经常在把递归函数转化为非递归形式时云里雾里，最近恰好在重温二叉树的非递归遍历算法，结合之前忘了哪本书上介绍的一种根据具体二叉树的图形画线求各个遍历序列的方法，大致归纳了一下递归函数如何迭代化。

首先分析递归向迭代转化时，困难的地方在哪，为什么递归可以轻松简短地写出，而迭代就必须绕来绕去。根据程序在内存中调用函数的过程可以知道，递归过程系统帮助我们记录了现场信息，而迭代过程我们需要自己判断对当前节点采用哪块代码块进行处理。通俗的说，迭代过程我们不知道当前访问的节点使第几次访问。

例如给出一段递归代码如下：

1. **void** fun(p1)
2. {
3. code1
4. fun(p2)
5. code2
6. fun(p3)
7. code3
8. fun(p4)
9. code4
10. }

函数执行过程大致为：

1. 第1次访问参数集合等于p1的节点时，执行操作code1后把参数集合等于p2的节点压入栈中。
2. 第2次访问参数集合等于p1的节点时，执行操作code2后把参数集合等于p3的节点压入栈中。
3. 第3次访问参数集合等于p1的节点时，执行操作code3后把参数集合等于p4的节点压入栈中。
4. 第4次访问参数集合等于p1的节点时，执行操作code4后把该节点退栈。

若定义struct Node结构体存储fun函数的各项参数。则将该函数转化为迭代的时候，栈stack结点定义为:

struct SNode{

struct Node e;

int cnt;

}

cnt表示当前结点是第几次访问。

设调用fun函数的初始参数为p，则可得迭代化之后的程序为：

1. **struct** SNode temp;
2. temp.e=p;
3. temp.cnt=0;
4. stack.push(temp);
5. whihle(!stack.empty())
6. {
7. stack.top().cnt++;
8. temp=stack.top();
9. **if**(temp.cnt==1)
10. {
11. code1;
12. **struct** SNode t;
13. t.e=p2;
14. t.cnt=0;
15. stack.push(t);
16. } **else** **if**(temp.cnt==2)
17. {
18. code2;
19. **struct** SNode t;
20. t.e=p3;
21. t.cnt=0;
22. stack.push(t);
23. } **else** **if**(temp.cnt==3)
24. {
25. code3;
26. **struct** SNode t;
27. t.e=p4;
28. t.cnt=0;
29. stack.push(t);
30. } **else** **if**(temp.cnt==4)
31. {
32. code4;
33. stack.pop();
34. }
35. }

可以看出，其实函数调用自身的命令将整个函数体划分成了一些代码块，记录访问次数即可轻松确定代码块，对于有返回值的递归函数，只需要在迭代过程中用一个辅助变量保存当前节点的计算记过即可。

一些实例会在同文件夹下的cpp中给出。