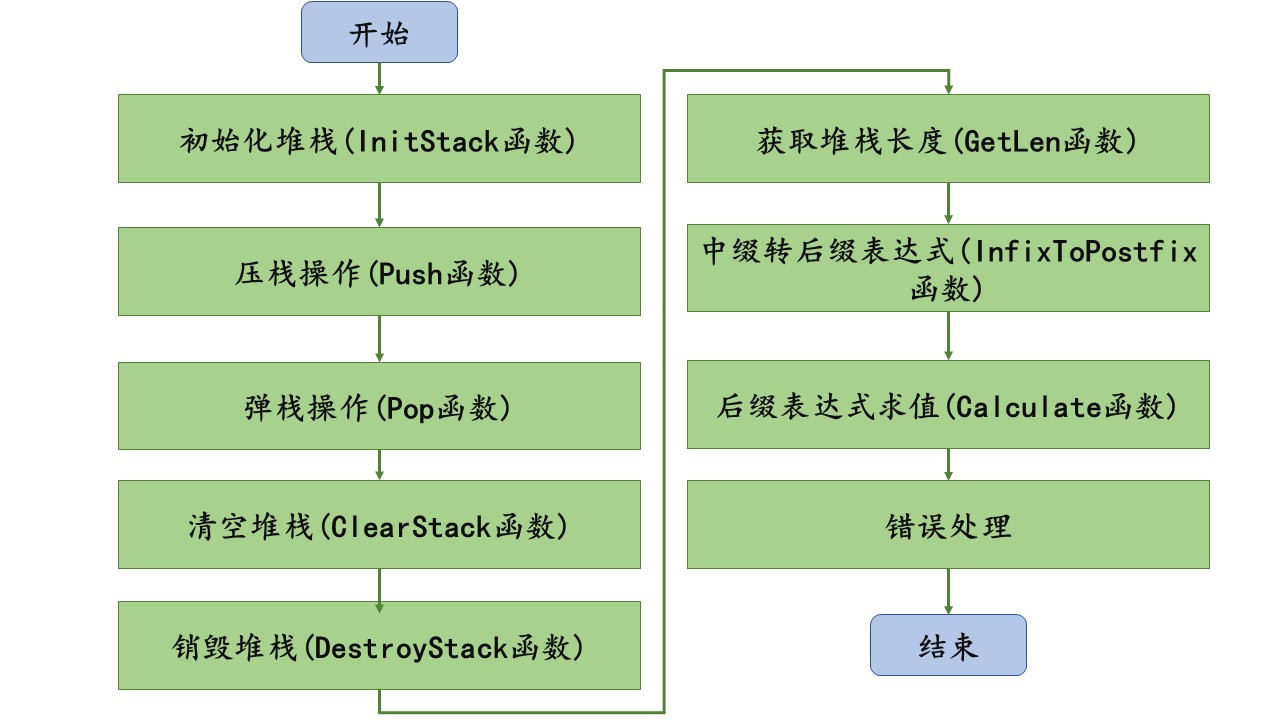
**堆栈应用: 逆波兰式解决四则运算报告**

（一）设计目标

本报告介绍了使用堆栈数据结构来解决四则运算表达式的计算问题。逆波兰式（Reverse Polish Notation，RPN）是一种无需括号的运算表达式表示方法，它通过将运算符放在操作数之后的方式来表示运算顺序。我们利用堆栈的先进后出（LIFO）特性，将逆波兰式表达式转换为计算步骤，并实现了一个逆波兰式计算器。

功能要求

输入算术表达式，输出表达式的值。如输入： 1 + 4 \* （4 - 1.5）- （5 / 2 ) ；输出结果：8.5

（二）总体设计

本设计主要实现了一个基于堆栈的中缀表达式计算器。以下是设计的要点和主要实现步骤：

1.数据结构：

使用了一个顺序堆栈数据结构（sqStack）来存储操作数和运算符。

堆栈的底部和顶部指针分别指向堆栈的起始位置和当前栈顶位置。

定义了堆栈的大小（stackSize）和元素类型的大小（typeSize）。

2.初始化堆栈：

通过调用InitStack函数初始化堆栈。

函数动态分配堆栈的内存空间，并进行错误检查。

初始化堆栈的底部和顶部指针，设置堆栈的大小和元素类型的大小。

3.压栈操作：

使用Push函数将元素压入堆栈。

检查堆栈是否已满，避免栈溢出。

使用memcpy函数将元素复制到堆栈的顶部，并更新栈顶指针。

4.弹栈操作：

使用Pop函数从堆栈中弹出元素。

检查堆栈是否为空，避免空栈错误。

更新栈顶指针，并使用memcpy函数将元素复制到指定的内存位置。

5.清空堆栈：

使用ClearStack函数清空堆栈。

将堆栈的顶部指针指向堆栈的底部，即将栈置空。

6.销毁堆栈：

使用DestroyStack函数销毁堆栈。

释放堆栈的内存空间，并将堆栈的指针和大小设置为初始状态。

7.获取堆栈长度：

使用GetLen函数获取堆栈中元素的个数。

通过计算栈顶指针和堆栈底部指针之间的距离，并除以元素类型的大小得到长度。

8.中缀转后缀表达式：

使用InfixToPostfix函数将中缀表达式转换为后缀表达式。

依次扫描中缀表达式的每个字符。

根据运算符的优先级和结合性，将运算符压栈或输出到后缀表达式中。

处理括号，并处理完后将剩余的运算符输出到后缀表达式中。

9.后缀表达式求值：

使用Calculate函数计算后缀表达式的值。

依次扫描后缀表达式的每个字符。

如果遇到操作数，将其压入堆栈。

如果遇到运算符，从堆栈中弹出相应数量的操作数，进行运算，并将结果压入堆栈。

最终堆栈中的唯一元素即为后缀表达式的计算结果。

10．错误处理：

在程序中定义了几个预定义的常量，如TRUE、FALSE和ERROR，用于表示操作的状态或错误。

如果发生错误，相应的函数将返回适当的错误代码，可以根据需要进行错误处理，例如打印错误信息并退出程序。

11.主函数：

主函数实现了整个计算器的交互过程。

首先，从标准输入中读取用户输入的中缀表达式。

然后，调用InfixToPostfix函数将中缀表达式转换为后缀表达式，并打印转换结果。

接下来，调用Calculate函数计算后缀表达式的值，并打印计算结果。

最后，退出程序。

（三）详细步骤：

1.功能函数的实现

1）定义常量

1. #define TRUE 1
2. #define FALSE 0
3. #define ERROR -1
4. #define STACKELEMENT 100 *//定义堆栈中可入栈元素个数*
5. #define BUFFERSIZE 100  *//定义临时缓冲区大小*

2）定义结构体

1. typedef int Status; *//定义返回状态*
2. typedef struct  //定义堆栈：堆栈中只定义栈大小和每次入栈元素大小，栈中一律使用空指针进行操作。这样，我们自己定义的栈就做到了和数据类型无关性。真正实现了栈的通用。
3. {
4. void \*base; *//指向栈底*
5. void \*top;  *//指向栈顶*
6. int stackSize; *//栈的空间总容量大小*
7. int typeSize; *//入栈时，单个元素占用空间大小*
8. } sqStack;

3）将中缀表达式转换为后缀表达式

参数:infix 指向中缀表达式，以回车键即\n结尾。

postfix 指向后缀表达式临时缓冲区，用来存放转换后的结果。

转换规则：从左到右遍历中缀表达式的每个数字和符号，若是数字则直接保存在postfix数组中；若是符号，则判断其与栈顶符号的优先级，是右括号或者优先级不大于栈顶符号，则栈顶元素依次出栈并输出，直到遇到左括号或者栈空时，才将刚才的那个符号入栈。

1. */\**
2. 将中缀表达式转换为后缀表达式
3. 参数:infix 指向中缀表达式，以回车键即\n结尾。
4. postfix 指向后缀表达式临时缓冲区，用来存放转换后的结果。
5. 附转换规则：从左到右遍历中缀表达式的每个数字和符号，若是数字则直接保存在postfix数组中；若是符号，则判断其与栈顶符号的优先级，是右括号或者优先级不大于栈顶符号，则栈顶元素依次出栈并输出，直到遇到左括号或者栈空时，才将刚才的那个符号入栈。
6. \*/
7. Status InfixToPostfix(char \*infix,char \*postfix)
8. {
9. sqStack s;
10. if(InitStack(&s,STACKELEMENT\*sizeof(char),sizeof(char))==ERROR)
11. my\_err("InfixToPostfix init stack error！");
12. char c,e;
13. int j=0,i=0;
14. c=\*(infix+i); *//取出中缀表达式中的第一个字符*
15. i++;
16. while('\n'!=c) *//遇到换行符，表示转换结束*
17. {
18. while(c>='0'&&c<='9'|| c=='.') *//先判断一下取出的字符是否是数字，如果是数字的话，则直接存入postfix数组*
19. {
20. postfix[j++]=c;
21. c=\*(infix+i);
22. i++;
23. if((c<'0'||c>'9')&&(c!='.')) *//如果不是数字，则在后面添加空格，以便区分各个符号*
24. {
25. postfix[j++]=' ';
26. }
27. }
28. if(')'==c) *//不是数字，则判断是否为右括号。[括号的优先级最高，所以，如果是右括号的话，就得先进行括号里的各种运算]*
29. {
30. Pop(&s,(void\*)&e);
31. while('('!=e) *//直到遇到左括号为止*
32. {
33. postfix[j++]=e;
34. postfix[j++]=' ';
35. Pop(&s,(void\*)&e);
36. }
37. }
38. else if('+'==c||'-'==c) *//如果是加减号,因为他俩的优先级最低了，所以此时先将栈里的所有符号出栈后(除非遇到左括号)，再把此符号入栈*
39. {
40. if(!GetLen(&s)) *//如果是空栈，则直接将加减号入栈*
41. {
42. Push(&s,(void\*)&c);
43. }
44. else
45. {
46. do
47. {
48. Pop(&s,(void\*)&e);
49. if('('==e)
50. {
51. Push(&s,(void\*)&e);
52. }
53. else
54. {
55. postfix[j++]=e;
56. postfix[j++]=' ';
57. }
58. }while(GetLen(&s)&&'('!=e);  *//将栈里的所有符号出栈(除非遇到左括号)*
59. Push(&s,(void\*)&c); *//最后将新来的加减号再入栈*
60. }
61. }
62. else if('\*'==c||'/'==c||'('==c) *//如果是乘除号或左括号，因为他们的优先级高，所以直接入栈。*
63. {
64. Push(&s,(void\*)&c);
65. }
66. else if('\n'==c) *//判断一下，所有符号是否都已转换完成*
67. {
68. break;
69. }
70. else *//能走到这个else的，都是我不认识的符号了*
71. {
72. *// printf("\nError:input error,the character %d cann't recognize!\n",c);*
73. return -1;
74. }
75. c=\*(infix+i); *//取出下一个字符进行转换*
76. i++;
77. }
78. while(GetLen(&s)) *//转换完成后，栈里可能还有没出栈的运算符号*
79. {
80. Pop(&s,(void\*)&e);
81. postfix[j++]=e;
82. postfix[j++]=' ';
83. }
84. DestroyStack(&s);
85. return TRUE;
86. }

4）计算后缀表达式的结果

参数：arr使用空格分隔的后缀表达式字符串。例：arr="31 5 + "

result 保存计算完毕后的结果

如何利用栈来计算后缀表达式的结果：依次取出后缀表达式中的符号进行比较，如果是数字，则直接入栈；如果是符号，则出栈两次，弹出两个要计算的因数，进行计算，之后再将计算结果入栈。知道后缀表达式中所有符号都已比较完毕。

1. */\**
2. 计算后缀表达式的结果
3. 参数：arr使用空格分隔的后缀表达式字符串。例：arr="31 5 + "
4. result 保存计算完毕后的结果
5. 注：如何利用栈来计算后缀表达式的结果：依次取出后缀表达式中的符号进行比较，如果是数字，则直接入栈；如果是符号，则出栈两次，弹出两个要计算的因数，进行计算，之后再将计算结果入栈。知道后缀表达式中所有符号都已比较完毕。
6. \*/
7. Status Calculate(char \*arr,void \*result)
8. {
9. *// printf("%s\n",arr);*
10. double d,e,f; *//d,e 存放两个因数。f存放d,e计算后的结果.*
11. sqStack s;
12. char \*op; *//存放后缀表达式中的每个因数或运算符*
13. char \*buf=arr; *//声明bufhe saveptr两个变量，是strtok\_r函数的需要。*
14. char \*saveptr=NULL;
15. if(InitStack(&s,STACKELEMENT\*sizeof(double),sizeof(double))==ERROR)
16. my\_err("Calculate init stack error!");
17. while((op=strtok\_r(buf," ",&saveptr))!=NULL) *//利用strtok\_r函数分隔字符串*
18. {
19. buf=NULL;
20. switch(op[0])
21. {
22. case '+':
23. Pop(&s,&d);
24. Pop(&s,&e);
25. f=d+e;
26. Push(&s,&f);
27. break;
28. case '-':
29. Pop(&s,&d);
30. Pop(&s,&e);
31. f=e-d;
32. Push(&s,&f);
33. break;
34. case '\*':
35. Pop(&s,&d);
36. Pop(&s,&e);
37. f=d\*e;
38. Push(&s,&f);
39. break;
40. case '/':
41. Pop(&s,&d);
42. Pop(&s,&e);
43. f=e/d;
44. Push(&s,&f);
45. break;
46. default:
47. d=atof(op); *//不是运算符，就肯定是因数了。所以，用atof函数，将字符串转换为double类型*
48. Push(&s,&d);
49. break;
50. }
51. }
52. Pop(&s,result);
53. DestroyStack(&s);
54. return TRUE;
55. }

5）初始化栈

参数：stackSize:栈的总容量大小

typeSize:以后要入栈的单个元素的大小

实现逻辑：InitStack函数用于初始化一个具有指定堆栈大小和元素类型大小的堆栈s。

s->base = malloc(stackSize); ：根据stackSize参数为堆栈动态地分配内存。堆栈的基点指针被设置为分配的内存。

if (!s->base) return ERROR;：检查内存分配是否成功。如果基点指针为NULL，表明内存分配失败，函数返回ERROR以表明错误情况。

s->top = s->base;：将堆栈的顶部指针设置为基点指针，表示一个空的堆栈。

s->stackSize = stackSize;：在堆栈结构的stackSize成员变量中存储堆栈大小。

s->typeSize = typeSize;：将元素类型大小存储在堆栈结构的typeSize成员变量中。

return TRUE;：如果初始化成功，该函数返回TRUE，表示成功。

InitStack函数通过为堆栈分配内存，设置基指针和顶指针，以及存储堆栈和元素类型大小来初始化堆栈。如果初始化成功，它返回TRUE，如果内存分配失败，则返回ERROR。

1. */\**
2. 初始化栈
3. 参数：stackSize:栈的总容量大小
4. typeSize:以后要入栈的单个元素的大小
5. \*/
6. Status InitStack(sqStack \*s,unsigned stackSize,unsigned typeSize)
7. {
8. s->base=malloc(stackSize);
9. if(!s->base)
10. return ERROR;
11. s->top=s->base;
12. s->stackSize=stackSize;
13. s->typeSize=typeSize;
14. return TRUE;
15. }

6）压栈

Push函数用于将一个元素e推到堆栈s上。

(int)s->top - (int)s->base + s->typeSize：通过从顶层指针中减去基层指针并加上元素类型大小，计算出堆栈的当前大小。

如果（(int)s->top - (int)s->base + s->typeSize > s->stackSize）：检查计算出来的大小是否超过了最大的堆栈大小（s->stackSize）。如果是，堆栈已经满了，函数返回FALSE，表示推送元素失败。

memcpy(s->top, e, s->typeSize)：如果堆栈未满，则使用memcpy函数将元素e的内容复制到s->top指向的内存位置。这假定为堆栈分配的内存有足够的空间来容纳该元素。

s->top = (void\*)((int)s->top + s->typeSize)：在推送元素后，顶部指针s->top被元素类型大小增加，指向堆栈中下一个可用的空空间。

return TRUE: 如果推送操作成功，该函数返回TRUE，表示成功。

Push函数检查堆栈中是否有足够的空间来推送一个元素。如果有，它将该元素复制到堆栈的顶部，更新顶部指针，并返回TRUE。如果堆栈已满，则返回FALSE。

1. */\**
2. 压栈
3. \*/
4. Status Push(sqStack \*s,void \*e)
5. {
6. if((int)s->top-(int)s->base+s->typeSize>s->stackSize)
7. return FALSE;
8. memcpy(s->top,e,s->typeSize);
9. s->top=(void\*)((int)s->top+s->typeSize);
10. return TRUE;
11. }

7）出栈

Pop函数用于从堆栈s中弹出（移除）最上面的元素，并将其值存储在e所指向的内存位置。

如果（s->top == s->base）：通过比较顶层指针s->top和底层指针s->base来检查栈是否为空。如果它们相等，说明堆栈是空的，该函数返回FALSE，表示弹出元素失败。

s->top = (void\*)((int)s->top - (int)s->typeSize) ：如果堆栈不是空的，用元素类型大小递减顶部指针s->top。这将使顶部指针移动到堆栈中的前一个元素。

memcpy(e, s->top, s->typeSize)：将更新后的顶部指针s->top处的元素内容复制到e所指的内存位置。

return TRUE: 如果pop操作成功，该函数返回TRUE以表示成功。

Pop函数检查堆栈是否为空。如果它不是空的，它将移除顶部元素，更新顶部指针，并将元素的值复制到e所指向的内存位置，然后返回TRUE。如果堆栈是空的，则返回FALSE。

1. */\**
2. 出栈
3. \*/
4. Status Pop(sqStack \*s,void \*e)
5. {
6. if(s->top==s->base)
7. return FALSE;
8. s->top=(void\*)((int)s->top-(int)s->typeSize);
9. memcpy(e,s->top,s->typeSize);
10. return TRUE;
11. }

8）清栈

ClearStack函数用于清除（清空）堆栈s，将顶层指针s->top重置为底层指针s->base。

s->top = s->base：将基指针s->base的值分配给顶指针s->top。这有效地将堆栈的顶部设置到与基数相同的位置，表示一个空堆栈。

return TRUE：返回TRUE，表示堆栈已经被成功清除了。

ClearStack函数只是将堆栈的顶部指针重置为基点，有效地清空了堆栈。

1. */\**
2. 清栈
3. \*/
4. Status ClearStack(sqStack \*s)
5. {
6. s->top=s->base;
7. return TRUE;
8. }

9）毁栈

DestroyStack函数用于通过释放为堆栈分配的内存和重置其成员来销毁堆栈s。

free(s->base)：通过使用free函数释放为堆栈分配的内存。这就把之前在InitStack函数中使用malloc分配的内存取消了。

s->top = s->base = NULL：将顶层指针s->top和底层指针s->base都设置为NULL。这确保堆栈不再被访问或使用。

s->stackSize = s->typeSize = 0: 将堆栈大小s->stackSize和类型大小s->typeSize设置为零，表明堆栈不再有任何大小或类型信息。

return TRUE：返回TRUE，表示堆栈已被成功销毁。

DestroyStack函数释放了为堆栈分配的内存，将指针和成员设置为NULL或零，并表示成功销毁了堆栈。

1. */\**
2. 毁栈
3. \*/
4. Status DestroyStack(sqStack \*s)
5. {
6. free(s->base);
7. s->top=s->base=NULL;
8. s->stackSize=s->typeSize=0;
9. return TRUE;
10. }

10）获取已入栈元素个数

GetLen函数用于计算并返回堆栈s的长度或大小，即它所包含的元素数量。

((int)s->top - (int)s->base)：计算s->top和s->base所指向的地址之间的差异。这给出了堆栈中的元素所占用的总字节数。

((int)s->top-(int)s->base)/s->typeSize：用总的字节数除以每个元素的大小（s->typeSize）。这就给出了堆栈中的元素数量。返回计算值作为堆栈的长度。

1. */\**
2. 获取已入栈元素个数
3. \*/
4. int GetLen(sqStack \*s)
5. {
6. return ((int)s->top-(int)s->base)/s->typeSize;
7. }

11）自定义错误处理函数

1. */\**
2. 自定义错误处理函数
3. \*/
4. void my\_err(char \*str)
5. {
6. perror(str);
7. exit(1);
8. }

2.主程序实现

打印一个提示，要求用户输入一个infix表达式。

声明并初始化变量，用于存储infix和postfix表达式。

使用 fgets 从标准输入中读取infix表达式并将其存储在infix缓冲区中。

打印输入的infix表达式。

调用InfixToPostfix函数将infix表达式转换为postfix表达式，并将其存储在postfix缓冲区。

打印结果的后缀表达式。

声明一个用于存储计算结果的变量 result。

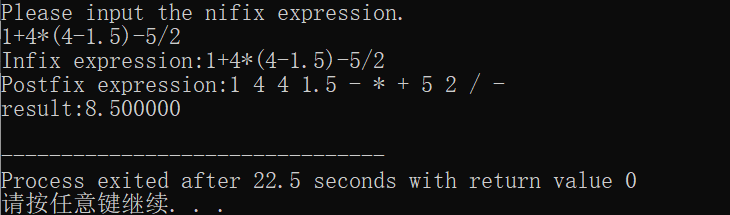
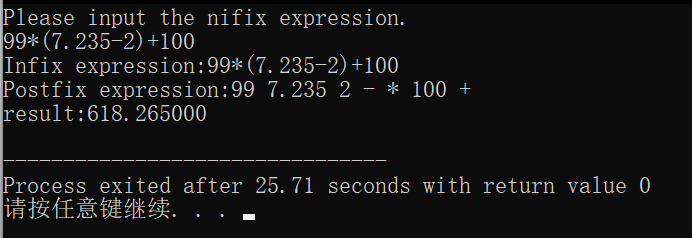
调用Calculate函数来计算后缀表达式，并将结果存储在结果变量中。

打印计算结果。

退出程序，成功终止。

读取一个infix表达式，将其转换为postfix，计算结果，并显示出来。

1. int main(int argc,char \*argv[])
2. {
3. printf("Please input the nifix expression.\n"); *//输入中缀表达式。*
4. char src[BUFFERSIZE]={'\0'}; *//存放中缀表达式的临时缓冲区*
5. char \*infix=src;
6. char postfix[BUFFERSIZE]={'\0'}; *//存放后缀表达式的临时缓冲区*
7. fgets(infix,BUFFERSIZE,stdin); *//从标准输入流中读取要计算的四则运算表达式*
8. printf("Infix expression:%s",infix);
9. InfixToPostfix(infix,postfix); *//将中缀转换为后缀表达式*
10. printf("Postfix expression:%s\n",postfix);
11. double result;
12. Calculate(postfix,&result); *//计算后缀表达式的结果*
13. printf("result:%f\n",result);
14. exit(0);
15. }

（四）运行结果展示

（五）设计总结

本设计实现了一个基于堆栈的中缀表达式计算器，通过将中缀表达式转换为后缀表达式，并利用堆栈进行求值，实现了四则运算的计算功能。设计使用了顺序堆栈数据结构，提供了初始化、压栈、弹栈、清空、销毁和获取长度等操作。通过合理的算法和数据结构设计，实现了中缀转后缀的转换和后缀表达式的求值，使得计算器具有了较好的扩展性和灵活性。同时，通过适当的错误处理机制，增强了程序的健壮性和用户体验。

在完成这个项目的过程中，我获得了以下技能和知识：

熟悉了堆栈数据结构的实现：通过设计和实现顺序堆栈，我深入理解了堆栈的基本概念和操作。我学会了如何初始化堆栈、入栈和出栈操作，以及堆栈的一些常见应用。

熟练运用指针和内存管理：在堆栈的实现中，我大量使用了指针操作，并进行了内存动态分配和释放。这让我更加熟悉了指针的概念、用法和内存管理的重要性。

掌握了中缀表达式转后缀表达式的算法：通过实现中缀表达式转后缀表达式的算法，我深入理解了算法的设计思路和实现步骤。这个过程让我对栈的应用有了更深入的认识，并提高了我解决问题的能力。

错误处理和异常情况处理：在项目中，我注意到了错误处理的重要性，并学会了使用预定义的常量和错误代码来标识和处理异常情况。这让我更加注重代码的健壮性和用户体验，同时也增加了我对代码质量的要求。

加强了编程和调试能力：通过实际编码和调试的过程，我提高了自己的编程技巧和调试能力。我学会了使用调试工具和技巧来定位和解决问题，同时也对代码的组织和结构有了更深入的思考。

通过完成这个项目，我不仅掌握了堆栈数据结构和相关算法，还提高了自己的编程能力和问题解决能力。这些收获将对我的日后学习和工作都有很大的帮助。