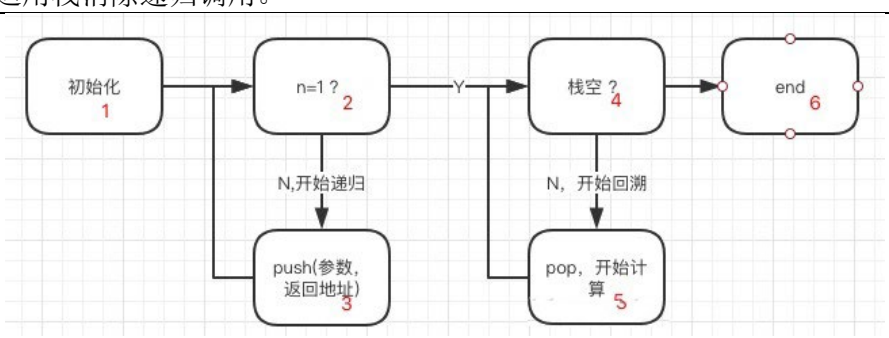


《数据结构》上机报告

2020 年 10 月 20 日

姓名： 林日中 学号： 1951112 班级： 10072602 得分：

实验题目	链表实验报告 - 运用栈模拟阶乘函数的调用过程	
问题描述	<p>栈（stack）又名堆栈，它是一种运算受限的线性表。限定仅在表尾进行插入和删除操作的线性表。这一端被称为栈顶，相对地，把另一端称为栈底。向一个栈插入新元素称作进栈、入栈或压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素；从一个栈删除元素称作出栈或退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素。</p> <p>实验目的：</p> <ol style="list-style-type: none"> 1、掌握栈的结构和基本操作； 2、理解函数调用的递归和回溯过程； 3、运用栈消除递归调用。 	
基本要求	 <ol style="list-style-type: none"> 1、参考这个状态机，用栈模拟 n 的阶乘的递归调用过程。 2、测试一下当 n 超过多少时，递归函数会出现堆栈溢出的错误。用栈消解递归后是否会出现错误。 	
	已完成基本内容（序号）：	1, 2
选做要求		
	已完成选做内容（序号）：	无
数据结构设计	<p>在本次上机实验中，我设计了两个数据结构：<code>struct MyData</code> 和 <code>struct MyStack</code>，分别表示栈中存储的基础数据类型和栈。</p> <p><code>struct MyData</code> 存储了数据 <code>int data</code>。</p> <p><code>struct MyStack</code> 存储了指向动态分配的基础数据类型数组 <code>stack_pointer base</code> 和 <code>stack_pointer top</code>；栈的容量 <code>size_t capacity</code>，初始容量 <code>size_t init_capacity</code>，每次栈满后新增的容量 <code>const size_t stack_increment</code>。</p>	

功能
(函数)
说明

```
struct MyData
{
    int data;
    // int returnAddress // only needed for linked stacks
    MyData(const int n = 0) : data(n) {}
    int get() const { return data; }
    int set(const int n) { return data = n; }
    MyData &operator=(const int n) ...
    operator int() const { return data; }
}; // struct MyData
```

struct MyData 的构造函数设置了 1 个默认参数 0，功能是在有参数传入时构造 data 值为参数 / 无参数传入时 data 值为 0 的 MyData 对象。成员函数 get() 和 set() 分别实现了取得 data 值和设置 data 值的功能。运算符=的重载函数 operator=() 和类型转换函数 operator int() 实现了便捷的赋值和类型转换操作。

```
struct MyStack
{
    mystack_ptr base; // pointer of data segment; !*DO NOT CHANGE ITS VALUE!*
    mystack_ptr top;
    size_t capacity;
    size_t init_capacity = 10;
    const size_t stack_increment = 50;
    MyStack(const size_t s = 10) : base(new SElemType[s]), top(base), capacity(s), init_capacity(s) {}
    ~MyStack() { delete[] base; }
    SElemType get_top() const { return top == base ? MyData() : *(top - 1); }
    SElemType push(const SElemType &e) ...
    SElemType pop() { return top == base ? SElemType(ERROR) : *--top; }
    int empty() const { return base == top; }
    int get_size() const { return top - base; }
}; // struct MyStack
```

struct MyStack 的构造函数的功能是动态申请一个基础元素类型数组，并将其赋给 top 和 base；将传入的 / 默认容量值赋给 capacity 和 init_capacity，至此完成构造的操作。成员函数 get_top() 能够在栈不空的时候返回栈顶元素的值，否则返回 data 值为 0 的 MyData 对象；push() 函数能在栈满时将动态申请的数组“扩容”，并将传入的参数压入栈内；pop() 函数能够将栈顶元素弹出，并返回它；函数 empty() 和 get_size() 分别返回栈是否满和栈当前的元素个数。

```
// test the functions of struct MyStack
long long test_mystack()
{
    // initialize the stack
    cout << "Welcome! This function is committed to calculating the sum of numbers from 1 to a particular number. " << endl;
    << "Please enter a positive integer: ";
    int num = 0;
    cin >> num;
    MyStack s(num);

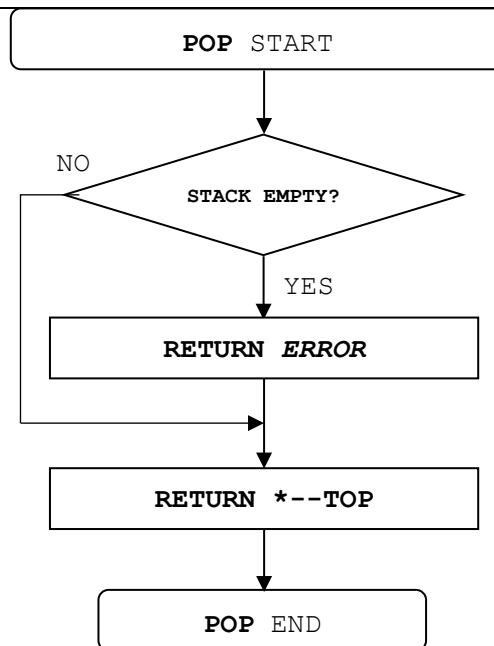
    // push elements
    for (int i = 1; i <= num; i++)
    {
        s.push(MyData(i));
    }
    cout << "Current size of my stack : " << s.get_size() << endl;

    // pop elements and sum them
    long long sum = 0; // sum of stack elements popped
    for (int i = 0; i < num; i++)
    {
        sum += s.pop().get();
    }
    cout << "The sum of integers within [1.." << num << "] : " << sum << endl;
    return sum;
}
```

test_mystack() 函数封装了测试栈功能的操作。首先输入一个数字 num，作为栈深度，然后循环依次向栈中压入 data 值为从 num 至 1 的 MyData 对象。循环结束后，依次从栈中弹出对象，并用变量 sum 累加各弹出元素的值。最后输出 sum，并返回 sum。（由于 100 的阶乘已经超过 long long int 的上限，不能很好地测试栈功能，故将乘法改为乘法，也能比较好地体现数据结构的功能。）

开发环境	Windows 10, C++ language Visual Studio Code with g++, Visual Studio Community 2019 with MSVC
调试分析	<div><pre>#include <iostream> const int num = 4000; int foo(const int n) { std::cout << num - n << std::endl; return n ? n + foo(n - 1) : 0; } int main() { std::cout << foo(num) << std::endl; return 0; }</pre></div> <p>首先测试当 n 超过多少时，递归函数会出现堆栈溢出的错误。源码如上。以 32bit MSVC 编译器为例。</p> <div><pre>3990 3991 3992 3993 3994 3995 3996 3997 3998 3999 4000 8002000</pre></div> <p>将 num 设为 4000 时，程序功能完好，成功输出 0~4000 的和。</p> <div><pre>const int num = 5000; int foo(const int n) { std::cout << num - n << std::endl; return n ? n + foo(n - 1) : 0; } int main() { std::cout << foo(num) << std::endl; return 0; }</pre><div>Exception Unhandled Unhandled exception at 0x7C932B19 (ucrtbased.dll) in new.exe: 0xC00000FD: Stack overflow (parameters: 0x00000000, 0x00302000).</div><pre>4656 4657 4658 4659 4660 4661 4662 4663 4664 4665 4666 4667 4668 4669 ...</pre></div> <p>将 num 设为 5000 时，发生了栈溢出。经过多次测试，栈深度约为 4660。</p> <p>接下来用 <code>test_mystack()</code> 函数测试设计的数据结构。</p> <div><pre>Welcome! This function is committed to calculating the sum of numbers from 1 to a particular number. Please enter a positive integer: 123456789 Current size of my stack : 123456789 The sum of integers within [1..123456789] : 7620789436823655</pre></div> <p>观察到函数封装的操作成功计算出了从 1 到 123456789 的和。</p> <p>综上，本实验设计的数据结构很好地完成了题目对栈的功能的要求，功能完好，使用方便，并且能够较好地处理非法数据的输入并反馈相关错误提示；程序界面友好，具有比较恰当的人性化提醒，具有一定的鲁棒性。</p>

心得体会	<div data-bbox="300 197 528 230"><p>一、 实验总结</p></div> <div data-bbox="300 268 1450 490"><p>在本次上机实验中，我复习了理论课上学到的栈的定义和作用，将课本上栈的顺序存储结构和与其相关的建立、压入元素、弹出元素等函数样例封装成了 <code>struct MyStack</code> 结构体。</p><p>在题中对 <code>struct Data</code> 结构的提示中，老师让我们判断 <code>int returnAddress</code> 是否必要。我的理解是，如果采用链栈，则需要该变量来存储栈中上一元素的值，而本实验中使用的顺序栈则不需要。</p></div> <div data-bbox="300 528 528 562"><p>二、 性能分析</p></div> <div data-bbox="312 600 619 710"><p>(1) 栈元素的压入 时间复杂度为 $O(1)$。 流程图如下：</p></div> <div data-bbox="619 710 1118 1355"><pre>graph TD; Start([PUSH START (IN E)]) --> Full{STACK FULL?}; Full -- NO --> Inc[*TOP++ = E]; Full -- YES --> Realloc[REALLOC AN EXPANSION]; Realloc --> Inc; Inc --> End([PUSH END]);</pre><p>The flowchart illustrates the process of pushing an element E onto a stack. It begins with a start node labeled "PUSH START (IN E)". An arrow leads to a decision diamond labeled "STACK FULL?". If the answer is "NO", the flow proceeds directly to a process rectangle labeled "$*TOP++ = E$". If the answer is "YES", the flow proceeds to a process rectangle labeled "REALLOC AN EXPANSION". From there, an arrow leads to the same process rectangle "$*TOP++ = E$". Finally, an arrow leads from this rectangle to an end node labeled "PUSH END".</p></div> <div data-bbox="312 1397 619 1507"><p>(2) 栈元素的弹出 时间复杂度为 $O(1)$。 流程图如下：</p></div>
------	---

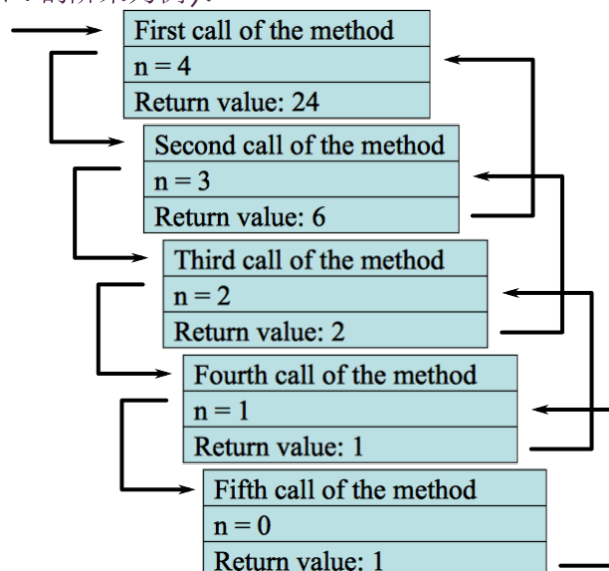


而通过栈来模拟递归函数的操作，本质是 n 个 push 操作和 n 个 pop 操作。其时间复杂度为 $O(n)$ 。

(3) 递归函数实现整数的阶乘（累加）

$$\begin{aligned}
 T(n) &= T(n-1) + O(1) \\
 &= T(n-2) + O(1) + O(1) \\
 &= T(n-3) + O(1) + O(1) + O(1) \\
 &= \dots\dots \\
 &= O(1) + \dots + O(1) + O(1) + O(1) \\
 &= n * O(1) \\
 &= O(n) \quad \text{故时间复杂度为 } O(n)。
 \end{aligned}$$

流程图如下（以 4 的阶乘为例）：



源码

```
1. #ifdef __GNUC__
2. #include <bits/stdc++.h>
3. #endif // __GNUC__
4. #ifdef _MSC_VER
5. #define _CRT_SECURE_NO_WARNINGS
6. #include <iostream>
7. #include <cstdio>
8. #include <cstdlib>
9. #include <stack>
10. #endif // _MSC_VER
11.
12. using namespace std;
13.
14. struct MyData
15. {
16.     int data;
17.     // int returnAddress; // only needed for linked stacks
18.     MyData(const int n = 0) : data(n) {}
19.     int get() const { return data; }
20.     int set(const int n) { return data = n; }
21.     MyData &operator=(const int n)
22.     {
23.         data = n;
24.         return *this;
25.     }
26.     operator int() const { return data; }
27. }; // struct MyData
28.
29. #define TRUE 1
30. #define FALSE 0
31. #define OK 1
32. #define ERROR 0
33. #define INFEASIBLE -1
34. typedef int Status;
35. typedef int Boolean;
36. typedef MyData SElemType, *mystack_ptr;
37.
38. struct MyStack
39. {
40.     mystack_ptr base; // pointer of data segment; !*DO NOT CHANGE ITS VALUE!*
41.     mystack_ptr top;
42.     size_t capacity;
43.     size_t init_capacity = 10;
44.     const size_t stack_increment = 50;
```

```

45.   MyStack(const size_t s = 10) : base(new SElemType[s]), top(base), capacity(s), init_c
    apacity(s) {}
46.   ~MyStack() { delete[] base; }
47.   SElemType get_top() const { return top == base ? MyData() : *(top - 1); }
48.   SElemType push(const SElemType &e)
49.   {
50.       if ((size_t)(top - base) >= capacity)
51.       { // realloc
52.           mystack_ptr new_base = new SElemType[capacity + stack_increment];
53.           copy(base, base + capacity - 1, new_base);
54.           top = new_base + capacity;
55.           capacity += stack_increment;
56.           delete[] base;
57.           base = new_base;
58.       }
59.       *top++ = e;
60.       return SElemType(e);
61.   }
62.   SElemType pop() { return top == base ? SElemType(ERROR) : *--top; }
63.   int empty() const { return base == top; }
64.   int get_size() const { return top - base; }
65. }; // struct MyStack
66.
67. // test the functions of struct MyStack
68. long long test_mystack()
69. {
70.     // initialize the stack
71.     cout << "Welcome! This function is committed to calculating the sum of numbers from 1
        to a particular number. " << endl
72.         << "Please enter a positive integer: ";
73.     int num = 0;
74.     cin >> num;
75.     MyStack s(num);
76.
77.     // push elements
78.     for (int i = 1; i <= num; i++)
79.     {
80.         s.push(MyData(i));
81.     }
82.     cout << "Current size of my stack : " << s.get_size() << endl;
83.
84.     // pop elements and sum them
85.     long long sum = 0; // sum of stack elements popped
86.     for (int i = 0; i < num; i++)

```

```
87.     {
88.         sum += s.pop().get();
89.     }
90.     cout << "The sum of integers within [1.." << num << "]" : " << sum << endl;
91.     return sum;
92. }
93.
94. int main()
95. {
96.     test_mystack();
97.     return 0;
98. }
```