

作业PA2-1报告

1. 涉及数据结构和相关背景

1.1 栈

后进先出LIFO，限定只在表的一端(表尾)进行插入和删除操作的线性表

允许插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)

- 基本操作：初始化、判空、push、pop、gettop、清空、返回长度
- 顺序栈
 - 预先定义好栈的容量
 - 限定在表尾进行插入和删除操作的顺序表
 - base称为栈底指针，始终指向栈底；当base == NULL时，表明栈结构不存在
 - 空栈：当top=base时为栈空的标记
 - 当栈非空时，top的位置：指向当前栈顶元素的下一个位置
 - 当栈满时再做进栈运算必定产生空间溢出，简称“上溢”；当栈空时，再做退栈运算也将产生溢出，简称为“下溢”
- 链栈（本实验实现的是模板链栈）
 - 无需预先定义好栈的容量
 - 不带头结点的单链表，其插入和删除操作仅限制在表头位置上进行。链表的头指针即栈顶指针
 - 栈空条件：s=NULL；栈满条件：无 / 无Free Memory可申请

2. 实验内容

2.1 问题描述

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：

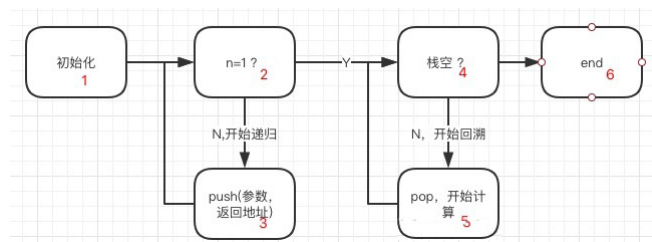
```
public long f(int n)
{
    if(n==1) return 1; //停止调用
    return n * f(n-1); //调用自身
}
```

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。

递归通常有两个过程：

- 递归过程：不断递归入栈push，直到停止调用 $n=1$
- 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。

可以把上述过程分为以下几个状态：



参考这个状态机，用栈模拟 n 的阶乘的递归调用过程。

2.2 基本要求

要存储的数据：

```
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<>();
```

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。

2.3 数据结构设计

利用栈的数据结构来进行阶乘运算，

用顺序表的下标模拟内存中的地址

- 当 n 大于1时向栈里push Data 类型数据，包括值以及地址
- 当 n 等于1时观察栈里是否有数据，有的话依次pop出来进行计算
- 将每一位的值返回到相应的地址即可

2.4 功能说明

2.4.1 Data类的实现

- Data类的实现

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

```

typedef struct Data
{
public:
    long long getter_num() //类公有函数，取值
    {
        return n;
    }

    long long getter_address() //类公有函数，取地址
    {
        return returnAddress;
    }

    void setter_num(long long num) //类共有函数，设置值
    {
        n = num;
    }
    void setter_address(long long add) //类公有函数，设置地址
    {
        returnAddress = add;
    }

private:
    long long n; //值
    long long returnAddress; //函数的返回地址

}Data;

```

这里用顺序表模拟内存，其实可以在内存中实在malloc申请一块内存，存放Data类型中

- 输出Data信息重载<<

```

ostream& operator<<(ostream& cout, Data& data)
{
    cout << data.getter_num() << endl;
    return cout;
}

```

重载<<，打印Data类型数据的值

2.4.2 Stack栈的实现

- abstract.h 抽象栈的实现

```

template<class dataType>
class MyStack
{
public:
    virtual bool isEmpty() = 0; //判空
    virtual void push(const dataType& d) = 0; //push
    virtual dataType pop() = 0; //pop
    virtual dataType top() = 0; //top

};

```

- stack.hpp模板类实现

```

#pragma once
#include "AbstractStack.h"
#include <iostream>
#include <stdio.h>
using namespace std;

template<class dataType>
struct node //链栈节点
{
    node(); // 无参构造
    node(const dataType& d, node<dataType>* n = NULL) //有参构造
    {
        data = d;
        next = n;
    }
    dataType data;
    node<dataType>* next;
};

template<class dataType>
class Stack : public MyStack<dataType>
{
public:
    Stack() //无参构造
    {
        top_p = NULL;
        //cout << "New LinkedStack Created!" << endl;
    }

    bool isEmpty() //判空
    {
        return top_p == NULL;
    }

    void push(const dataType& d) //push
    {
        node<dataType>* add = new node<dataType>(d, top_p);
        top_p = add;
    }

```

```

        //cout << "Push data successfully" << endl;
    }

    dataType pop() //pop
    {
        if (isEmpty())
        {
            throw 0;
        }

        dataType d = top_p->data;
        node<dataType>* tmp = top_p;
        top_p = top_p->next;
        delete tmp; //释放内存
        return d;
    }

    dataType top() //取栈顶
    {
        if (isEmpty()) //检查是否空
        {
            throw 0;
        }

        return top_p->data;
    }

    ~Stack() //析构
    {
        while (top_p != NULL) //依次出栈释放内存
        {
            node<dataType>* tmp = top_p;
            top_p = top_p->next;
            delete tmp;
        }
    }

    node<dataType>* top_p; //头指针
};

```

链栈节点和链栈的实现 注意模板的使用

2.4.3 递归函数实现

- factorial函数实现

```

long long factorial(Stack<Data>& st, long long n)
{
    /*
    * @param st : 数据栈
    * @param n : 计算阶乘的数字
    */
}

```

```

long long num = n;
if (n < 0) //Robust examination
{
    cout << "Factorial Range Error." << endl;
    return -1;
}

while (n > 1) //大于1时候Data信息入栈
{
    Data tmp;
    tmp.setter_num(n);
    tmp.setter_address(n);
    st.push(tmp);
    n--;
}

long long ans = 1; 初始化结果

while (!st.isEmpty()) //栈非空的时候不断出栈
{
    Data out = st.top();
    st.pop();
    ans *= out.getter_num();
    address[out.getter_address()] = ans;
}

return address[num]; //返回地址中存有的元素值
}

```

模拟状态机展示过程完成函数构建

- main函数初始化

```

int main()
{
    address[1] = 1;
    address[0] = 1;

    Stack<Data>* factorialStack = new Stack<Data>(); //新建栈
    long long n1;
    cout << "Enter the number to begin stack factorial process" << endl;
    cin >> n1;
    long long answer = factorial(*factorialStack, n1);
    cout << answer << endl;
    return 0;
}

```

2.5 调试分析

- 链栈功能调试
 - 判空及push功能

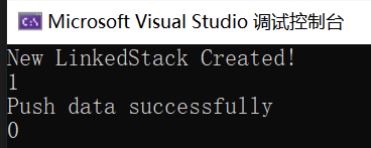
```
int main()
{
    Stack<Data>* factorialStack = new Stack<Data>();
    cout << factorialStack->isEmpty() << endl;

    /*push data*/
    Data test;
    test.setter_num(1); test.setter_address(2);
    factorialStack->push(test);

    /*Empty or not ?*/
    cout << factorialStack->isEmpty() << endl;

    return 0;
}
```

效果:



```
Microsoft Visual Studio 调试控制台
New LinkedStack Created!
1
Push data successfully
0
```

- pop功能

```
int main()
{
    Stack<Data>* factorialStack = new Stack<Data>();

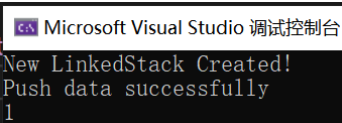
    Data test;
    test.setter_num(1); test.setter_address(2);

    factorialStack->push(test);

    Data poper = factorialStack->pop();
    cout << poper << endl;

    return 0;
}
```

效果:



```
Microsoft Visual Studio 调试控制台
New LinkedStack Created!
Push data successfully
1
```

- top功能

```

int main()
{
    Stack<Data>* factorialStack = new Stack<Data>();

    Data test;
    test.setter_num(1); test.setter_address(2);

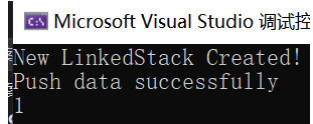
    factorialStack->push(test);

    Data poper = factorialStack->top();
    cout << poper << endl;

    return 0;
}

```

效果

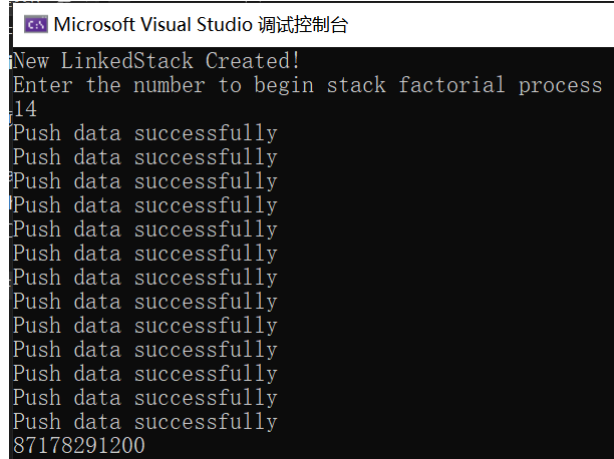


```

Microsoft Visual Studio 调试控制
New LinkedStack Created!
Push data successfully
1

```

- 阶乘功能调试
 - 效果



```

Microsoft Visual Studio 调试控制台
New LinkedStack Created!
Enter the number to begin stack factorial process
14
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
Push data successfully
87178291200

```

- 关于递归函数的堆栈溢出
 - 在正常写递归函数的情况下，当n在4700左右(4703?)时会有堆栈溢出(StackOverflow)的错误
 - 正常实现递归函数计算阶乘


```
long long factorial(long long n)
{
    if (n == 1) return 1;

    return factorial(n - 1) * n;
}
```

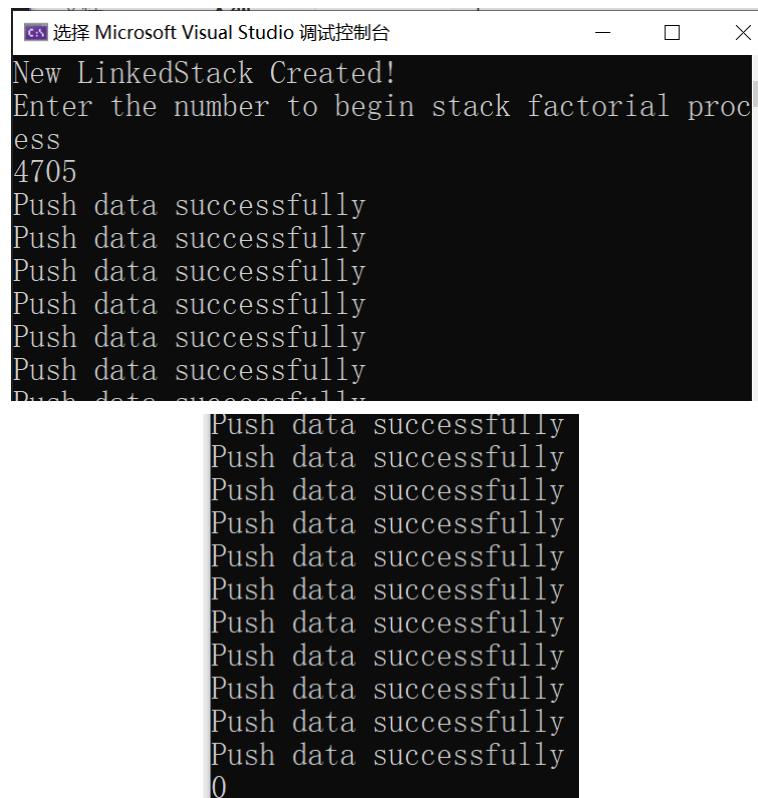
- 主函数进行检测

```
int main()
{
    int aout = 1;
    for (int i = 1; ; i++)
    {
        cout << factorial(i) << " " << i << endl;
    }
    return 0;
}
```

运行结果:



- 栈消解后在4700左右



可以看到其实栈的使用消解掉了递归函数的最大深度问题，虽然没有最后结果（远远超过64位表示上限）但是并没有报错

2.6 递归计算阶乘时间复杂度

其函数的时间复杂度有以下递归方程

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ T(n-1) + O(1) & n > 1 \end{cases}$$

递推有(并不严谨)

$$\begin{aligned} T(n) &= T(n-1) + O(1) = T(n-2) + O(2) \\ &= \dots = O(1) + (n-1)O(1) = O(n) \end{aligned}$$

阶乘不是分治，所以无法用主定理进行求解

关于主定理计算分治递归算法时间复杂度：

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

https://blog.csdn.net/qq_40673808

2.7 总结和体会

- 栈的使用有效消了递归最大深度的问题，实际上递归是自顶向下的将规模为n的问题变成了规模为n-1的问题，继续下去，而在pop的过程中实现了自底向上的地推过程
- 关于returnAddress是否需要：需要，在一次计算中，返回的地址中存的过程量可以直接作为0~n之间的数的阶乘结果，一次计算后其实可以给出0~n所有数阶乘的结果（类似动态规划的思想）

3. 实验总结

本实验涉及到

- 手动实现简单链栈模板及其功能，包括判空，入栈，出栈，取栈顶
- 理解与复现递归函数的调用过程，回溯过程
- 将栈运用到消解递归调用中

4. 源代码

- AbstractStack.h

```
template<class dataType>

class MyStack
{
public:
    virtual bool isEmpty() = 0;
    virtual void push(const dataType& d) = 0;
    virtual dataType pop() = 0;
    virtual dataType top() = 0;
};
```

- Stack.hpp

```
#pragma once
#include "AbstractStack.h"
#include <iostream>
#include <stdio.h>
using namespace std;

template<class dataType>
struct node
{
    node();
    node(const dataType& d, node<dataType>* n = NULL)
    {
        data = d;
        next = n;
    }
    dataType data;
    node<dataType>* next;
};

template<class dataType>
```

```

class Stack : public MyStack<dataType>
{
public:
    Stack()
    {
        top_p = NULL;
        cout << "New LinkedStack Created!" << endl;
    }

    bool isEmpty()
    {
        return top_p == NULL;
    }

    void push(const dataType& d)
    {
        node<dataType>* add = new node<dataType>(d, top_p);
        top_p = add;
        cout << "Push data successfully" << endl;
    }

    dataType pop()
    {
        if (isEmpty())
        {
            throw 0;
        }

        dataType d = top_p->data;
        node<dataType>* tmp = top_p;
        top_p = top_p->next;
        delete tmp;
        return d;
    }

    dataType top()
    {
        if (isEmpty())
        {
            throw 0;
        }

        return top_p->data;
    }

    ~Stack()
    {
        while (top_p != NULL)
        {
            node<dataType>* tmp = top_p;
            top_p = top_p->next;
            delete tmp;
        }
    }
}

```

```

node<dataType>* top_p;

};

```

- StackData.h

```

#pragma once
#include <iostream>
#include <stdio.h>
using namespace std;

typedef struct Data
{
public:
    long long getter_num()
    {
        return n;
    }

    long long getter_address()
    {
        return returnAddress;
    }

    void setter_num(long long num)
    {
        n = num;
    }
    void setter_address(long long add)
    {
        returnAddress = add;
    }

private:
    long long n; //函数的输入参数
    long long returnAddress; //函数的返回地址 //需要，这样小于等于n的阶乘都可以找到

}Data;

ostream& operator<<(ostream& cout, Data& data)
{
    cout << data.getter_num() << endl;
    return cout;
}

```

- stackFactorial.cpp

```

#include "Stack.hpp"

```

```

#include "StackData.h"
#include <iostream>
using namespace std;
const int N = 1e7 + 5;
long long address[N];

long long factorial(Stack<Data>& st, long long n)
{
    long long num = n;
    if (n < 0)
    {
        cout << "Factorial Range Error." << endl;
        return -1;
    }

    while (n > 1)
    {
        Data tmp;
        tmp.setter_num(n);
        tmp.setter_address(n);
        st.push(tmp);
        n--;
    }

    long long ans = 1;

    while (!st.isEmpty())
    {
        Data out = st.top();
        st.pop();
        ans *= out.getter_num();
        address[out.getter_address()] = ans;
    }

    return address[num];
}

int main()
{
    address[1] = 1;
    address[0] = 1;

    Stack<Data>* factorialStack = new Stack<Data>();
    long long n1;
    cout << "Enter the number to begin stack factorial process" << endl;
    cin >> n1;
    long long answer = factorial(*factorialStack, n1);
}

```

```
cout << answer << endl;
```

```
return 0;
```

```
}
```