

程序设计 Programming

Lecture 4: 函数



两种代码风格

```
#include<stdio.h>
int main()
{
    int sum,a,b;
    a = 1;
    for(sum=0; a > 0; a=a)
    {
        scanf("%d", &b);
        a = b;
        if (a % 2 == 1)
            sum += a;
    }
    printf("%d", sum);
    return 0;
}
```

左括号在行首

```
#include<stdio.h>
int main()
{
    int sum, n;
    sum=0;
    do{
        scanf("%d",&n);
        if(n%2==1){
            sum=sum+n;
        }
    }while(n>0);
    printf("%d\n",sum);
    return 0;
}
```

左括号在行末

风格有瑕疵的代码示例

```
#include<stdio.h>
int main(void)
{
    int n,sum;
    sum=0;
    scanf("%d",&n);
    while (n>0)
    {
        if(n==1)
            sum=sum+1;
        else if(n%2==1)
            sum=sum+n;
        scanf("%d",&n);
    }
    printf("%d\n",sum);
    return 0;
}
```

```
#include <stdio.h>
int main(void)
{
    int a = 0;
    int sum = 0;
    scanf("%d",&a);
    while (a > 0) {
        if (a % 2) { (sum += a); }
        else sum += 0;
        scanf("%d",&a);
    }

    printf("%d", sum);

    return 0;
}
```

```
#include <stdio.h>
int main()
{
    int i, n,k,Sum=0;
    int a[1000000]={0};
    for(n=0;n<=1000000;n++){
        scanf("%d",&a[n]);
        if (a[n]<=0){
            k=n;
            break;}
    }
    for(i=0;i<=k;i++)
    {
        if(a[i]%2==1)
            Sum= Sum +a[i];
    }
    printf("%d",Sum);
    return 0;
}
```

糟糕的风格

```
#include <stdio.h>
int main()
{int n;
  n>0;
  int sum=0;
  if (n>0)
  {scanf("%d",&n);
    if (n%2!=0)
    {sum=n+sum;
    }
  }
  if (n<=0)
  {sum=sum+0;
  }

  printf("%d",sum);
  return 0;
}
```



1、函数

函数的定义

- 完成特定工作的独立模块，其一般形式为

返回值类型 函数名（形式参数表）

{

 函数过程

}

函数的定义

- 完成特定工作的独立模块，其一般形式为

返回值类型 函数名（形式参数表）

{

}

函数过程

函数返回结果
的数据类型

标识符

0个到n个形参变量
类型1 形参1, 类型2 形参2, ..., 类型n 形参n

函数的定义

- 完成特定工作的独立模块，其一般形式为

```
int func (int a, int b)
{
    int sum = a + b;
    return sum;
}
```

返回值类型 函数名 (形式参数表)

{

}

函数过程

函数返回结果
的数据类型

标识符

0个到n个形参变量
类型1 形参1, 类型2 形参2, ..., 类型n 形参n

函数的定义

- 完成特定工作的独立模块，其一般形式为

```
int func (int a, int b)
{
    int sum = a + b;
    return sum;
}
```

返回值类型 函数名 (形式参数表)

{

}

函数过程

函数返回结果
的数据类型

标识符

0个到n个形参变量
类型1 形参1, 类型2 形参2, ..., 类型n 形参n

- 形参（形式参数）必须是变量
- 函数的声明必须在函数调用之前（否则报错或warning）

函数的调用

- 一个函数定义之后，就可以在其他函数中调用它执行其功能
- 函数调用的形式

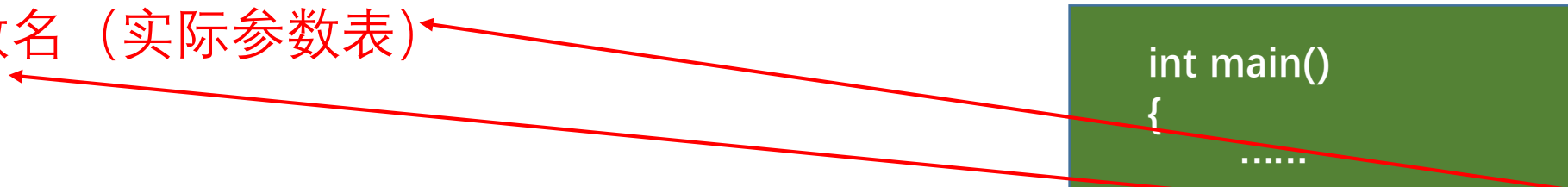
函数名（实际参数表）

函数的调用

- 一个函数定义之后，就可以在其他函数中调用它执行其功能
- 函数调用的形式

函数名（实际参数表）

```
int main()
{
    .....
    int sum = func (b, c);
    .....
    return 0;
}
```



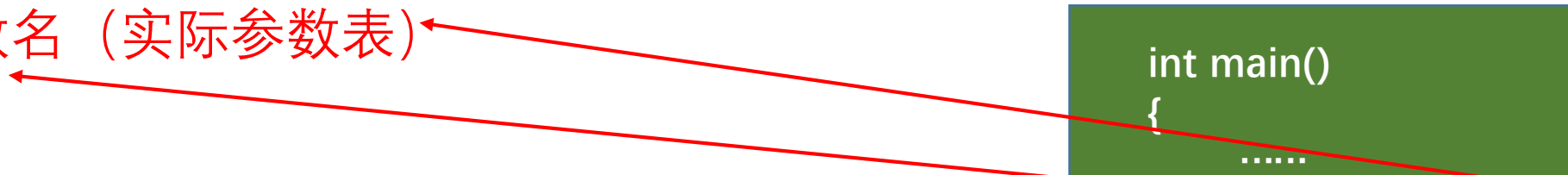
函数的调用

- 一个函数定义之后，就可以在其他函数中调用它执行其功能
- 函数调用的形式

函数名（实际参数表）

- 实际参数可以是常量、变量或者表达式

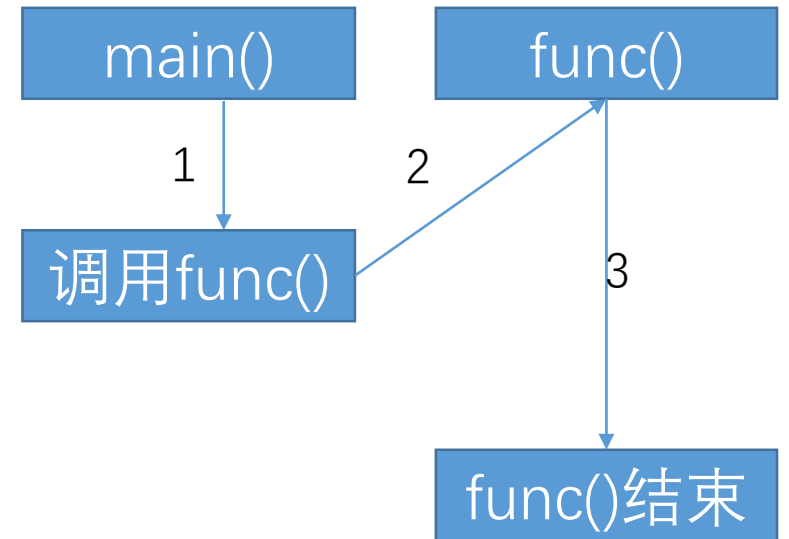
```
int main()
{
    .....
    int sum = func (b, c);
    .....
    return 0;
}
```



函数调用的过程

- 在程序执行过程中遇到函数调用时，主调函数（比如main）暂停，执行被调函数的功能（此时才给被调函数分配内存）

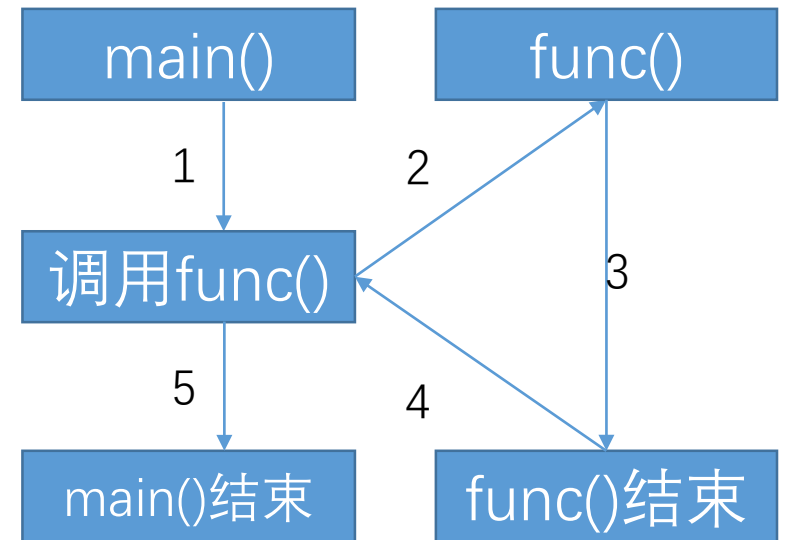
```
int main()
{
    .....
    int sum = func (b, c);
    .....
    return 0;
}
```



函数调用的过程

- 在程序执行过程中遇到函数调用时，主调函数（比如main）暂停，执行被调函数的功能（此时才给被调函数分配内存）
- 当被调函数执行完毕，主调函数从暂停点继续执行

```
int main()
{
    .....
    int sum = func (b, c);
    .....
    return 0;
}
```



函数的参数传递

- 函数定义时函数头部的参数称为形式参数（形参）
- 函数调用时接受的主调函数的参数称为实际参数（实参）

```
int func(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

形参

```
int main()
{
    int b = 1, c = 2;
    int sum = func(b, c);
    .....
    return 0;
}
```

实参

函数的参数传递

- 函数定义时函数头部的参数称为形式参数（形参）
- 函数调用时接受的主调函数的参数称为实际参数（实参）
- 形参和实参命名可以相同，数量必须相同，类型可以不同
- 函数调用时，实参的值依次赋予形参的过程叫做参数传递

int func (int a, int b)

形参

```
{  
    int sum = a + b;  
    return sum;  
}
```

int main()

```
{  
    int b = 1, c = 2;  
    int sum = func (b, c);  
    .....  
    return 0;  
}
```

实参

函数定义和调用的例子

```
1 #include <stdio.h>
2 int addition(int num1, int num2)
3 {
4     int sum;
5     sum = num1+num2;
6     return sum;
7 }
8
9 int main()
10 {
11     int var1, var2;
12     printf("Enter number 1: ");
13     scanf("%d",&var1);
14     printf("Enter number 2: ");
15     scanf("%d",&var2);
16
17     int res = addition(var1, var2);
18     printf("Output: %d", res);
19
20     return 0;
21 }
```

定义函数addition，将形参num1和num2相加并返回结果

调用函数addition，将实参var1和var2传递给形参，执行函数功能并将返回结果赋值给变量res

C语言参数传递的规则

- 只能将实参的值传递给形参，而不能将形参的值传递给实参
 - ✓ 实参的值可以赋给被调函数中的形参
 - ✓ 形参的值无法直接赋给主调函数中的实参，函数调用结束后，实参的值不会被形参改变

```
9  #include <stdio.h>
10
11 void changeA(int a)
12 {
13     a = 5;
14 }
15
16 int main()
17 {
18     int a = 3;
19     changeA(a);
20     printf("%d\n", a);
21
22     return 0;
23 }
24
```

C语言参数传递的规则

- 只能将实参的值传递给形参，而不能将形参的值传递给实参
 - ✓ 实参的值可以赋给被调函数中的形参
 - ✓ 形参的值无法直接赋给主调函数中的实参，函数调用结束后，实参的值不会被形参改变

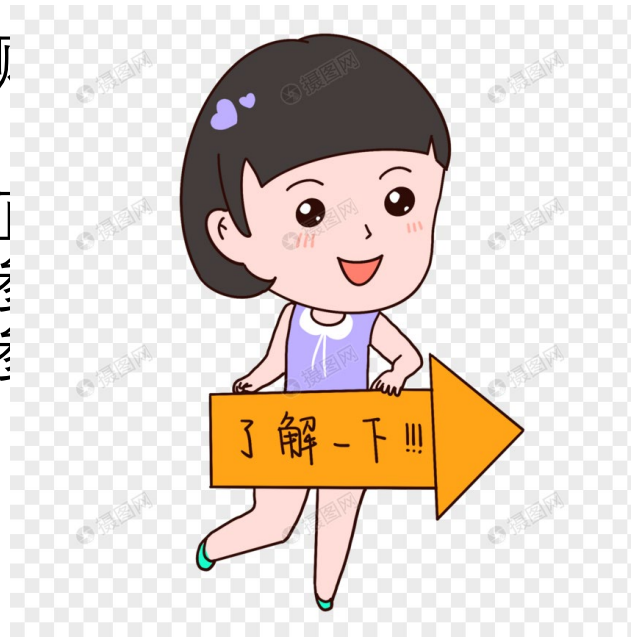
```
9  #include <stdio.h>
10
11 void changeA(int a)
12 {
13     a = 5;
14 }
15
16 int main()
17 {
18     int a = 3;
19     changeA(a);
20     printf("%d\n", a);
21 }
```

形参不能改变实参！！

C语言参数传递的规则

- 只能将实参的值传递给形参，而不能将形参的值传递给实参

- ✓ 实参的值可以影响函数中的形参
- ✓ 形参的值无法影响调用函数中的实参，调用结束后，实参不会被形参改变



```

9  #include <stdio.h>
10
11 void changeA(int *a)
12 {
13     *a = 5;
14 }
15
16 int main()
17 {
18     int a = 3;
19     changeA(&a);
20     printf("%d\n", a);
21
22     return 0;
23 }

```

参数传递示例（交换两个变量的值）

```

1  #include <stdio.h>
2  int swap(int a, int b)
3  {
4      int c;
5      c = b;
6      b = a;
7      a = c;
8      return 0;
9  }
10
11 int main()
12 {
13     int var1 = 10;
14     int var2 = 5;
15     swap(var1, var2);
16
17     printf ("var1 is: %d\n", var1);
18     printf ("var2 is: %d\n", var2);
19
20     return 0;
21 }

```

```

1  #include <stdio.h>
2  int swap(int* a, int* b)
3  {
4      int c;
5      c = *b;
6      *b = *a;
7      *a = c;
8      return 0;
9  }
10
11 int main()
12 {
13     int var1 = 10;
14     int var2 = 5;
15     swap(&var1, &var2);
16
17     printf ("var1 is: %d\n", var1);
18     printf ("var2 is: %d\n", var2);
19
20     return 0;
21 }

```

参数传递示例（交换两个变量的值）

```

1  #include <stdio.h>
2  int swap(int a, int b)
3  {
4      int c;
5      c = b;
6      b = a;
7      a = c;
8      return 0;
9  }
10
11 int main()
12 {
13     int var1 = 10;
14     int var2 = 5;
15     swap(var1, var2);
16

```

var1 is: 10
var2 is: 5

...Program finished with exit code 0
Press ENTER to exit console.

```

1  #include <stdio.h>
2  int swap(int* a, int* b)
3  {
4      int c;
5      c = *b;
6      *b = *a;
7      *a = c;
8      return 0;
9  }
10
11 int main()
12 {
13     int var1 = 10;
14     int var2 = 5;
15     swap(&var1, &var2);
16

```

var1 is: 5
var2 is: 10

...Program finished with exit code 0
Press ENTER to exit console.

先了解下，讲“指针”时详细说明

```

20     return 0;
21 }

```

```

20     return 0;
21 }

```

返回值类型

- return语句返回的数据类型尽量与函数返回值类型保持一致
✓ 否则可能会。。。
- 返回值类型为void的函数可以没有return语句

```
3 void printA(int a)
4 {
5     if (a == 1)
6         printf("a == 1");
7     else
8         printf("a != 1");
9 }
```

```
3 int printA(int a)
4 {
5     if (a == 1)
6         printf("a == 1");
7     else
8         printf("a != 1");
9     return 0;
10 }
```


返回值类型

- return语句返回的数据类型尽量与函数返回值类型保持一致
 - ✓ 否则可能会。。。
- 返回值类型为void的函数可以没有return语句
- C语言中return语句只能返回一个值
 - ✓ 返回多个值可以使用指针、结构体等（之后会讲）

```

3 void printA(int a)
4 {
5     if (a == 1)
6         printf("a == 1");
7     else
8         printf("a != 1");
9 }

```

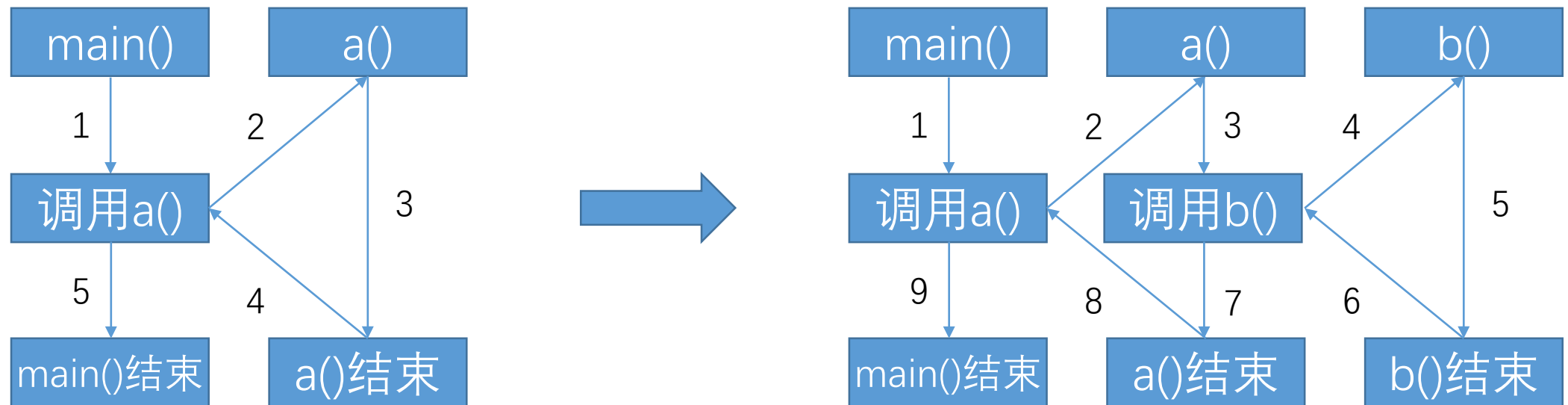
```

3 int printA(int a)
4 {
5     if (a == 1)
6         printf("a == 1");
7     else
8         printf("a != 1");
9     return 0;
10 }

```


函数的嵌套调用

- 被调函数中可以继续调用其他函数



函数嵌套调用示例

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int quad_func(int x)
5  {
6      int y = pow(x, 2);
7      return y+x;
8  }
9
10 int main()
11 {
12     int x = 3;
13     printf("%d\n", quad_func(x));
14     return 0;
15 }
16
```

2、递归函数

递归函数

- 自我调用的函数称为递归函数 (Recursive Function)
 - ✓ 是**分治算法** (divide-and-conquer) 的一种代码体现
 - ✓ 先将大问题分解为小问题，用**相同的方法**分别解决这些小问题，然后再将小问题的结果逐层返回给大问题

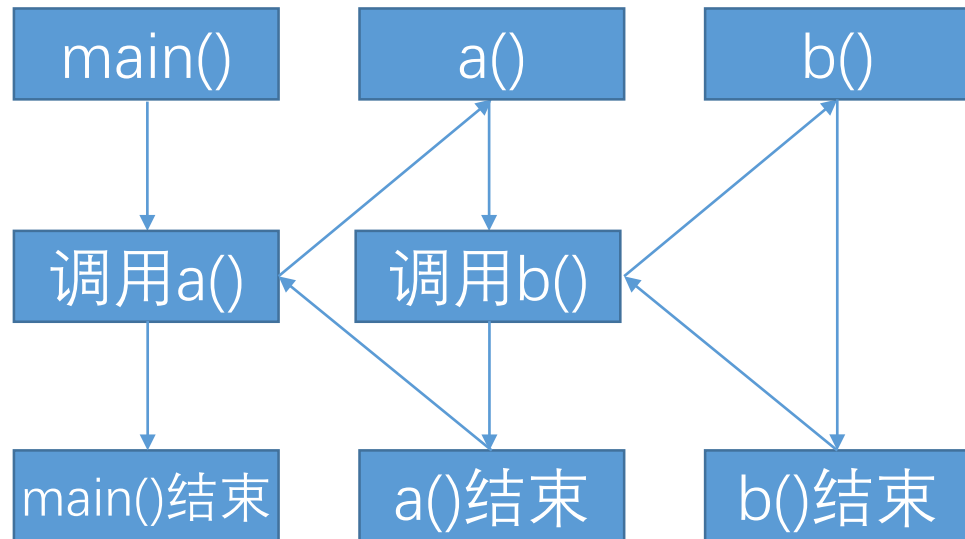


递归函数

- 自我调用的函数称为递归函数 (Recursive Function)
 - ✓ 是分治算法 (divide-and-conquer) 的一种代码体现
 - ✓ 先将大问题分解为小问题，用相同的方法分别解决这些小问题，然后再将小问题的结果逐层返回给大问题
- 在解决复杂问题时，代码实现会非常简洁高效
- 缺点：运行效率相对较低
 - ✓ 大量的函数调用
 - ✓ 可能有大量重复计算过程

递归函数：自我嵌套调用

普通嵌套调用

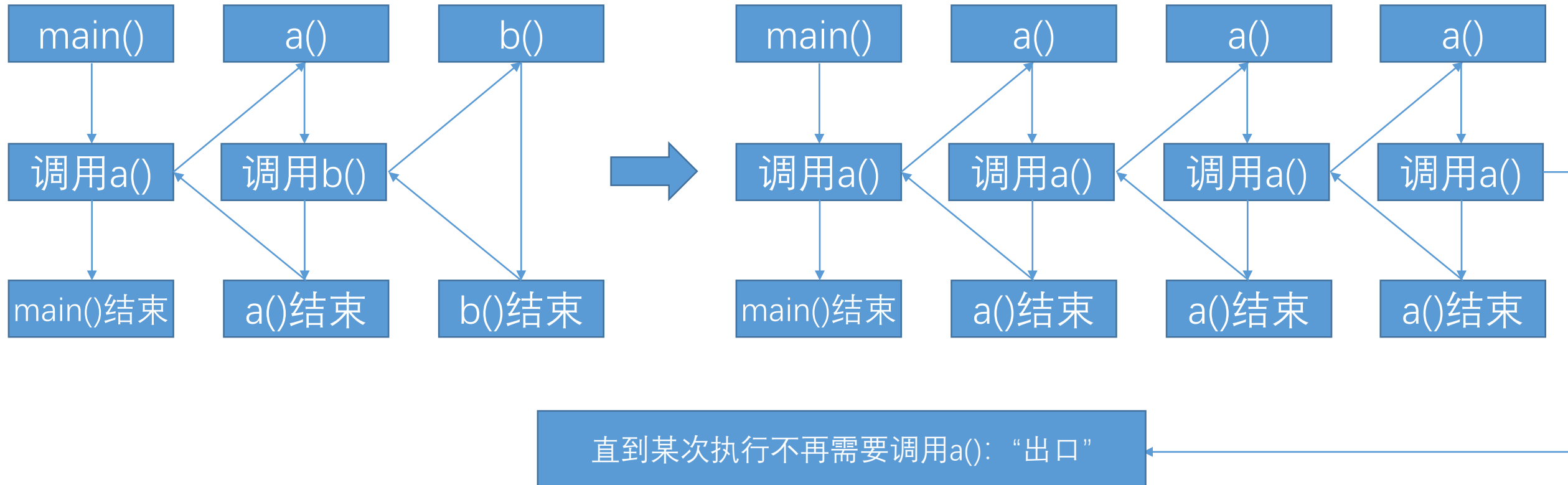


递归函数：自我嵌套调用

普通嵌套调用

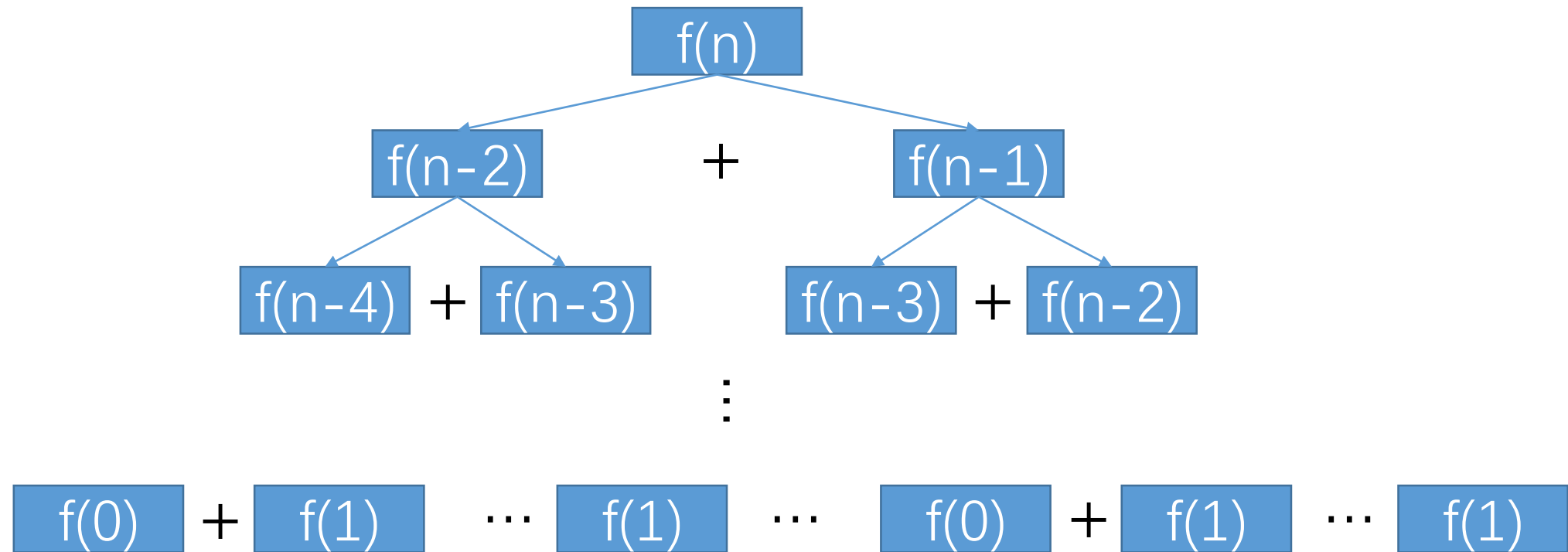
递归嵌套调用

问题规模逐渐变小



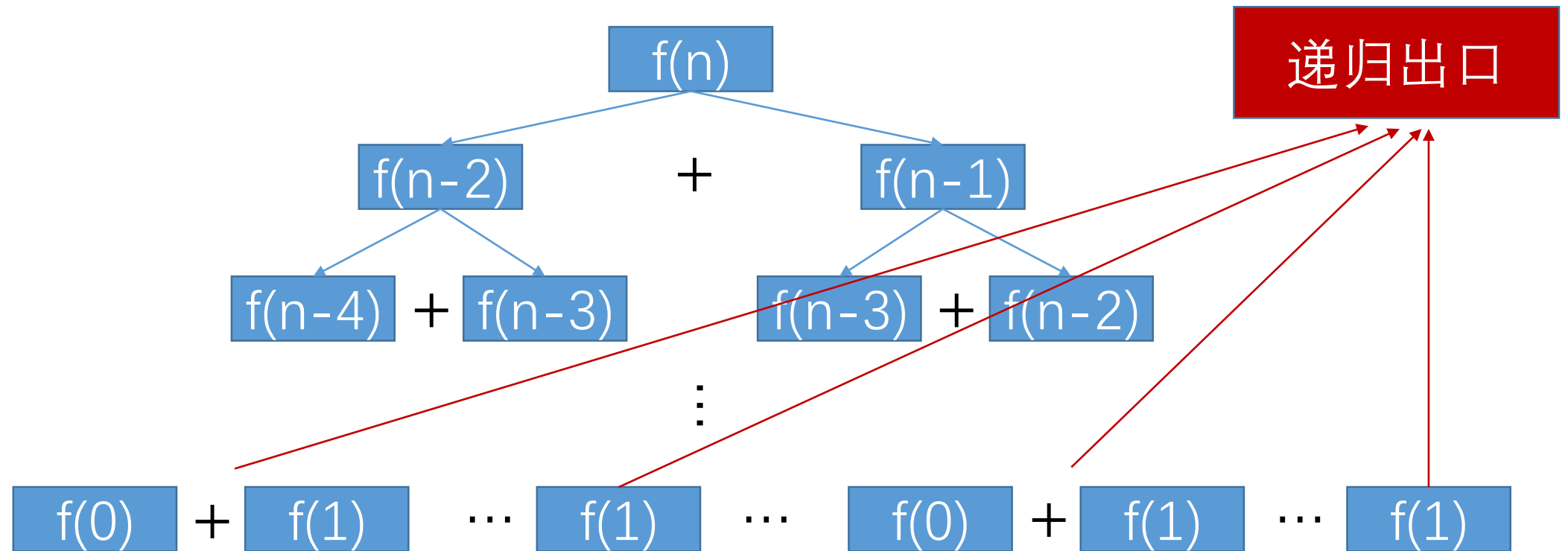
一个简单的递归函数：斐波那契数列

- 又称黄金分割数列：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- $f(0)=0$, $f(1)=1$, $f(n)=f(n-2)+f(n-1)$, for $n \geq 2$



一个简单的递归函数：斐波那契数列

- 又称黄金分割数列：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- $f(0)=0$, $f(1)=1$, $f(n)=f(n-2)+f(n-1)$, for $n \geq 2$



递归函数：斐波那契数列

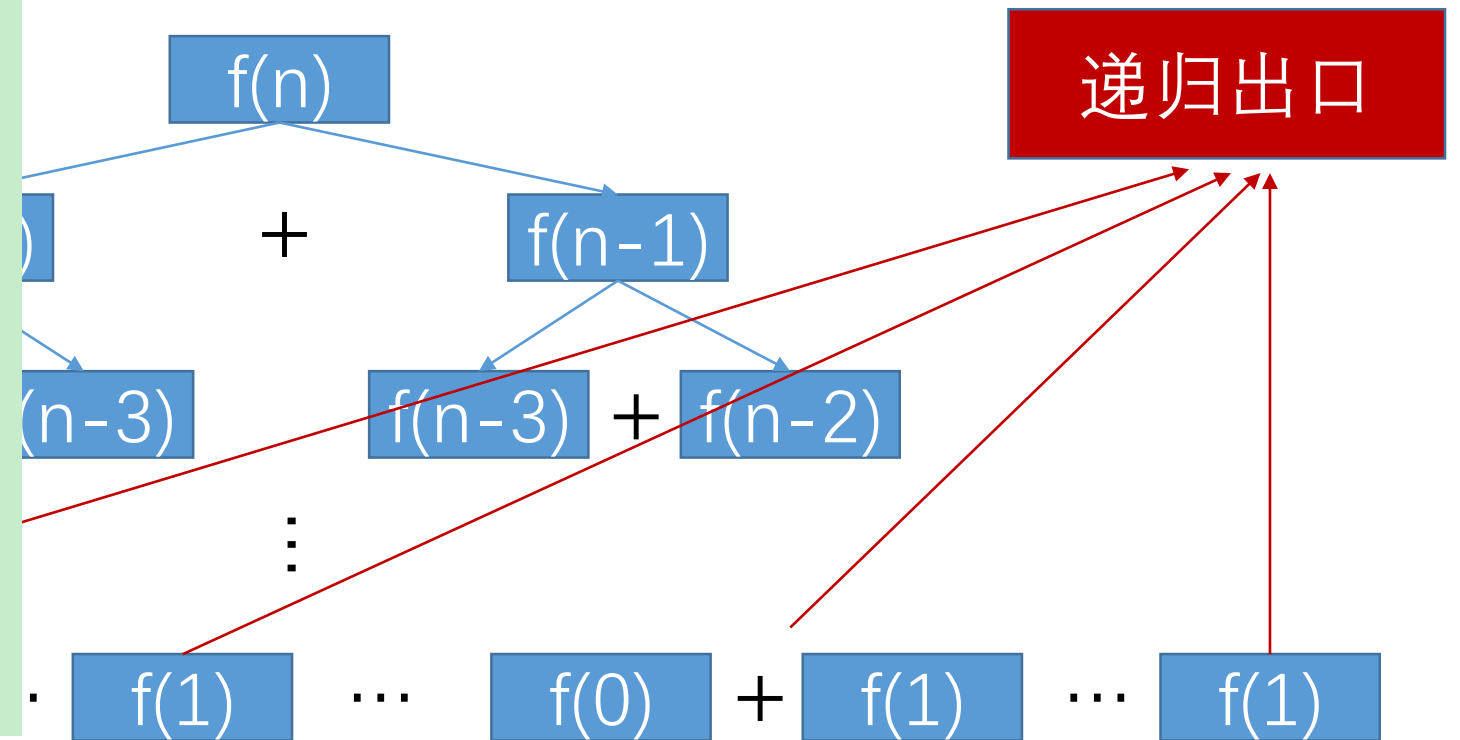
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

$f(n) = f(n-2) + f(n-1)$, for $n \geq 2$

```

1  #include<stdio.h>
2
3  int fib(int n)
4  {
5      if (n == 0)
6          return 0;
7      if (n == 1)
8          return 1;
9      return fib(n-2) + fib(n-1);
10 }
11
12 int main()
13 {
14     printf("\n");
15     for(int i = 0; i < 10; i++)
16     {
17         printf("%d ", fib(i));
18     }
19     printf("\n\n");
20
21     return 0;
22 }

```

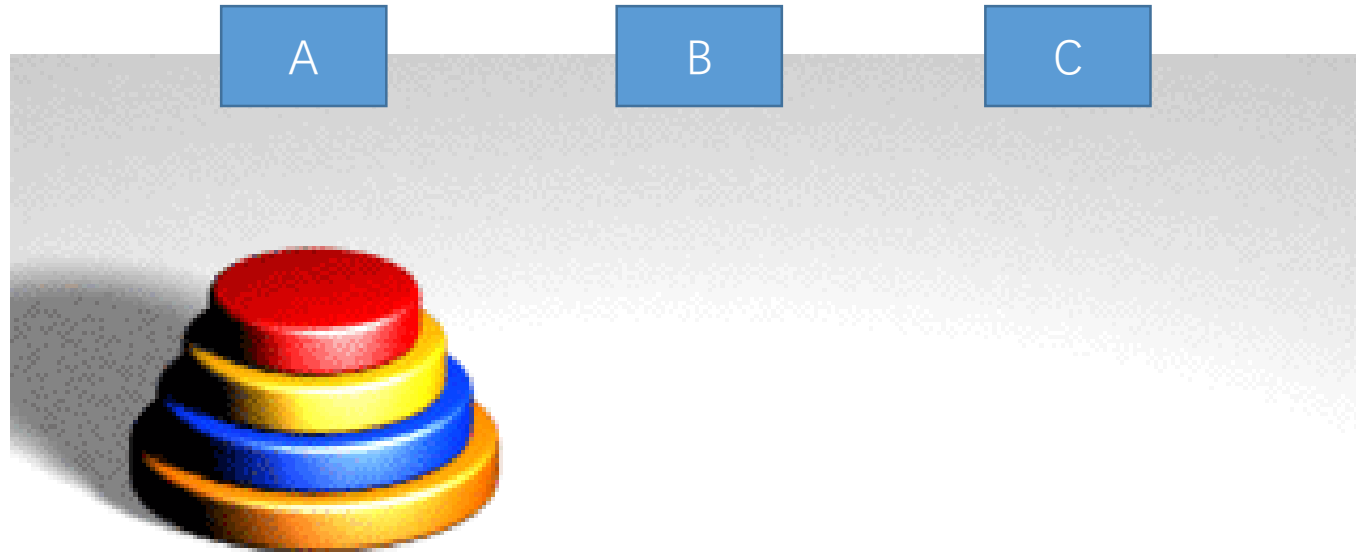


递归函数可以解决的经典问题

- 汉诺塔 (Tower of Hanoi)
 - ✓ 将 n 个盘子从A搬到C
 - ✓ 一次只能搬运一个，放在另外两个地点其中之一
 - ✓ 大盘永远只能在小盘下面

递归函数可以解决的经典问题

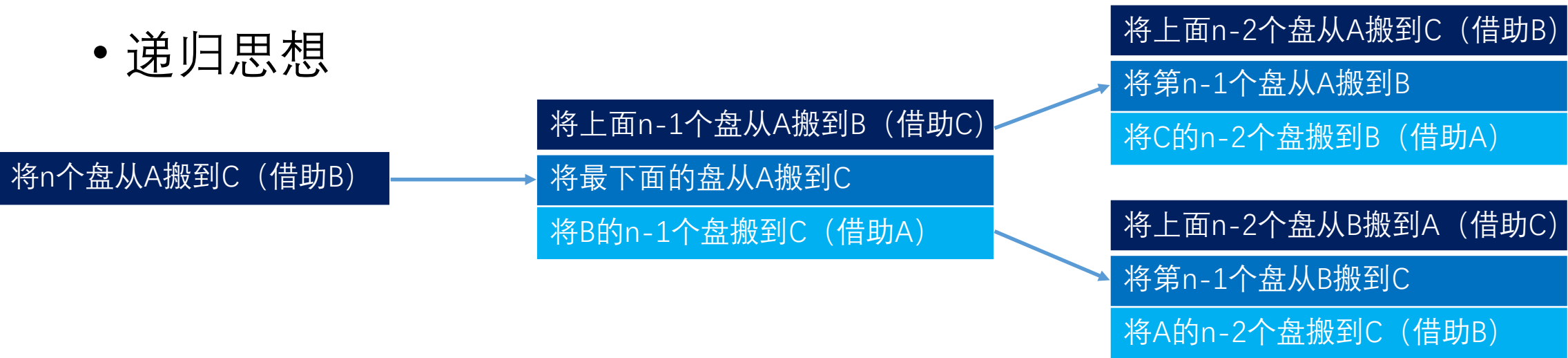
- 汉诺塔 (Tower of Hanoi)
 - ✓ 将 n 个盘子从A搬到C
 - ✓ 一次只能搬运一个，放在另外两个地点其中之一
 - ✓ 大盘永远只能在小盘下面



递归函数可以解决的经典问题

- 汉诺塔 (Tower of Hanoi)
 - ✓ 将 n 个盘子从A搬到C
 - ✓ 一次只能搬运一个，放在另外两个地点其中之一
 - ✓ 大盘永远只能在小盘下面

递归思想



递归函数可以解决的经典问题

- 汉诺塔 (Tower of Hanoi)

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

```
1  #include <stdio.h>
2
3  //Move all disks from A to C via B
4  void towerOfHanoi(int n, char A, char C, char B)
5  {
6      if (n == 1)
7      {
8          printf("\n Move disk 1 from %c to %c", A, C);
9          return;
10     }
11     towerOfHanoi(n-1, A, B, C);
12     printf("\n Move disk %d from %c to %c", n, A, C);
13     towerOfHanoi(n-1, B, C, A);
14 }
15
16 int main()
17 {
18     int n = 3; // Number of disks
19     towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
20     return 0;
21 }
```

其他经典问题

- 快速排序 (quicksort)
- 二分查找 (binary search)
- 最长相同子序列 (longest common subsequences, LCS)
- 八皇后问题 (eight queens puzzle)

<https://www.geeksforgeeks.org/recursion-practice-problems-solutions/>

有兴趣的同学可以深入研究

3、局部变量和全局变量

局部变量和全局变量

- 之前所遇到的变量都是局部变量
 - ✓ 变量定义在函数中或者控制语句中
 - ✓ 作用域分别限在函数内或者控制语句内
 - ✓ 保证了变量间的独立性

```
int income (int bonus, int allowance)
{
    int total =      bonus + allowance;
    return total;
}

int main()
{
    int bonus = 20000, allowance = 1000;

    int total = income (bonus, allowance);

    return 0;
}
```

局部变量和全局变量

- 之前所遇到的变量都是局部变量
 - ✓ 变量定义在函数中或者控制语句中
 - ✓ 作用域分别限在函数内或者控制语句内
 - ✓ 保证了变量间的独立性
- 定义在函数外的变量称为全局变量, 一般写在程序开头 (代码中的base)
 - ✓ 作用域从定义开始到程序结束
 - ✓ 可以被作用域内的函数共用

```
int base = 10000;

int income (int bonus, int allowance)
{
    int total = base + bonus + allowance;
    return total;
}

int main()
{
    int bonus = 20000, allowance = 1000;
    base = base * 2;
    int total = income (bonus, allowance);

    return 0;
}
```

变量的生命周期

- 指变量从被分配内存到内存被回收的过程
 - ✓ 局部变量：在函数被调用时分配内存，在调用结束时内存被收回

```
int base = 10000;

int income(int bonus, int allowance)
{
    int total = base + bonus + allowance;
    return total;
}

int main()
{
    int bonus = 20000, allowance = 1000;
    base = base * 2;
    int total = income(bonus, allowance);

    return 0;
}
```

1. 调用income时分配内存
2. 调用income结束，内存被收回

变量的生命周期

- 指变量从被分配内存到内存被回收的过程
 - ✓ 局部变量：在函数被调用时分配内存，在调用结束时内存被收回
 - 思考：1) 形参不能改变实参的原因！

```
int base = 10000;

int income(int bonus, int allowance)
{
    int total = base + bonus + allowance;
    return total;
}

int main()
{
    int bonus = 20000, allowance = 1000;
    base = base * 2;
    int total = income(bonus, allowance);

    return 0;
}
```

1. 调用income时分配内存
2. 调用income结束，内存被收回

变量的生命周期

- 指变量从被分配内存到内存被回收的过程
 - ✓ 局部变量：在函数被调用时分配内存，在调用结束时内存被收回
 - 思考：1) 形参不能改变实参的原因！
2) main函数中的变量？

```
int base = 10000;
```

```
int income (int bonus, int allowance)  
{  
    int total = base + bonus + allowance;  
    return total;  
}
```

1. (系统)调用main时分配内存
2. 调用main结束，内存被收回

```
int main()  
{  
    int bonus = 20000, allowance = 1000;  
    base = base * 2;  
    int total = income (bonus, allowance);  
  
    return 0;  
}
```



变量的生命周期

- 指变量从被分配内存到内存被回收的过程
 - ✓ 局部变量：在函数被调用时分配内存，在调用结束时内存被收回
 - 思考：1) 形参不能改变实参的原因！
2) main函数中的变量？
 - ✓ 全局变量：内存分配从程序开始到结束！

```
int base = 10000;
```

```
int income (int bonus, int allowance)  
{  
    int total = base + bonus + allowance;  
    return total;  
}
```

```
int main()  
{  
    int bonus = 20000, allowance = 1000;  
    base = base * 2;  
    int total = income (bonus, allowance);  
  
    return 0;  
}
```



变量的生命周期 Vs 作用域

- 生命周期和作用域是不同概念
 - ✓ 生命周期是变量在内存中的时期
 - ✓ 作用域是变量起作用的范围
 - ✓ 举例：main函数中for循环的局部变量的生命周期从系统调用main到main执行结束，但是不能在for循环外中起作用

4、静态变量

静态变量

- 使用static声明的变量，存储在静态存储区中（下次课会提到）

```
static int a;
```

```
int income (int bonus, int allowance)
{
    static int base = 10000;
    int total = base + bonus + allowance;
    return total;
}

int main()
{
    int bonus = 20000, allowance = 1000;
    int total = income (bonus, allowance);

    return 0;
}
```

静态变量

- 使用static声明的变量，存储在静态存储区中（下次课会提到）

```
static int a;
```

- 一旦分配内存，就会被保存直至程序结束
 - ✓主函数中从被声明开始被保存，直至程序结束
 - ✓普通函数中从该函数被调用开始被保存，直至程序结束

```
int income (int bonus, int allowance)
{
    static int base = 10000;
    int total = base + bonus + allowance;
    return total;
}

int main()
{
    int bonus = 20000, allowance = 1000;
    int total = income (bonus, allowance);

    return 0;
}
```

静态局部变量（掌握）

- 静态局部变量的初始化赋值 **只执行一次**，即包含静态变量的函数第一次被调用时

```
int income (int bonus, int allowance)
{
    static int base = 10000;
    base = base * 2;
    int total = base + bonus + allowance;
    return total;
}
```

第一次调用income
时，初始化base

```
int main()
{
    int bonus = 20000, allowance = 1000;
    int year1 = income (bonus, allowance);
    int year2 = income (bonus, allowance);
    int year3 = income (bonus, allowance);

    return 0;
}
```

静态局部变量（掌握）

- 静态局部变量的初始化赋值 **只执行一次**，即包含静态变量的函数第一次被调用时
- 后续调用时，跳过base的初始化，直接使用内存中保存的base的值

```
int income (int bonus, int allowance)
{
    static int base = 10000;
    base = base * 2;
    int total = base + bonus + allowance;
    return total;
}
```

第一次调用income
时，初始化base

```
int main()
{
    int bonus = 20000, allowance = 1000;
    int year1 = income (bonus, allowance);
    int year2 = income (bonus, allowance);
    int year3 = income (bonus, allowance);

    return 0;
}
```

静态全局变量（了解）

- 生命周期和普通全局变量相同
- 不同点
 - ✓ 静态全局变量只在定义该变量的源文件中可以使用
 - ✓ 普通全局变量在同一个工程的所有源文件中都可以使用

Next

- 数组