PA2 实验报告

——运用栈模拟阶乘函数的调用过程

姓名：刘博洋 学号：2153538 日期：2022 年 10 月 24 日

1. 涉及数据结构和相关背景

本次实验涉及的主要数据结构是栈。从数据结构的角度看，栈也是一种线性表，是一种操作受限的线性表，栈的特点是后进先出，仅仅在表尾进行插入和删除操作。本实验的主要任务是用栈来模拟函数的递归调用。当一个函数运行期间调用另一函数时，在运行被调用函数之前，系统需要先将所有实参、返回地址等传递给被调用函数，并且将被调用函数的局部变量分配储存区域，并且将控制转移到被调用函数的入口。在返回调用函数之前，系统将保留被调用函数的计算结果，并且释放调用函数的数据区。递归函数是函数调用的一种特殊形式，即函数不断调用自身，在达到某种条件后停止调用得到结果。为了保证递归函数正确执行，系统需要设立一个递归工作栈作为整个递归函数运行期间的数据储存区。每一层工作区构成一个工作记录。每进入一层递归，就把所有的实参、局部变量和返回地址压入栈中，每退出一层递归，就从栈顶弹出一个工作记录，则当前执行曾的工作记录笔试递归工作栈栈顶的工作记录。为了在系统外部模拟其调用过程，可以用栈来模拟，即函数每一次调用自身时，编译器会将参数和返回地址入栈，保留函数执行的现场；当函数返回时，编译器将这些值出栈。

递归通常有两个过程：

（1）递归过程：不断递归调用入栈，直到满足停止调用的条件。

（2）回溯过程：不断回溯执行并出栈，计算结果，直到栈空，结束整个过程。

给出栈的抽象数据类型定义：

ADT Stack {

数据对象：D={ai | ai ∈ElemSet, i=1,2, …,n, n≥0}

数据关系：R1={<ai-1, ai > | ai-1, ai ∈D, i=2, …, n}

约定 an 端为栈顶，a1 端为栈底

基本操作：

① InitStack(&s) 构造一个空栈 s。

② StackEmpty(s) 若 s 为空栈，返回 1，否则返回 0。

③ Push(&s, x)

进栈。插入 x 为栈顶元素。

④ Pop(&s, x)

退栈。若 s 非空，删除栈顶元素，

并用 x 返回其值。

⑤ GetTop(s)

返回栈顶元素。

⑥ ClearStack (&s)

将栈 s 清为空栈。

⑦ StackLength(s) 返回栈 s 的元素个数。

⑧ DestroyStack(&s) 销毁栈

⑨ StackTraverse(S,visit()) 遍历栈

}

2. 实验内容

2.1 问题描述

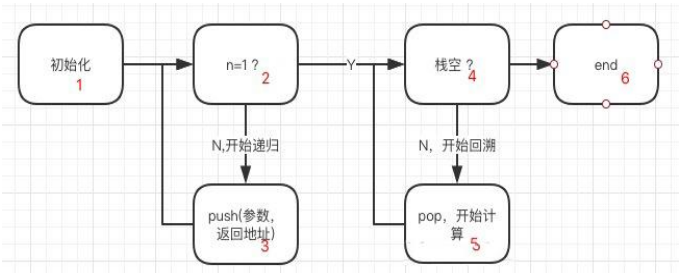

图 1 递归实现阶乘的流程图

参考这个状态机，用栈模拟 n 的阶乘递归调用过程。

```
public long f(int n){
   if(n==1) return 1; //停止调用
   return n*f(n-1); //调用自身
}
```

2.2 基本要求

(1)程序要添加适当的注释，程序的书写要采用缩进格式 。

(2)程序要具在一定的健壮性，即当输入数据非法时，程序也能适当地做出反应,如插入删除时指定的位置不对等等。

(3)程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。

(4)测试当 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。

2.3 数据结构设计

要在系统外部模拟递归函数调用，即需要弄清楚栈中包含的数据元素，由以上分析，系统中的递归工作栈每进入一层递归，就把所有的实参、局部变量和返回地址压入栈中，每退出一层递归，就从栈顶弹出一个工作记录，则当前执行曾的工作记录笔试递归工作栈栈顶的工作记录。为了模拟这一过程，我们可以在自己建立的栈中放置两个数据元素，一个为当前的 n 值，一个为回溯函数时得到的上一个函数的返回值 returnaddress。由此，在每一次开始递归时，把当前的 n 值压入栈中，然后 n--，再递归进入下一层，再将 n 值压入栈中，直到最后一次 n--后 n=1，此时开始回溯操作，每一次将深一层的返回值回溯到外层的函数并进行计算，并将函数的工作形式出栈，直到栈中得到最后的结果。　　以 n=5 为例：

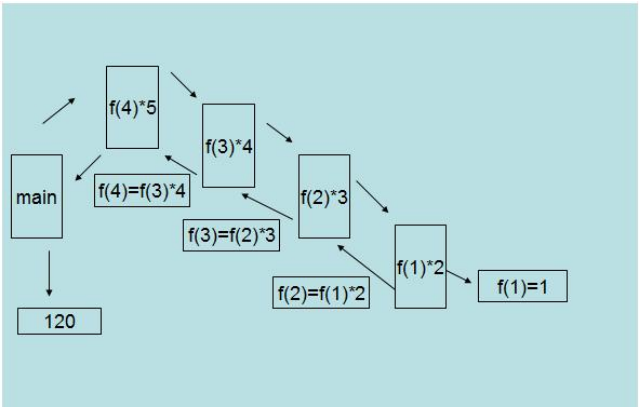

图 2 函数递归和回溯过程

层次/层数	实参/形参	调用形式	需要计算的表达式	需要等待的结果
1	n=5	factorial(5)	factorial(4) * 5	factorial(4) 的结果
2	n=4	factorial(4)	factorial(3) * 4	factorial(3) 的结果
3	n=3	factorial(3)	factorial(2) * 3	factorial(2) 的结果
4	n=2	factorial(2)	factorial(1) * 2	factorial(1) 的结果
5	n=1	factorial(1)	1	无

表 1 递归入栈过程

层次/层数	调用形式	需要计算的表达式	从内层递归得到的结果 (内层函数的返回值)	表达式的值 (当次调用的结果)
5	factorial(1)	1	无	1
4	factorial(2)	factorial(1) * 2	factorial(1) 的返回值，也就是 1	2
3	factorial(3)	factorial(2) * 3	factorial(2) 的返回值，也就是 2	6
2	factorial(4)	factorial(3) * 4	factorial(3) 的返回值，也就是 6	24
1	factorial(5)	factorial(4) * 5	factorial(4) 的返回值，也就是 24	120

表 2 回溯计算过程

2.4 功能说明（函数、类）

```
#include<iostream>
#include<string.h>
#include<conio.h>
using namespace std;
#define OVERFLOW 3
#define OK 1
#define ERROR -1
typedef int Status;
typedef struct {
    int n; //函数的输入参数
    int returnAddress;
        //构造器及 getter、setter
}Data;//节点类型，两个数据域
typedef struct {
    Data *base;
    Data* top;
    int stacksize;
    int stacklength;
}sqstack;
sqstack L;
Status Initstack(sqstack& L,int n)
```

```
{
    L.base = (Data*)malloc(n * sizeof(Data));
    if (!L.base)
        exit(OVERFLOW);//内存分配失败
    L.top = L.base;
    L.stacksize = n;
    return OK;
}
Status push(sqstack& L,Data a)
{
    if (L.top - L.base == L.stacksize)//如果栈已经满，则需要重新分配内存
        L.base = (Data*)realloc(L.base, 100 * sizeof(Data));
    if (!L.base)
        exit(OVERFLOW);
    L.top++;
    *L.top = a;
    return OK;
}
Status pop(sqstack& L ,Data a)
{
    if (L.top == L.base)
        return ERROR;
    L.top--;
}
long long int model(sqstack& L, Data a)
{
    int num = a.n;
    while (a.n > 1)
    {
        push(L, a);
        a.n--;//只要n>1，一直进栈，模拟递归调用的过程，每一次调用n都要--
    }
    L.top->returnAddress = a.n;//最内层的函数返回值为n的末尾值1
    while (a.n < num)
    {
        Data* p = --L.top;//储存上一层的地址
        L.top++;
        p->returnAddress = L.top->n * L.top->returnAddress;计算结果
        pop(L, a);//出栈，
        a.n++; //回溯
    }
    return L.base->returnAddress;//最后得到结果
}
int main()
```

```
{
    int n;
    Data a;
    cout << "请输入阶乘数 n：" << endl;
    cin >> n;
    a.n = n;
    Initstack(L,n);
    cout << model(L, a);
}
```

2.5 分析堆栈溢出问题

首先构造一个测试程序，测试调用递归函数时 n 等于多少时堆栈溢出。由于堆栈大小与编译器、系统、进程等等有关，而且可以修改，并非一个确定的值，在此仅仅测试本电脑系统运行递归函数时的堆栈溢出情况。我们可以设计一个计数器，用于计算递归函数的调用次数，每一次执行递归函数便输出一次，然后值++，在运行时输入一个比较大的值，等到程序崩溃时，记录最后一次输出的计数器的值便可以判断递归函数一共执行了多少次。

构造测试程序代码如下：

```
#include<iostream>
using namespace std;
int i = 1;
long f(int n) {
    if (n== 1) return 1;
    i++;
    cout << i << endl;
    return n * f(n-1); //调用自身
}
int main()
{
    int n;
    cin >> n;
    cout << f(n);
}
```

键盘输入：1000000

VS 中运行结果：n=3996 时堆栈溢出

图 3

在 DEV 中运行：n=43169 时堆栈溢出

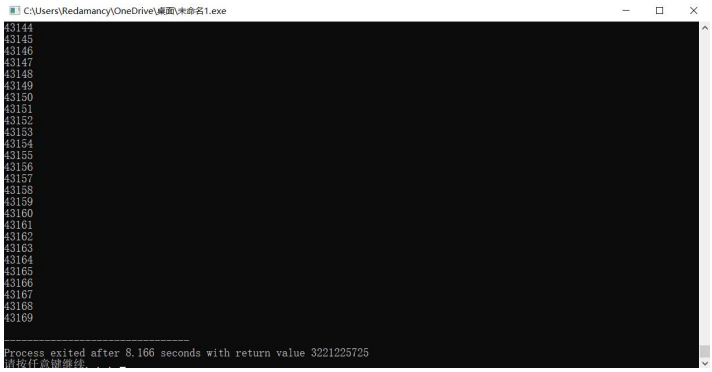

图 4

栈空间是由编译器自动分配释放的，一般用于保存指令地址，函数参数与局部变量值等。之所以会出现堆栈溢出的问题，主要是由于函数调用层次太深。函数递归调用时，系统要在栈中不断保存函数调用时的现场和产生的变量，如果递归调用太深，就会造成栈溢出，这时递归无法返回。再有，当函数调用层次过深时也可能导致栈无法容纳这些调用的返回地址而造成栈溢出。

如果用自己构造的栈模拟递归也有可能出现问题，但是在错误出现的范围在可接受范围之内，并且可以运行的范围比递归函数大的多。由于用户自己控制和定义的空间是储存在堆中，保存内容由用户决定，若程序结束时用户未回收，程序将自动回收。如果这里 n 如果过大，在分配空间时就会导致堆溢出而 exit（OVERFLOW）提示内存分配错误，否则便不会出现问题。这里就用栈模拟的情形进行测试：

输入 $n=10^{10}$ 此时退出代码为 3，表示堆溢出，分配空间超出最大空间

图 5

输入 $n=10^9$ 发现程序光标一直闪烁，表示耗费较长时间分配储存空间，，打开任务管理器发现 VS 占据的内存已经使本电脑运行内存占有达到 99%，表示空间分配成功，没有出现堆溢出，只需等待较长时间后可以得到 n 的阶乘值。不过此数据过大，已经远远超出 long long int 的储存范围，在本题中没有进行高精度模拟的操作，所以最后答案为 0。但是能够计算出结果仍然表明只要有较大的运行内存是可以执行成功的。通过此分析可以发现，用栈模拟递归过程可以允许的数量级远远大于执行递归函数的次数，是执行相对数量级较大的工程和计算时更优的选择。

图 6

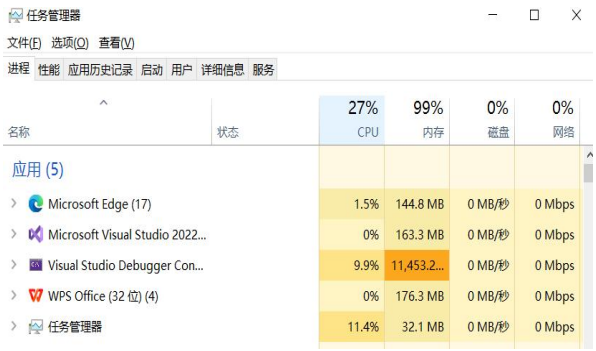

图 7

3. 实验总结

3.1 程序分析

本程序主要是通过自己构造栈来模拟递归函数的调用过程，首先在理解递归函数调用原理图的情况下，通过在栈的节点中加入两个数据域 n 和 returnaddress 来实现和递归函数相同的先递归再回溯过程。模拟递归操作时，将每一个 n 压入栈表示调用函数，再 n--继续调用下一层函数，直到达到递归函数的终止递归条件即 n=1。模拟回溯操作时，先建立一个临时指针指向 L.top 的前驱节点，然后进行计算，即前驱节点的返回值等于内层函数的返回值乘以当层函数的 n 的值，不断循环直到栈只剩下最后的结果，即为所求。其中主要的操作便是 push 和 pop 操作，都比较容易实现，关键是需要理解计算机内部递归函数的调用过程。

3.1.1 健壮性分析

本程序的健壮性主要体现在堆空间溢出时 exit（OVERFLOW）提示空间分配失败，不会导致堆溢出导致程序崩溃；还体现在栈空间满时，通过 realloc 来重新分配空间，避免数组越界和指针越界。

3.1.2 效率分析

本程序的主要操作 pop 和 push 的时间复杂度均为 O(1)，模拟函数递归调用时由于先要递归调用，然后再原路回溯，时间复杂度为 O(2n)。空间效率方面，根据键盘输入的 n 值先分配一个空间为 n 的栈，保证每次 push 和 pop 操作时不会越界，空间效率较高。

3.2 总结和收获

本次实验是对函数递归过程这一问题的细致研究。利用数据结构栈的知识，我们比较轻松的用栈模拟出了阶乘计算递归函数的执行过程，对二者的理解也更加深刻。同时通过测试栈溢出和堆溢出，我自行了解了计算机中堆和栈空间等的概念，加深了对编译器编译过程和内部函数调用的理解，对栈操作和使用也更加熟练。同时，通过比较发现了用

栈来模拟函数递归操作的实用性更强，可以允许的数据范围远大于递归函数的数据范围，对于大的数据和计算可以通过模拟的方法进行编程。推而广之，之前通过线性表也进行过高精度加法和乘法的模拟，可能计算机的硬件不支持我们计算一定数量级的数据，但是模拟的算法是人们运用有限的空间和时间转变思维而达到目标的一种算法，这也是一种重要的思想，它启示我们“山穷水复疑无路，柳暗花明又一村”，即使在程序的实现遇到了硬性的阻碍，可以通过智慧的思想方法去转变思维进行实现，这就是算法的目标，不仅是要将原本难以实现的问题进行实现，更要将已经能够实现的事情进行不断优化，这是一种精益求精，勇攀高峰的科研精神。