][MAX_NUM +1] = { '\0' };
    Copy(temp, expr);
    for (int i = 0; temp[i][0] != '\0'; i++) {
        for (int j = 0; temp[i][j] != '\0'; j++) {
```

```
                //检查非法字符
                if (!(temp[i][j] >= '0' && temp[i][j] <= '9')
&& !LegalCharacter(temp[i][j]) && temp[i][j] != '(' && temp[i][j] != ')' &&
temp[i][j] != '.') {
                    std::cout << "计算表达式存在非法字符输入，请重新输
入！ \n";
                    return false;
                }
                //检查输入格式
                if(temp[i][j] == '('|| temp[i][j] == ')'|| LegalCharacter(temp[i][j]))
                    if ((j != 0 || temp[i][j + 1] != '\0') && !((temp[i][j] == '+' ||
temp[i][j] == '-') && j == 0 )) {
                        std::cout << "计算表达式未使用空格分隔不同对象，
请重新输入！ \n";
                        return false;
                    }
                if((temp[i][j] == '+' || temp[i][j] == '-') && j ==
0&&temp[i][1]!='\0' &&i != 0 && temp[i - 1][0] !='(') {
                    std::cout << "计算表达式除第一项外，所有带正负号的数
前一项必须为左括号！ \n";
                    return false;
                }
                //检查小数点是否单独出现
                if (temp[i][j] == '.')
                    if (!(LegalFigure(temp[i][j + 1]) || j != 0 &&
LegalFigure(temp[i][j - 1]))) {
                        std::cout << "计算表达式中小数点不能单独出现，请
重新输入！ \n";
                        return false;
                    }
            }
        }
        for (int i = 0; temp[i][0] != '\0'; i++)
            if (temp[i][1] != '\0') {
```

```
                int num_dot = 0;
                for (int j = 0; temp[i][j] != '\0'; j++)
                    if (temp[i][j] == '.')
                        num_dot++;
                //检查小数点是否合理
                if (num_dot > 1) {
                    std::cout << "计算表达式中存在有某个数有多个小数点，
请重新输入！ \n";
                    return false;
                }
            }
        Stack<char> parentheses;// 用于存储括号
        char previous = '\0';
        for (int i = 0; temp[i][0] != '\0';i++) {
            char ch = temp[i][0];
            if (temp[i][1] != '\0' && (temp[i][0] == '+' || temp[i][0] == '-'))
                ch = temp[i][1];
            // 检查公式是否以乘号或除号开始
            if (i==0 && LegalCharacter(ch)) {
                std::cout << "计算表达式不能以运算符开始，请重新输入！
\n";
                return false;
            }
            // 检查空括号
            if (previous == '(' && ch == ')') {
                std::cout << "计算表达式存在空括号，请重新输入！ \n";
                return false;
            }

            // 检查变量与括号的连接
            if ((LegalFigure(ch) && previous == ')') || (ch == '(' &&
LegalFigure(previous))) {
                std::cout << "计算表达式中变量与括号的连接不正确，请重新
输入！ \n";
```

```
            return false;
        }
        // 检查运算符与括号的连接
        if ((LegalCharacter(ch) && previous == '(') || (ch == ')' &&
        LegalCharacter(previous))) {
            std::cout << "计算表达式中运算符与括号的连接不正确，请重
        新输入！\n";
            return false;
        }
        // 存储左括号
        if (ch == '(')
            parentheses.Push(ch);
        // 处理右括号
        else if (ch == ')') {
            if (parentheses.IsEmpty()) {
                std::cout << "计算表达式括号不匹配，请重新输入！\n";
                return false;// 右括号没有匹配的左括号
            }
            parentheses.Pop();// 匹配到一对括号，弹出栈顶元素
        }
        if ((LegalFigure(ch)||ch=='.') && (LegalFigure(previous)||
previous=='.')) {
            std::cout << "计算表达式中常数间必须使用运算符连接，请重
新输入！\n";
            return false;
        }
        // 检查运算符前后的连接
        if (LegalCharacter(ch)&& (!LegalFigure(previous) && previous !=
')'&& previous!='.')) {
            std::cout << "计算表达式中每个二元运算符前后必须连接变
量，请重新输入！\n";
            return false;
        }
        previous = ch;
```

```
        }
        // 检查公式是否以运算符结尾
        if (LegalCharacter(previous)) {
            std::cout << "计算表达式不能以运算符结尾，请重新输入！\n";
            return false;
        }
        // 检查括号是否匹配
        if (!parentheses.IsEmpty()) {
            std::cout << "计算表达式括号不匹配，请重新输入！\n";
            return false;
        }
        return true;// 表达式合法
    }
```

3.4.2 动态内存申请失败的异常处理

在进行 Stack 类与 String 类的动态内存申请，以及二叉树建立过程中，程序使用 new(std::nothrow) 来尝试分配内存。new(std::nothrow)在分配内存失败时不会引发异常，而是 返回一个空指针（NULL 或 nullptr），代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，对于内存分配失败，可以采用两种方式处理：

1.判断 new(std::nothrow)是否返回 nullptr，是则使用 exit 函数退出程序，并限定返回值，示例：

```
template <typename T>
void Stack<T>::Push(const T& x)
{
    if (top ==nullptr) {
        top = new(std::nothrow) Node<T>(x);
        if (top == nullptr) {
            std::cout << "内存分配错误！\n";
            exit(MEMORY_ALLOCATION_ERROR);
        }
        size++;
        return;
    }
```

```
        auto* newNode = new(std::nothrow) Node<T>(x);
        if (newNode == nullptr) {
            std::cout << "内存分配错误！\n";
            exit(MEMORY_ALLOCATION_ERROR);
        }
        newNode->next = top;
        top = newNode;
        size++;
    }
```

2. 通过 assert 语句判断 new(std::nothrow)是否返回 nullptr，示例：

```
    inline String::String(const char* str)
    {
        _str_len = Strlen(str);
        _str_cap = _str_len;
        _str = new(std::nothrow) char[_str_cap + 1];//多开一位用来存放'\0'
        assert(_str != nullptr);
        memcpy(_str, str, _str_len + 1);
    }
```

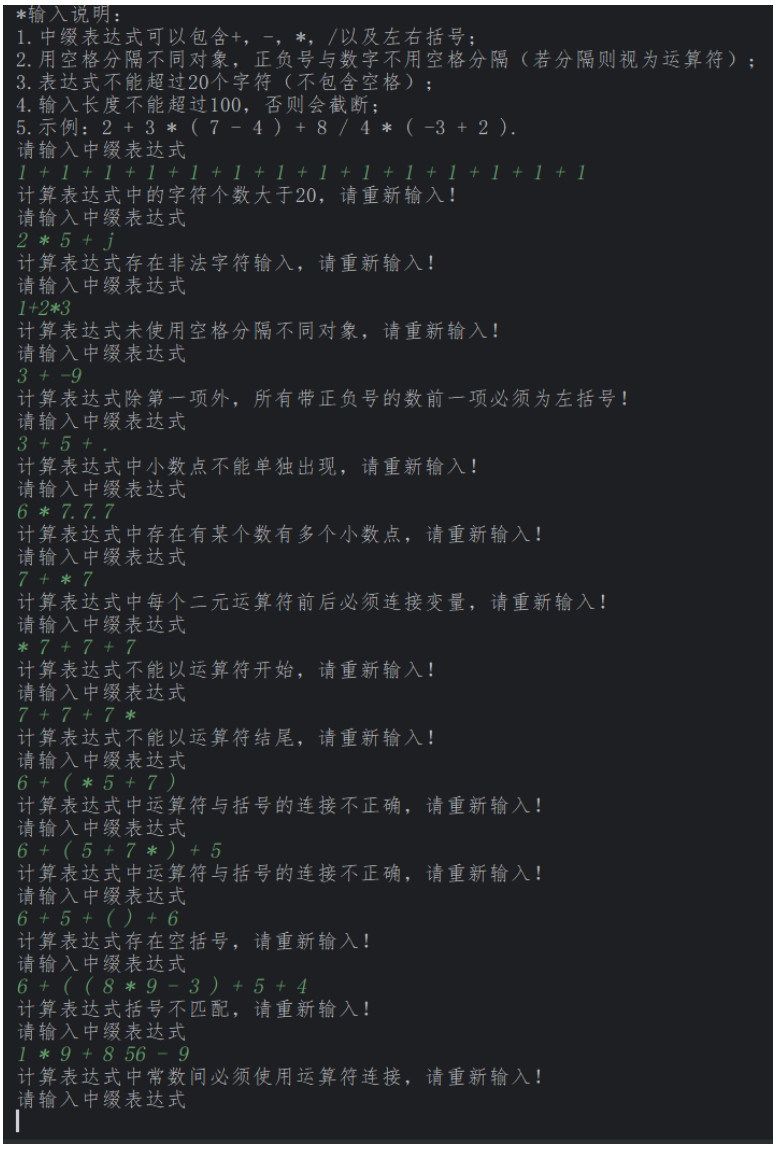
3.4.3 索引越界的异常处理

在 String 类中，许多操作，例如 Insert、Erase、Find、Substr 等函数，都需要通过索引访问，通过 assert 函数来判断索引是否正确，一般为 assert(pos < _str_len)，也可能为 assert(pos <= _str_len)。

第 4 章 项目测试

4.1 输入验证功能测试测试

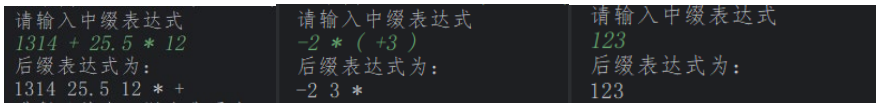
当用户输入的表达式错误时，会提示相关的错误信息



4.2 表达式转换功能测试

当输入正确的中缀表达式时，可以输出正确的后缀表达式


请输入中缀表达式
2 + 3 * (7 - 4) + 8 / 4
后缀表达式为:
2 3 7 4 - * + 8 4 / +
请输入中缀表达式
((2 + 3) * 4 - (8 + 2)) / 5
后缀表达式为:
2 3 + 4 * 8 2 + - 5 /


请输入中缀表达式
1314 + 25.5 * 12
后缀表达式为:
1314 25.5 12 * +
请输入中缀表达式
-2 * (+3)
后缀表达式为:
-2 3 *
请输入中缀表达式
123
后缀表达式为:
123

第 5 章 相关说明

5.1 编程语言

本项目全部.cpp 文件以及.h 文件均使用 C++编译完成，使用 UTF-8 编码。

5.2 Windows 环境

Windows 系统：Windows 11 x64

Windows 集成开发环境：CLion 2024

工具集：MinGW 11.0 w64

5.3 Linux 环境

基于 Linux 内核的操作系统发行版：Ubuntu 24.04.1

Linux 命令编译过程为：

1. 定位包含项目所在文件夹，包括.pp 与.h 文件；具体命令为: cd /home/bruce/programe/ expression_conversion

2.编译项目，生成可执行文件；具体命令为: g++ -static -o expression_conversion_linux expression_conversion.cpp my_stack.h my_string.h;

其中指令含义分别为：

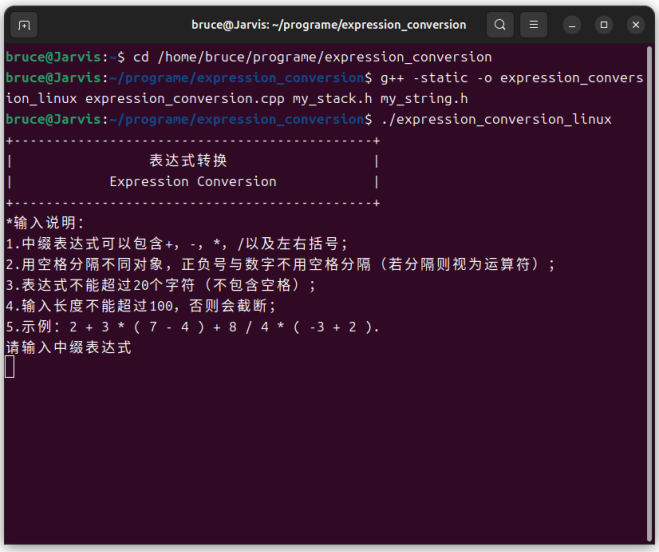
g++: 调用 GNU 的 C++编译器

-static: 使用静态链接而非动态链接，将所有依赖库直接嵌入到可执行文件，文件存储空间变大，但可以单独运行

-o expression_conversion_linux: -o 表示输出文件选项，expression_conversion_linux 为可执行文件名

expression_conversion.cpp my_stack.h my_string.h:编译所需要的文件。

3.运行可执行文件；具体命令为 ./ expression_conversion_linux


bruce@Jarvis: ~/programe/expression_conversion
bruce@Jarvis:~$ cd /home/bruce/programe/expression_conversion
bruce@Jarvis:~/programe/expression_conversion$ g++ -static -o expression_conversion_linux expression_conversion.cpp my_stack.h my_string.h
bruce@Jarvis:~/programe/expression_conversion$ ./expression_conversion_linux
+---+
| 表达式转换 |
| Expression Conversion |
+---+
*输入说明:
1.中缀表达式可以包含+, -, *, /以及左右括号;
2.用空格分隔不同对象, 正负号与数字不用空格分隔（若分隔则视为运算符）;
3.表达式不能超过20个字符（不包含空格）;
4.输入长度不能超过100, 否则会截断;
5.示例: 2 + 3 * (7 - 4) + 8 / 4 * (-3 + 2).
请输入中缀表达式