

《数据结构》上机报告

2020 年 10 月 24 日

姓名： 王上游 学号： 1850767 班级： 19 计科 2 班 得分：

实验 题目	运用栈模拟阶乘函数的调用过程
实验 目的	实验目的： 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。
问题 描述	<p>实验内容： 以 n 的阶乘为例，递归是一种函数调用自身的方法，代码可以如此实现：</p> <pre>public long f(int n){ if(n==1) return 1; //停止调用 return n*f(n-1); //调用自身 }</pre> <p>当调用一个函数时，编译器会将参数和返回地址入栈；当函数返回时，这些值出栈。</p> <p>递归通常有两个过程： (1) 递归过程：不断递归入栈 push，直到停止调用 $n=1$ (2) 回溯过程：不断回溯出栈 pop，计算 $n*f(n-1)$，直到栈空，结束计算。</p> <p>可以把上述过程分为以下几个状态：</p> <pre>graph LR; 1[初始化 1] --> 2{n=1? 2}; 2 -- Y --> 4{栈空? 4}; 2 -- N, 开始递归 --> 3[push(参数, 返回地址) 3]; 3 --> 2; 4 -- Y --> 6[end 6]; 4 -- N, 开始回溯 --> 5[pop, 开始计算 5]; 5 --> 4;</pre> <p>参考这个状态机，用栈模拟 n 的阶乘的递归调用过程。</p> <p>参考信息：</p>

	<p>要存储的数据：</p> <pre>typedef struct{ int n; //函数的输入参数 int returnAddress; //函数的返回地址（是否需要，留给大家思考） //构造器及 getter、setter ... }Data; Stack<Data> myStack = new Stack<>();</pre>	
基本要求	<p>实验要求：</p> <p>(1) 程序要添加适当的注释，程序的书写要采用 缩进格式。</p> <p>(2) 程序要具有一定的 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。</p> <p>(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。</p> <p>(4) 根据实验报告模板详细书写实验报告, 在实验报告中给出主要算法的复杂度分析。</p> <p>(5) 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。</p>	
选做要求	无	
	已完成选做内容（序号）	无
数据结构设计	<p>以下是测试程序数据结构，Data 类是栈的元素类型，栈使用 STL 的 stack 和自己实现的 SqStack 两种。</p> <pre>class Data { private: long long n; //函数的输入参数 int returnAddress; //函数的返回地址（是否需要，留给大家思考） //构造器及getter、setter public: friend ostream& operator<<(ostream& out, const Data& data); //重载<<输出 friend void fac1(long long n); //栈模拟求阶乘递归过程 friend void fac2(); //栈模拟求阶乘回溯过程 }; template <class SElemType> class SqStack { protected: SElemType* base; //存放动态申请空间的首地址</pre>	

```

SElemType* top;    //栈顶指针
int stacksize; //当前分配的元素个数
public:
/* P.46-47的抽象数据类型定义转换为实际的C++语言 */
SqStack();//构造函数，替代InitStack
~SqStack(); //析构函数，替代DestroyStack
Status ClearStack();//清空栈
Status StackEmpty();//判断栈是否为空
int StackLength();//栈长度
Status GetTop(SElemType& e)const;//取栈顶元素
Status Push(SElemType e);//e入栈
Status Pop(SElemType& e);//出栈，出栈元素赋值给e
Status StackTraverse(Status(*visit)(SElemType e));//遍历栈
};

```

基类型 Data 及 SqStack 栈的类成员/友元函数

class Data

long long n; //函数的输入参数

int returnAddress; //函数的返回地址 (是否需要, 留给大家思考)

friend ostream& operator<<(ostream& out, const Data& data);//重载<<输出

friend void fac1(long long n);//栈模拟求阶乘递归过程

friend void fac2();//栈模拟求阶乘回溯过程

class SqStack<Data>

SElemType* base; //存放动态申请空间的首地址

SElemType* top; //栈顶指针

int stacksize; //当前分配的元素个数

SqStack(); //构造函数, 替代 InitStack

~SqStack(); //析构函数, 替代 DestroyStack

Status ClearStack();//清空栈

Status StackEmpty();//判断栈是否为空

int StackLength();//栈长度

Status GetTop(SElemType& e)const;//取栈顶元素

Status Push(SElemType e);//e 入栈

Status Pop(SElemType& e);//出栈, 出栈元素赋值给 e

Status StackTraverse(Status(*visit)(SElemType e));//遍历栈

(1) SqStack的实现函数

/* 重载<<输出Data对象 */

/* 复杂度O(1) */

```
ostream& operator<<(ostream& out, const Data& data)
{
    out << "n = " << data.n << endl;
    out << "returnAddress = " << data.returnAddress << endl;
    return out;
}
```

/* 构造函数（初始化栈），堆分配 */

/* 复杂度O(1) */

```
template <class SElemType>
SqStack<SElemType>::SqStack()
{
    base = new SElemType[STACK_INIT_SIZE];
    if (base == NULL)
        exit(LOVERFLOW);
    top = base; //栈顶指针指向栈底，表示栈空
    stacksize = STACK_INIT_SIZE;
}
```

功能
(函数)
说明

/* 析构函数（删除栈） */

/* 复杂度O(1) */

```
template <class SElemType>
SqStack<SElemType>::~~SqStack()
{
    /* 若未执行 InitStack，直接执行本函数，则可能出错，因为指针初始值未定 */
    if (base)
        delete base; //要考虑空栈删除的情况，因此要判断
    top = NULL;
    stacksize = 0;
}
```

/* 清空栈（已初始化，不释放空间，只清除内容） */

/* 复杂度O(1) */

```
template <class SElemType>
Status SqStack<SElemType>::ClearStack()
```

```

{
    /* 如果栈曾经扩展过，恢复初始的大小 */
    if (stacksize > STACK_INIT_SIZE) {
        /* 释放原空间并申请 */
        delete base;
        base = new SElemType[STACK_INIT_SIZE];
        if (base == NULL)
            exit(LOVERFLOW);
        stacksize = STACK_INIT_SIZE;
    }

    top = base; //栈顶指针指向栈顶，表示栈空
    return OK;
}

/* 判断是否为空栈 */
/* 复杂度O(1) */
template <class SElemType>
Status SqStack<SElemType>::StackEmpty()
{
    if (top == base)
        return TRUE;
    else
        return FALSE;
}

/* 求栈的长度 */
/* 复杂度O(1) */
template <class SElemType>
int SqStack<SElemType>::StackLength()
{
    return top - base; //指针相减，值为相差的元素个数
}

/* 取栈顶元素 */
/* 复杂度O(1) */
template <class SElemType>
Status SqStack<SElemType>::GetTop(SElemType& e) const
{
    if (top == base)
        return ERROR;
}

```

```

    e = *(top - 1); //下标从0开始, top是实际栈顶+1
    return OK;
}

/* 元素入栈 */
/* 复杂度O(n) (有memcpy)*/
template <class SElemType>
Status SqStack<SElemType>::Push(SElemType e)
{
    /* 如果栈已满, 则扩充空间 */
    if (top - base >= stacksize) {
        SElemType* newbase;
        newbase = new SElemType[stacksize + STACKINCREMENT];
        if (!newbase)
            return LOVERFLOW;

        /* 原来的listsize个ElemType空间进行复制 */
        memcpy(newbase, base, stacksize * sizeof(SElemType));

        delete base;
        base = newbase;
        top = base + stacksize; //base可能与原来不同, top也要移动
        stacksize += STACKINCREMENT;
    }

    *top++ = e; //先*top, 再top++
    return OK;
}

/* 元素出栈 */
/* 复杂度O(n) (有memcpy)*/
template <class SElemType>
Status SqStack<SElemType>::Pop(SElemType& e)
{
    int length;
    if (top == base)
        return ERROR;
    e = *--top;

    /* 如果栈缩小, 则缩小动态申请空间的大小 */
    length = top - base;

```

```

    if (stacksize > STACK_INIT_SIZE && stacksize - length >=
STACKINCREMENT) {
        SElemType* newbase;
        newbase = new SElemType[stacksize - STACKINCREMENT];
        if (newbase == NULL)
            return LOVERFLOW;

        /* 原来的listsize个ElemType空间进行复制 */
        memcpy(newbase, base, (stacksize - STACKINCREMENT) *
sizeof(SElemType));

        delete base;
        base = newbase;
        top = base + length; //若S->base变化, 则修正S->top的值
        stacksize -= STACKINCREMENT;
    }

    return OK;
}

/* 遍历栈 */
/* 复杂度O(n) */
template <class SElemType>
Status SqStack<SElemType>::StackTraverse(Status(*visit)(SElemType e))
{
    SElemType* t = top - 1;

    while (t >= base && (*visit)(*t) == TRUE)
        t--;

    if (t < top)
        return ERROR;

    cout << endl; //最后打印一个换行, 只是为了好看, 与算法无关
    return OK;
}

template <class SElemType>
/* 复杂度O(1) */
Status myvisit(SElemType e)
{
    cout << e << ' ';
}

```

```

        return OK;
    }

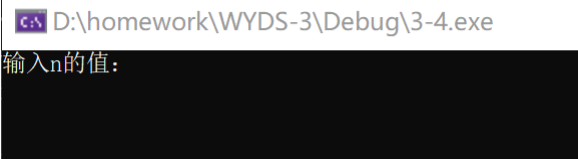
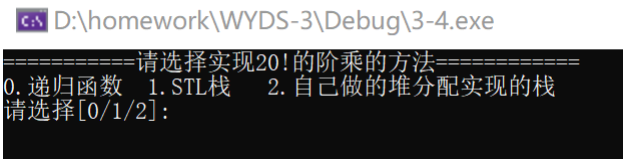
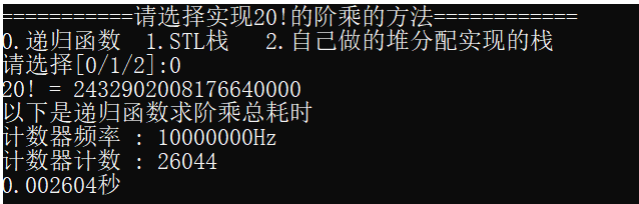
    SqStack<Data> s2;
    stack<Data> s1;

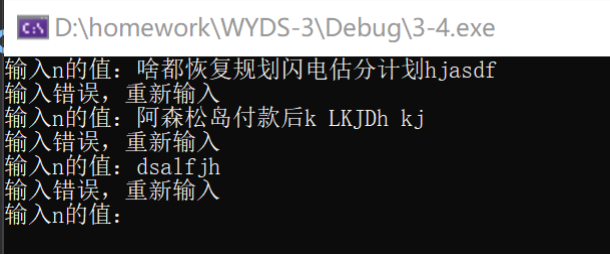
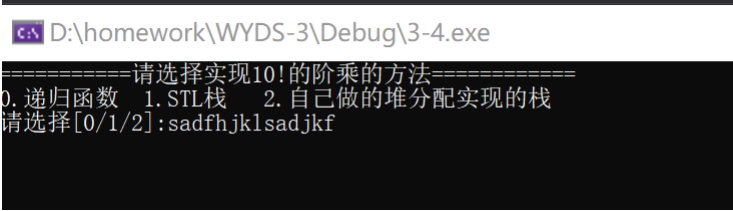

    long long ans = 1ll;
    int com = -1;

    /* 复杂度O(1) */
    void fac1(long long n)//栈模拟的递归过程
    {
        Data data;
        data.n = n;
        data.returnAddress = int(fac1);
        if(com == 2)
            s2.Push(data);
        if(com == 1)
            s1.push(data);
    }

    /* 复杂度O(1) */
    void fac2()//栈模拟的回溯过程
    {
        if (com != 2 && com != 1)return;
        Data data;
        if (com == 2)
            s2.Pop(data);
        if (com == 1)
        {
            data = s1.top();
            s1.pop();
        }
        ans *= data.n;
        cout << data;
        cout << "当前计算到" << data.n << "! = " << ans << endl;
    }

```


	<h2>(2) 递归函数求阶乘</h2> <p>时间复杂度$O(n)$,空间复杂度$O(kn)$(k:保存现场和恢复现场所需空间)</p> <pre>long long fac(long long n)//递归函数 { if (n == 0 n == 1) return 1ll; return fac(n - 1) * n; }</pre>
界面设计和使用说明	<p>(1) 进入程序，首先输入所需求的阶乘 n 的值。</p>  <p>(2) 以 n 输入 20 为例，然后选择运行模式 输入[0/1/2]分别以递归函数/STL stack 栈容器模拟/自己实现的 SqStack 栈模拟</p>  <p>(3) 程序加入了计时功能，选择递归函数，直接输出递归最终结果和所用时间</p>  <p>选择 2 种栈模拟，程序会输出每一步进栈出栈元素，当前计算到阶数，最后输出递归过程、回溯过程、总用时三个消耗时间。</p>

	<pre>returnAddress = 4069043 当前计算到15! = 1307674368000 n = 16 returnAddress = 4069043 当前计算到16! = 20922789888000 n = 17 returnAddress = 4069043 当前计算到17! = 355687428096000 n = 18 returnAddress = 4069043 当前计算到18! = 6402373705728000 n = 19 returnAddress = 4069043 当前计算到19! = 121645100408832000 n = 20 returnAddress = 4069043 当前计算到20! = 2432902008176640000 以下是栈模拟递归求阶乘，递归过程所用时间 计数器频率：10000000Hz 计数器计数：460 0.000046秒 以下是栈模拟递归求阶乘，回溯过程所用时间 计数器频率：10000000Hz 计数器计数：253531 0.025353秒 以下是栈模拟求阶乘总耗时 计数器频率：10000000Hz 计数器计数：253991 0.025399秒</pre>	
调试分析	<p>(1) 健壮性测试</p> <div></div> <div></div> <p>(2) 基于 n 的测试，比较三种实现方式的消耗时间 N = 5</p> <div></div>	

```
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：165
0.000017秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：24731
0.002473秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：24896
0.002490秒
```

```
当前计算到5! = 120
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：7
0.000001秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：22664
0.002266秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：22671
0.002267秒
```

N = 10

```
=====请选择实现10!的阶乘的方法=====
0. 递归函数 1. STL栈 2. 自己做的堆分配实现的栈
请选择[0/1/2]:0
10! = 3628800
以下是递归函数求阶乘总耗时
计数器频率：10000000Hz
计数器计数：6903
0.000690秒
```

```
当前计算到10! = 3628800
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：306
0.000031秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：55245
0.005524秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：55551
0.005555秒
```

```
当前计算到10! = 3628800
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：14
0.000001秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：57837
0.005784秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：57851
0.005785秒
```

N = 50

```
=====请选择实现50!的阶乘的方法=====
0. 递归函数 1. STL栈 2. 自己做的堆分配实现的栈
请选择[0/1/2]:0
50! = -3258495067890909184
以下是递归函数求阶乘总耗时
计数器频率：10000000Hz
计数器计数：8889
0.000889秒
```

```
当前计算到50! = -3258495067890909184
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：955
0.000096秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：2234773
0.223477秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：2235728
0.223573秒
```

```
当前计算到50! = -3258495067890909184
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：34
0.000003秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：2324388
0.232439秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：2324422
0.232442秒
```

N = 100

```
=====请选择实现100!的阶乘的方法=====
0. 递归函数 1. STL栈 2. 自己做的堆分配实现的栈
请选择[0/1/2]:0
100! = 0
以下是递归函数求阶乘总耗时
计数器频率：10000000Hz
计数器计数：88742
0.008874秒
```

```
当前计算到100! = 0
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：69
0.000007秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：2740628
0.274063秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：2740697
0.274070秒
```

```
当前计算到50! = -3258495067890909184
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：34
0.000003秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：2324388
0.232439秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：2324422
0.232442秒
```

N = 1000

```
=====请选择实现1000!的阶乘的方法=====
0. 递归函数 1. STL栈 2. 自己做的堆分配实现的栈
请选择[0/1/2]:0
1000! = 0
以下是递归函数求阶乘总耗时
计数器频率：10000000Hz
计数器计数：4196
0.000420秒
```

```
当前计算到1000! = 0
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：14710
0.001471秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：12725155
1.272515秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：12739865
1.273986秒
```

```
当前计算到1000! = 0
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：517
0.00052秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：11158871
1.115887秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：11159388
1.115939秒
```

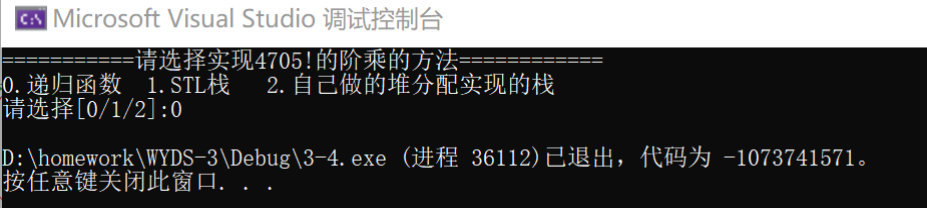
N = 10000

```
=====请选择实现10000!的阶乘的方法=====
0. 递归函数  1. STL栈  2. 自己做的堆分配实现的栈
请选择[0/1/2]:0
```

D:\homework\WYDS-3\Debug\3-4.exe (进程 56644) 已退出，代码为 -1073741571。

```
当前计算到10000! = 0
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：143431
0.014343秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：108580458
10.858046秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：108723889
10.872389秒
```

```
当前计算到10000! = 0
以下是栈模拟递归求阶乘，递归过程所用时间
计数器频率：10000000Hz
计数器计数：5990
0.000599秒
以下是栈模拟递归求阶乘，回溯过程所用时间
计数器频率：10000000Hz
计数器计数：107294959
10.729496秒
以下是栈模拟求阶乘总耗时
计数器频率：10000000Hz
计数器计数：107300949
10.730095秒
```

心得体会	<p>通过程序中添加的计时功能以及对 n 大小的测试，我们可以得出以下结论：</p> <ol style="list-style-type: none"> 1. 栈模拟递归求阶乘，解决了递归函数递归层数太深导致栈溢出的问题。这是因为系统栈 STL 可以使用堆空间，我自己做的栈从开始就是堆分配内存，而递归函数保存恢复现场使用的是栈空间，栈空间大小远小于堆空间，且递归函数保存现场、恢复现场所需辅助空间更多。递归函数实现，大约在 n 达到 4700 左右发生栈溢出。而栈方式实现则基本不会溢出。  <ol style="list-style-type: none"> 2. 计时结果显示，递归函数速度大大快于栈方式，且栈方式中，绝大多数时间用于回溯过程，用于递归过程的时间远远小于用于回溯过程。 3. 以上 1、2 可知，递归函数实现，速度快但可用空间少；栈方式实现，速度慢但可用空间大。 4. 返回地址 returnAddress 没必要，同一函数地址不因多次执行改变。可以看到打印的地址每次都一样。
代码实现	<pre> #include <iostream> #include <cstdlib> //malloc/realloc函数 #include <cstring> #include <string> #include <stack> #include <iomanip> #include <windows.h> //取系统时间 using namespace std; /* P.10 的预定义常量和类型 */ #define TRUE 1 #define FALSE 0 #define OK 1 #define ERROR 0 #define INFEASIBLE -1 #define LOVERFLOW -2 //避免与<math.h>中的定义冲突 typedef int Status; /* P.46 结构体定义 */ #define STACK_INIT_SIZE 10000 //初始大小定义为10000（可按需修改） #define STACKINCREMENT 1000 //若空间不够，每次增长10（可按需修改） class Data { private: long long n; //函数的输入参数 </pre>

```

    int returnAddress; //函数的返回地址（是否需要，留给大家思考）
    //构造器及getter、setter
public:
    friend ostream& operator<<(ostream& out, const Data& data); //重载<<输出
    friend void fac1(long long n); //栈模拟求阶乘递归过程
    friend void fac2(); //栈模拟求阶乘回溯过程
};

ostream& operator<<(ostream& out, const Data& data)
{
    out << "n = " << data.n << endl;
    out << "returnAddress = " << data.returnAddress << endl;
    return out;
}

template <class SElemType>
class SqStack {
protected:
    SElemType* base;    //存放动态申请空间的首地址
    SElemType* top;     //栈顶指针
    int stacksize; //当前分配的元素个数
public:
    /* P.46-47的抽象数据类型定义转换为实际的C++语言 */
    SqStack(); //构造函数，替代InitStack
    ~SqStack(); //析构函数，替代DestroyStack
    Status ClearStack(); //清空栈
    Status StackEmpty(); //判断栈是否为空
    int StackLength(); //栈长度
    Status GetTop(SElemType& e) const; //取栈顶元素
    Status Push(SElemType e); //e入栈
    Status Pop(SElemType& e); //出栈，出栈元素赋值给e
    Status StackTraverse(Status(*visit)(SElemType e)); //遍历栈
};

/* 构造函数（初始化栈） */
template <class SElemType>
SqStack<SElemType>::SqStack()
{
    base = new SElemType[STACK_INIT_SIZE];
    if (base == NULL)
        exit(LOVERFLOW);
    top = base; //栈顶指针指向栈底，表示栈空
    stacksize = STACK_INIT_SIZE;
}

```

```

}

/* 析构函数（删除栈） */
template <class SElemType>
SqStack<SElemType>::~~SqStack()
{
    /* 若未执行 InitStack，直接执行本函数，则可能出错，因为指针初始值未定 */
    if (base)
        delete base; //要考虑空栈删除的情况，因此要判断
    top = NULL;
    stacksize = 0;
}

/* 清空栈（已初始化，不释放空间，只清除内容） */
template <class SElemType>
Status SqStack<SElemType>::ClearStack()
{
    /* 如果栈曾经扩展过，恢复初始的大小 */
    if (stacksize > STACK_INIT_SIZE) {
        /* 释放原空间并申请 */
        delete base;
        base = new SElemType[STACK_INIT_SIZE];
        if (base == NULL)
            exit(LOVERFLOW);
        stacksize = STACK_INIT_SIZE;
    }

    top = base; //栈顶指针指向栈顶，表示栈空
    return OK;
}

/* 判断是否为空栈 */
template <class SElemType>
Status SqStack<SElemType>::StackEmpty()
{
    if (top == base)
        return TRUE;
    else
        return FALSE;
}

/* 求栈的长度 */
template <class SElemType>

```



```

int SqStack<SElemType>::StackLength()
{
    return top - base; //指针相减，值为相差的元素个数
}

/* 取栈顶元素 */
template <class SElemType>
Status SqStack<SElemType>::GetTop(SElemType& e) const
{
    if (top == base)
        return ERROR;

    e = *(top - 1); //下标从0开始，top是实际栈顶+1
    return OK;
}

/* 元素入栈 */
template <class SElemType>
Status SqStack<SElemType>::Push(SElemType e)
{
    /* 如果栈已满，则扩充空间 */
    if (top - base >= stacksize) {
        SElemType* newbase;
        newbase = new SElemType[stacksize + STACKINCREMENT];
        if (!newbase)
            return LOVERFLOW;

        /* 原来的listsize个ElemType空间进行复制 */
        memcpy(newbase, base, stacksize * sizeof(SElemType));

        delete base;
        base = newbase;
        top = base + stacksize; //base可能与原来不同，top也要移动
        stacksize += STACKINCREMENT;
    }

    *top++ = e; //先*top，再top++
    return OK;
}

/* 元素出栈 */
template <class SElemType>
Status SqStack<SElemType>::Pop(SElemType& e)
{

```

```

int length;
if (top == base)
    return ERROR;
e = *--top;

/* 如果栈缩小，则缩小动态申请空间的大小 */
length = top - base;
if (stacksize > STACK_INIT_SIZE && stacksize - length >=
STACKINCREMENT) {
    SElemType* newbase;
    newbase = new SElemType[stacksize - STACKINCREMENT];
    if (newbase == NULL)
        return LOVERFLOW;

    /* 原来的listsize个ElemType空间进行复制 */
    memcpy(newbase, base, (stacksize - STACKINCREMENT) *
sizeof(SElemType));

    delete base;
    base = newbase;
    top = base + length; //若S->base变化，则修正S->top的值
    stacksize -= STACKINCREMENT;
}

return OK;
}

/* 遍历栈 */
template <class SElemType>
Status SqStack<SElemType>::StackTraverse(Status(*visit)(SElemType e))
{
    SElemType* t = top - 1;

    while (t >= base && (*visit)(*t) == TRUE)
        t--;

    if (t < top)
        return ERROR;

    cout << endl; //最后打印一个换行，只是为了好看，与算法无关
    return OK;
}

template <class SElemType>

```

```

Status myvisit(SElemType e)
{
    cout << e << ' ';
    return OK;
}

SqStack<Data> s2;
stack<Data> s1;

long long ans = 1ll;
int com = -1;

void fac1(long long n)//栈模拟的递归过程
{
    Data data;
    data.n = n;
    data.returnAddress = int(fac1);
    if(com == 2)
        s2.Push(data);
    if(com == 1)
        s1.push(data);
}

void fac2()//栈模拟的回溯过程
{
    if (com != 2 && com != 1)return;
    Data data;
    if (com == 2)
        s2.Pop(data);
    if (com == 1)
    {
        data = s1.top();
        s1.pop();
    }
    ans *= data.n;
    cout << data;
    cout << "当前计算到" << data.n << "! = " << ans << endl;
}

long long fac(long long n)//递归函数
{

```

```

    if (n == 0 || n == 1)
        return 1ll;
    return fac(n - 1) * n;
}

int main()
{
    int n;
    long long ans = 1ll;

    while (1)
    {
        cout << "输入n的值: ";
        if (!(cin >> n))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误, 重新输入" << endl;
        }
        else if (n >= 0)
            break;
        else
            cout << "输入错误, 重新输入" << endl;
    }

    while (1)
    {
        system("cls");
        cout << "=====请选择实现" << n << "!的阶乘的方法"
        << endl;
        cout << "0.递归函数  1.STL栈  2.自己做的堆分配实现的栈" << endl;
        cout << "请选择[0/1/2]:";
        if (!(cin >> com))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误, 重新输入" << endl;
        }
        else if (com >= 0 && com <= 2)
            break;
        else
            cout << "输入错误, 重新输入" << endl;
    }
}

```

```

LARGE_INTEGER tick, begin, end, middle;

QueryPerformanceFrequency(&tick);    //获得计数器频率
QueryPerformanceCounter(&begin);      //获得初始硬件计数器计数

if (com)
    for (long long i = n; i >= 1; i--)
        fac1(i);
if (com)
{
    QueryPerformanceCounter(&middle);    //获得终止硬件计数器计数
}
if (com == 2)
    while (!s2.StackEmpty())
        fac2();
if (com == 1)
    while (!s1.empty())
        fac2();

if (!com)
    cout << n << "! = " << fac(n) << endl;

QueryPerformanceCounter(&end);          //获得终止硬件计数器计数
if (com)
{
    cout << "以下是栈模拟递归求阶乘，递归过程所用时间" << endl;
    cout << "计数器频率：" << tick.QuadPart << "Hz" << endl;
    cout << "计数器计数：" << middle.QuadPart - begin.QuadPart <<
endl;
    cout << setiosflags(ios::fixed) << setprecision(6) <<
double(middle.QuadPart - begin.QuadPart) / tick.QuadPart << "秒" << endl;

    cout << "以下是栈模拟递归求阶乘，回溯过程所用时间" << endl;
    cout << "计数器频率：" << tick.QuadPart << "Hz" << endl;
    cout << "计数器计数：" << end.QuadPart - middle.QuadPart << endl;
    cout << setiosflags(ios::fixed) << setprecision(6) <<
double(end.QuadPart - middle.QuadPart) / tick.QuadPart << "秒" << endl;
}

cout << "以下是" << (com ? "栈模拟" : "递归函数") << "求阶乘总耗时" <<
endl;
cout << "计数器频率：" << tick.QuadPart << "Hz" << endl;

```

```
cout << "计数器计数：" << end.QuadPart - begin.QuadPart << endl;
cout << setiosflags(ios::fixed) << setprecision(6) <<
double(end.QuadPart - begin.QuadPart) / tick.QuadPart << "秒" << endl;

return 0;
}
```