2023—2024 学年第二学期

《数据结构与算法导论》实验报告

班级：_____

姓名：_____

学号：_____

班内序号：_____

报告日期：_____

数据结构实验报告

1．实验要求

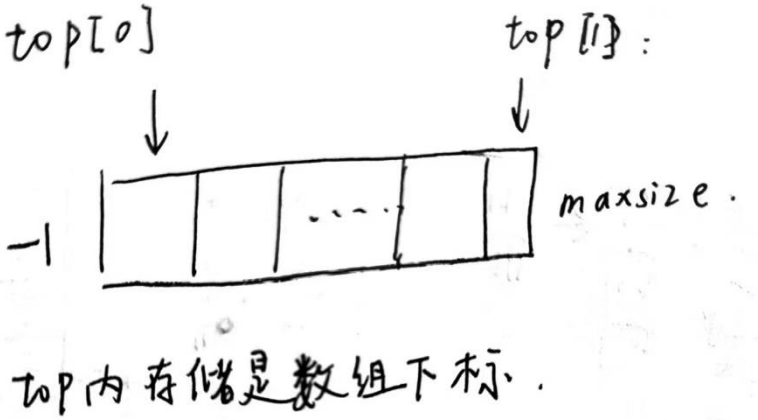
根据栈和队列的抽象数据类型的定义，按要求实现一个栈或一个队列的基本功能（四选一）。

要求：
1、实现一个共享栈
2、实现一个链栈
3、实现一个循环队列
4、实现一个链队列
编写测试 main()函数测试栈或队列的正确性

2. 程序分析

2.1 存储结构

数组构成栈，示意图如下：



2.2 关键算法分析

一、关键算法：
1、构造函数
（1）无参构造函数 sharestack()：

初始化 stacksize 为 maxsize（4）。
动态分配大小为 maxsize 的栈数组。
初始化 top[0] 为 -1，top[1] 为 stacksize。

（2）有参构造函数 sharestack(const int size)：

初始化 stacksize 为给定值 size。
动态分配大小为 stacksize 的栈数组。
初始化 top[0] 为 -1，top[1] 为 stacksize。
2、判断栈是否满
函数 isfull()：
如果 top[0] + 1 == top[1]，则栈满，返回 true。
否则，返回 false。
3、入栈操作
函数 push(T x, bool flag)：
如果栈满，抛出异常 "栈满"。
根据 flag 决定向左栈或右栈入栈：
flag == 0：左栈入栈，top[0]++，将元素 x 放入 stack[top[0]]。
flag == 1：右栈入栈，top[1]--，将元素 x 放入 stack[top[1]]。
4、出栈操作
函数 pop(bool flag)：
根据 flag 决定从左栈或右栈出栈：
flag == 0：左栈出栈，如果 top[0] != -1，top[0]--，返回 stack[top[0] + 1]，否则抛出异常 "栈空"。
flag == 1：右栈出栈，如果 top[1] != stacksize，top[1]++，返回 stack[top[1] - 1]，否则抛出异常 "栈空"。
5、打印栈
函数 printstack()：
遍历并打印左栈的元素。
打印左栈和右栈之间的 null。
遍历并打印右栈的元素。

二、代码详细分析：

1、无参构造函数
sharestack()：
a.设置栈大小 stacksize 为默认值 maxsize（4）。
b.动态分配一个大小为 maxsize 的数组，并将指针赋给 stack。
c.初始化 top[0] 为 -1，表示左栈为空。
d.初始化 top[1] 为 stacksize，表示右栈为空。
2、带参构造函数:
a.sharestack(const int size)：
b.将 stacksize 设置为传入的参数 size。
c.动态分配一个大小为 size 的数组，并将指针赋给 stack。
d.初始化 top[0] 为 -1，表示左栈为空。
e.初始化 top[1] 为 stacksize，表示右栈为空。
3、判断栈是否满:
isfull() 函数:
a.判断条件：如果 top[0] + 1 == top[1]，则栈满，返回 true。

否则，返回 false。

4、入栈操作

push(T x, bool flag) 函数:

a.如果栈已满，抛出异常 "栈满"。

b.根据 flag 值决定将元素 x 入哪个栈：

c.如果 flag == 0：

d.将元素 x 入左栈，top[0] 增加 1，stack[top[0]] 设置为 x。

e 如果 flag == 1：

f.将元素 x 入右栈，top[1] 减少 1，stack[top[1]] 设置为 x。

5、出栈操作

pop(bool flag) 函数:

a.根据 flag 值决定从哪个栈出元素：

b.如果 flag == 0：

c 如果左栈非空（top[0] != -1），将 top[0] 减少 1，返回 stack[top[0] + 1]。

d.否则，抛出异常 "栈空"。

e.如果 flag == 1：

f.如果右栈非空（top[1] != stacksize），将 top[1] 增加 1，返回 stack[top[1] - 1]。

g.否则，抛出异常 "栈空"。

h.如果 flag 值不为 0 或 1，抛出异常 "flag 错误"。

6、打印栈

printstack() 函数:

a.打印左栈的所有元素：

b.遍历从 0 到 top[0] 的索引，打印对应的 stack 元素。

c.打印左栈和右栈之间的空位（用 null 表示）：

d.从 top[0] + 1 到 top[1] - 1 的位置，打印 null。

e.打印右栈的所有元素：

f.从 top[1] 到 stacksize - 1 的索引，打印对应的 stack 元素。

三、时间空间复杂度分析：

时间复杂度分析

入栈操作 push()

入栈操作的时间复杂度为 O(1)。

无论是向左栈还是右栈入栈，都只需要执行常数次操作，即更新栈顶指针和存储元素。

出栈操作 pop()

出栈操作的时间复杂度为 O(1)。

类似入栈操作，无论是左栈还是右栈出栈，都只需执行常数次操作。

判断栈是否满 isfull()

判断栈是否满的时间复杂度为 O(1)。

只需进行一次比较操作，即判断两个栈顶指针的差值是否为 1。

打印栈 printstack()

打印栈的时间复杂度为 O(n)，其中 n 为栈的大小。

打印栈需要遍历整个栈的元素，因此时间复杂度与栈的大小成正比。

总体时间复杂度

由于各操作的时间复杂度都是 O(1)，因此整体时间复杂度取决于执行的操作次数，可以视为 O(1)。

空间复杂度分析

栈的空间复杂度：

栈的空间复杂度为 O(n)，其中 n 为栈的最大容量。

每个栈的容量都是固定的，取决于初始化时设定的大小。

额外空间：

除了栈本身的存储空间外，还需要额外的空间存储栈顶指针和类的成员变量。

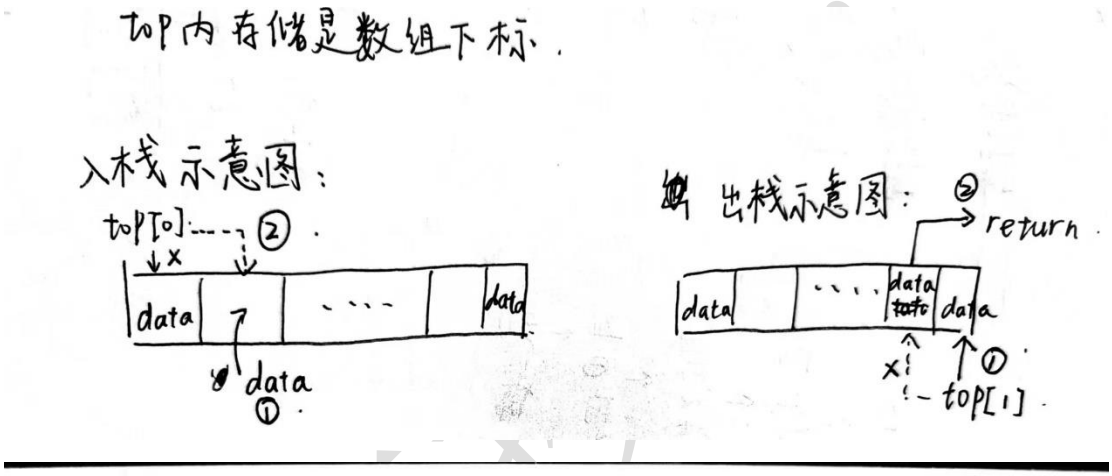
额外空间的大小与栈的最大容量无关，因此可以视为常数空间，即 O(1)。

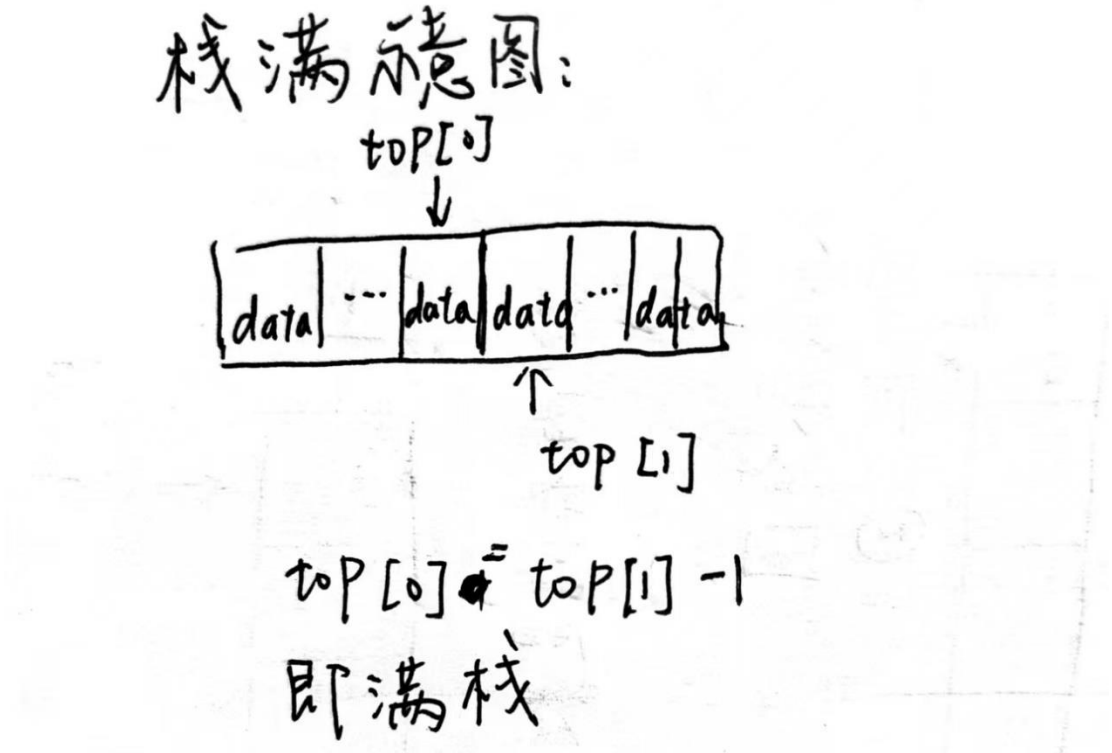
总体空间复杂度

整体空间复杂度由栈的空间复杂度和额外空间复杂度组成，可以视为 O(n)。

综上所述，代码的时间复杂度为 O(1)，空间复杂度为 O(n)

示意图如下：





3. 程序运行结果

测试主函数流程

1. 创建共享栈实例:

   创建两个共享栈实例 `stack01` 和 `stack02`，一个使用默认大小，另一个使用自定义大小。

2. 测试无参构造的共享栈

   - 对 `stack01` 进行入栈、出栈和打印操作，以测试栈的基本功能。

3. 测试自定义大小的共享栈:

   - 对 `stack02` 进行入栈、出栈和打印操作，以测试栈的基本功能。

测试条件

1. 共享栈功能测试:

   - 测试共享栈的入栈、出栈和打印操作是否能够正常工作。
   - 包括左栈和右栈的操作，以及打印栈内容。

2. 栈满条件测试:

   - 在入栈操作时，测试当栈满时是否能够正确抛出异常 "栈满"。

3. 栈空条件测试:

   - 在出栈操作时，测试当栈为空时是否能够正确抛出异常 "栈空"。

测试结论

1. 共享栈功能测试结论:

   共享栈的入栈、出栈和打印操作能够正常工作。

   左栈和右栈之间能够正确共享数组空间。

   打印栈内容能够显示左栈和右栈的元素以及空位。

2. 栈满条件测试结论:

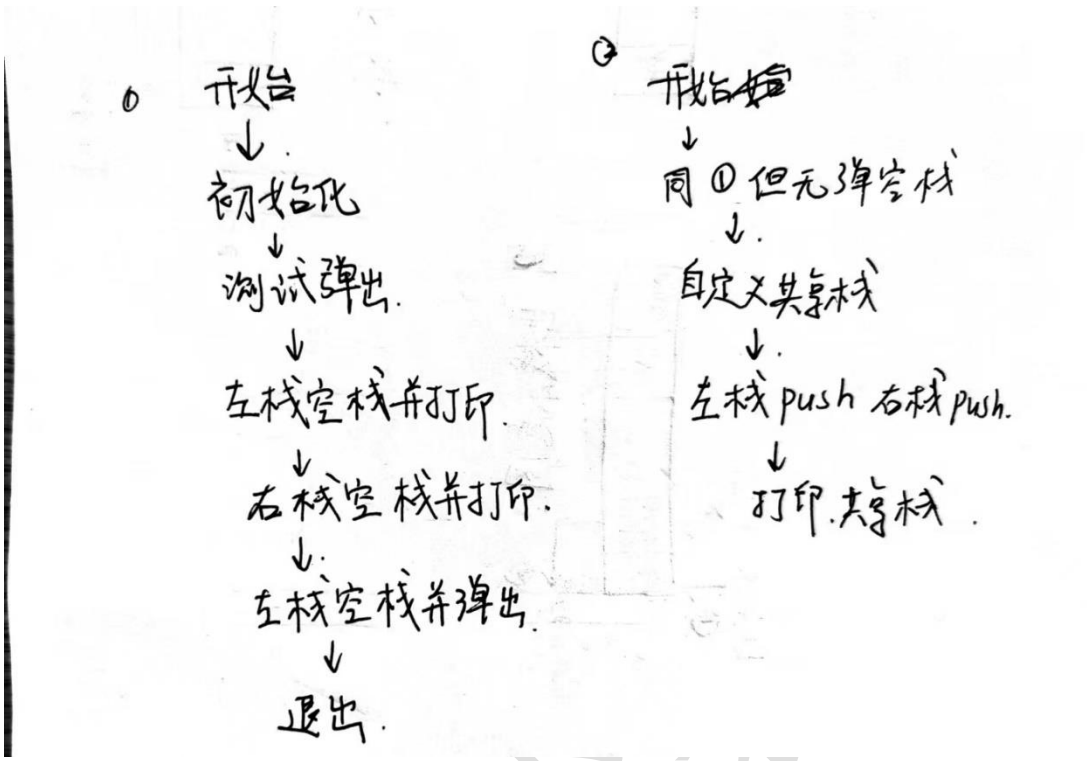
   当栈满时，入栈操作能够正确抛出异常 "栈满"，不会造成栈溢出。

3. 栈空条件测试结论

   - 当栈为空时，出栈操作能够正确抛出异常 "栈空"，避免了非法操作。

通过以上测试,可以确认共享栈的基本功能实现正确,并且在边界情况下能够正确处理异常。

示意图如下：



Main 函数代码：

```
int main()
{
    sharestack<int> stack01;//构造无参共享栈
    sharestack<int> stack02(6);//构造长度为 6 的共享栈
    cout<<"进行无参构造初始化测试"<<endl;
    stack01.push(23,0);
    stack01.push(88,0);
    stack01.push(45,1);
    stack01.push(99,1);
    stack01.printstack();
    cout<<endl;
    cout<<"进行弹出测试"<<endl;
    int temp=0;
    temp = stack01.pop(0);
    cout<<"弹出的是："<<temp<<endl;
    stack01.push(77,1);
    stack01.printstack();
    cout<<endl;
    cout<<"进行左栈空栈测试"<<endl;
    int temp1=0;
    temp1=stack01.pop(0);
    cout<<"弹出的是:"<<temp1<<endl;
    stack01.printstack();
```

```
    cout<<endl;
    cout<<"进行右栈的空栈测试"<<endl;
    stack01.pop(1);
    stack01.printstack();
    cout<<endl;
    stack01.pop(1);
    stack01.pop(1);
    stack01.printstack();
    //stack01.pop(0);//会退出程序的代码
    cout<<endl;
    cout<<"进行自定义构造共享栈"<<endl;
    stack02.push(1,0);
    stack02.push(2,1);
    stack02.printstack();
    cout<<endl;
    system("pause");
    return 0;
}
```

Main 函数测试结果：

1、 不含空栈弹出的异常处理代码

```
PS D:\C++working> ^C
PS D:\C++working>
PS D:\C++working>  & 'c:\Users\lenovo\.vscode\extensions\ms-vscode.cpptools-1.20.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-xuuws24n.eb2' '--stdout=Microsoft-MIEngine-Out-y4xe31wu.0gu' '--stderr=Microsoft-MIEngine-Error-2psauis2.hem' '--pid=Microsoft-MIEngine-Pid-sax053b3.lvg' '--dbgExe=D:\mingw64\bin\gdb.exe' '--interpreter=mi'
进行无参构造初始化测试
23|88|99|45
进行弹出测试
弹出的是：88
23|77|99|45
进行左栈空栈测试
弹出的是:23
null|77|99|45
进行右栈的空栈测试
null|null|99|45
null|null|null|null|
进行自定义构造共享栈
1|null|null|null|2
```

含有异常处理代码：

```
PS D:\C++working>  & 'c:\Users\lenovo\.vscode\extensions\ms-vscode.cpptools-1.20.5-win32-x64\debugA
5d0.mkj' '--stdout=Microsoft-MIEngine-Out-w1fu5eko.5et' '--stderr=Microsoft-MIEngine-Error-ijru5ygh
\gdb.exe' '--interpreter=mi'
进行无参构造初始化测试
23|88|99|45
进行弹出测试
弹出的是：88
23|77|99|45
进行左栈空栈测试
弹出的是:23
null|77|99|45
进行右栈的空栈测试
null|null|99|45
null|null|null|null|terminate called after throwing an instance of 'char const*'
PS D:\C++working>
```

4. 总结

本实验难度总体不大，在写代码的过程中遇到的最大问题应该是打印栈这里了，因为这里的弹栈写的都是伪弹栈，即只将 top 进行移动，而并不真正的删除栈的数值，所以在输出时要以 top 为标准，而不要以 stack 指向的数组。

还有一点就是在输出 null 的空栈时，我写了三个 if 语句，这里是很容易重复的，注意 if 语句之间的逻辑关系。

1．实验要求

试设计一个算术四则运算表达式求值的简单计算器。

要求：

操作数均为非负整数常数

操作符仅为+、-、*、/、(、)

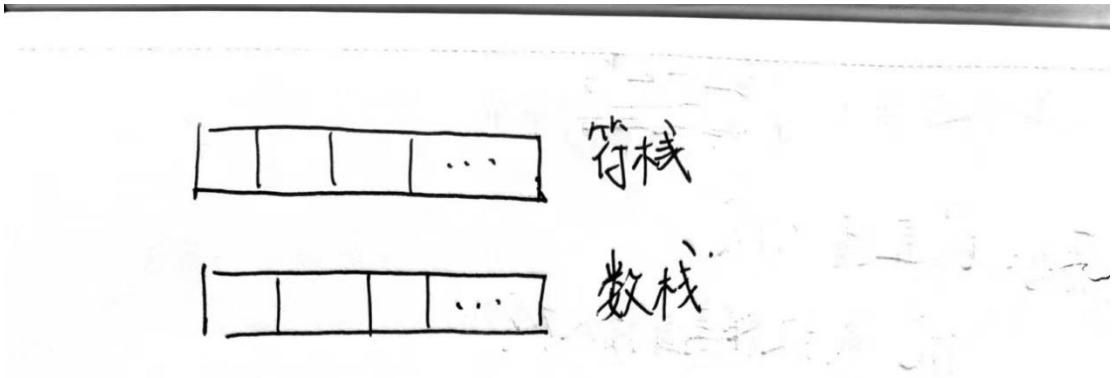
编写 main 函数进行测试

2. 程序分析

2.1 存储结构

存储结构为库函数中的栈

示意图如下：



2.2 关键算法分析

1、关键算法分析：

a. 设置优先级函数

```
FUNCTION setOper():
    // 设置操作符的优先级
    FOR 每个操作符 op IN {'+', '-', '*', '/', '('}:
        IF op 是 '+' 或 '-':
            x[op] = ADD_SUB
        ELSE IF op 是 '*' 或 '/':
            x[op] = MUL_DEV
        ELSE IF op 是 '(':
            x[op] = LEFT_BR
```

b. 判断是否为数字函数

```
FUNCTION isDigital(c: char) -> bool:
    // 判断字符是否是数字
    IF c 在 '0' 和 '9' 之间:
        RETURN true
    ELSE:
        RETURN false
```

c. 将字符转换为数字

```
FUNCTION TurnToNum(a: char) -> float:
    // 将字符转换为数字
    RETURN (a - '0')
```

D. 关键函数：计算函数！

```
FUNCTION calc(k: char[]) -> float:
    DECLARE fu AS mystack<char> // 符号栈
```

```
DECLARE shu AS mystack<float> // 数字栈
DECLARE status AS PRE // 记录前一个数据类型
CALL setOper() // 设置操作符的优先级
DECLARE num AS float // 用于接收数字
DECLARE ch AS char // 用于接收操作符
WHILE k 中还有字符:
    IF 当前字符是数字:
        IF 前一个状态是数字:
            将当前字符转换为数字，加到已有数字的末尾
        ELSE:
            将当前字符转换为数字，赋给 num
        更新状态为 NUM
    ELSE: // 当前字符是操作符
        将当前字符赋给 ch
        IF 前一个状态是数字:
            将已经累积的数字压入数字栈
        IF 符号栈为空 或 当前操作符优先级高于栈顶操作符:
            将当前操作符压入符号栈
        ELSE IF 栈顶操作符是左括号:
            将当前操作符压入符号栈
        ELSE IF 当前操作符是右括号:
            WHILE 符号栈顶不是左括号:
                从数字栈弹出两个数字 a 和 b
                根据栈顶操作符进行计算，并将结果压入数字栈
                弹出符号栈顶元素
            IF 符号栈顶是左括号:
                弹出左括号
        ELSE: // 当前操作符的优先级低于栈顶操作符
            从数字栈弹出两个数字 a 和 b
            根据栈顶操作符进行计算，并将结果压入数字栈
            从符号栈弹出栈顶操作符
            将当前操作符压入符号栈
        更新状态为 OPER
    移动到下一个字符
IF 最后一个字符是数字且状态为 NUM:
    将已经累积的数字压入数字栈
WHILE 符号栈不为空:
    从数字栈弹出两个数字 a 和 b
    根据栈顶操作符进行计算，并将结果压入数字栈
    从符号栈弹出栈顶操作符
返回数字栈顶元素
```

2、代码详细分析：

a.枚举类型定义

定义了两个枚举类型：

PRIO：用于表示操作符的优先级，包括右括号、加减法、乘除法和左括号。
PRE：用于表示前一个数据类型，包括起始状态、数字和操作符。

b.数组初始化和栈重写
定义了全局字符数组 x[255] 用于存储操作符的优先级。
重写了 mystack 类继承自标准库的 stack 类，修改了 pop() 函数的行为，使得在栈为空时抛出异常。

c.设置操作符优先级和判断函数
setOper() 函数设置了操作符的优先级，将每个操作符映射到其对应的优先级。
isDigital() 函数用于判断一个字符是否是数字，如果字符是数字则返回 true，否则返回 false。

d.转换字符为数字函数
TurnToNum() 函数用于将一个字符表示的数字转换为对应的实际数字。

e.关键函数：
calc() 函数详细分析

1 初始化栈和状态变量：
声明两个自定义栈 fu（用于存储操作符）和 shu（用于存储操作数）。
声明状态变量 status，用于记录前一个数据类型，初始状态为 START。

2 遍历输入表达式：
通过循环遍历输入表达式的每一个字符。
对于每个字符进行不同的处理：
如果是数字字符，则转换为实际数字，并根据状态累加或覆盖数字。
如果是操作符字符，则根据当前状态和栈内操作符优先级进行处理。

3 处理数字字符：
如果当前字符是数字字符：
如果前一个状态是数字，将当前数字字符转换为实际数字，并累加到已有的数字后面。
如果前一个状态不是数字，将当前数字字符转换为实际数字，并作为新的数字。

4 处理操作符字符：
如果当前字符是操作符字符：
如果前一个状态是数字，将已经累积的数字压入数字栈。
根据当前操作符和栈内操作符的优先级进行不同的处理：
如果栈为空或者当前操作符优先级高于栈顶操作符，则将当前操作符压入操作符栈。
如果栈顶操作符是左括号，直接将当前操作符压入操作符栈。
如果当前操作符是右括号，则执行弹栈操作，直到遇到左括号为止，并计算中间结果。

5 操作符优先级比较：
在处理操作符字符时，根据当前操作符和栈顶操作符的优先级进行比较，决定压栈还是弹

栈计算。

如果当前操作符优先级高于栈顶操作符，直接将当前操作符压入栈。

如果当前操作符优先级低于或等于栈顶操作符，则执行弹栈操作，直到满足压栈条件。

6 结束字符处理：

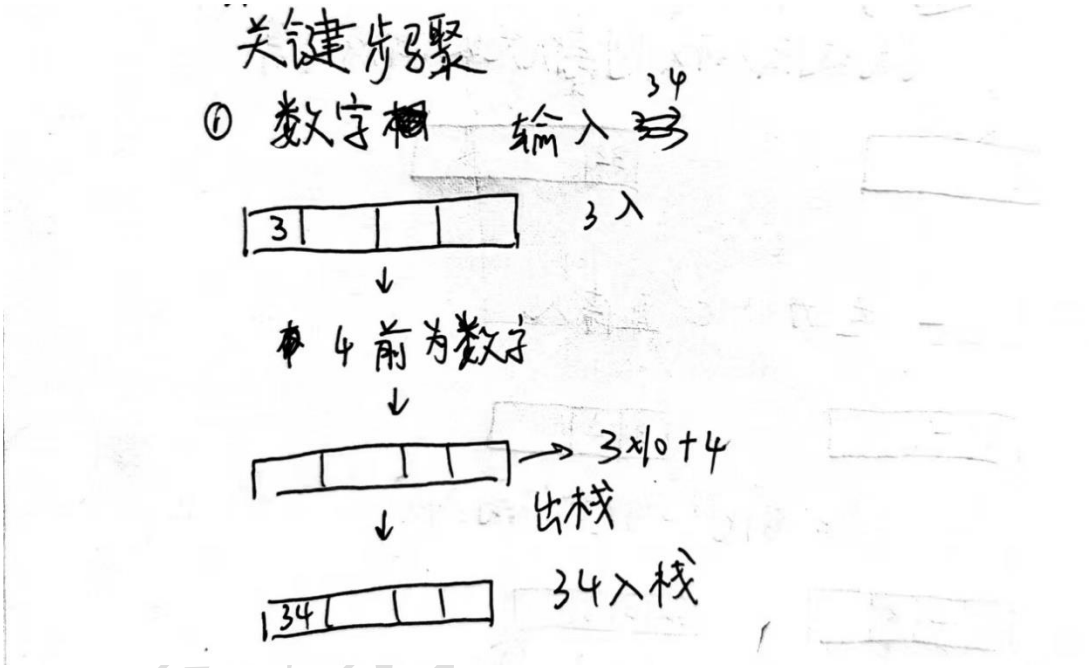
如果遍历结束后，最后一个字符是数字且状态为数字，则将已经累积的数字压入数字栈。

7 清空栈并计算结果：

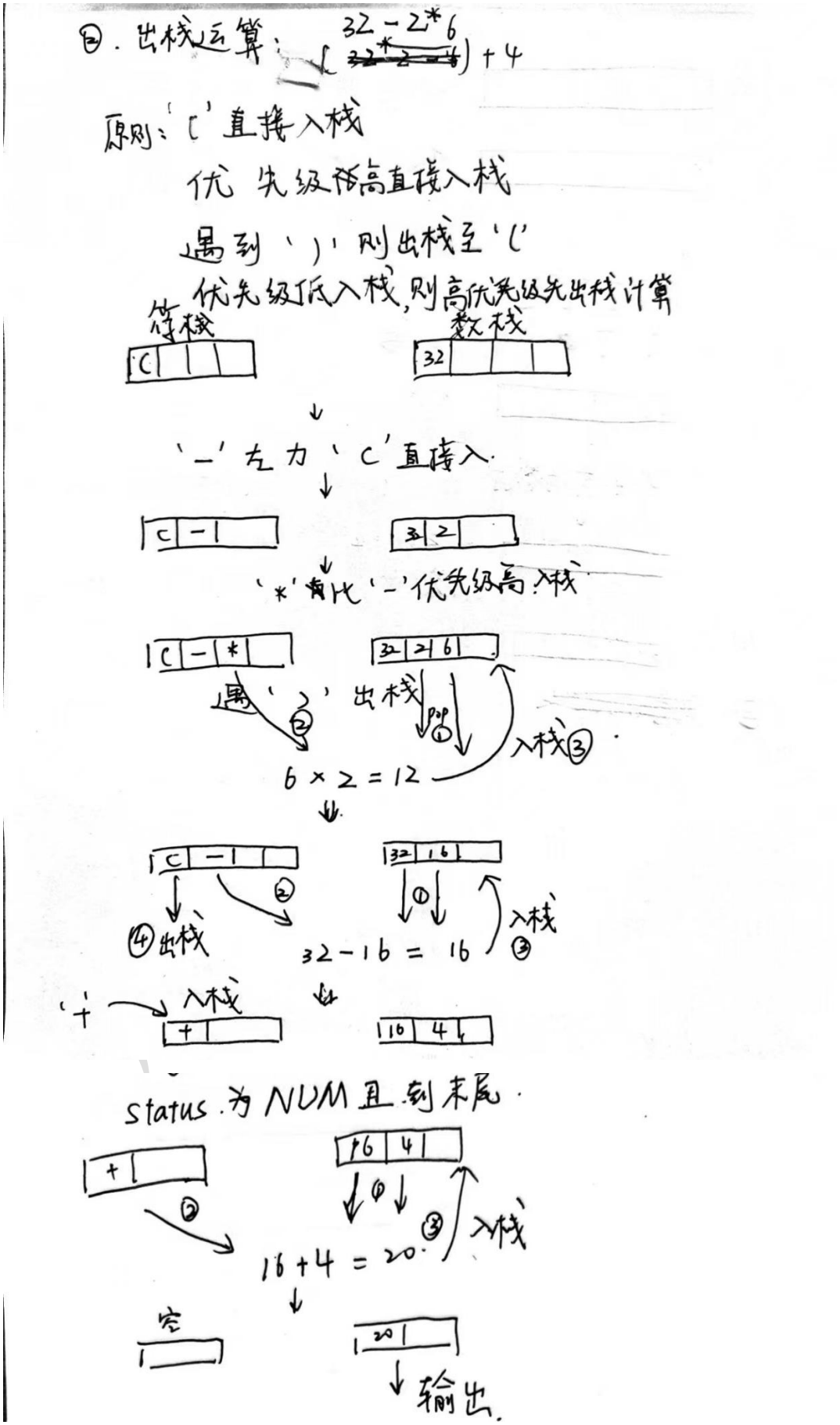
如果操作符栈不为空，则继续弹栈并计算中间结果，直到操作符栈为空。

最终返回数字栈顶元素，即为表达式的计算结果。

示意图如下：



计算函数关键示意图：


③. 出栈运算：32-2*6
(32-2*6)+4
原则：'('直接入栈
优先级低直接入栈
遇到')'则出栈至'('
优先级低入栈，则高优先级先出栈计算
符栈
数栈
'-'左力'('直接入
'*'比'-'优先级高入栈
遇')'出栈
入栈
6×2=12
32-16=16
'+'入栈
status为NUM且到末尾
16+4=20
空
输出

三、代码时间和空间复杂度分析：

时间复杂度分析

遍历输入表达式：

时间复杂度为 O(n)，其中 n 为表达式的长度，因为需要遍历每个字符。

处理数字字符：

对于每个数字字符的处理是常数时间操作，因此时间复杂度为 O(1)。

处理操作符字符：

涉及到栈操作，包括压栈、弹栈和比较操作，因此时间复杂度取决于栈的大小和操作。

如果没有括号，栈的大小最多为 O(n/2)，其中 n 为表达式长度，因此栈操作的时间复杂度为 O(n)。

如果有括号，则每个括号对应的操作可能会导致一系列的弹栈操作，但总体仍然是线性时间复杂度。

结束字符处理：

如果最后一个字符是数字，则将数字压入栈，为常数时间操作，时间复杂度为 O(1)。

清空栈并计算结果：

如果栈中还有剩余元素，则需要进行弹栈和计算操作，时间复杂度为 O(n)。

总体时间复杂度：

综合考虑以上各个步骤的时间复杂度，整体时间复杂度为 O(n)。

空间复杂度分析

栈的空间复杂度：

包含两个栈 fu 和 shu，用于存储操作符和操作数。

如果没有括号，则栈的大小最多为 O(n/2)，其中 n 为表达式长度。

如果有括号，则栈的大小可能会达到表达式长度的大小，最大为 O(n)。

因此，栈的空间复杂度为 O(n)。

其他变量和数组：

除了栈外，还有一些辅助变量和数组，包括状态变量 status、全局字符数组 x 等。

这些辅助变量和数组的空间复杂度为常数级别，因此可以忽略不计。

总体空间复杂度：

根据以上分析，整体空间复杂度主要由栈的空间复杂度决定，为 O(n)。

通过以上分析，我们了解到 calc() 函数的时间复杂度为 O(n)，空间复杂度为 O(n)。

3. 程序运行结果

1、主函数流程

a.输出欢迎信息和使用注意事项：

输出欢迎信息，提示用户欢迎使用计算器。

b.输出使用注意事项，提示用户注意事项，如输入法设置、输入限制等。

确认用户准备好使用计算器：

提示用户确认是否准备好使用计算器，并确认用户输入。

c.获取用户输入并进行计算：

如果用户确认准备好使用计算器，获取用户输入的待计算表达式。

调用 calc() 函数进行表达式计算，并输出计算结果。

2、测试条件

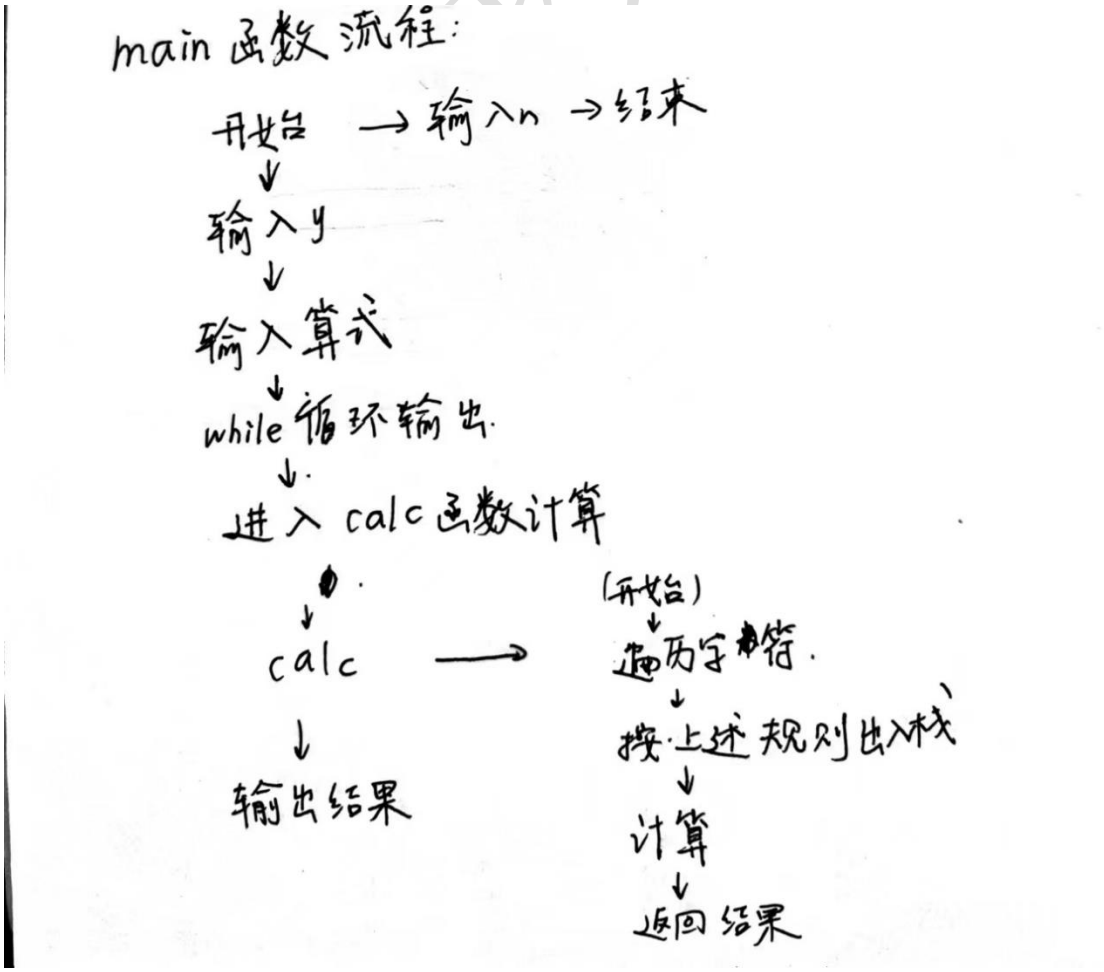
输入待计算表达式：
用户输入了待计算的合法表达式，包含数字、加减乘除和括号等。
计算结果输出：
程序成功计算出表达式的结果，并将结果正确输出。

3、测试结论

a. 程序功能正常：
用户能够顺利使用计算器，输入合法表达式。
程序能够正确识别和计算输入的表达式，并输出正确的计算结果。

b. 输入约束生效：
用户必须确认准备好使用计算器，并且输入法设置为英文输入法。
用户输入的表达式必须是合法的，只包含数字、加减乘除和括号等。

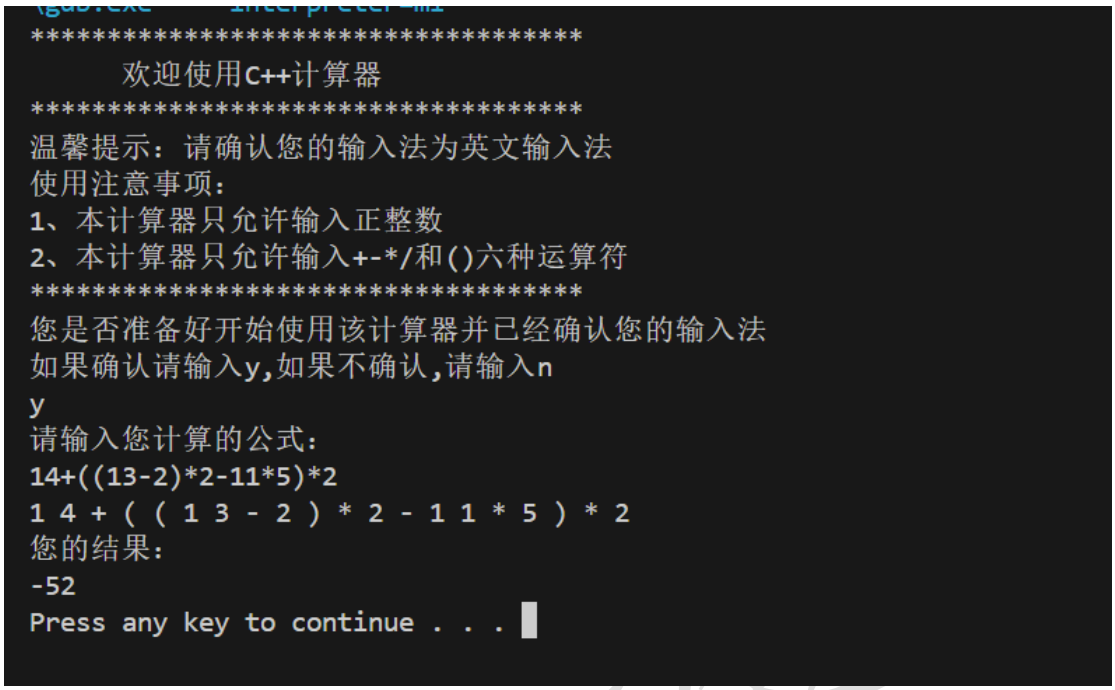
c. 计算结果正确：
程序能够正确处理不同优先级的运算符，以及括号的运算顺序。
输出的计算结果与预期结果一致，证明计算器功能正常。
通过以上测试，我们可以确认程序能够正常运行，用户能够顺利使用计算器进行表达式计算，并且计算结果正确。
示意图如下：



Main 函数如下：

```
int main()
 {
    char duihua;
    cout<<"************************************"<<endl;
    cout<<"       欢迎使用 C++计算器       "<<endl;
    cout<<"************************************"<<endl;
    cout<<"温馨提示：请确认您的输入法为英文输入法"<<endl;
    cout<<"使用注意事项："<<endl;
    cout<<"1、本计算器只允许输入正整数"<<endl;
    cout<<"2、本计算器只允许输入+-*/和()六种运算符"<<endl;
    cout<<"************************************"<<endl;
    cout<<"您是否准备好开始使用该计算器并已经确认您的输入法"<<endl;
    cout<<"如果确认请输入 y,如果不确认,请输入 n"<<endl;
    cin>>duihua;
    if(duihua=='y'){
    char question[100];
    cout<<"请输入您计算的公式："<<endl;
    char non[1];
    cin.getline(non,1);//因为输入 y 产生的回车会默认使 getline 获取所以需要有
缓冲
    cin.getline(question,100);
    char *q=question;
    while((*q)!='\0')
    {
        cout<<*q<<" ";
        q++;
    }
    cout<<endl;
    float m =calc(question);
    cout<<"您的结果："<<endl;
    cout<<m<<endl;
    }
    else
    {
        return 0;
    }
    system("pause");
    return 0;
 }
```

Main 函数测试结果如下：



测试结果与计算相符，这里是因为感觉课件上的测试比较完成包含了各种条件

4、总结

这个题目还是耗费了很多很多心血。他的过程很复杂而且步骤很多，最需要付出努力的就是在完成代码后的调试阶段。其实计算器的背后逻辑比较清晰，就是遵循几个符号入栈弹栈的原则即可，但是在写代码的时候很容易出现一些小问题，这就导致在调试的时候需要付出很大努力。我在调试的时候就是从最简单的 1+1 开始进行测试，竟然发现我的初始代码连这个都跑不动，当时真的崩溃。后来逐渐增加了（），然后增加了十位数，和不同的运算。经过这次调试，完全懂得了使用 C++的调试功能。

这次的代码有一些值得提出来的点：首先是设置优先级，这里其实涉及到了一个强制值类型转化，就是将 x 数组【】里面的符号转成对应的 ascii 码值，然后给相应的下角标赋值不同的优先级。这个思路是借鉴了课上的。然后是在写符号出栈入栈规则时，if 条件语句的判断一定要十分的清晰，不同类型的判断，而且要弄明白 if 和 else if 的关系，不能把两者的判断条件写重，要不符号入栈会错乱。还有一点就是最后在美化我的计算器时，有一个输入的细节，是无意中偶然发现的，就是需要在里面设置一个小的 cin.getline 的缓冲区，否则在输入 y 后回车会自动认为 cin.getline 里面的，就会导致跳过下一个 cin 阶段，所以需要有一个类似于缓冲区的东西。