《数据结构》上机报告

2019 年 9 月 27 日

姓名：李佳庚 学号： 1852409 班级： 计算机1班 得分：

<table>
<tr><td>实验题目</td><td colspan="2">栈</td></tr>
<tr><td>问题描述</td><td colspan="2">利用栈模拟阶乘函数的调用过程。
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。

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。需要利用栈来对函数的递归和回溯进行模拟。</td></tr>
<tr><td rowspan="2">基本要求</td><td colspan="2">(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) 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。</td></tr>
<tr><td>已完成基本内容（序号）：</td><td>1，2，3，4，5</td></tr>
<tr><td>选做要求</td><td colspan="2">1. 利用参考信息中的函数指针思想重写有关地址的 stack 类

2. 模拟函数入栈出栈时的现场保存。

3. 模拟函数入栈时的地址传参。</td></tr>
</table>

	4. 模拟函数入栈时的参数传递。 5. 通过函数指针模拟函数的递归调用
	已完成选做内容（序号） | 1，2，3，4，5
数据结构设计	``` class stackLoc { public: using _Lc = long long(*)(long long); static const int MAX = 10000; private: _Lc *stackBottle; int num; bool popFlag; ``` 设计了名为 stackLoc 的栈类。 其中，stackBottle 为整个栈的基指针。 num 为目前栈中元素的个数。 PopFlag 是函数是否进行出栈操作的标志。 另外，定义了_Lc 这个类型，使得整个程序能够按照不同的数据类型进行测试。 MAX 为静态常变量，为栈容纳的最大函数指针数目。
功能（函数）说明	``` public: stackLoc() :num(0), popFlag(false), stackBottle(new _Lc[MAX]) {} ``` 普通构造函数，有且仅有的构造函数，可以满足本题目的所有需要。 ``` void push(_Lc loc) { stackBottle[num] = loc; num++; } ``` 设计的栈的栈顶指针相当于指向下一次 push 操作应该指向的区域。 于是直接 push，而后增加 num，表示栈中元素的增加。 ``` _Lc pop() { num--; return stackBottle[num]; } ``` 虽然 num 的数目首先减小，但是栈中元素却没有被消灭。仍可以通过栈底指针的随机访问得到名义上弹出的元素。 ``` constexpr bool isEmpty() { return (num == 0) ? 1 : 0; } constexpr bool isPop() { return popFlag; } ``` 两个判断函数

第一个判断栈是否为空。
第二个判断现在是否应该弹出栈中的函数指针。

```
void changePopFlag() {
    popFlag = popFlag ? 0 : 1;
}
```

利用 popFlag 本身的值对其身进行赋值。
可以通过本函数，告诫入栈出栈过程控制的函数。令其合理地操作函数。

```
long long factorial(long long input) {

    if (stl.isPop())
        goto pop;

    if (1 != input && !stl.isPop()) {
        stl.push(factorial);
        factorial(--input);
        goto popend;
    }

    stl.changePopFlag();

pop:
    if (stl.isEmpty())
        return result;
    else {
        long long(*funcp)(long long) = stl.pop();
        funcp(++input);
        result *= input;
    }
popend:
    return result;
}
```

阶乘函数。
个人以为完美模拟了函数的四大特点：保存现场、传参、返回、恢复现场。

1. 在递归过程当中，本函数的 input 参数如果不是 1，并且现在还没有到弹出函数指针的时候，那么就应该执行：

   ```
   stl.push(factorial);
   factorial(--input);
   goto popend;
   ```

   将正在执行的函数的函数指针压入栈，然后嵌套调用 factorial 函数，改变输入的参数值。
2. 直到在上面代码框中的 input 参数传入为 1 时，不符合

   ```
   if (1 != input && !stl.isPop())
   ```

   本条件

   跳过 if 语句进行下一个：

   ```
   stl.changePopFlag();
   ```

   改变了类中 popflag 的值。
   从现在开始，函数应该弹出函数指针，而不是压入函数指针。
3. 如果栈中还有函数指针，那么就一直 pop 出指针来。
   直到栈为空为止。
   在此过程当中，通过不断执行 funcp 所代表的函数来进行阶乘。

```
if (stl.isEmpty())
    return result;
else {
    long long(*funcp)(long long) = stl.pop();
    funcp(++input);
    result *= input;
}
```

4. 在上述代码框中，本函数的执行：`funcp(++input);`
需要将目光重新放到 factorial 函数开始的地方：

```
if (stl.isPop())
    goto pop;
```

现在的 popflag 已经被 changePopFlag 函数修改过了。
所以直接利用 goto 语句跳到 pop 段
这也就是模拟函数保存现场、恢复现场的地方。

栈中最后一个被压入的函数，只执行到了`stl.changePopFlag();`

```
pop:
    if (stl.isEmpty())
        return result;
```

而它的下一句话就是：
在 goto 到 pop 段的这段过程当中，由于栈中不为空，所以函数仍然会执行到：

```
long long(*funcp)(long long) = stl.pop();
funcp(++input);
```

语句
从而继续看似恢复现场，实则继续保护现场的递归调用，逐步将 input 恢复，使得后面的 result *= input 可以顺利执行。

5. 这之后，pop 的函数已经执行完毕，函数又重新回到第一波压入函数指针的时候：

```
    factorial(--input);
    goto popend;
}
```

这个时候，由于 factorial 已经执行完毕。所以直接跳至 popend，返回 result。

开发环境

Win10 Microsoft Visual Studio Community 2017 15.9.3 Debug x86

调试分析

（运行结果截图）

当 input = 7 时：

```
int main() {
    cout << factorial(7);
}
```

当 input = 6 时：

```
int main() {
    cout << factorial(6);
}
```

```
720
C:\Users\Aober\source\repos\dsoj\Debug\6.4_函数调用地址.exe (进程 12776)已退出，返回代码为: 0。
```

心得体会

（对整个实验过程做出总结，对重要的算法做出性能分析。）

1. 普通递归函数的探索：

```
long long fac(int input) {
    if (input == 1)
        return 1;
    else
        return input * fac(input - 1);
}

using namespace std;


int main() {

    cout << fac(4795);

}
```

当输入 4795 时，会弹出堆栈溢出的提示

而输入 4790 一下的数字，则可以保证二十次输入之内没有依次报错。

```
long long fac(int input) {
    if (input == 1)
        return 1;
    else
        return input * fac(input - 1);
}

using namespace std;


int main() {

    cout << fac(4790);

}
```

```
Microsoft Visual Studio 调试控制台
0
C:\Users\Aober\source\repos\dsoj\Debug\6.4_函数调用地址.exe (进程 9640)已退出，返回代码为: 0。
若要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台
按任意键关闭此窗口...
```

由此可以基本确认，普通递归函数可以支持的范围在[0，4790]左右。

2. 本 stack 类的最大值探索

```
int main() {
    cout << factorial(2220);

}
```

这个是可以的。
但在

```
int main() {
    cout << factorial(2230);

}
```

左右的区域就会报错：stack overflow

```
constexpr bool isPop() noexcept {
    return popFlag;
}

void changePopFlag() {
    popFlag = popFlag ? 0 : 1;
}


ackLoc stl;
ng long result = 1;
```

```
未经处理的异常
0x00323027 处有未经处理的异常(在 6.4_函数调用地址.exe 中):
0xC00000FD: Stack overflow (参数: 0x00000001, 0x01002F8C)。
复制详细信息
异常设置
```

可能其他机器的其他代码运行出来的结果与我不同。
但是我的确是找到了一组范围比较小的区间。
一旦数字大于 2230，几乎是每次运行都会报出堆栈溢出的错误。
然而如果数字小于 2220，每二十次运行可能只会报出一条错误。
所以我找到的区间为[0, 2220]，超过这个数字，爆栈的可能性就会大大增加。

3. 阶乘函数递归时间复杂度分析：
每次进行递归调用的时候，时间复杂度为常数。
然而完成一次阶乘需要 n 次调用。
所以时间复杂度为 O(n)。