

《数据结构》上机报告

2022 年 9 月 29 日

姓名：郑博远 学号：2154312 班级：计科 1 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	<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。	
问题描述	<p>递归是一种函数调用自身的方法，可以如此实现 n 的阶乘：</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：</p> <ol style="list-style-type: none">（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>用栈模拟 n 的阶乘的递归调用过程。</p>	
基本要求	<ol style="list-style-type: none">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；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. 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；5. 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。	
选做要求		
	已完成选做内容（序号）	

<p>数据结构设计</p>	<p>本题探讨通过运用栈的消解方式来模拟递归函数求算 n 的阶乘，因此主要涉及到的数据结构即为栈。我选择了顺序表、链表两种不同方式来实现栈，并用模板类将其封装。值得注意的是，本题需要比对递归函数出现堆栈溢出时，用栈消解递归是否能规避这样的错误。但由于 n 的阶乘随 n 上升而上升的速率很快，当递归函数出现堆栈溢出时，n 的阶乘大小早已超过 <code>long long</code> 数据类型，无法进行有效的数据计算；因此需要进行大整数的高精度计算。考虑到使用的方便性，将其封装其类，并通过运算符重载的方式使得操作上与普通数据类型无异。具体数据结构如下：</p> <pre>/* 顺序表方式实现的栈 */ template <class SElemType> //元素的数据类型 struct SqStack { private: SElemType* base; //栈底 SElemType* top; //栈顶 int stacksize; //栈的大小 public: //此处省略若干成员函数 由下一部分逐一介绍 } /* 链表结点 */ template <class LElemType> class LNode { public: LElemType data; //存放的数据 LNode* next; //直接后继的指针 }; /* 链表方式实现的栈 */ template <class LElemType> struct LinkStack { private: LNode<LElemType>* head; //链表的头指针 public: //此处省略若干成员函数 由下一部分逐一介绍 }</pre>
---------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

	<pre>/* 栈的元素Data - 模拟函数存放的信息 */ struct Data { int n; //传入的参数, 表示当前的数n int returnAddress; //函数的返回地址 此处实际用于表示该函数进行到的模块 0/1 } #define MAXN 1024//最长存放的大整数位数(十进制) /* 表示大整数的类 */ class LONG_LONG_INT { private: char number[MAXN]; //存放每一位数字的char数组 int length; //该大整数数字的位数 public: //此处省略若干成员函数 由下一部分逐一介绍 }</pre>
功能(函数) 说明	<p>1. 顺序表栈类的成员函数</p> <pre>/** * @brief 构造空栈 * @tparam LElemType 栈的元素类型 */ template <class LElemType> SqStack<LElemType>::SqStack() /** * @brief 栈的销毁 * @tparam LElemType 栈的元素类型 */ template <class LElemType> SqStack<LElemType>::~~SqStack() /** * @brief 栈的清空 * @tparam LElemType 栈的元素类型 */ template <class LElemType> Status SqStack<LElemType>::ClearStack() /**</pre>

```

* @brief 取栈顶元素
* @tparam LElemType 栈的元素类型
* @param e 取的栈顶元素值
*/
template <class LElemType>
Status SqStack<LElemType>::Top(LElemType& e)

/**
* @brief 弹出栈顶元素
* @tparam LElemType 栈的元素类型
* @param e 弹出的栈顶元素
*/
template <class LElemType>
Status SqStack<LElemType>::Pop(LElemType& e)

/**
* @brief 新元素入栈
* @tparam LElemType 栈的元素类型
* @param e 入栈的元素
*/
template <class LElemType>
Status SqStack<LElemType>::Push(LElemType e)

/**
* @brief 栈是否为空
* @tparam LElemType 栈的元素类型
*/
template <class LElemType>
bool SqStack<LElemType> ::StackEmpty()

```

2. 链表栈类的成员函数

（与上述一致，此处省略）

3. 大整数类的成员函数、友元函数

```

/**
* @brief 大整数的无参构造（置0）
*/
LONG_LONG_INT::LONG_LONG_INT()

/**
* @brief 通过long long构造大整数
* @param n 大整数的值n
*/
LONG_LONG_INT::LONG_LONG_INT(long long n)

```

```

/**
 * @brief 重载运算符* 大整数乘法
 * @param that 参与乘法的另一个大整数
 * @return 乘积
 */
const LONG_LONG_INT LONG_LONG_INT::operator*(const
LONG_LONG_INT& that) const

/**
 * @brief 重载运算符+ 大整数加法
 * @param that 参与加法的另一个大整数
 * @return 运算的和
 */
const LONG_LONG_INT LONG_LONG_INT::operator+(const
LONG_LONG_INT& that) const

/**
 * @brief 重载>> 流输入大整数
 * @param in 输入流
 * @param that 输入的大整数对象
 */
istream& operator >> (istream& in, LONG_LONG_INT& that)

/**
 * @brief 重载<< 流输出大整数
 * @param out 输出流
 * @param that 输出的大整数对象
 */
ostream& operator << (ostream& out, const LONG_LONG_INT&
that)

```

4. 求阶乘的两种方式的函数（递归、栈模拟）

这两个函数是本题探究的关键部分，因此摘录完整函数代码，以展示栈消解方式实现递归的具体方法。

1) 传统递归方式求阶乘：

```

/**
 * @brief 递归方式求阶乘
 * @param n 求阶乘的数n
 * @return 阶乘结果
 */
LONG_LONG_INT Factorial_1(int n)
{
    if (n == 1)

```

```

        return 1;
    else
        return LONG_LONG_INT(n) * Factorial_1(n - 1);
}

```

2) 栈模拟递归函数方式求阶乘:

```

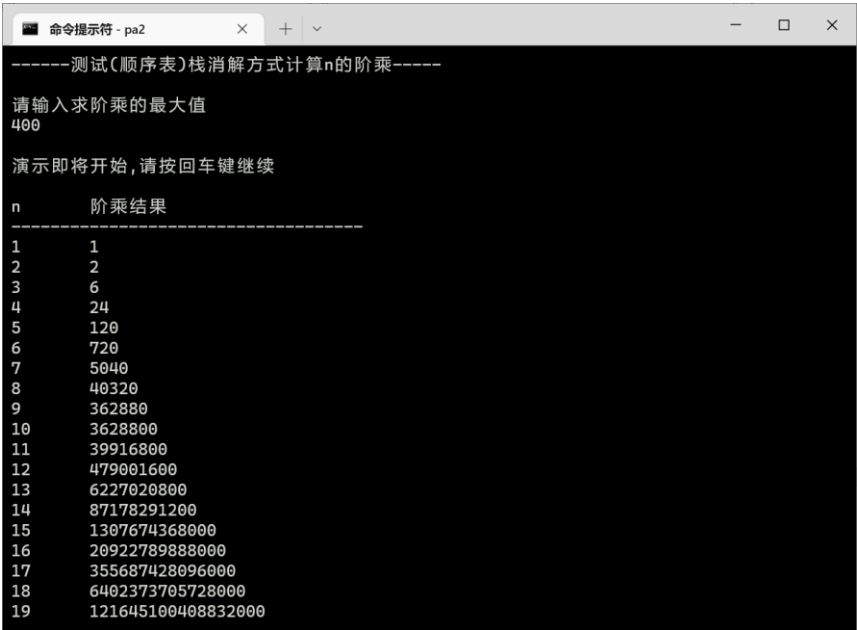
/**
 * @brief 栈消解方式求阶乘
 * @tparam stack 栈的类型
 * @param n 求阶乘的数n
 * @param function 存放函数信息的栈
 * @return 阶乘结果
 */
template <class stack>
LONG_LONG_INT Factorial_2(int n, stack& function)
{
    LONG_LONG_INT ret;
    //模拟通过寄存器传递的各个函数返回值

    function.Push({ n, 0 }); //模拟第一个函数被main调用
    while (!function.StackEmpty()) {
        Data now;
        function.Top(now);
        switch (now.returnAddress) {
            //递归调用的部分
            case 0:
                function.Pop(now);
                if (now.n > 1) {
                    function.Push({ now.n, 1 });
                    //保存现场
                    function.Push({ now.n - 1, 0 });
                    //调用下一层
                }
                else {
                    ret = 1; //返回
                }
                break;
            //回溯的部分
            case 1:
                function.Pop(now);
                //恢复现场
                ret = ret * LONG_LONG_INT(now.n);
                //计算返回值
                break;

```

	<pre> } } return ret; } 5. 模拟函数信息的结构体的构造函数 /* * @brief 无参构造函数 */ Data() /** * @brief 双参构造函数 * @param _n 用于构造的层数n * @param _ra 用于构造的返回地址returnAddress */ Data(int _n, int _ra)</pre>
界面设计和使用说明	<div></div> <p>程序运行开始，进入菜单项选择页面。选择数字 1，进入栈（顺序表实现）消解方式模拟递归演示；选择数字 2，进入栈（链表实现）消解方式模拟递归演示；选择数字 3，进入普通函数递归方式求递归演示；选择数字 0，程序退出运行。</p> <p>选择数字 1 后，程序通过栈（顺序表实现）消解的方式模拟递</p>

归函数调用来计算 n 的阶乘。用户输入期望得到的最大阶乘数后，程序开始运行。每一行输出分别为一个数字 n（从 1 开始每行增加 1）以及其对应的阶乘结果。演示结果如下图所示：



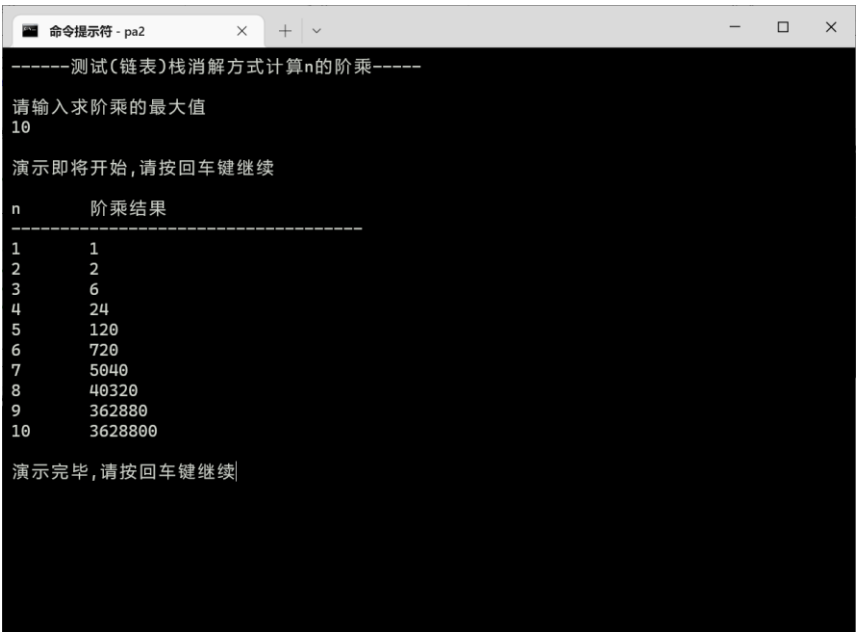
```
命令提示符 - pa2
-----测试(顺序表)栈消解方式计算n的阶乘-----

请输入求阶乘的最大值
400

演示即将开始,请按回车键继续

n      阶乘结果
-----
1      1
2      2
3      6
4      24
5      120
6      720
7      5040
8      40320
9      362880
10     3628800
11     39916800
12     479001600
13     6227020800
14     87178291200
15     1307674368000
16     20922789888000
17     355687428096000
18     6402373705728000
19     121645100408832000
```

选择数字 2 后，程序通过栈（链表实现）消解的方式模拟递归函数调用来计算 n 的阶乘。用户输入期望得到的最大阶乘数后，程序开始运行。每一行输出分别为一个数字 n（从 1 开始每行增加 1）以及其对应的阶乘结果。演示结果如下图所示：



```
命令提示符 - pa2
-----测试(链表)栈消解方式计算n的阶乘-----

请输入求阶乘的最大值
10

演示即将开始,请按回车键继续

n      阶乘结果
-----
1      1
2      2
3      6
4      24
5      120
6      720
7      5040
8      40320
9      362880
10     3628800

演示完毕,请按回车键继续|
```


	<p>选择数字 3 后，程序通过递归调用的方式模拟递归函数调用来计算 n 的阶乘。用户输入期望得到的最大阶乘数后，程序开始运行。每一行输出分别为一个数字 n（从 1 开始每行增加 1）以及其对应的阶乘结果。显示界面与前两个功能类似，不再另外贴图赘述。</p> <p>当大整数数组采用 1024 大小的 int 数组来记录整数时，当求阶乘到 105 时使用递归函数的方式计算阶乘由于递归的层数太多，从而产生了堆栈溢出，使得程序错误地退出。如下图：</p>
--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

当大整数数组采用 1024 大小的 int 数组来记录整数时，当求阶乘到 105 时使用递归函数的方式计算阶乘由于递归的层数太多，从而产生了堆栈溢出，使得程序错误地退出。如下图：

```
C:\Users\BoyanZheng>source\repos\数据结构\Debug>
```

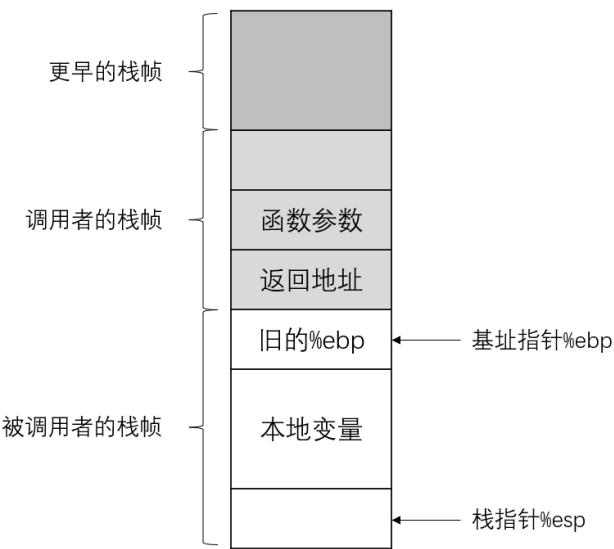
当大整数数组采用 1024 大小的 char 数组来记录，由于每一层函数压栈时所占用空间变小，更不容易产生堆栈溢出，因此采用递归调用方式时直到求 337 的阶乘时才出现溢出，程序错误退出。具体表现如下图：

[illegible]

	<p>而如果使用栈的方式来模拟递归函数调用,当阶乘乘到 105 时,也并没有产生堆栈溢出,程序继续正确运行。当阶乘持续扩大 (n 到 460, 数组大小为 1024), 由于阶乘答案位数过长会导致结果的溢出,可以考虑后续通过动态内存申请的方式来实现大整数类。测试时,列举了从 1 到 400 的阶乘情况,程序均能正确得出结果:</p> <div data-bbox="461 533 1329 1128"></div>
调试分析	<p>1. 起初在计算阶乘时,没有考虑到结果可能位数极大的状况,因而选择使用 long long 类型来存储阶乘的结果。但由于要测得递归调用函数时堆栈溢出的情况,并在此情况下比较栈模拟是否能解决该问题,实际测试使用的时候 n 值会比较大 (在本机 64 位 Windows10 系统 Microsoft Visual Studio 2022 x86 Debug 模式的环境下,1024 大小的 int 数组记录的大整数堆栈溢出时 n 的值为 105,1024 大小的 char 数组记录的大整数堆栈溢出时 n 的值为 337),导致阶乘数远超 long long 数据类型的最大值。因此在 n 略大时, long long 类型无法存储对应大小的数字,导致结果全部出错,不符合预期。为解决该问题,引入大整数的高精度乘法,具体使用方式与 HW0 中的相关题目以及 HW1 中的级数一题类似。即,使用大数组逐位按照从低位到高位从前往后的方式存储较大的整数,并为了代码简洁将其封装在类当中,通过运算符重载的方式使得在阶乘调</p>

	<p>用时代码也能简洁易懂、可读性强；</p> <p>2. 对函数调用，尤其是递归调用自身的压栈、出栈方式不够熟悉，导致起初不能很好的设计出存储调用函数信息的结构体。对于结构体的 returnAddress 变量，起初不能很好的使用。这导致函数递归之后回溯的过程中，程序无法判断是要继续调用下一级函数、还是到了逐层返回 ret 值的情况，使得结果与预期不符合。在实际的函数调用过程中，returnAddress 在被调用函数入栈前被压入栈中；本质上，函数的返回地址 returnAddress 是为了在被调用 callee 返回调用函数 caller 时，能够恢复到调用前执行的下一步语句。而求阶乘作为递归的一种简单调用，每个入栈的函数体都可以被分为递归调用前后两个部分。因此，虽然无法直接获取到 caller 下一步语句对应的地址，但是可以通过在保护现场压栈时记录下一次进入函数时执行的是哪一个部分，来实现与递归相同的效果。此处记录下一步执行的是哪一个部分的变量虽然实际记录方式与 returnAddress 不同，但其起到的效果大致类似。通过上述方式，便可以通过栈的方式来模拟递归函数的调用过程，得到正确的阶乘值且不会产生堆栈溢出。</p>
心得体会	<p>本次实验中，我用顺序表与链表两种方式实现栈，以 C++ 方式将操作其的函数都封装在类中。同时为了使封装之后的类能够适应不同的数据类型，我使用模板类，使其能够以诸如本题中“调用函数信息”的结构体之类的不同数据类型为其栈中的元素。</p> <p>想要理解并实现本题，首先要对函数的调用栈有所了解。查阅相关资料后，我大致熟悉了函数的调用方式。函数调用过程中，需要将不同的数据及地址压入或者弹出栈。栈帧是一种专门用于保存函数调用过程中的各种信息（例如参数、返回地址等）的栈。栈帧中，栈顶的地址最低，栈底的地址最高，以 x86-32bit 为例，基址指针 %ebp 指向栈底，栈指针 %esp 指向栈顶。</p> <p>调用函数时，首先会将函数的参数按照从右往左的顺序压入栈中，然后将函数的返回地址压入栈中。此后，将旧的 %ebp 压入栈，</p>

并将%ebp 指向此处，便可以开始新的函数执行。返回调用函数 caller 时，则执行相反的操作，返回到返回地址处。此外，函数的返回值通过寄存器在不同的函数体之间传递。



而用栈来模拟函数的调用，关键在于函数参数、返回值的传递，以及如何实现返回地址 returnAddress 的传递与使用。函数参数的传递比较简单，在调用者函数入栈时同时压入即可。返回地址的使用上，由于在本题中函数体是递归的一种简单应用——阶乘；因此，每个入栈的函数体（除 n 为 1 递归终止时）都可以被分为递归调用前后两个部分。从而，可以通过在入栈时记录下一次进入函数时执行的是哪一个部分，从而判断和区分是在初次进入的递归的阶段还是在回溯并返回的阶段。虽然实际记录方式与 returnAddress 直接记录地址不同，但其起到的效果大致类似。而对于返回值的传递，为模拟函数之间通过寄存器的传递，可以直接另开一个变量 ret 来实现不同函数体之间值的传递。

有了如上大体思路后，本题的基本框架便构建完成了。还需要注意例如用高精度处理大整数乘法等细节，便可以达到题目的要求。易得，通过递归方式计算阶乘的时间复杂度为 $O(n)$ 。

由于栈先进后出的结构特性，取栈顶元素 Top、在栈顶插入新

	<p>元素 Push、弹出栈顶元素 Pop 的时间复杂度均为 $O(1)$。在实现方式上，顺序表由于其性质比较简单，链表通过头插法在头部插入删除从而达到 $O(1)$ 的时间复杂度。</p> <p>在实现大整数高精度运算时，起初采用的是遍历整个数组大小 MAXN 进行乘、加操作，这会导致大量无意义的位置的运算，运行十分缓慢；因此在大整数类中增加了记录长度的 length 成员变量，在加法与乘法中进行相应优化，从而实现改进。改进之后的时间复杂度分析如下：若两个操作数的位数分别为 m、n，实现高精度乘法、加法的时间复杂度分别为 $O(mn)$ 以及 $O(\max(m, n))$。</p>
--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

附.完整代码

1.用顺序表实现栈的 hpp 文件 (sqstack.hpp)

```
#pragma once
#include <iostream>
using namespace std;

#define STACK_INIT_SIZE 100000
#define STACKINCREMENT 10

typedef int Status;
#define SOVERFLOW -2
#define ERROR -1
#define OK 0

/* 顺序表方式实现的栈 */
template <class LElemType>
struct SqStack {
private:
    LElemType* base;
    LElemType* top;
    int stacksize;
};
```

```

public:
    SqStack();           //构造空栈
    ~SqStack();          //销毁已有的栈
    Status ClearStack(); //把现有栈置空栈
    Status Top(LElemType& e); //取栈顶元素
    Status Pop(LElemType& e); //弹出栈顶元素
    Status Push(LElemType e); //新元素入栈
    bool StackEmpty();    //是否为空栈
};

/**
 * @brief 构造空栈
 * @tparam LElemType 栈的元素类型
 */
template <class LElemType>
SqStack<LElemType>::SqStack()
{
    base = new(nothrow) LElemType[STACK_INIT_SIZE];
    if (!base)
        exit(SOVERFLOW);
    top = base;
    stacksize = STACK_INIT_SIZE;
}

/**
 * @brief 栈的销毁
 * @tparam LElemType 栈的元素类型
 */
template <class LElemType>
SqStack<LElemType>::~~SqStack()
{
    if (base)
        delete base;
    stacksize = 0;
}

/**
 * @brief 栈的清空
 * @tparam LElemType 栈的元素类型
 */
template <class LElemType>
Status SqStack<LElemType>::ClearStack()
{

```

```

    //先销毁原有空间
    if (base)
        delete base;
    //重新申请
    base = new(nothrow) LElemType[STACK_INIT_SIZE];
    if (!base)
        exit(SOVERFLOW);
    top = base;
    stacksize = STACK_INIT_SIZE;
    return OK;
}

/**
 * @brief 取栈顶元素
 * @tparam LElemType 栈的元素类型
 * @param e 取的栈顶元素值
 */
template <class LElemType>
Status SqStack<LElemType>::Top(LElemType& e)
{
    if (top == base)
        return ERROR;
    e = *(top - 1);
    return OK;
}

/**
 * @brief 弹出栈顶元素
 * @tparam LElemType 栈的元素类型
 * @param e 弹出的栈顶元素
 */
template <class LElemType>
Status SqStack<LElemType>::Pop(LElemType& e)
{
    if (top == base)
        return ERROR;
    e = *(--top);
    return OK;
}

/**
 * @brief 新元素入栈
 * @tparam LElemType 栈的元素类型

```

```

    * @param e 入栈的元素
    */
template <class LElemType>
Status SqStack<LElemType>::Push(LElemType e)
{
    if (top - base >= stacksize) {
        LElemType* newbase = new(nothrow) LElemType[stacksize +
STACKINCREMENT];
        if (!newbase)
            exit(SOVERFLOW);
        memcpy(newbase, base, sizeof(LElemType) * stacksize);
        delete base;
        base = newbase;
        top = base + stacksize;
        stacksize += STACKINCREMENT;
    }
    *top = e;
    top++;
    return OK;
}

/**
 * @brief 栈是否为空
 * @tparam LElemType 栈的元素类型
 */
template <class LElemType>
bool SqStack<LElemType> ::StackEmpty()
{
    return base == top;
}

```

2.用链表实现栈的 hpp 文件 (linkstack.hpp)

```

#pragma once
#include <iostream>
using namespace std;

typedef int Status;
#define LOVERFLOW -2
#define ERROR -1

```



```

#define OK 0

/* 链表节点 */
template <class LElemType>
class LNode {
public:
    LElemType data;        //存放的数据
    LNode* next;           //直接后继的指针
};

/* 链表方式实现的栈 */
template <class LElemType>
struct LinkStack {
private:
    LNode<LElemType>* head;    //链表的头指针
public:
    LinkStack();              //构造空栈
    ~LinkStack();             //销毁已有的栈
    Status ClearStack();      //把现有栈置空栈
    Status Top(LElemType& e);  //取栈顶元素
    Status Pop(LElemType& e);  //弹出栈顶元素
    Status Push(LElemType e);  //新元素入栈
    bool StackEmpty();         //是否为空栈
};

/**
 * @brief 构造空栈
 * @tparam LElemType 栈的元素类型
 */
template <class LElemType>
LinkStack<LElemType>::LinkStack()
{
    head = new(nothrow) LNode<LElemType>;    //头结点的建立
    if (!head)
        exit(LOVERFLOW);
    head->next = NULL;
}

/**
 * @brief 栈的销毁
 * @tparam LElemType 栈的元素类型
 */
template <class LElemType>
LinkStack<LElemType>::~~LinkStack()

```

```

{
    LNode<LElemType>* p = head, * q;
    while (p) {
        q = p->next;
        delete p;
        p = q;
    }
}

/**
 * @brief 栈的清空
 * @tparam LElemType 栈的元素类型
 */
template <class LElemType>
Status LinkStack<LElemType>::ClearStack()
{
    LNode<LElemType>* p = head->next, q;
    while (p) {
        q = p->next;
        delete p;
        p = q;
    }
    head->next = NULL; //头结点指向空
}

/**
 * @brief 取栈顶元素
 * @tparam LElemType 栈的元素类型
 * @param e 取的栈顶元素值
 */
template <class LElemType>
Status LinkStack<LElemType>::Top(LElemType& e)
{
    if (!head->next)
        return ERROR;
    e = head->next->data;
    return OK;
}

/**
 * @brief 弹出栈顶元素
 * @tparam LElemType 栈的元素类型
 * @param e 弹出的栈顶元素
 */

```

```

template <class LElemType>
Status LinkStack<LElemType>::Pop(LElemType& e)
{
    if (!head->next)
        return ERROR;
    LNode<LElemType>* p;
    p = head->next;
    e = p->data;
    head->next = p->next;
    delete p;
    return OK;
}

/**
 * @brief 新元素入栈
 * @tparam LElemType 栈的元素类型
 * @param e 入栈的元素
 */
template <class LElemType>
Status LinkStack<LElemType>::Push(LElemType e)
{
    LNode<LElemType>* q;
    q = new(nothrow) LNode<LElemType>;
    if (!q)
        exit(LOVERFLOW);
    q->data = e;
    q->next = head->next;
    head->next = q;
    return OK;
}

/**
 * @brief 栈是否为空
 * @tparam LElemType 栈的元素类型
 */
template <class LElemType>
bool LinkStack<LElemType>::StackEmpty()
{
    return head->next == NULL;
}

```

3.大整数类的头文件 (long_long_int.h)

```
#pragma once
#include <iostream>
#include <cstring>
using namespace std;
#define MAXN 1024//最长存放的大整数位数(十进制)

//表示大整数的类
class LONG_LONG_INT {
private:
    //存放大整数的数组
    char number[MAXN];
    //大整数长度
    int length;

public:
    //无参构造大整数(0)
    LONG_LONG_INT();
    //用long long构造大整数
    LONG_LONG_INT(long long n);
    //重载大整数加法
    const LONG_LONG_INT operator+ (const LONG_LONG_INT& that) const;
    //重载大整数乘法
    const LONG_LONG_INT operator* (const LONG_LONG_INT& that) const;
    //重载>> 输入大整数
    friend istream& operator >> (istream& in, LONG_LONG_INT& that);
    //重载<< 输出大整数
    friend ostream& operator << (ostream& out, const LONG_LONG_INT&
that);
};
```

4.大整数类的源文件 (long_long_int.cpp)

```
#include "../long_long_int.h"

/**
 * @brief 大整数的无参构造（置0）
```

```

*/
LONG_LONG_INT::LONG_LONG_INT()
{
    memset(this->number, 0, sizeof(this->number));
    length = 1;
}

/**
 * @brief 通过long long构造大整数
 * @param n 大整数的值n
 */
LONG_LONG_INT::LONG_LONG_INT(long long n)
{
    memset(this->number, 0, sizeof(this->number));
    for (int i = 0; ; i++) {
        this->number[i] = n % 10;
        n /= 10;
        if (!n) {
            length = i + 1;
            return;
        }
    }
}

/**
 * @brief 重载运算符* 大整数乘法
 * @param that 参与乘法的另一个大整数
 * @return 乘积
 */
const LONG_LONG_INT LONG_LONG_INT:: operator*(const LONG_LONG_INT&
that) const
{
    LONG_LONG_INT answer;
    for (int i = 0; i < length; i++) {
        for (int j = 0; j < that.length; j++) {
            int newn = answer.number[i + j] + this->number[i] *
that.number[j];
            answer.number[i + j] = newn % 10;
            answer.number[i + j + 1] += newn / 10;
        }
    }

    //特判0

```

```

        if ((length == 1 && !number[0]) || (that.length == 1
&& !that.number[0])) {
            answer.length = 1;
            return answer;
        }

        answer.length = answer.number[length + that.length - 1] ? length +
that.length : length + that.length - 1;
        return answer;
    }

/**
 * @brief 重载运算符+ 大整数加法
 * @param that 参与加法的另一个大整数
 * @return 运算的和
 */
const LONG_LONG_INT LONG_LONG_INT:: operator+(const LONG_LONG_INT&
that) const
{
    LONG_LONG_INT answer;
    for (int i = 0; i < max(that.length, length); i++) {
        answer.number[i] += number[i] + that.number[i];
        answer.number[i + 1] += answer.number[i] / 10;
        answer.number[i] %= 10;
    }

    answer.length = answer.number[max(that.length, length)] ?
max(that.length, length) + 1 : max(that.length, length);
    return answer;
}

/**
 * @brief 重载>> 流输入大整数
 * @param in 输入流
 * @param that 输入的大整数对象
 */
istream& operator >> (istream& in, LONG_LONG_INT& that)
{
    char input[2048];
    in >> input;

    that.length = strlen(input);

```

```

        for (int i = that.length - 1; i >= 0; i--)
            that.number[that.length - 1 - i] = input[i] - '0';

        return in;
    }

/**
 * @brief 重载<< 流输出大整数
 * @param in 输出流
 * @param that 输出的大整数对象
 */
ostream& operator << (ostream& out, const LONG_LONG_INT& that)
{
    for (int i = that.length - 1; i >= 0; i--)
        out << that.number[i] + 0;    //整型提升, int输出

    return out;
}

```

5. 测试普通递归与栈模拟方式的主程序 (main.cpp)

```

#include <iostream>
#include <iomanip>
#include <conio.h>
#include "../long_long_int.h"
#include "../sqstack.hpp"
#include "../linkstack.hpp"
using namespace std;

/* 用栈消解方式模拟函数存放的信息 */
struct Data {
    int n;                //传入的参数, 表示当前的数n
    int returnAddress;    //函数的返回地址 此处实际用于表示该函数进行到
                          //的模块0/1
    Data(){                //无参构造函数
        n = 1;
        returnAddress = 0;
    };
    Data(int _n, int _ra) //双参构造函数
    {

```

```

        n = _n;
        returnAddress = _ra;
    }
};

/**
 * @brief 递归方式求阶乘
 * @param n 求n!
 * @return 阶乘结果
 */
LONG_LONG_INT Factorial_1(int n)
{
    if (n == 1)
        return 1;
    else
        return LONG_LONG_INT(n) * Factorial_1(n - 1);
}

/**
 * @brief 栈消解方式求阶乘
 * @tparam stack 栈的类型
 * @param n 求n!
 * @param function 存放函数信息的栈
 * @return 阶乘结果
 */
template <class stack>
LONG_LONG_INT Factorial_2(int n, stack& function)
{
    LONG_LONG_INT ret;          //模拟通过寄存器传递的各个函数返回值

    function.Push({ n, 0 });    //模拟第一个函数被main调用
    while (!function.StackEmpty()) {
        Data now;
        function.Top(now);
        switch (now.returnAddress) {
            //递归调用的部分
            case 0:
                function.Pop(now);
                if (now.n > 1) {
                    function.Push({ now.n, 1 });    //保存现场
                    function.Push({ now.n - 1, 0 }); //调用下一层
                }
                else {

```



```

        ret = 1;        //返回
    }
    break;
//回溯的部分
case 1:
    function.Pop(now);        //恢复现场
    ret = ret * LONG_LONG_INT(now.n);    //计算返回值
    break;
}
}

return ret;
}

/**
 * @brief 等待回车输入
 * @param prompt 提示语
 */
void wait_for_enter(const char* prompt)
{
    cout << endl << prompt << ", 请按回车键继续";
    while (getchar() != '\n')
        ;
    cout << endl << endl;
}

/**
 * @brief 可视化菜单界面
 * @return 选择的菜单项
 */
int Menu()
{
    //system("mode con: cols=83 lines=30");
    cout << endl << endl << endl << endl;
    cout << "\t\t\t\t\t PA2-栈的应用" << endl;

    cout << endl << endl;
    cout << "\t\t\t\t\t 菜单选择" << endl;
    cout << '\t' << setw(65) << setfill('-') << "" << endl;
    cout << "\t|\t1\t|\t2\t|\t3\t|\t0\t|" << endl;
    cout << '\t' << setw(65) << setfill('-') << "" << endl;
    cout << "\t|\t\t|\t\t|\t\t|\t\t|" << endl;
    cout << "\t| 顺序表栈\t| 链表栈\t| 普通递归\t| 退出演示\t|"

```

```
<< endl;
cout << "\\t|\\t\\t|\\t\\t|\\t\\t|\\t\\t|" << endl;
cout << '\\t' << setw(65) << setfill('-') << "" << endl << endl <<
endl;
cout << "\\t\\t\\t    [请按对应数字选择功能]" << endl;
cout << "\\t\\t\\t\\t\\t\\t";
while (char ch = _getch())
    if (ch >= '0' && ch <= '7')
        return ch - '0';
return 0;
}

int main()
{
    LinkStack<Data> linkstack;
    SqStack<Data> sqstack;
    int ret;
    int n;
    while (ret = Menu()) {
        switch (ret){
            case 1:
                system("cls");
                cout << "-----测试(顺序表)栈消解方式计算n的阶乘-----" <<
endl << endl;
                cout << "请输入求阶乘的最大值" << endl;
                cin >> n;
                wait_for_enter("演示即将开始");
                cout << "n\\t阶乘结果" << endl;
                cout << "-----" << endl;
                for (int i = 1; i <= n; i++)
                    cout << i << "\\t" << Factorial_2(i, sqstack) << endl;
                wait_for_enter("演示完毕");
                system("cls");
                break;
            case 2:
                system("cls");
                cout << "-----测试(链表)栈消解方式计算n的阶乘-----" << endl
<< endl;
                cout << "请输入求阶乘的最大值" << endl;
                cin >> n;
                wait_for_enter("演示即将开始");
                cout << "n\\t阶乘结果" << endl;
                cout << "-----" << endl;
```

```

        for (int i = 1; i <= n; i++)
            cout << i << "\t" << Factorial_2(i, linkstack) << endl;
        wait_for_enter("演示完毕");
        system("cls");
        break;
    case 3:
        system("cls");
        cout << "-----测试递归函数方式计算n的阶乘-----" << endl <<
endl;

        cout << "请输入求阶乘的最大值" << endl;
        cin >> n;
        wait_for_enter("演示即将开始");
        cout << "n\t阶乘结果" << endl;
        cout << "-----" << endl;
        for (int i = 1; i <= n; i++)
            cout << i << "\t" << Factorial_1(i) << endl;
        system("cls");
        break;
    }
}

return 0;
}

```