《数据结构》课程实践报告

院、系	计算机学院	年级专业	21 计算机科学与技术	姓名	赵鹏	学号	2127405037
实验布置日期	2022.9.5		提交日期	2022.12.31		成绩	

课程实践实验 2：中缀表达式

一、问题描述及要求

中缀表达式求解

中缀表达式是我们熟悉的表达式形式。为了能正确表示运算的先后顺序，中缀表达式中难 免要出现括号。假设我们的表达式中只允许有圆括号。 读入一个浮点数为操作数的中缀表达式后，对该表达式进行运算。 要求中缀表达式以一个字符串的形式读入，可含有加、减、乘、除运算符和左、右括号， 并假设该表达式以“#”作为输入结束符。

如输入“3.5*(20+4)-1#”，则程序运行结果应为 83。 要求可单步显示输入序列和栈的变化过程。

并考虑算法的健壮性，当表达式错误时，要给 出错误原因的提示。 附：大家可扩展考虑后缀表达式求值、中缀表达式求值、括号匹配等栈的综合应用

二、概要设计

（1）问题分析

中缀表达式求值是一个非常常见的问题。中缀表达式的求解不能简单的从左向右依次计算，对于+-*/有着一些约定的运算顺寻（*/优先于+-），其余部分则需要由括号来决定。我们可以用栈来很好的进行计算这一操作的实现。

（2）系统功能

本项目实现的功能相对简单，主要是由用户输入表达式求出相应的值，可以选择显示细节和不显示细节两种模式。在实现过程中进行了较多的异常处理，使得程序的健壮性有较好的保证。

（3）程序运行界面设计

本程序在运行的过程中首先由用户输入是否显示运算细节，随后用用户输入表达式，输出相应的结果。

（4）总体设计思路

本程序在设计过程中主要使用的数据结构为链栈，在实现过程中设计了一个InfixExpression类，用于实现给定中缀表达式的计算。为了使计算流程较为简洁以及提升程序健壮性的需求，函数在设计过程中进行了较多的封装，在设计过程中将各种操作进行了较细的拆分，并在多个函数里进行了异常处理。在设计InfixExpreesion类的过程中设计了以下的方法。

函数名	作用
getOperands	从操作数栈顶取出一对数
saveOperand	保存运算后的操作数
getOperator	从运算符栈顶取出一个运算符
saveOperator	保存运算符进栈
stepCalculate	单步计算，取出一对数和一个操作符运算
digitCompute	进行一次运算
getResult	获得最后结果
checkBracketMatch	判断括号是否匹配
splitOperatorsOperands	分离操作符和操作数
parseExpression	解析表达式
judgeDigitString	判断是否为数字串
judgeValidChar	判断是否为合法字符
judgeMinus	判断该符号是减号还是负号
compareOperator	比较两个操作符优先级

（5）程序结构设计

本项目在实现过程中复用了链栈这一数据结构，具体实现在 LinkStack.hpp 文件中。还额外设计了以下文件:

文件	功能
InfixExpression.h	声明表达式类
InfixExpression.cpp	实现表达式类的各种函数
Utility.h	包含常用头文件
main.cpp	主函数

本程序在实现过程中使用的主要数据结构为栈，主要算法为模拟。整体思路是用栈模拟计算的的过程中。由于运算符具由不同的优先级，较先出现的运算符可能滞后运算如+-等，对于滞后这一特性，我们可以使用栈来进行保存，很好的实现运算。

三、详细设计

在本程序中，计算中缀表达式的重点主要有以下几个部分:

1. **数据在程序中的存储**

   由于运算符有着不同的优先级，排在前面的运算符可能滞后被计算，因此很自然的可以想到我们可以用两个栈来暂存运算符和数字。根据后续读取到的字符的优先级来决定是继续入栈还是弹出栈中现有的数据进行运算。

2. **运算符优先级的比较**

   在中缀表达式中，运算的优先级由运算符和()来决定，+-*/这四种运算和我们日常计算时的约定一样，*/运算优先于+-运算，在同一级别内，先出现的优先级高于后出现的。对于括号的处理，若遇到左括号，则优先级最高，之间入栈即可。若遇到右括号，则计算匹配的左括号开始的表达式，计算完后入栈。

3. **对于表达式的预处理部分**

   1.1. 括号匹配判断

   定义一个栈用于存储左括号，依次从前往后扫描整个表达式，若遇到左括号则入栈，若遇到右括号则从栈中弹出一个左括号，若在弹出左括号的时候栈为空或者扫描结束时栈不为空则说明表达式的括号不合法。

   3.2 表达式的解析

   在表达式的解析过程中主要有三个部分。

   3.2.1 判断非法字符

   这一步较为简单，只要判断表达式中是否出现 0123456789+-*/(). 以外的字符即可。

   3.2.2 用空格分离数字和操作符

   这一步也较为简答，只需要使用字符串的 repalce 函数把操作符替换为空格+操作符+空格即可。

3.2.3 表达式的修改

这一步操作主要是为了在表达式计算时区分出减号与负号。对于‘-’有以下几种特征：1.若出现在表达式开头则必定为负号

2.若出现在数字或者‘）’后则必定为减号

3.若出现在表达式开头则必定为负号

4.若出现在另一个操作符后则必定是负号

对于判定为减号的符号，我们可以用~进行替代，这样就能区别出减号与负号了。

4. 表达式的计算部分

函数的计算过程由多个函数相互调用实现。

calculateExpression 函数的主流程为：

1. 检查括号匹配
2. 解析表达式并存储到字符串流中
3. 依次从字符串流中读取对象，根据优先级进行入栈、计算等操作。

对于计算的重点部分，我们可以在进行预处理后获得一个保存在字符串流中的用空格分离操作符操作数且减号与负号被区分开的表达式。程序在读取的过程中边计算边读取，对于读入的运算符，若当前运算符栈为空或者运算符优先级高于栈顶优先级则入栈，否则从栈中弹出一个运算符，并弹出两个操作数，进行单步运算，再将结果入栈，往复次过程，直到操作符栈空或者栈顶优先级更高为止，再将这个运算符入栈。

当处理到栈的结尾时，运算符的各种括号关系已经被处理完成，只要不断的单步计算（弹出一个操作符和两个操作数，计算并将结果入栈）即可。

最后操作数栈顶的元素即为最终结果。(若弹出栈顶后栈不为空还需处理异常)。

4. 运行步骤的显示

这一步较为简单，可以在类中使用一个静态成员 detail 用于表示是否选择显示细节。在进行各种操作数若 detail==True 则输出相应信息即可。

5. 程序健壮性的提升

为了使代码结构较为清晰，代码在编写时对各种函数进行了较多的封装。如将操作数入栈出栈、操作符入栈出栈、检查括号匹配、单步计算等函数都进行了封装，并在各个函数里进行了分别进行了异常处理，使得在执行出错时可以通过返回的错误信息来判断到底是在那一步出现了错误。

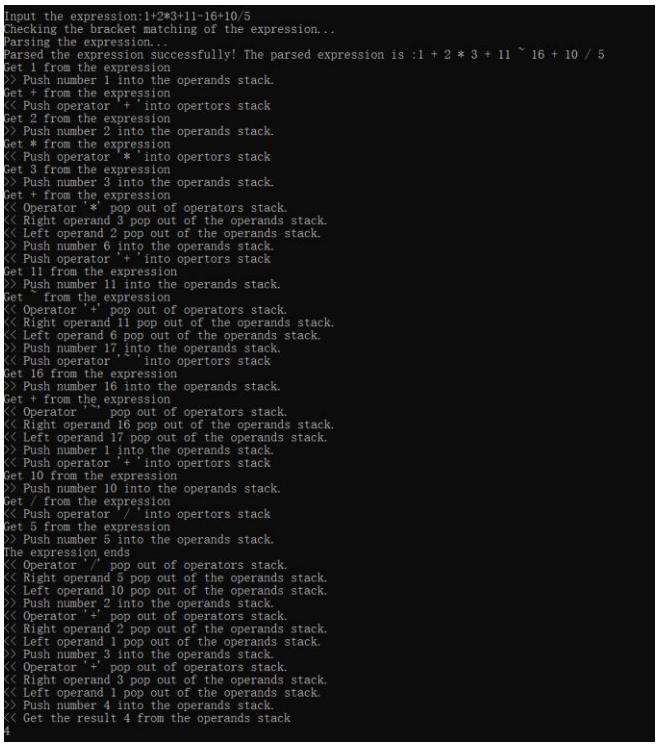
四、实验结果

测试输入： 1+2*3+11-16+10/5

测试目的： 计算整数的四则运算

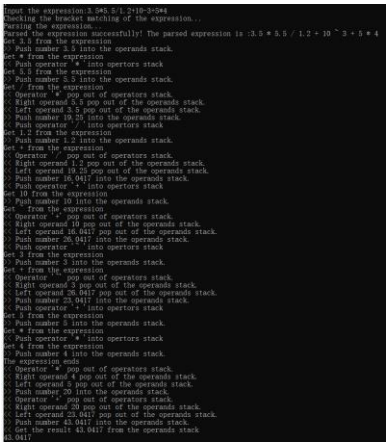
正确输出： 4

实际输出： 4



测试结论： 通过

测试输入：3.5*5.5/1.2+10-3+5*4
测试目的： 测试包含小数的四则运算
正确输出： 43.0416666
实际输出： 43.0417



测试结论： 通过

测试输入：(1+5)*(3-4+6*10)
测试目的： 计算包含括号的表达式的正确计算
正确输出： 354
实际输出： 354

```
<< Left operand -1 pop out of the operands stack.
>> Push number 59 into the operands stack.
<< Operator '(' pop out of operators stack.
The calculation of sub expression finished.
The expression ends
<< Operator '*' pop out of operators stack.
<< Right operand 59 pop out of the operands stack.
<< Left operand 6 pop out of the operands stack.
>> Push number 354 into the operands stack.
<< Get the result 354 from the operands stack
354
```

测试结论： 通过

测试输入：((1+1)
测试目的： 括号不匹配报错
正确输出： 异常：括号错误
实际输出：

```
Input the expression:((1+1)
Checking the bracket matching of the expression...
Unexpected line in <stdin> : ((1+1)
[Runtime_error] Invalid bracket matches of the expression :missing ') When checking the bracket matching of the expression
```

测试结论： 通过

测试输入：(1+1+1+!)
测试目的： 判断非法字符
正确输出： 异常：非法字符
实际输出：

```
Input the expression:(1+1+1+!)
Checking the bracket matching of the expression...
Parsing the expression...
Unexpected line in <stdin> : (1+1+1+!)
[Runtime_error] Invalid expression is given : Expression containing invalid character When parsing the expression
```

测试结论： 通过

测试输入：10+5++
测试目的： 判断缺少运算数是栈下溢的异常处理
正确输出： 异常：栈下溢
实际输出：

```
Input the expression:10+5++
Checking the bracket matching of the expression...
Parsing the expression...
Parsed the expression successfully! The parsed expression is :10 + 5 +  +
Get 10 from the expression
>> Push number 10 into the operands stack.
Get + from the expression
<< Push operator '+' into opertors stack
Get 5 from the expression
>> Push number 5 into the operands stack.
Get + from the expression
<< Operator '+' pop out of operators stack.
<< Right operand 5 pop out of the operands stack.
<< Left operand 10 pop out of the operands stack.
>> Push number 15 into the operands stack.
<< Push operator '+' into opertors stack
Get + from the expression
<< Operator '+' pop out of operators stack.
<< Right operand 15 pop out of the operands stack.
Unexpected line in <stdin> : 10+5++
[Underflow_error] Invalid expression is given: mising operand(s) When popping operands
```

测试结论： 通过

五、实验分析与探讨

本程序通过了以上的多种测试，且能够处理多种异常，有着较好的健壮性。对于中缀表达式的求解，还可以先将中缀表达式转为后缀，然后仅需一个栈便可进行计算。

在程序实现的过程中遇到了较多的问题，主要问题如下：

1. 减号与负号的二义性
   解决办法：通过一定的判断区分出减号与负号，并用~代替减号进行计算。
2. 添加异常处理后代码结构较为复杂。
   解决办法：对各个操作的函数均进行封装，在函数内部进行异常处理，使得代码结构简洁清晰。
3. 程序抛出异常后程序直接终止运行、
   解决办法：在主函数内添加 try catch 语句，捕获异常并输出异常信息。

六、小结

通过本次实验，我对表达式求值的问题有了更加深刻的理解，对于栈的利用能力也有所增强，我的面向对象设计的能力也得到了一定的提示。

在本次实验中，我完成了中缀表达式求值的计算并支持选择显示计算细节。但程序仍有所不足，例如没有实现文件读入等、仅仅支持(),不支持{}[]等其他括号等。

附录：源代码

实验环境：Windows11 Visual Studio 2022

(1) main.cpp

```
#include "InfixExpression.h"
int main()
{
    string line, detail;
    cout << "Whether to show the detail ?（Y/N）" << endl;
    cin >> detail;
    if (detail[0] == 'Y' || detail[0] == 'y')
        InfixExpression::showDetail(1);
    else
        InfixExpression::showDetail(0);
    cout << "Input the expression:";
    while (getline(cin,line))
    {
        if (cin.eof() || line == "exit")
            return 0;
        if (line.empty()) {
            continue;
        }
        InfixExpression expression(line);
        try
        {
```

```
                cout << expression.calculateExpression() << endl;
            }
            catch (runtime_error& e)
            {
                SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_INTENSITY | FOREGROUND_RED);
                cout << "Unexpected line in <stdin> : " << line << endl<< e.what()
<< endl;
                SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_INTENSITY | FOREGROUND_RED | FOREGROUND_GREEN |
FOREGROUND_BLUE);
            }
            cout                                                                <<
"------------------------------------------------------------" << endl;
            cout << "Input the expression:";
        }

      }
```

(2) utility.h

```
#pragma once
#include<iostream>
#include<fstream>
#include<sstream>
#include<cstring>
#include<Windows.h>

using namespace std;
```

(3) InfixExpression.h

```
#pragma once
#include"utility.h"
#include"LinkStack.hpp"
class InfixExpression
{
public:

 static void showDetail(bool = false);
 explicit InfixExpression(string);
 double calculateExpression();//计算

private:

 string expression;//表达式
 Stack<char>operators;//操作符栈
```

```
    Stack<double>operands;//操作数栈

    static bool detail;//显示细节选项



    void getOperands(double&, double&);//获得一对操作数
    void saveOperand(double);//保存操作数
    char& getOperator(char&);//获得操作符
    void saveOperator(char);//保存操作符

    void stepCalculate();//单步计算并保存
    double digitCompute(double, char, double);//数字计算
    double getResult();//获得最终运算结果



    void checkBracketMatch(const string&);//判断括号匹配
    void splitOperatorsOperands(string&);//分离操作符和操作数
    string parseExpression(const string&);//解析表达式



    bool judgeDigitString(const std::string&);//判断是否是数字串
    bool judgeValidChar(char ch);//判断是否为合法字符
    void judgeMinus(string&, int);//判断并修改减号和负号
    int compareOperators(char, char);//比较两操作符的优先级
};
```

(4) InfixExpression.cpp

```
#include "InfixExpression.h"

bool InfixExpression::detail = false;


void InfixExpression::showDetail(bool choice)
{
    InfixExpression::detail = choice;
}


InfixExpression::InfixExpression(string str) //除去表达式里的空格
{
    (*this).expression = str;
```

```
        auto p = expression.begin();
        while (p != expression.end())
        {
            if (*p == ' ' || *p == '#')p = expression.erase(p);
            else ++p;
        }
    }



    double InfixExpression::calculateExpression()
    {
        //清除上一次的计算结果
        operands.clear();
        operators.clear();

        //输出细节

        if (detail)
        {
            cout << "Checking the bracket matching of the expression..." << endl;
        }

        //判断括号匹配
        checkBracketMatch(expression);


        //输出细节
        if (detail)
        {
            cout << "Parsing the expression..." << endl;

        }

        //将表达式解析用空格分离每一段，并将结果存储到字符串流中，用于存储
        stringstream source(parseExpression(expression));

        if (detail)
        {
            cout << "Parsed the expression successfully! The parsed expression is :"
<< source.str() << endl;
        }
```

```
        string nextString;
        //开始处理表达式分解后的每一段
        while (source >> nextString)
        {
            if (detail)
                cout << "Get " << nextString << " from the expression" << endl;

            if (judgeDigitString(nextString))//如果这一部分是数字
            {
                char* temp;
                saveOperand(strtod(nextString.c_str(), &temp));//将数字字符串转为
double 并压入操作数栈
                continue;
            }

            char op = nextString[0];//取出这一部分
            if (op == ')') //如果是右括号，则说明需要进行运算了
            {
                if (detail) { cout << "The sub expression in brackets will be calculated
in advance." << endl; }

                while (!operators.empty() && operators.top() != '(')
                {
                    stepCalculate(); //运算一次
                }

                getOperator(op);  // 弹出左括号
                if (detail) { cout << "The calculation of sub expression finished." <<
endl; }
                continue;
            }
            if (!operators.empty() && compareOperators(operators.top(), op) > 0) //如
果优先级可以运算
            {
                stepCalculate(); //进行一次运算
            }
            saveOperator(op);//存储这个操作符
        }

        //计算剩余部分
        if (detail)
        {
            cout << "The expression ends" << endl;
```

```
        }

        while (!operators.empty()) { stepCalculate(); }


        return getResult();
    }

    inline void InfixExpression::stepCalculate()
    {
        char op;
        double left, right;

        getOperator(op);//取出栈顶操作符
        getOperands(left, right);//取出两个操作数

        saveOperand(digitCompute(left, op, right));//运算一次并将结果存入操作数栈

    }

    inline void InfixExpression::getOperands(double& left, double& right)
    {
        try
        {
            right = operands.top(); operands.pop();//取出两个操作数
            if (detail) { cout << "<< Right operand " << right << " pop out of the operands stack." << endl; }
            left = operands.top(); operands.pop();
            if (detail) { cout << "<< Left operand " << left << " pop out of the operands stack." << endl; }
        }
        catch (underflow_error) //栈空则报错
        {
            throw  runtime_error{ "[Underflow_error] Invalid expression is given: mising operand(s) When popping operands" };
        }
    }

    inline void InfixExpression::saveOperand(double result)
    {
        //存储操作数
        try
        {
            operands.push(result);
```

```
            if (detail) { cout << ">> Push number " << result << " into the operands stack." << endl; }

        }
        catch (overflow_error(&e))
        {
            throw  runtime_error{ string("[Overflow_error] The expression too long, operands , When saving operands") + e.what() };
        }
    }

    inline char& InfixExpression::getOperator(char& op)
    {
        //获得操作符
        try
        {
            op = operators.top();
            operators.pop();

            if (detail)
            {
                cout << "<< Operator '" << op << "' pop out of operators stack." << endl;
            }
        }
        catch (underflow_error&)//如果栈空则报错
        {
            throw  runtime_error{ "[Runtime_error],Invalid expression is given:mising operator(s),When popping an operator" };

        }
        //返回取得的操作符
        return op;

    }

    inline void InfixExpression::saveOperator(char op)
    {
        //存储操作符
        if (!(op == '+' || op == '~' || op == '*' || op == '/' || op == '('))
        {
            throw  runtime_error{ "[Runtime_error] Invalid operator is given When Pushing the operator" };
```

```
            }
            try {
                operators.push(op);
                if (detail) { cout << "<< Push operator '" << op << " 'into opertors stack" << endl; }
            }
            catch (overflow_error& e)
            {
                throw   runtime_error{ string("[Overflow_error] The expression too long, operators ") + e.what() };
            }
        }

        inline double InfixExpression::getResult()
        {

            //返回结果
            double result;
            try
            {
                result = operands.top(); operands.pop();
            }
            catch (underflow_error&)
            {
                throw   runtime_error{ "[Runtime_error] Invalid expression is given,missing operand(s) ,When getting result" };
            }
            if (!operands.empty())//如果弹出栈顶后栈中还有元素则抛出错误
            {
                if (detail)
                {
                    cout << "Fail to get the result ! there are 2 or more operand(s) in the operands stack:" << result;
                    while (!operands.empty()) { cout << ", " << operands.top(); operands.pop(); }
                    cout << endl;
                }
                throw    runtime_error{ "[Runtime_error] Invalid expression is given: missing operator(s),When getting result" };
            }
            if (detail) { cout << "<< Get the result " << result << " from the operands stack" << endl; }
            return result;
        }
```

```
void InfixExpression::checkBracketMatch(const  string& expression)
{
    Stack<char>stack;
    for (char p : expression)
    {
        if (p == '('//如果是左括号则入栈
        {
            stack.push(p);
            continue;
        }
        if (p == ')' && (stack.empty() || stack.pop() != '('))//如果右括号无法匹配则报错
        {
            throw  runtime_error
            {
                "[Runtime_error] Invalid bracket matches of the expression : missing '(' When checking the bracket matching of the expression"
            };
        }
    }
    //如果左括号有剩余的报错
    if (!stack.empty())
        throw  runtime_error{
        "[Runtime_error] Invalid bracket matches of the expression :missing ') When checking the bracket matching of the expression"
    };
}
string InfixExpression::parseExpression(const  string& expression)//解析表达式
{
    string temp_expression(expression);//存储表达式解析后的临时结果
    for (int i = 0; i < temp_expression.length(); i++)
    {
        if (!judgeValidChar(temp_expression[i]))//如果出现非法字符则直接抛出错误
        {
            throw  runtime_error{ "[Runtime_error] Invalid expression is given : Expression containing invalid character When parsing the expression" };

        }
        judgeMinus(temp_expression, i);//由于-有两种含义：减号和负号，所以需要进行区分，这里把减号统一转换为~表示，-表示负号
    }
```

```
        splitOperatorsOperands(temp_expression);
        return temp_expression;
    }

    int InfixExpression::compareOperators(char op1, char op2)   {

        switch (op1)
        {
        case '(':     return -1;
        case '*':
        case '/': if (op2 == '(')   return -1;
            return 1;
        case '+':
        case '~':
            if (op2 == '+' || op2 == '~') return 1;
            return -1;
        }
        exit(1);
    }

    double InfixExpression::digitCompute(double l, char op, double r)
    {
        //进行运算并返回结果
        if (op == '+')return l + r;
        else if (op == '~')return l - r;
        else if (op == '*')return l * r;
        else if (op == '/')
        {
            if (r == 0)//除 0 则报错
            {
                throw runtime_error{ "[Runtime_error] Division by 0 error When computing" };

            }
            return l / r;
        }
        return 0;
    }

    bool InfixExpression::judgeDigitString(const   string& str)//判断是否为数字字符串
    {
        for (char ch : str)//如果字符串中全为数字或小数点或负号则为
        {
            if (!(ch == '-' || ch == '.' || isdigit(ch)))
```

```
                return false;
        }
        return true;
    }
    void InfixExpression::splitOperatorsOperands(string& expression)
    {
        //把操作符和数字分离开来
        char ops[6] = { '+', '~', '*', '/' ,'(',')' };
        for (auto op : ops)
        {
            auto pos = expression.find(op);
            //把所有操作符替换为空格+操作符+空格

            while (pos != string::npos)
            {
                expression = expression.replace(pos, 1, { ' ',op,' ' });//替换
                pos = expression.find(op, pos + 2);//继续替换后面的内容
            }

        }
    }
    bool  InfixExpression::judgeValidChar(char  ch)//用于判断表达式中是否出现非法字符
    {
        string validString = "0123456789+-*/().";
        if (validString.find(ch) == string::npos)
            return false;
        return true;
    }

    void InfixExpression::judgeMinus(string& expression, int pos)
    {
        if (pos != 0 && pos != expression.length() - 1 && expression[pos] == '-')
        {
            if (expression[pos - 1] == ')' || isdigit(expression[pos - 1]) || !isdigit(expression[pos + 1]))//如果-出现在右括号后或者数字后或者后一位是非数字,则-只能表示减号
            {
                expression[pos] = '~';//把-转为~表示减号
            }
        }

        if (pos == 0 && expression[pos] == '-' && !isdigit(expression[pos + 1])) //下一位不是数字则表示减号
```

```
            {
                expression[pos] = '~';
            }
            if (pos == expression.length() - 1 && expression[pos] == '-') //最后一位只能表
示减号
            {
                expression[pos] = '~';
            }

    }
```

(5) LinkStack.hpp

```
    #pragma once
    template<typename data_type>
    struct Node
    {
        data_type data;
        Node<data_type>* next;
    };
    template<typename data_type>
    class Stack
    {
        public:
        Stack();
        ~Stack();
        void clear();
        bool empty()const;
        void push(const data_type &item);
        data_type pop();
        data_type top();

        private:
        Node<data_type> * top_node;
    };
    template<typename data_type>
    Stack<data_type>::Stack()
    {
        top_node=NULL;
    }
    template<typename data_type>
    void Stack<data_type>:: push(const data_type &item)
    {
        Node<data_type> *s=NULL;
        s=new Node<data_type>;
```

```
        s->data=item;
        s->next=top_node;
        top_node=s;

    }
    template<typename data_type>
    bool Stack<data_type>:: empty()const
    {
        return top_node==NULL;
    }
    template<typename data_type>
    Stack<data_type>::~Stack()
    {
        while(!empty())
            pop();
    }
    template<typename data_type>
    data_type    Stack<data_type>::pop()
    {

      auto x = top();
     Node<data_type>*p=top_node;
     if (empty())
          throw underflow_error{ "" };
     top_node=top_node->next;
     delete p;
     return x;
    }
    template<typename data_type>
    data_type Stack<data_type>::top()
    {
        if(empty())
            throw underflow_error{""};
        data_type x=top_node->data;
        return x;
    }
    template<typename data_type>
    void Stack<data_type>::clear()
    {
        while (!empty())
            pop();
}
```