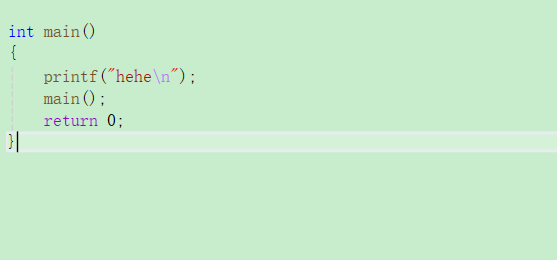
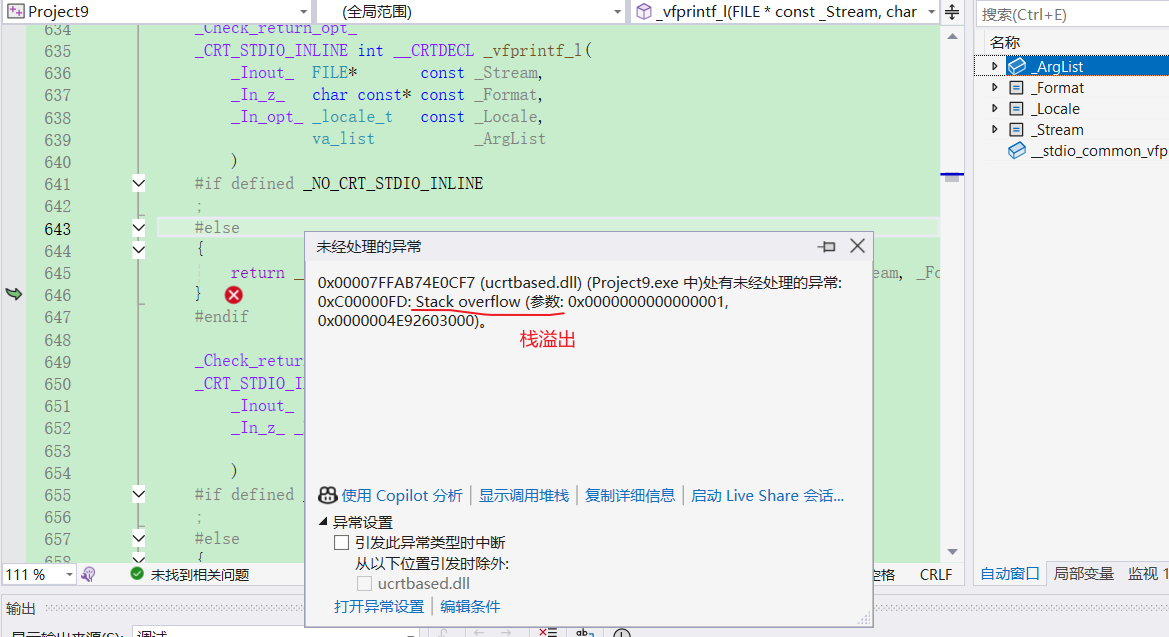
第9讲：函数递归

1、什么是递归

递归是基于C语言函数：**函数自己调用自己**。



调试代码，出现**栈溢出**：



递归思想：把一个大型复杂问题层层转化为一个与原问题相似，但规模较⼩的子问题来求解；直到子问题不能再被拆分，递归就结束了。所以递归的思考方式就是把大事化小的过程。

**递归中的递就是递推的意思，归就是回归的意思。**

2、递归的限制条件

2.1 递归存在限制条件，当满足这个限制条件的时候，递归便不再继续。

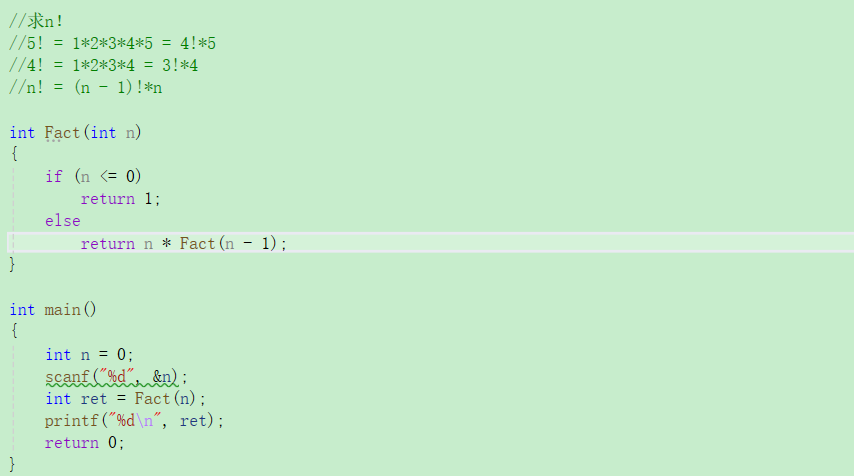
2.2 每次递归调用之后越来越接近这个限制条件。

以上条件为必要条件，必须有但是有可能不够。

3、递归例子

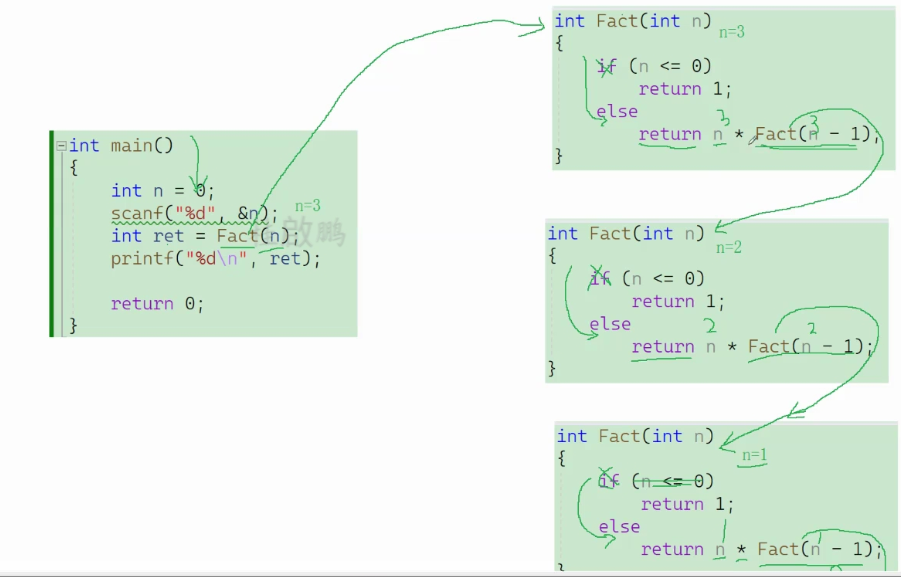
3.1 n的阶乘

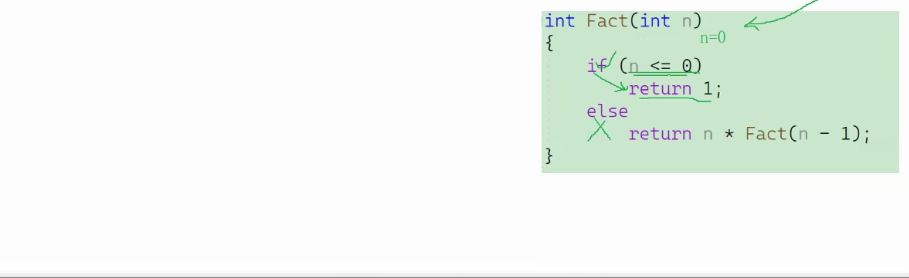
由5! = 1\*2\*3\*4\*5 = 4!\*5、4! = 1\*2\*3\*4 = 3!\*4得出：n! = (n - 1)!\*n



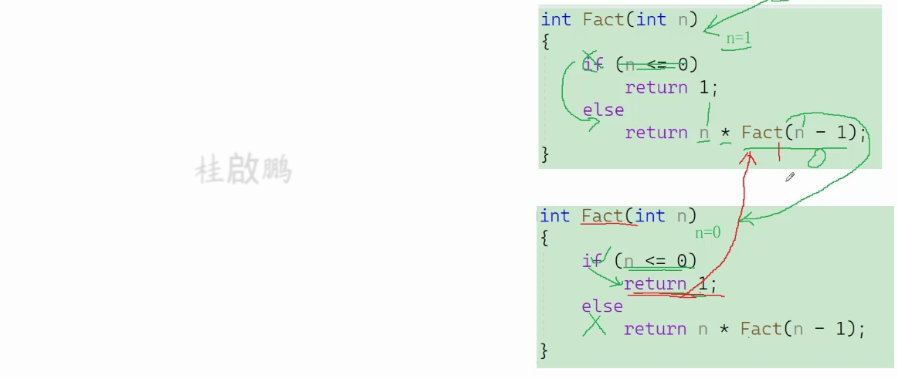
代码实现原理：

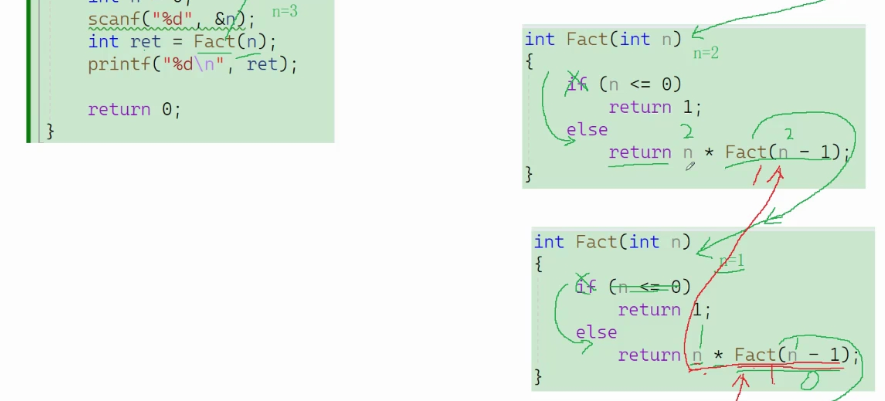
先递推：

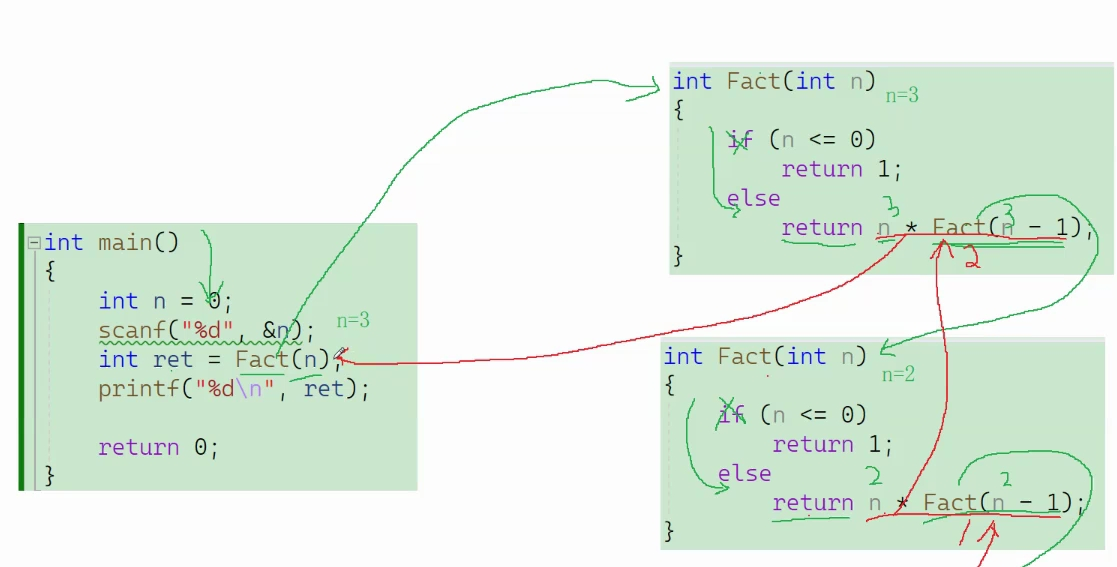




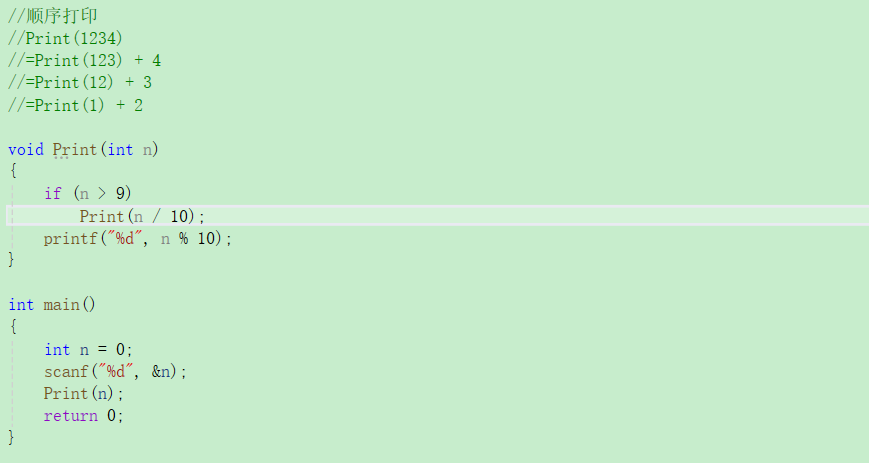
后回归：





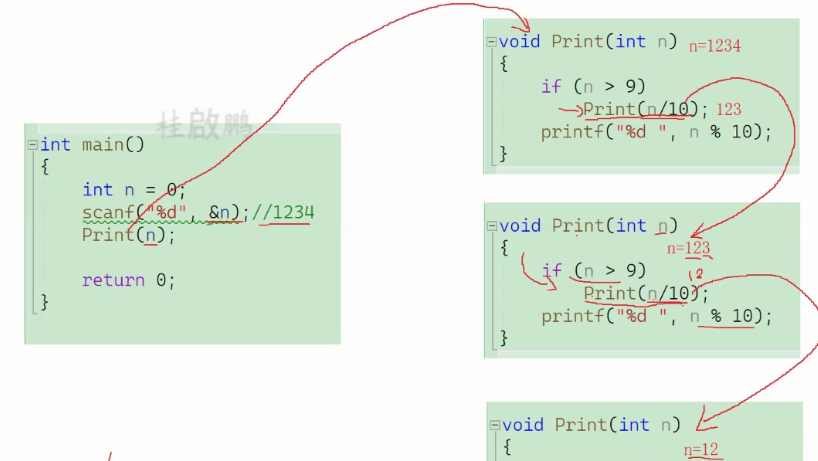


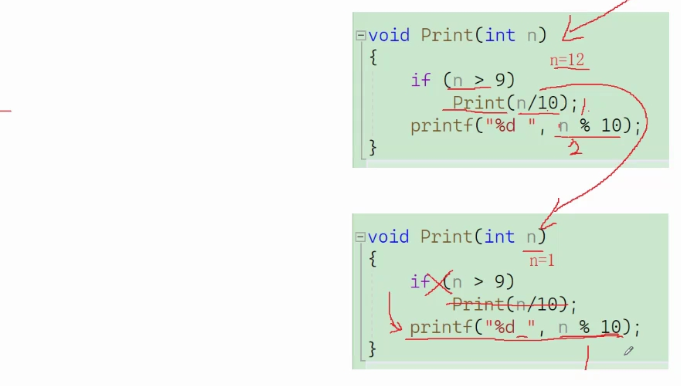
3.2 按照顺序打印整数每一位



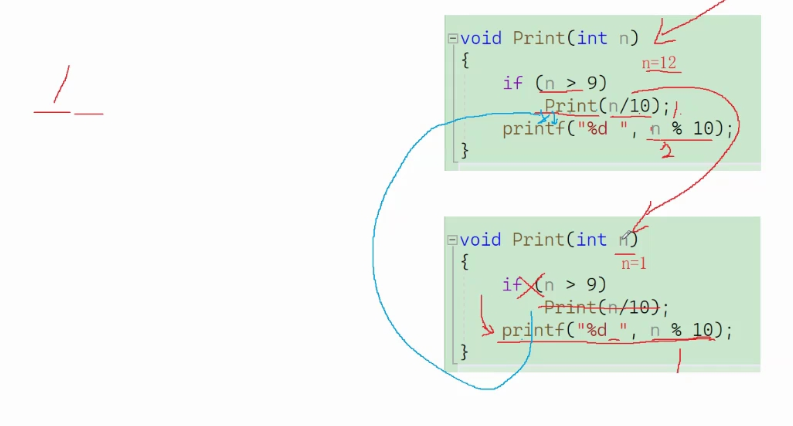
代码实现原理：

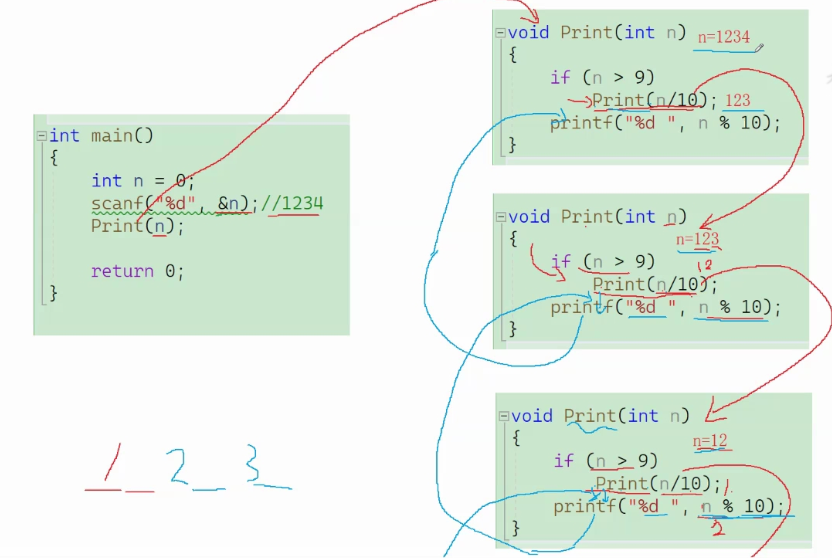
先递推：

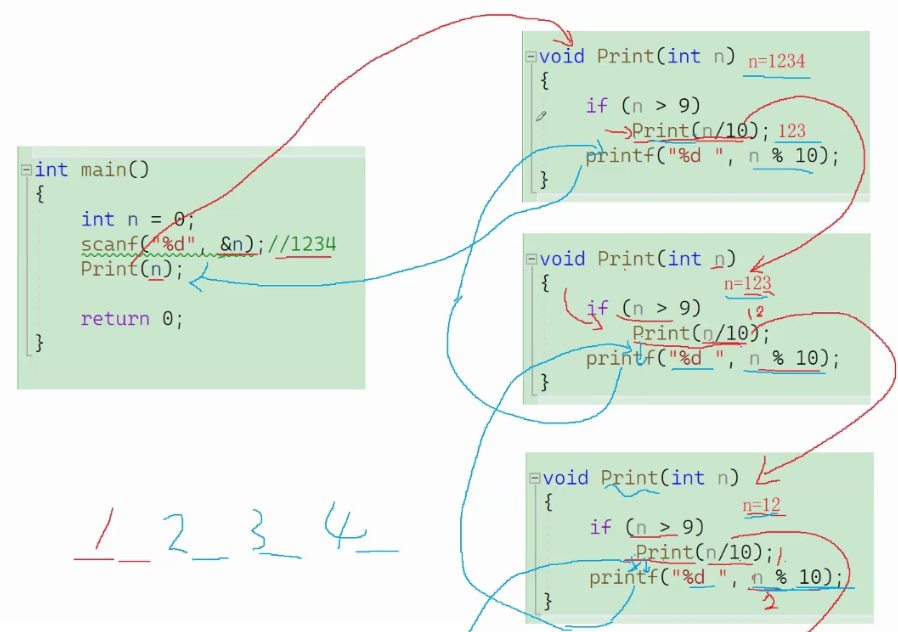




后回归：

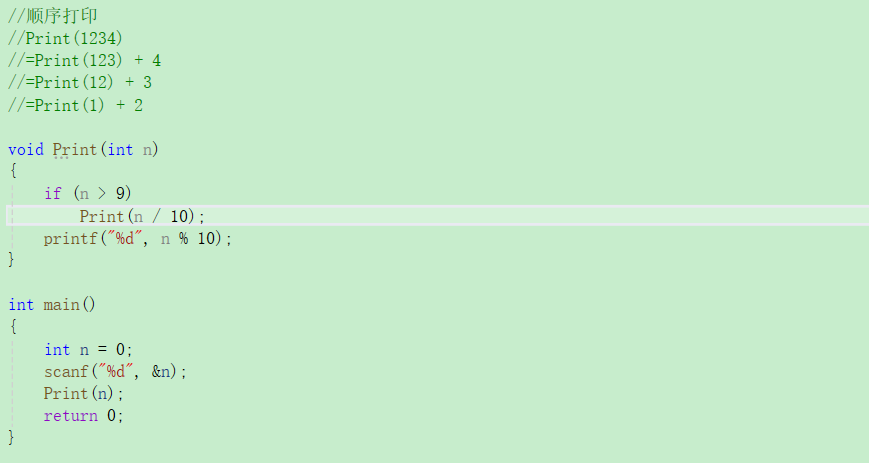






上述代码不能使用if…else…语句。上述代码递归限制条件：n > 9，每次n/10就越接近n < 10。

4、递归和迭代



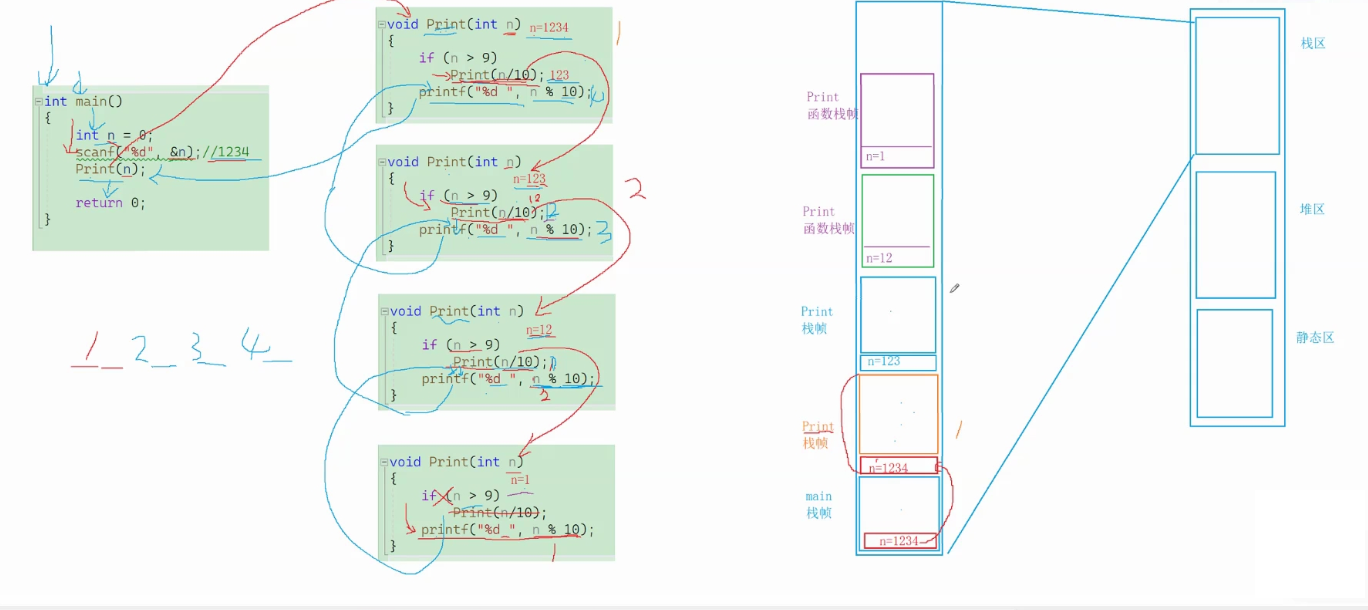
上述代码中，Print函数是可以产生正确的结果，但是在递归函数调用的过程中涉及一些运行时的开销。

运行时的开销：在C语言中每一次函数调用，都要需要为本次函数调用在栈区申请一块内存空间来保存函数调用期间的各种局部变量的值，这块空间被称为运行时的堆栈，或者函数栈帧。

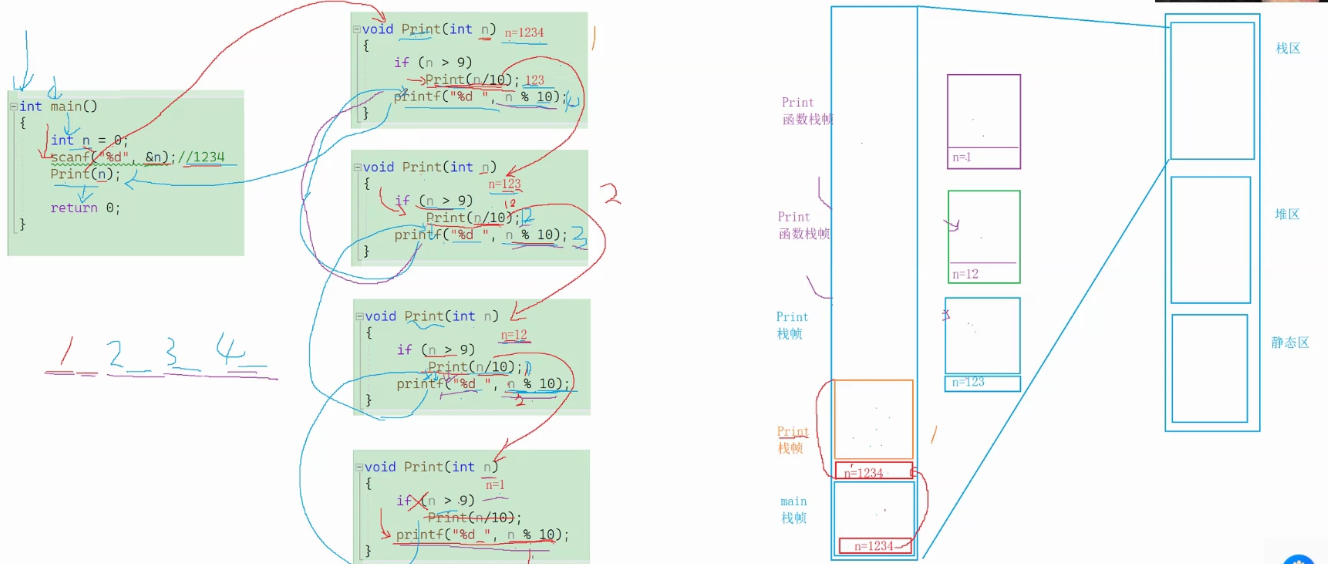
如果函数不返回，函数对应的栈帧空间就一直占用，所以如果函数调用中存在递归调用的话，每一次递归函数调用都会开辟属于自己的栈帧空间，直到函数递归不再继续，开始回归，才逐层释放栈帧空间。

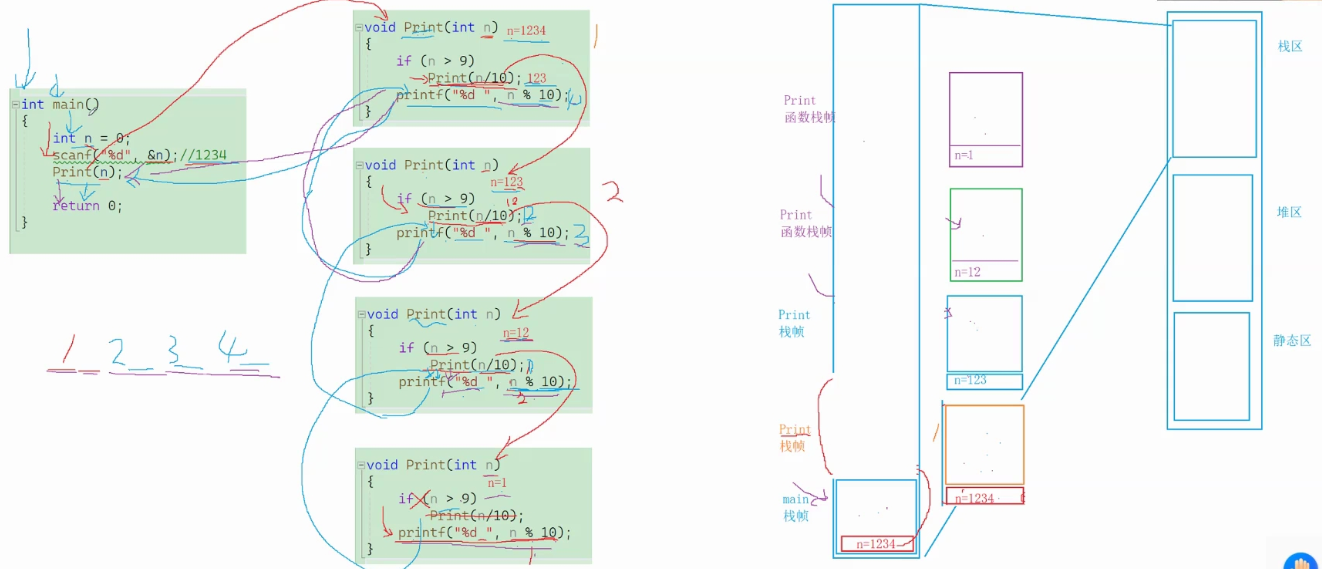
如果采用函数递归方式完成代码，递归层次太深，就会浪费太多的栈帧空间，也可能引起栈溢出（Stack Overflow）的问题。

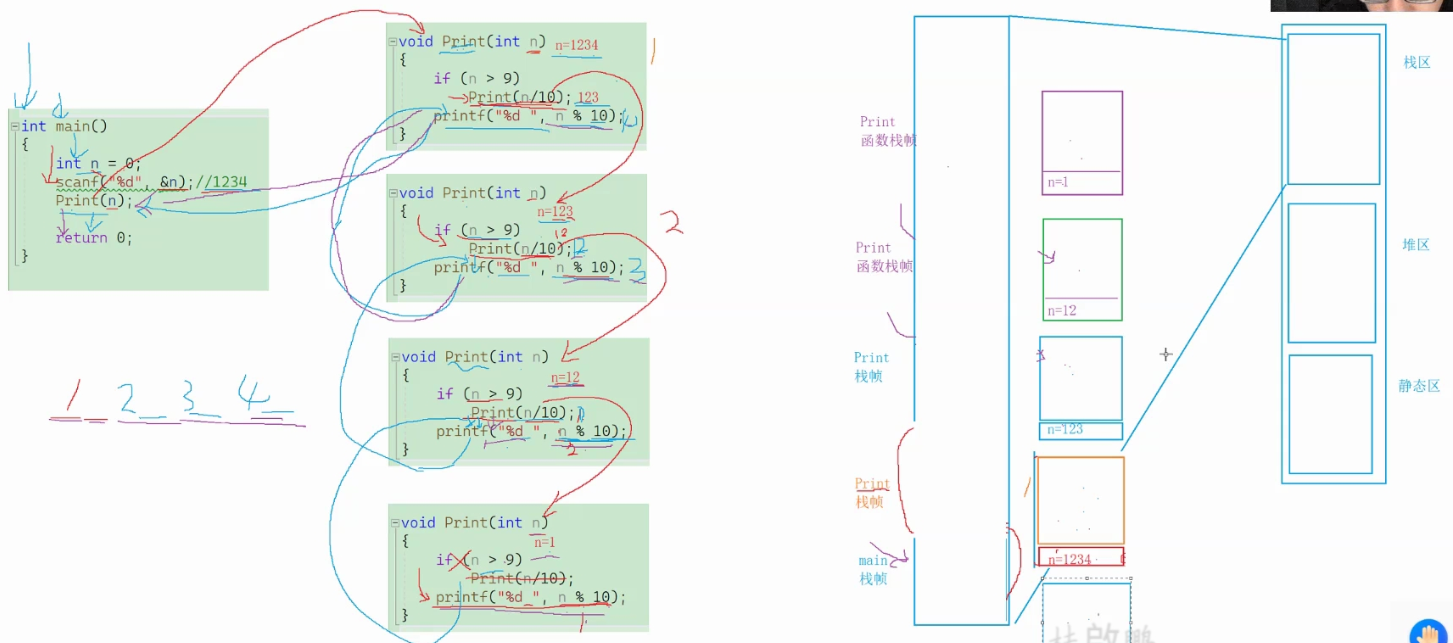
图解上述代码内存布局：



最开始调用main函数，在栈区空间上申请一块内存，每一次调用Print函数又会在栈区空间上申请一块内存，一共要申请5块内存空间。（以n=1234为例）

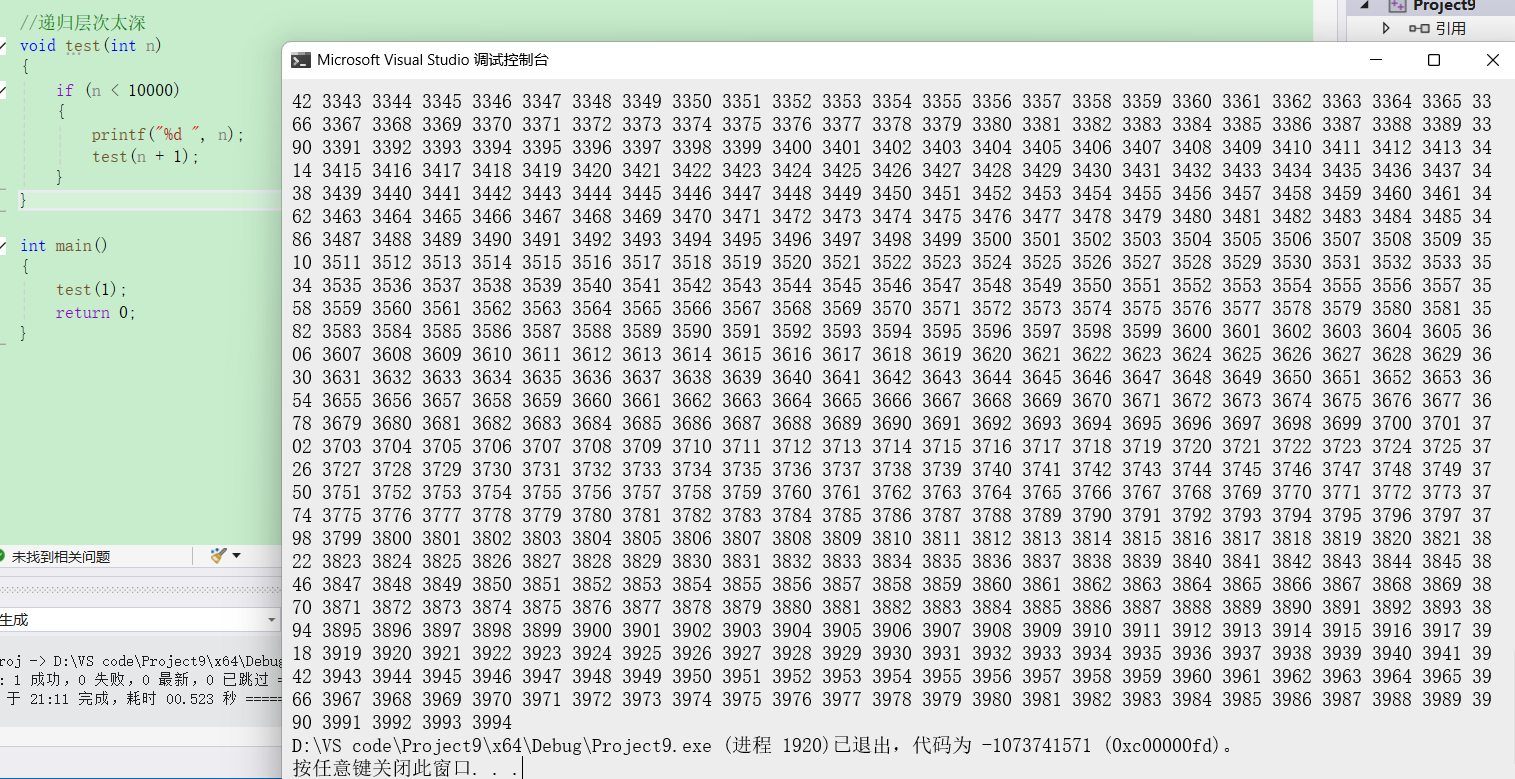




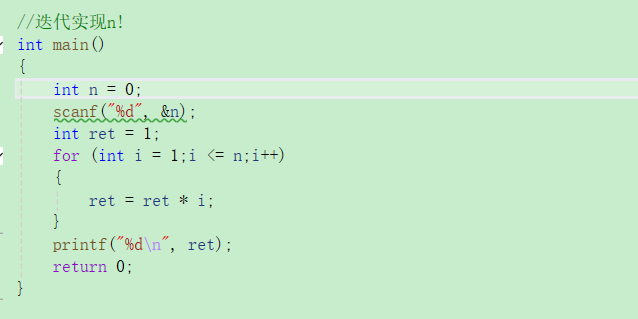


当n=1，Print函数执行完printf(“%d ”, n % 10);开始逐层释放内存空间，直到main函数结束，占用的内存空间全部还给操作系统。

详细的过程：函数如何调用、如何传参、如何开辟空间、空间如何还给操作系统… — 《函数栈帧的创建和销毁》



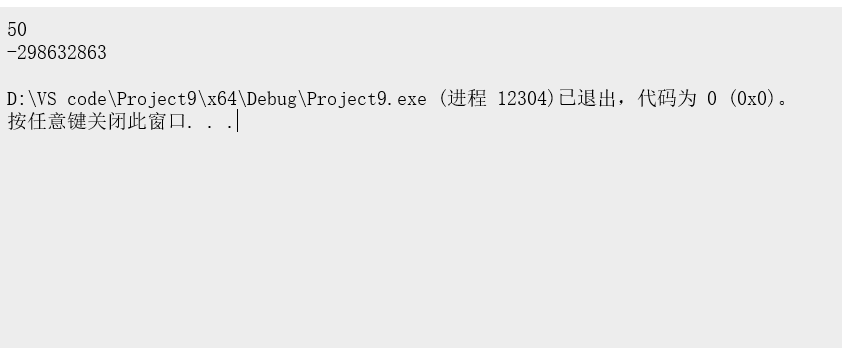
如果不想使用递归出现栈溢出的问题，就得想其他的办法，通常就是**迭代。（通常就是循环的方式）**



很多问题以递归的形式进行解释的，这只是因为它比非递归的形式更加清晰， 但是这些问题的迭代实现往往比递归实现效率更高。 当一个问题非常复杂，难以使用迭代的方式实现时，此时递归实现的简洁性便可以补偿它所带来的运行的开销。

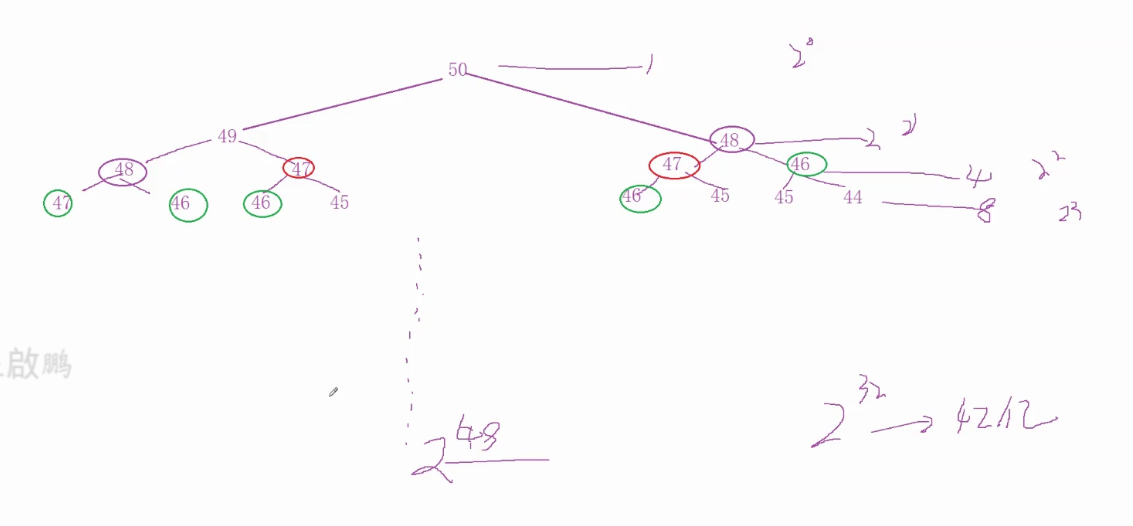
斐波那契数问题：





当使用递归的方式求解，运算的时间会非常长。

图解上述代码逻辑：



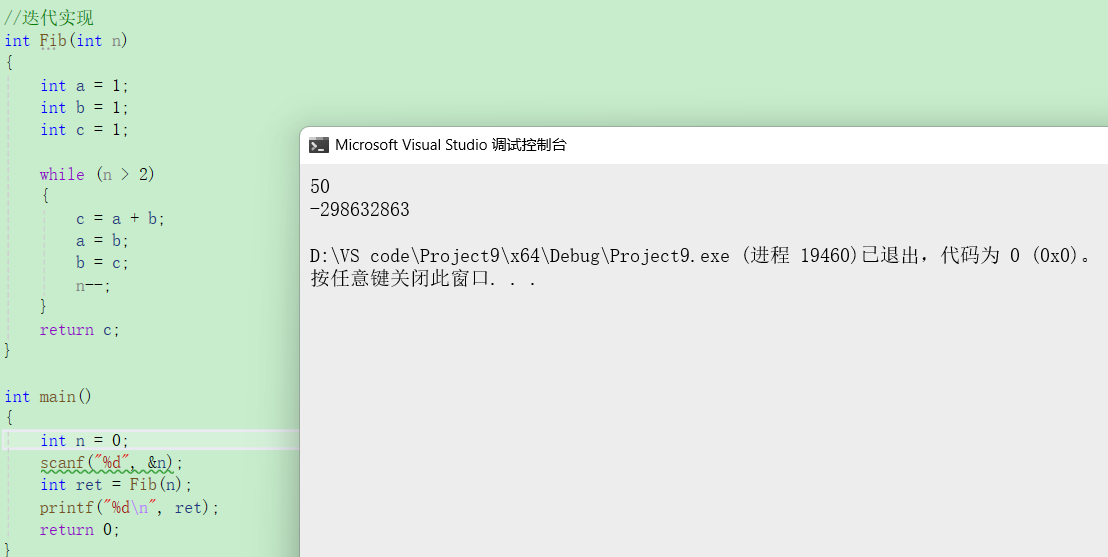
我们求第50个斐波那契数，需要先求得第49个和第48个斐波那契数，求第49个、48个斐波那契数又需要求第48个和47个、第47个和第46个斐波那契数…随着递推展开，每一行的数个数也是呈现指数增长（最后一行数字个数249个）。而且在递推过程中，还有些数字是重复计算。



我们可以发现，当我需要求第30个斐波那契数，第三个斐波那契数被计算了317811次，效率太低。

为什么不会出现栈溢出：其实代码在运行的时候，先执行的是左边部分。即return (Fib(n - 1) + Fib(n - 2));先执行的Fib(n - 1)，等到满足n <= 2条件，再逐层返回，而且递推的层数不多。

采用迭代的方式解决问题：



我们发现这次结果出来的非常快，但是结果依然是错误的，这是因为第50个斐波那契数非常大，已经超过了int类型最大值。

如何选择哪种方式来解决问题：1、如果一个问题使用递归的方式去写代码，是非常方便的、简单的，写出的代码没有明显缺陷，这个时候使用递归就可以。2、如果使用递归写的代码，存在明显缺陷：栈溢出、效率低下…就必须使用其他方式，如迭代。

递归经典问题：1、汉诺塔问题；2、青蛙跳台阶问题