

# 《计算机语言与程序设计》

## 第6周 递归

清华大学 自动化系  
范 静 涛

# 函数与变量规则：多文件多函数

## unit1.c

#include <必要的系统头文件>

#include "unit1.h"

#include 必要的其他头文件

可被其他文件访问的全局变量定义（对别的文件来说，是外部的）

static 不可被其他文件访问的全局变量

static 不可被其他文件调用的函数声明

所有函数定义

## unit1.h

#ifndef \_UNIT1\_H\_

#define \_UNIT1\_H\_

#include 必要的其文件

extern 可被其他文件访问的全局变量声明

可被其他文件调用的函数声明

#endif

# 数组作为函数参数：数组名参数

用数组名作函数实参时，地址发生了值传递，实际上形参和实参存储了相同的值（地址）。因此通过形参改变数组元素时，实参对应的数组元素也随之变化。

```
#include <stdio.h>

void sort(int arr[10]); /*函数声明*/

int main(void)
{
    //...
    int i;
    int a[10];
    sort(a);
    for (i = 0; i < 10; i++) {
        printf("%d,", a[i]);
    }
    printf("\n");
    return 0;
}
```

```
/* 函数定义*/
void sort(int arr[10])
{
    int i;
    int j;
    int temp;
    for (j = 0; j < 9; j++) {
        for (i = 0; i < 9 - j; i++) {
            if (arr[i] > arr[i + 1]) {
                temp = arr[i + 1];
                arr[i + 1] = arr[i];
                arr[i] = temp;
            }
        }
    }
}
```

**Succeed !!!**

# 数组作为函数参数：数组名参数

## 注意：

- 实参数组与形参**数组类型应一致**，如不一致，结果出错；
- 在**函数定义中声明形参数组的大小是不起任何作用的**；
- **形参数组可以不指定大小**，有时为了在被调用函数中处理数组元素个数的需要，**可以另外设一个形参**；
- 用数组名作为函数实参时，不是把数组元素的值传递给形参，**而是（数组首地址）作为实参进行数值传递，两个数组就共占同一段内存空间，在被调函数中访问（读或写）形参数组元素，是通过地址访问数值。**

# 数组作为函数参数：多维数组参数

用多维数组名作为函数实参和形参，参数声明中必须指明数组的列数

- ❓ 在被调函数中对形参数组定义时可以指定每一维的大小
- ❓ 也可以省略最高维的大小，但不能省略最高维的[]
- ❓ 在维度相同的前提下，最高维大小可以与实参数组不同

例，求3x4矩阵所有元素中的最大值。

```
int max_value(int array[][4]);
int main(void) {
    int a[3][4] = {
        {1,3,5,7},
        {2,4,6,8},
        {1, 4, 9, 16}
    };
    printf("max value is %d\n", max_value(a));
}
```

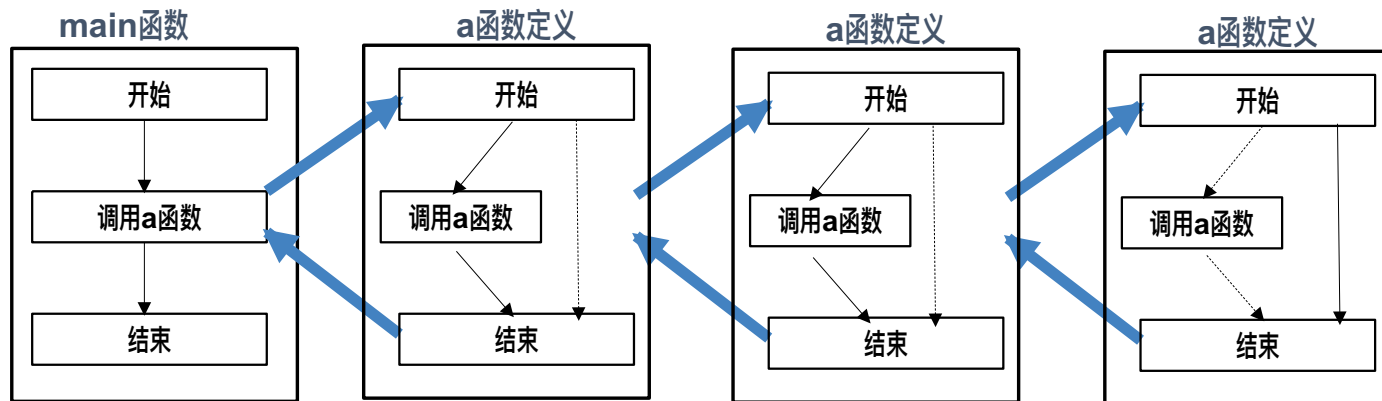
```
int max_value(int array[][4]) {
    int max = array[0][0];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            if(array[i][j] > max) {
                max= array[i][j];
            }
        }
    }
    return max;
}
```

# 本节课主要内容

## 分治与递归



# 递归函数



# 递归函数：从实例讲起

## 用递归方法求n!

### 问题分析：

假如n是5，那么5!等于5×4!，而4!=4×3!，  
...，1!=1。

可用下面的递归公式表示：

$$n! = 1 \quad (n=0, 1)$$

$$n! = n \cdot (n-1)! \quad (n > 1)$$

```
int fact(int n) {
    if(n == 1) {
        return 1;
    }
    else {
        return n * fact(n-1);
    }
}

int main(void)
{
    int n;
    scanf("%d", &n);
    printf("%d! = %d \n", n, fact(n));
    return 0;
}
```



# 递归函数：基本思想

## 找出递归子结构性质

- ❓ 原问题的解包含子问题的解
- ❓ 用子问题的解递归定义原问题的解

## 递归算法设计的关键

- ❓ 找出递归关系式，使得问题越来越简单，规模越来越小

- ❓ 找出递归终止条件

$n! = 1$

$(n=0,1)$

```
int fact(int n) {  
    if(n == 1) {  
        return 1;  
    }  
    else {  
        return n * fact(n - 1);  
    }  
}
```

```
int main(void)  
{  
    int n;  
    scanf("%d", &n);  
    printf("%d! = %d\n", n, fact(n));  
    return 0;  
}
```

# 递归函数：编程模式

```
if(最简单情形)
```

```
{
```

```
    直接得到最简单情形的解;
```

```
}
```

```
else
```

```
{
```

```
    将原始问题转化为简单一些（降低问题规模）的一个或多个子问题;
```

```
    以递归方式逐个求解上述子问题;
```

```
    以合理有效的方式将这些子问题的解组装成原始问题的解;
```

```
}
```

# 递归函数：实例2

计算Fibonacci数列



斐波那契，L.

# 递归函数：实例1

## 怎样编写这种程序？

先从最简单的情况分析起；

逐渐增加难度，归纳总结出思路。



也可以反过来，先建立一般递归关系，然后分析和确定边界

# 递归函数：实例1

```
#include <stdio.h>
long int Fibonacci(int n);
int main(void) {
    int n;
    int i;
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
        printf("%12ld", Fibonacci(i));
        if(i % 4 == 0) {
            printf("\n");
        }
    }
}
```

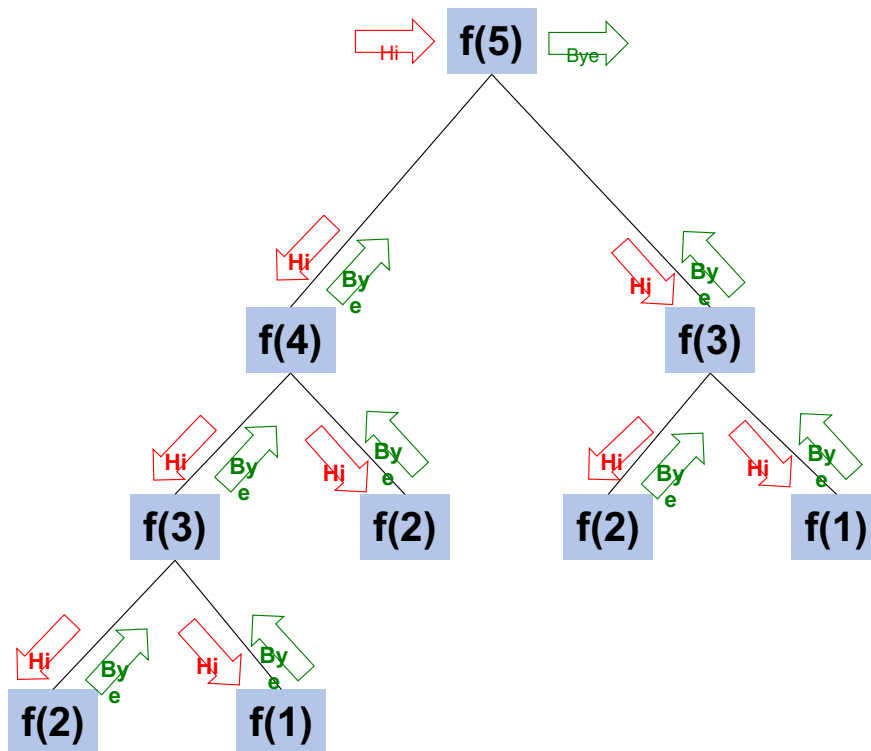
```
long int Fibonacci(int n) {
    long int f;
    if(n == 1 || n == 2) {
        f = 1;
    }
    else{
        f = Fibonacci(n-1) + Fibonacci(n-2);
    }
    return f;
}
```

**printf("f(%d) begins\n", n);**

**printf("f(%d) ends\n", n);**

# 递归函数：实例1

- ⇒ f(5) begins
- ⇒ f(4) begins
- ⇒ f(3) begins
- ⇒ f(2) begins
- ⇒ f(2) ends
- ⇒ f(1) begins
- ⇒ f(1) ends
- ⇒ f(3) ends
- ⇒ f(2) begins
- ⇒ f(2) ends
- ⇒ f(4) ends
- ⇒ f(3) begins
- ⇒ f(2) begins
- ⇒ f(2) ends
- ⇒ f(1) begins
- ⇒ f(1) ends
- ⇒ f(3) ends
- ⇒ f(5) ends



# 递归函数：实例2

给定一个整型数组 $a$ ，寻找最大元素（用递归实现）。

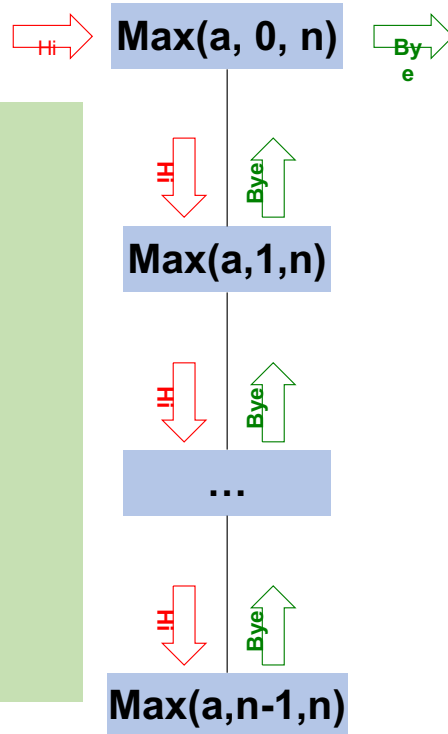
- 如何来设计相应的递归算法？
- 求解目标： $\max\{a[0], a[1], \dots, a[n-1]\}$
- 任务分解：分两组求最大值，然后比较。
- 递归关系： $\max\{a[0], \max\{a[1], \dots, a[n-1]\} \}$
- 递归边界： $\max\{x\} = x$



任务分解方式唯一么？  
如果不唯一，如何选择？

# 递归函数：实例2

```
int Max(int a[], int first, int n)
{
    int max;
    //任务分解
    if(first == n - 1) {
        return a[first];
    }
    else {
        max = Max(a, first+1, n);
        //结果合并
        return max < a[first] ? a[first] : max;
    }
}
```





# 递归函数：实例3

## 查找 (Searching

)：根据给定的某个值，在一组数据（尤其是一个数组）当中，确定有没有出现相同取值的数据元素。

❓ 顺序查找

如果该数组为有序数组，是否需要遍历所有元素？

❓ 折半查找

❓ 将目标值与数组的中间元素进行比较

若相等，查找成功；

否则根据比较的结果将查找范围缩小一半，然后重复此过程。

# 递归函数：实例3

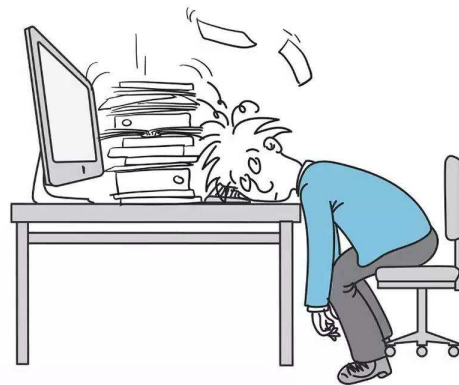
## 如何来设计相应的递归算法？

- 递归的形式？
- 递归的边界？

1个元素的数组？ 2个？ 3个？

- 函数原型：

```
int BinaySearch(int b[], int x, int L, int R);
```



# 递归函数：实例3

```
#include <stdio.h>
int search(int b[], int x, int L, int R);
int main(void)
{
    int a[] = {05, 13, 19, 21, 37, 56, 64, 75, 80, 88, 92};
    int x = 21;
    int res = search(a, x, 0, 10);
    if (res == -1) {
        printf("x does not exist!");
    }
    else {
        printf("x's position: %d\n", res);
    }
}
```

```
int search(int b[], int x, int L, int R)
{
    int mid;
    if (L > R) {
        return -1;
    }
    else {
        mid = (L + R) / 2;
        if (x == b[mid]) {
            return mid;
        }
        else if (x < b[mid]) {
            return search(b, x, L, mid-1);
        }
        else {
            return search(b, x, mid+1, R);
        }
    }
}
```

# 递归函数：实例3

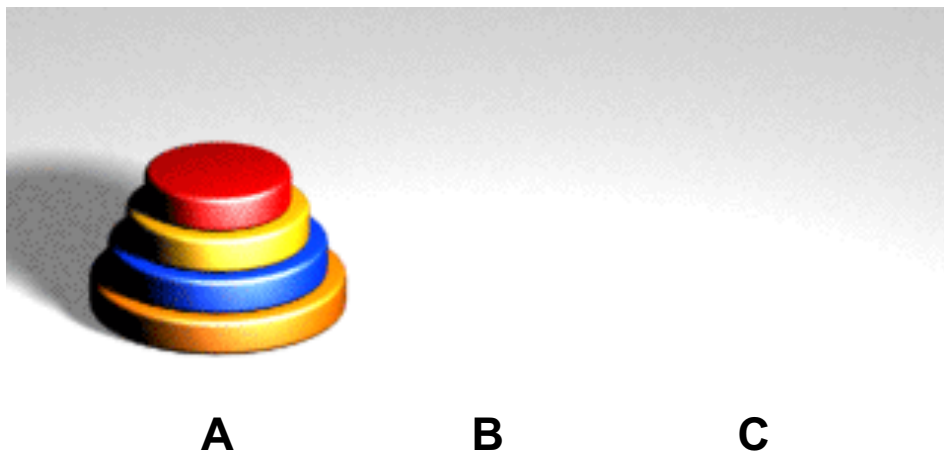
例如：4565926  
是否在此列表当中？

4120243	4120243	4120243	4120243	4120243	4120243
4276013	4276013	4276013	4276013	4276013	4276013
4328968	4328968	4328968	4328968	4328968	4328968
4397700	4397700	4397700	4397700	4397700	4397700
4462718	4462718	4462718	4462718	4462718	4462718
4466240	4466240	4466240	4466240	4466240	4466240
4475579	4475579	4475579	4475579	4475579	4475579
4478964	4478964	4478964	4478964	4478964	4478964
4480332	4480332	4480332	4480332	4480332	4480332
4494763	4494763	4494763	4494763	4494763	4494763
4499043	4499043	4499043	4499043	4499043	4499043
4508710	4508710	4508710	4508710	4508710	4508710
4549243	4549243	4549243	4549243	4549243	4549243
4563697	4563697	4563697	4563697	4563697	4563697
4564531	4564531	4564531	4564531	4564531	4564531
4565926	4565926	4565926	4565926	4565926	4565926
4566088	4566088	4566088	4566088	4566088	4566088
4572874	4572874	4572874	4572874	4572874	4572874

# 递归函数：实例4

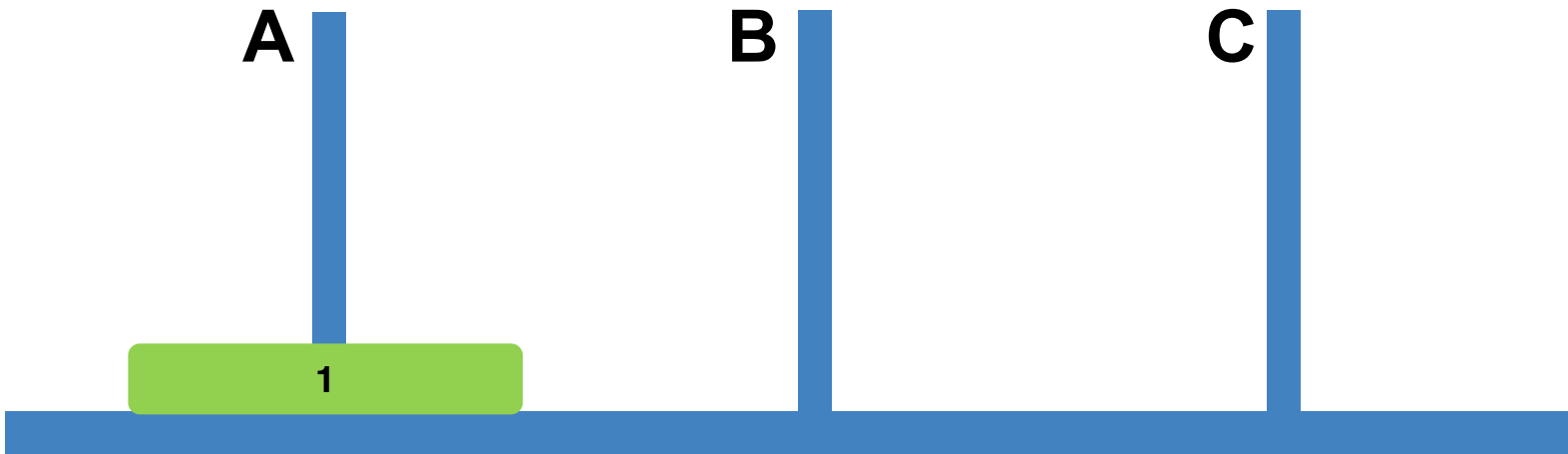
## 问题描述c: 汉诺(Hanoi)塔问题

相传在古印度Bramah庙中，有位僧人整天把三根柱子上的金盘倒来倒去，原来他是想把64个一个比一个小的金盘从一根柱子上移到另一根柱子上去。移动过程中遵守以下规则：每次只允许移动一只盘，每根柱子上都是小盘在上大盘在下（简单吗？若每秒移动一只盘子，需5800亿年）



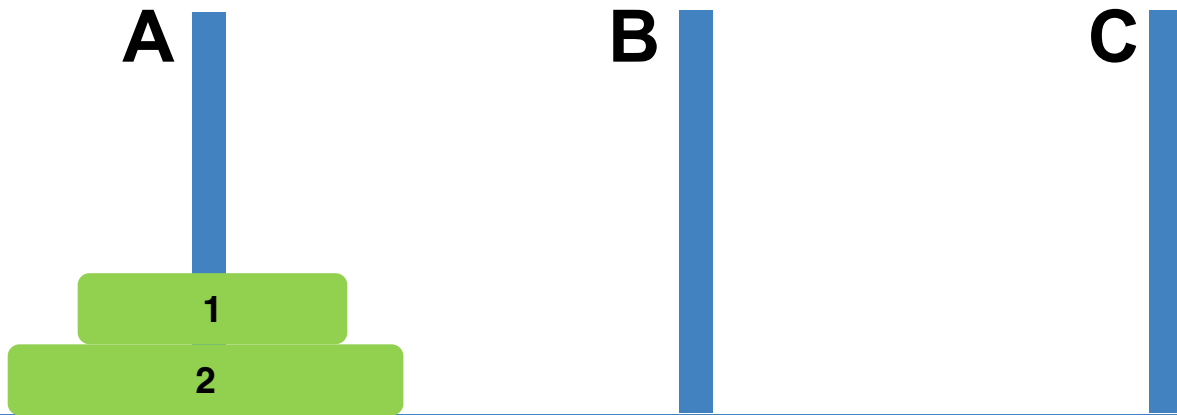
# 递归函数：实例4

在A柱上只有一只盘子，这时只需将该盘直接从A搬至C，记为A→C



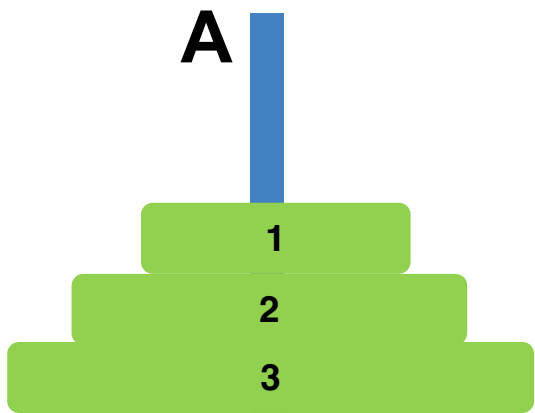
# 递归函数：实例4

若有两只圆盘，则小盘A→B，大盘直接A→C，小盘B→C；



# 递归函数：实例4

若有 $n$ 块圆盘时，可以将上面 $n-1$ 块当作“一块”， $A \rightarrow B$ ，第 $n$ 块直接 $A \rightarrow C$ ， $n-1$ 块 $B \rightarrow C$ ；





# 递归函数：实例4

**边界条件**为 $n=1$

❓ 将(1个) 盘子从“起点”移到“终点”；

**递归关系**：将 $n$ 个盘子从“起点”移到“终点”可以分解为

❓ 将 $n-1$ 个盘子从“起点”移到“临时”；

❓ 将剩余的1个盘子从“起点”移到“终点”；

❓ 将 $n-1$ 个盘子从“临时”移到“终点”；

几个盘子？从哪里移到哪里？经过中间传递的盘子？

定义Hanoi函数

函数原型： `void Hanoi(int n, char From, char Temp, char To);`

❓ 返回值： 无

❓ 函数参数：  $n$ --圆盘数

# 递归函数：实例4

```
#include <stdio.h>
void Hanoi(int n, char From, char Temp, char To);
int main(void) {
    int n;
    printf("input n:");
    scanf("%d", &n);
    Hanoi(n, 'A', 'B', 'C');
    return 0;
}

void Hanoi(int n, char From, char Temp, char To) {
    if (n==1) {
        printf("%c->%c ", From, To); //只有一个盘子直接A->C
    }
    else {
        Hanoi(n-1, From, To, Temp); //上面n-1块盘子A->B
        printf("%c->%c ", From, To); //第n块盘子直接A->C
        Hanoi(n-1, Temp, From, To); //B塔n-1块盘子B->C
    }
}
```

# 递归函数：总结

优点：

- ❑ 递归使一个蕴含递归关系且结构复杂的程序简洁精炼
- ❑ 只需要少量的步骤就可描述解题过程中所需要的多次重复计算，代码量大幅减小。

难点：

- ❑ 如何找到递归形式？**目标驱动！**
- ❑ 如何找到递归边界？**极力简化！**

缺点：

- ❑ 递归的运行效率往往很低，频繁内存调用，会消耗额外的时间和存储空间。

# 递归函数：总结

在算法分析上，要建立分治递归的思维方式。

- ❑ 分析得出递归关系式
- ❑ 确定边界条件

在编程实现上，要建立递归信心 (To trust the recursion, Jerry Cain, Stanford)。

- ❑ 系统对栈空间管理有条不紊
- ❑ 栈结构严格记录递归路径

# 递归函数：总结

原问题应该分为多少个子问题才较适宜？

每个子问题的规模应该怎样才为适当？这些问题很难予以肯定的回答。

但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。换言之，将一个问题分成大小相等的 $k$ 个子问题的处理方法是行之有效的。

# 递归函数：总结

## 吓得我抱起了

抱着抱着抱着我的小鲤鱼的我的我的我



```
#include <stdio.h>
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
int main(void) {
    printf("吓得我抱起了\n");
    Recursion(2);
}
```

# 递归函数：总结

```
#include <stdio.h>
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
int main(void) {
    printf("吓得我抱起了\n");
    Recursion(2);
}
```

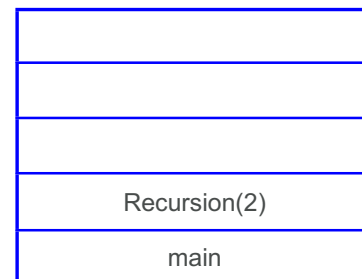
```
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
```

```
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
```

不符合实际情况，但**work**的多函数副本思考法

# 递归函数：总结


```
#include <stdio.h>
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
int main(void) {
    printf("吓得我抱起了\n");
    Recursion(2);
}
```





# 递归函数：总结

```
#include <stdio.h>
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
int main(void) {
    printf("吓得我抱起了\n");
    Recursion(2);
}
```




```
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
```

Recursion(1)
Recursion(2)
main

# 递归函数：总结

```
#include <stdio.h>
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
int main(void) {
    printf("吓得我抱起了\n");
    Recursion(2);
}
```

```
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
```



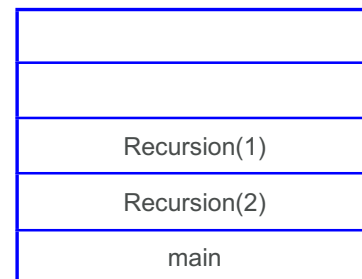
```
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
```

Recursion(0)
Recursion(1)
Recursion(2)
main

# 递归函数：总结

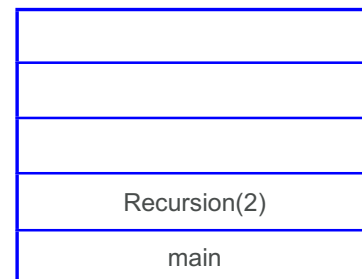

```
#include <stdio.h>
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
int main(void) {
    printf("吓得我抱起了\n");
    Recursion(2);
}
```

```
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
```



# 递归函数：总结

```
#include <stdio.h>
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
int main(void) {
    printf("吓得我抱起了\n");
    Recursion(2);
}
```



# 递归函数：总结

```
#include <stdio.h>
void Recursion(int depth) {
    printf("抱着");
    if (!depth)
        printf("我的小鲤鱼");
    else
        Recursion(depth-1);
    printf("的我");
}
int main(void) {
    printf("吓得我抱起了\n");
    Recursion(2);
}
```

