递归

Wang Houfeng
CCES, PKU
wanghf@pku.edu.cn

内容

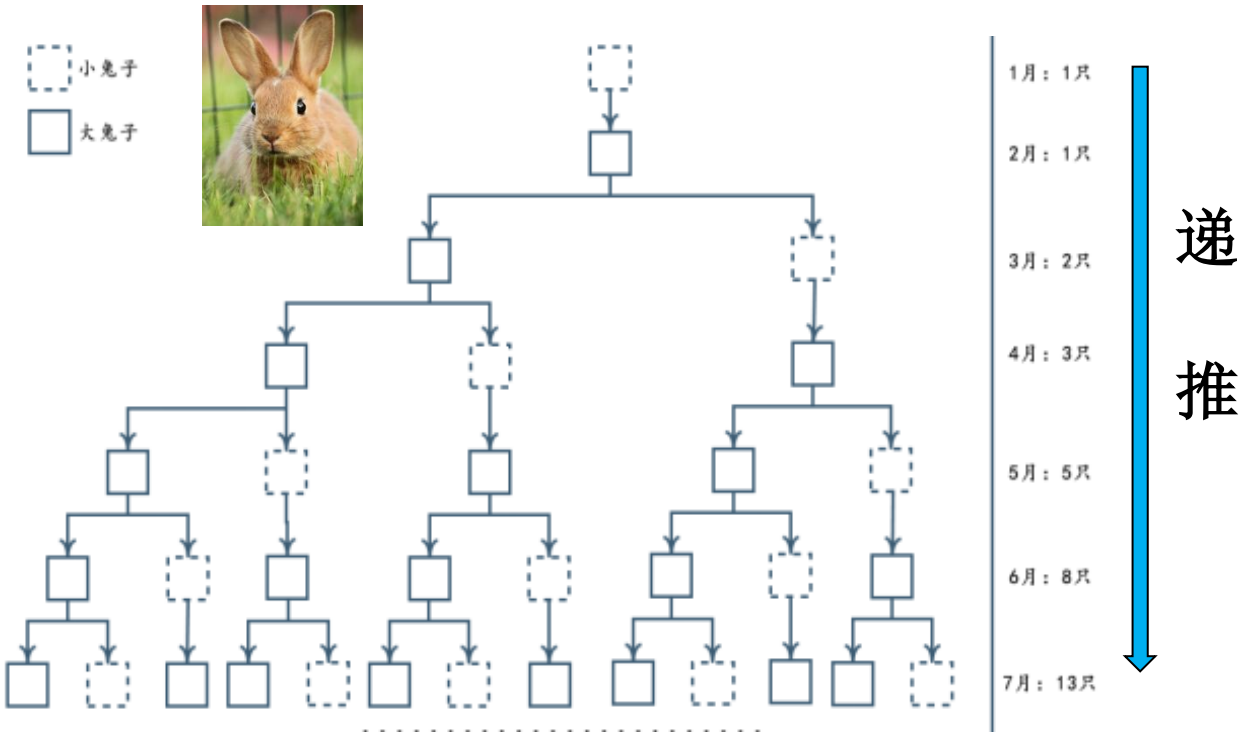
- **递推**
- 递归
- 递归与递推

递推：一种重要程序设计方法

- 递推是程序设计的一类重要算法，可以将复杂的运算化为若干重复的简单运算，充分发挥计算机长于重复处理的特点
- 递推算法是通过已知条件，利用特定关系得出中间推论，直至得到结果的算法
- 递推也称为迭代法
  - **顺推：从已知条件出发，逐步推算出要解决的问题的方法**
  - **逆推：从已知问题的结果出发，用迭代表达式逐步推算出问题的开始的条件**

递推例1

- **兔子（斐波那契）问题**：一对刚出生的小兔一个月后就能长成大兔，再过一个月就能生下一对小兔，并且此后每个月都生一对小兔，一年内没有发生死亡，那么一对刚出生的兔子，在一年内繁殖成多少对兔子?



$f(0)=0$

$f(1)=1$

$f(2)=f(0)+f(1)=1$

$f(3)=f(1)+f(2)=2$

$f(4)=f(2)+f(3)=3$

$f(5)=f(3)+f(4)=5$

$f(6)=f(4)+f(5)=8$

$f(7)=f(5)+f(6)=13$

数列生成的一般形式

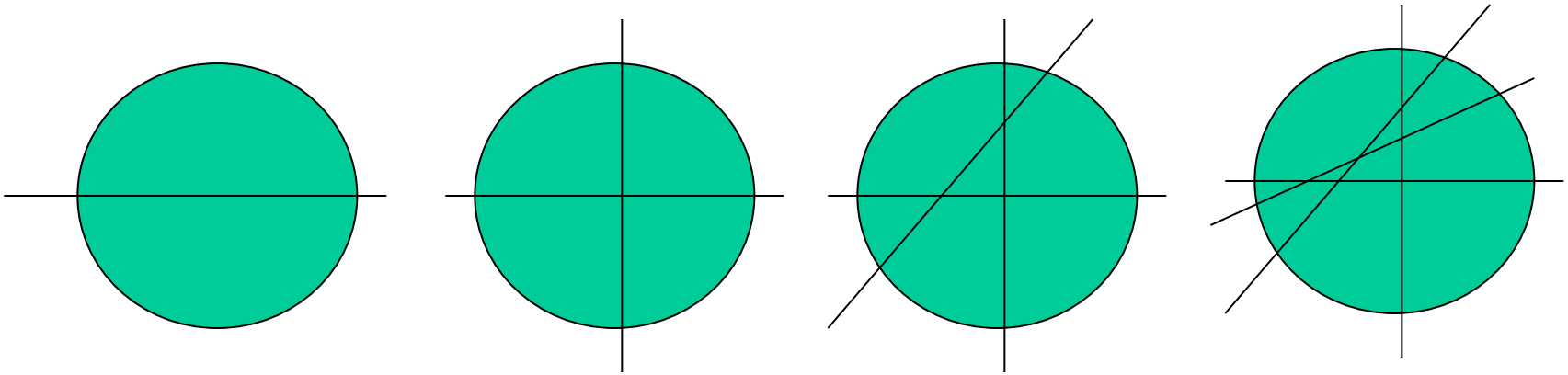
- 斐波那契数列：1,1,2,3,5,8,13,21,….
  - fab(n)= fab(n-1)+fab(n-2)（通项公式）;
  - fab(1)=1,fab(2)=1;（初始条件）
- 一个数列从某一项起，它的任何一项都可以用它前面的若干项来确定，这样的数列称为递推数列：
  - 1!,2!,3!,…n!
  - fact(n) = n * fact(n-1)（通项公式）;
  - fact(1) = 1 （初始条件）

斐波那契数列的实现

```
#include <iostream>
using namespace std;
int main()
{
    int fib[40] = {0,1};
    int i;
    for(i=2;i<40;i++)
    {
        fib[i ] = fib[i-1]+fib[i-2];
    }
    for(i=0;i<40;i++)
    {
        cout<<"Fib"<<i<<"="<< fib[i]<<endl;
    }
    return 0;
}
```

递推例2

- **切饼，100刀最多能切多少块？**



- q(0)=1;
- q(1)=1+1=2;
- q(2)=1+1+2=4;
- q(3)=1+1+2+3=7;
- q(4)=1+1+2+3+4=11;
- q(n)=q(n−1)+n;

实现

```
#include <iostream.h>
using namespace std;
int main()
{   int q[101];
    q[0] = 1;
    for (int i = 1; i <= 100; i++)
    {  q[i] = q[i-1] + i;}
    cout<<"100刀最多可切"<<q[100]<<"块"<<endl;
    return 0;
}
```

递推例3：分鱼

- A、B、C、D、E 五人合伙夜间捕鱼，凌晨时都疲惫不堪，各自在湖边的树丛中找地方睡着了。日上三竿，A第一个醒来，他将鱼平分作五份，结果多一条，于是把多余的一条扔回湖中，自己从五份中取走一份。B第二个醒来，也将剩余的鱼平分为五份，同样多一条，也扔掉了多余的一条，从五份中拿走了自己的一份。接着 C、D、E 依次醒来，与A、B按相同方式处理，即，将剩余的鱼平分为五份，同样多一条扔掉了，从五份中拿走了自己的一份。 问五人至少合伙捕到多少条鱼？每个人醒来后看到的鱼数是多少条？

分鱼：解题思路

假定A、B、C、D、E 五人的编号分别为1、2、3、4、5，整数数组 fish[k] 表示第 k 个人所看到的鱼数。fish[1] 表示A所看到的鱼数，fish[2] 表示 B 所看到的鱼数……

fish[1]= A所看到的鱼数，合伙捕到鱼的总数

fish[2]=(fish[1]-1)*4/5　B所看到的鱼数

fish[3]=(fish[2]-1)*4/5　C所看到的鱼数

fish[4]=(fish[3]-1)*4/5　D所看到的鱼数

fish[5]=(fish[4]-1)*4/5　E所看到的鱼数

分鱼：问题一般形式

$$fish[\,i\,] = (\,fish[\,i\,-1\,] - 1\,) * 4 / 5$$

$$i = 2, 3, \ldots, 5$$

上述公式可用于知道A 看到的鱼数去推算 B 看到的，再推算 C 看到的，…….。现在要求的是 A 看到的。现在需要先知 E 看到的，再反推 D 看到的，……，直到A看到的。为此，可将上式改写为：

$$fish[\,i-1\,] = fish[\,i\,] * 5 / 4 + 1$$

$$i = 5, 4, \ldots, 2$$

分鱼：进一步分析

1. 当 i = 5 时，fish[5] 表示 E 醒来所看到的鱼数，该数应满足被 5 整除后余 1 ，即

fish[5] % 5 == 1

2. 当 i = 5 时，fish[i-1] 表示 D 醒来所看到的鱼数，这个数既要满足

fish[4] = fish[5] * 5 / 4 + 1

又要满足

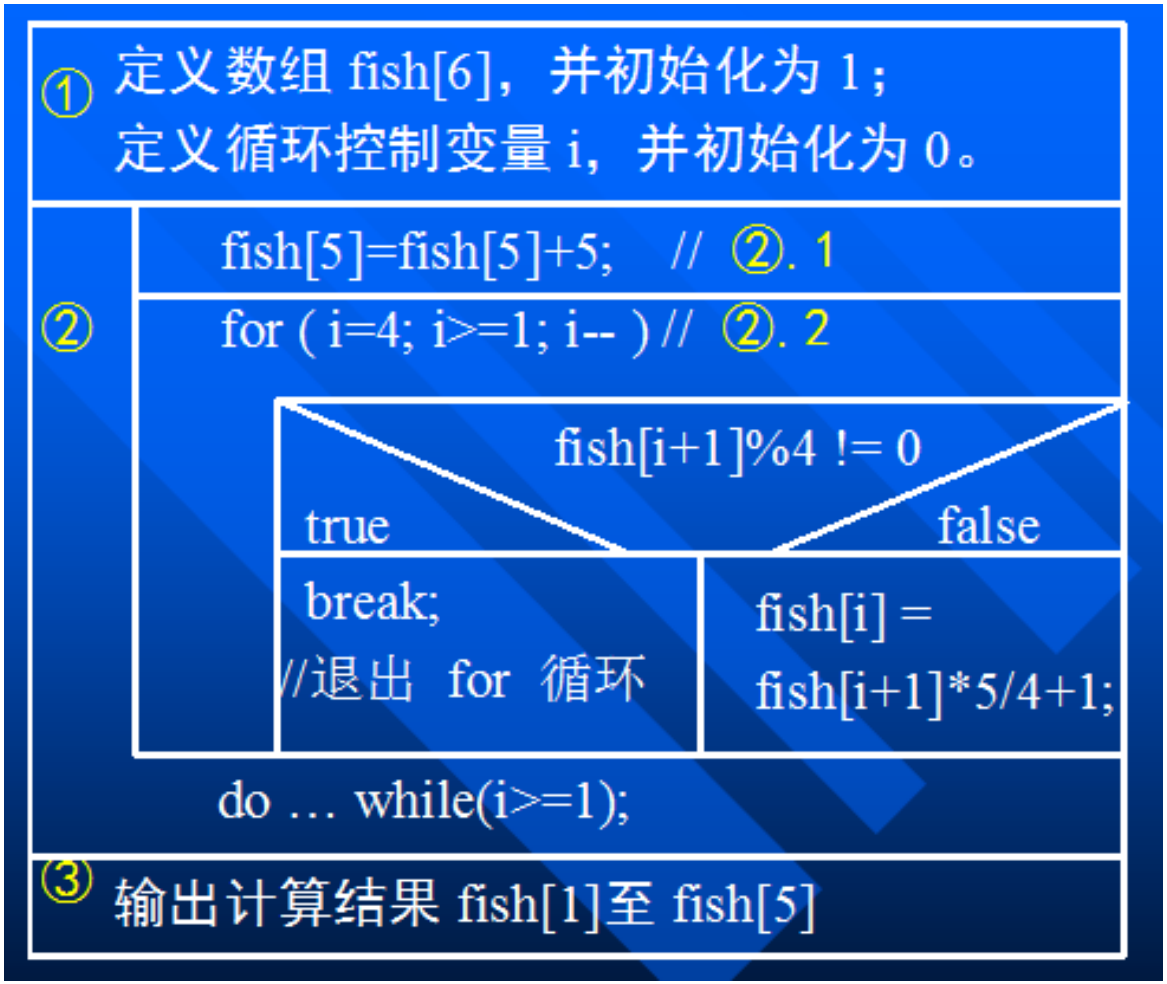
fish[4] % 5 == 1

显然，fish[4] 不能不是整数，这个结论同样可以用至 fish[3], fish[2] 和 fish[1]

分鱼：进一步分析(续)

3 . 按题意要求 5 人合伙捕到的最少鱼数，可以从小往大枚举，可以先让 E 所看到的鱼数最少为 6 条，即 fish[5] 初始化为 6 来试，之后每次增加 5 再试，直至递推到 fish[1] 得整数且除以 5 之后的余数为 1。

NS图表示算法



是前面留下的4份！

三部分的解释

(1) 是说明部分：包含定义数组 fish[6]，并初始化为 1 和定义循环控制变量 i，并初始化为 0。

(2) 是 do….while 循环，其中又含两块：

(2).1 是枚举过程中的 fish[5] 的初值设置，一开始 fish[5]=1+5; 以后每次增 5。

(2).2 是一个 for 循环，i的初值为 4，终值为 1，步长为 -1，该循环的循环体是一个分支语句，如果 fish[i+1]不能被 4 整除，则跳出 for 循环（使用 break 语句;）否则，从 fish[i+1] 算出fish[i]。(前面那个人留下的4份)

当由 break 语句让程序退出循环时，意味着某人看到的鱼数不是整数，当然不是所求，必须令fish[5] 加 5 后再试，即重新进入直到型循环 do while 的循环体。

当着正常退出 for 循环时，一定是控制变量 i 从初值 4，一步一步执行到终值 1，每一步的鱼数均为整数，最后 i = 0，表示计算完毕，且也达到了退出直到型循环的条件。

(3) 输出结果

```
#include <iostream>              // 预编译命令
using namespace std;
int  main()                      //主函数
{
   int fish[6]={1,1,1,1,1,1}; // 整型数组，记录每人醒来后看到的鱼数
   int i=0;
   do
   {
      fish[5]=fish[5]+5;         // 让E看到的鱼数增5。
      for (i=4; i>=1; i--)
      {
           if ( fish[i+1]%4 !=0 )
                    break;       // 跳出for循环
           else
                    fish[i]=fish[i+1]*5/4+1;// 计算第i人看到的鱼数
      }
   } while( i>=1 ); // 当 i>=1 继续做do循环
   // 输出计算结果
   for (i=1; i<=5; i++)
           cout << fish[i] << endl;
   return 0;
}
```

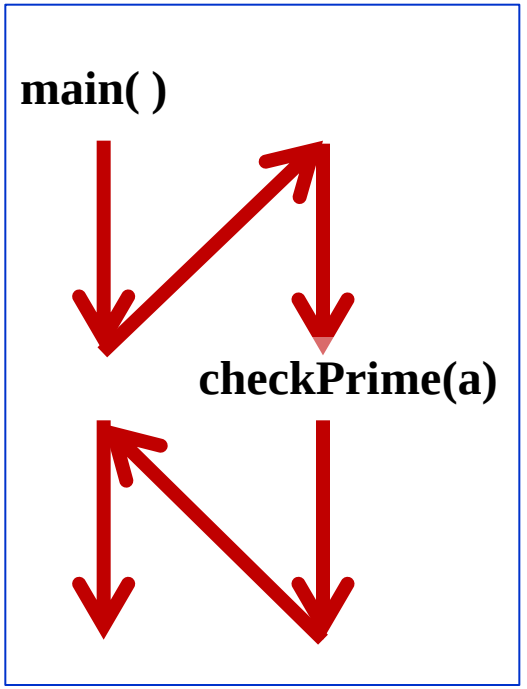
```
3121
2496
1996
1596
1276
请按任意键继续. . .
```

内容

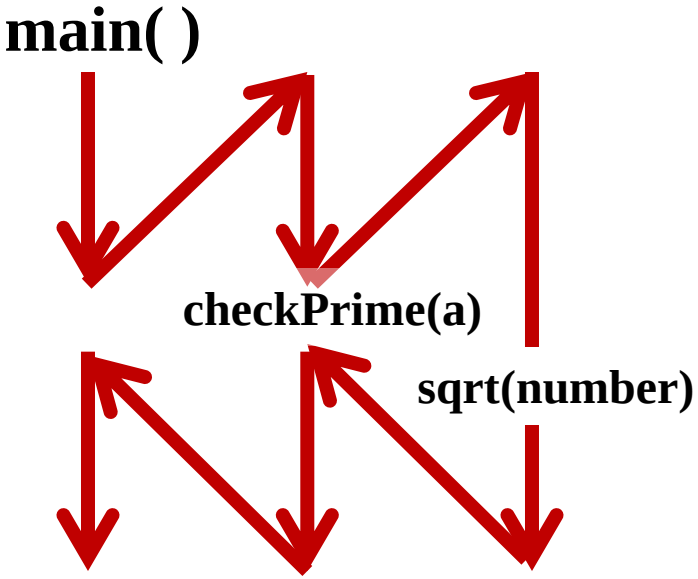
- 递推
- 递归
- 递归与递推

再看嵌套调用：素数判断

```
#include <iostream>
#include <cmath>
using namespace std;
bool checkPrime(int);       //判断素数
int main()
{
        int a;
        cout << "请输入一个整数" << endl;
        while (cin >> a)
        {
                if (checkPrime(a))
                        cout << "是质数" << endl;
                else
                        cout << "不是质数" << endl;
        }
        return 0;
}
```



素数判断




```
bool checkPrime(int number)
{
        int i, k;
        k = sqrt(number);
        for (i = 2; i <= k; i++)
        {
                if (number % i == 0)
                                                //只要有一个数被除尽
                        return 0;               //则不是素数
        }
        return 1;       //走到这一步，说明没能被除尽
}
```

函数关系

- 函数之间的关系
  - 函数**不能嵌套定义**
    - 所有函数一律平等
  - 函数**可以嵌套调用**
    - 无论嵌套多少层，原理都一样
- 问题
  - 函数能调用“自己”吗？

递归

- 递归：嵌套调用中，存在自己调用自己的语句
  - 间接递归：

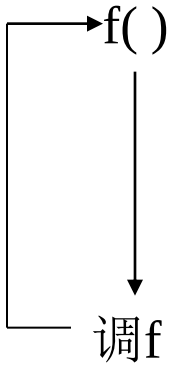
    A 调用 B， B 又调用 A 的方式
  - 直接递归：函数直接调用自身（A 调用 A）
- 递归的 2-Step 思想
  - 基始值（初始值）定义；
  - 归纳方法（向初始值靠拢）

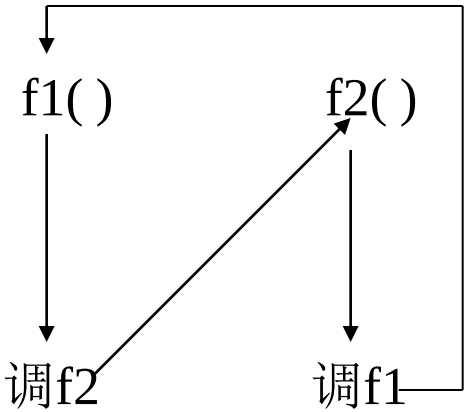
```
int  f(int x)
{    int y,z;
      ……
     z=f(y);
     …….
     return(2*z);
}
```

```
int  f1(int x)
{    int y,z;
      ……
     z=f2(y);
     …….
     return(2*z);
}
```

```
int  f2(int t)
{    int a,c;
      ……
     c=f1(a);
     …….
     return(3+c);
}
```

f 直接调用 f 自己





f 1 借助 f 2调用 f 1

- 说明
  - C编译系统对递归函数的自调用次数没有限制
  - 每调用函数一次，在内存堆栈区分配空间，用于存放函数变量、返回值等信息，递归次数过多，可能引起堆栈溢出

例子

- 求和：
  - 计算 f(n) = 1+2+…+n
- 递归描述

  f(1) = 1, （基始）

  f(n) = n+f(n-1) （归纳）
- **注意**
  - 函数递归一定要保证函数向结束的方向逼近，收敛于某一点
  - (n-1)! =n!/n 是要求的递归吗？

递归的直观解释

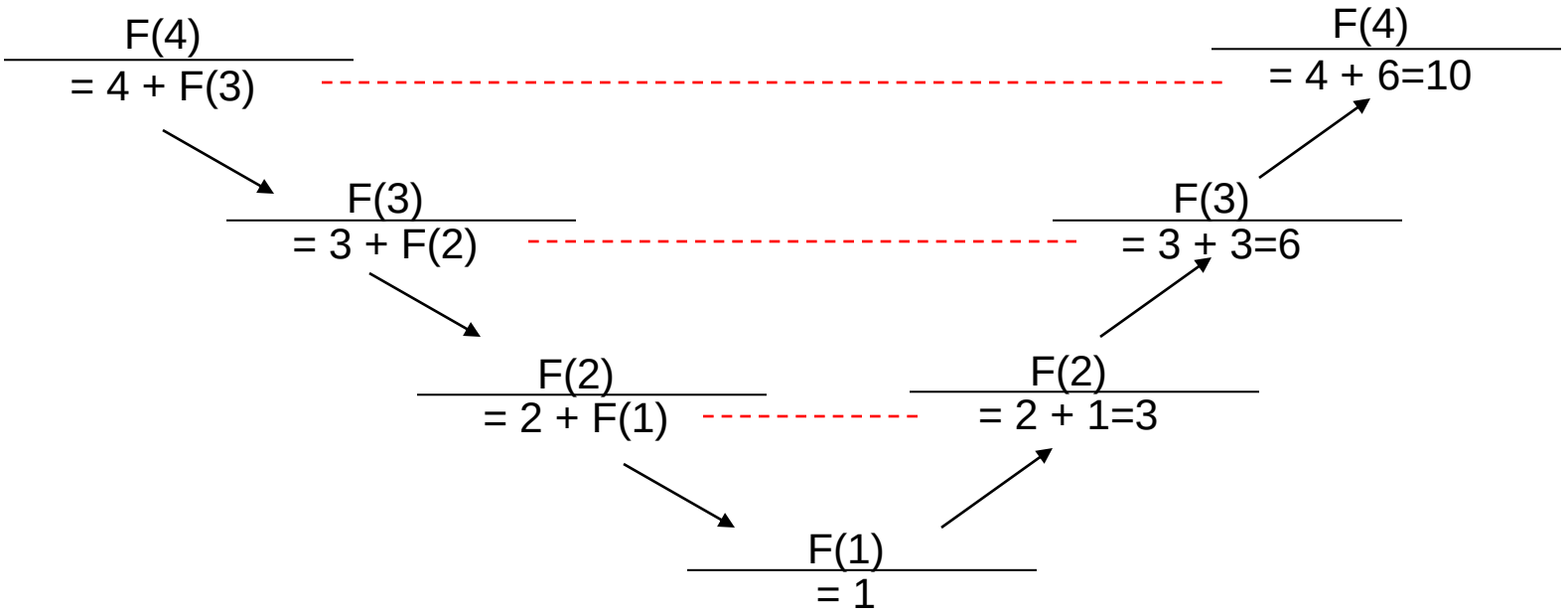
- 计算 1+2+…+n (n>0, n 为正整数)
- 定义：

$$f(n) = \begin{cases} 1, \text{ 当 } n=1 \text{ 时} \\ n+f(n-1), \text{ 当 } n > 1 \text{时} \end{cases}$$

- 含义：要计算 f(n)，如果能先计算 f(n-1)，就能在基础上加上 n 而计算出 f(n)。因此，需要先计算 f(n-1)

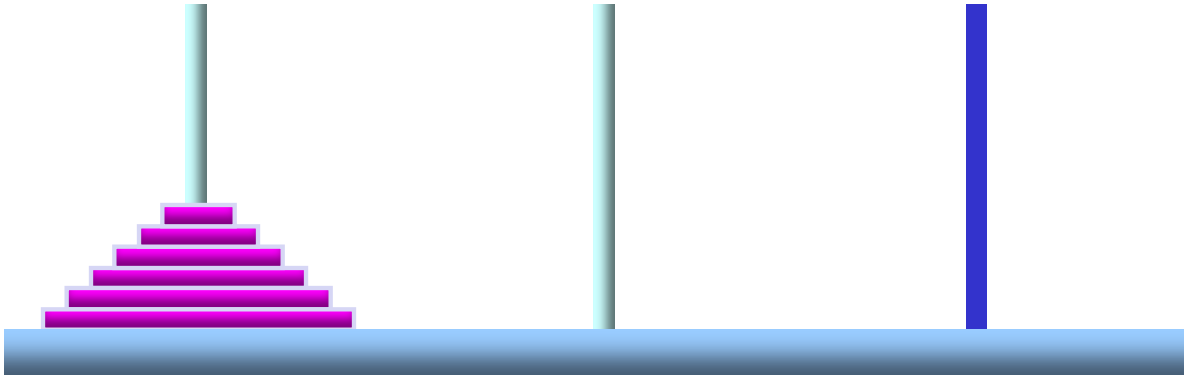
```
int f(int n)
{
  if (n=1) return(1);
 else return (n + f(n-1));  // 向初始值 f(1) 逼近。
}

int main()
{   int n, y;
    cout<<"Input a integer number:"<<endl;
    cin>>n;
    cout<<"The result of f(n): "<<f(n)<<endl;
    return 0;
}
```


F(4)
= 4 + F(3)
F(4)
= 4 + 6=10
F(3)
= 3 + F(2)
F(3)
= 3 + 3=6
F(2)
= 2 + F(1)
F(2)
= 2 + 1=3
F(1)
= 1

一个复杂的问题——— Hanoi Tower

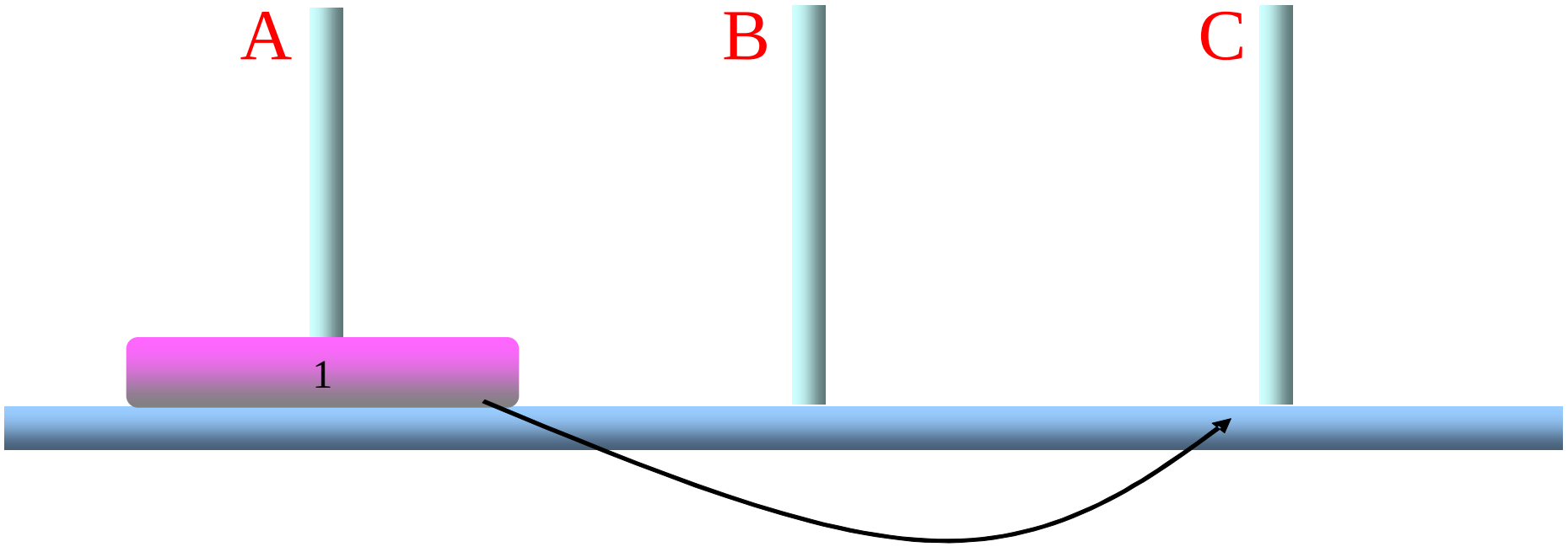
- 故事：相传在古代印度的Bramah庙中，有位僧人整天把三根柱子上的金盘倒来倒去，原来他是想把64个一个比一个小的金盘从一根柱子上移到另一根柱子上去。移动过程中恪守下述规则：每次只允许移动一只盘，且大盘不得落在小盘上面。
- 有人会觉得这很简单，真的动手移盘就会发现，如以每秒移动一只盘子的话，按照上述规则将64只盘子从一个柱子移至另一个柱子上，所需时间约为5800亿年。



逐步分析

1、在A柱上只有一只盘子，假定盘号为 1，这时只需将该盘从 A 搬至 C，一次完成，记为

move 1# from A to C　　(演示)



2块盘情况

在 A 柱上有二只盘子，1 为小盘，2 为大盘。

第 1 步将 1 号盘从A移至B，这是为了让 2号盘能动；

第 2 步将 2 号盘从A 移至 C；

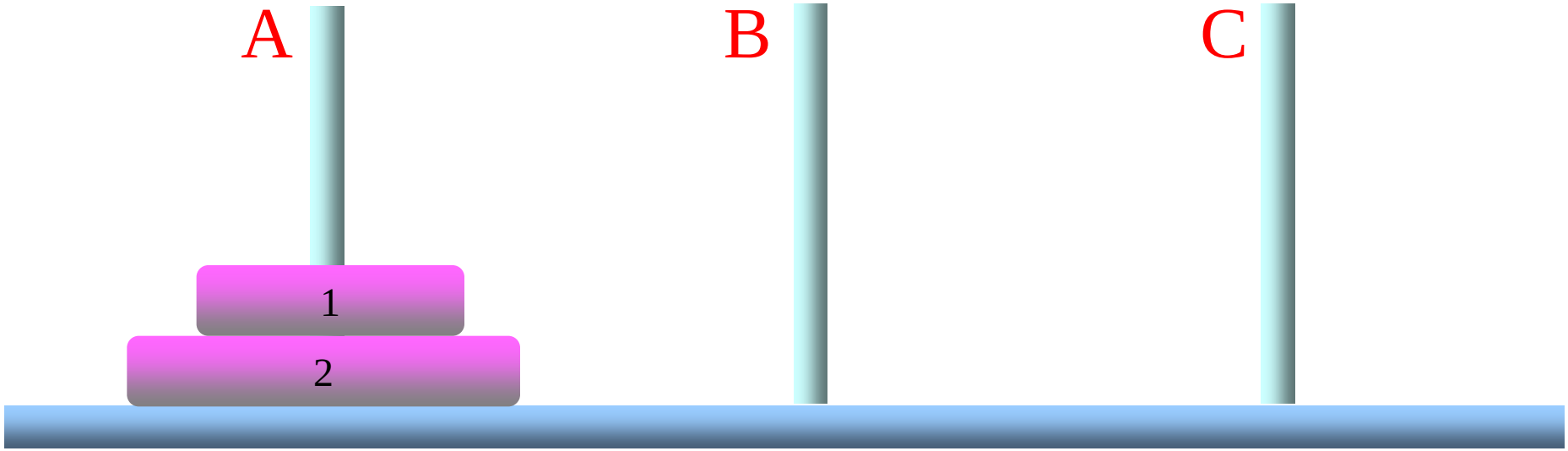
第 3 步再将 1 号盘从 B 移至 C；

这三步记为（演示）：

move 1# from A to B;

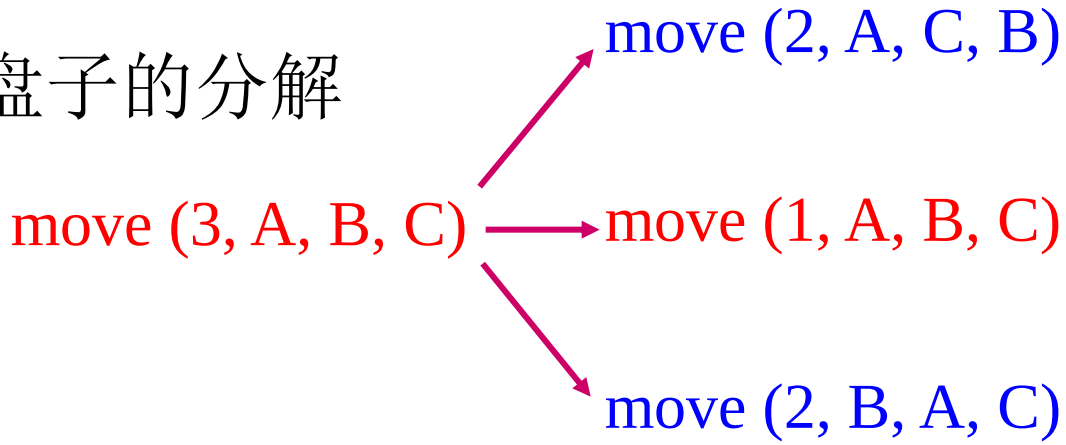
move 2# from A to C;

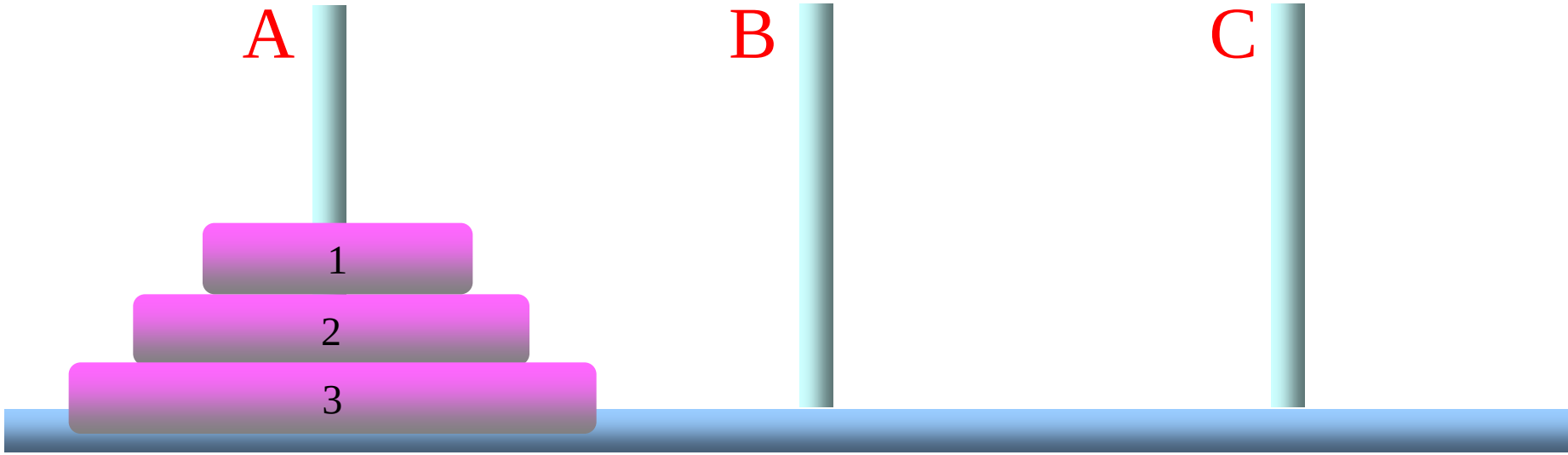
move 1# form B to C;



3 块盘情况

演示：移动3个盘子的分解

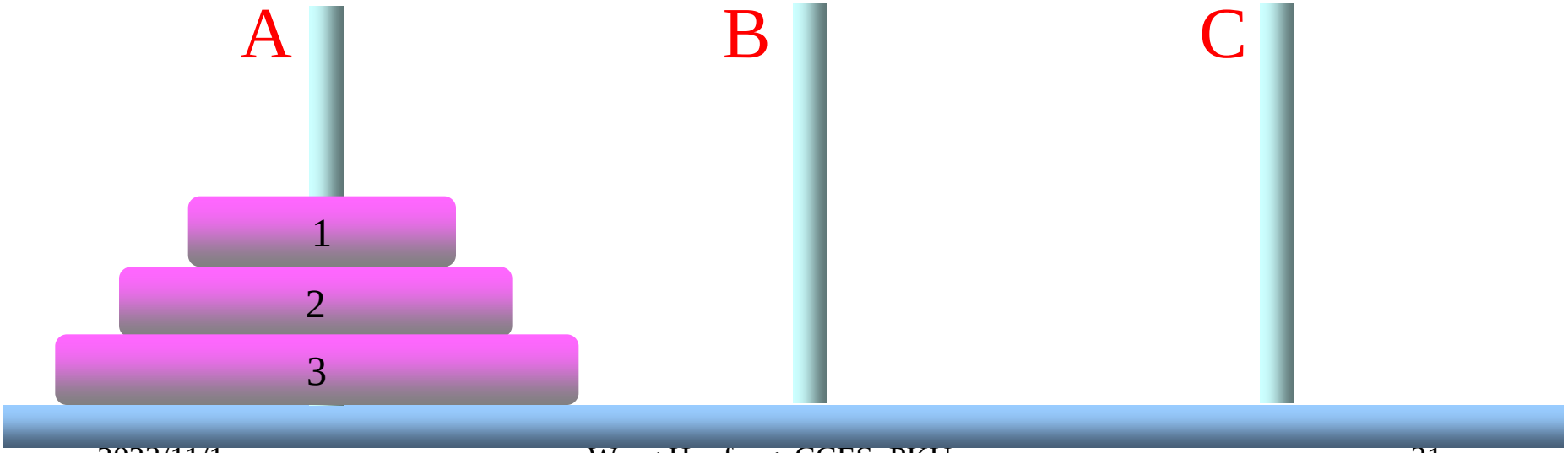




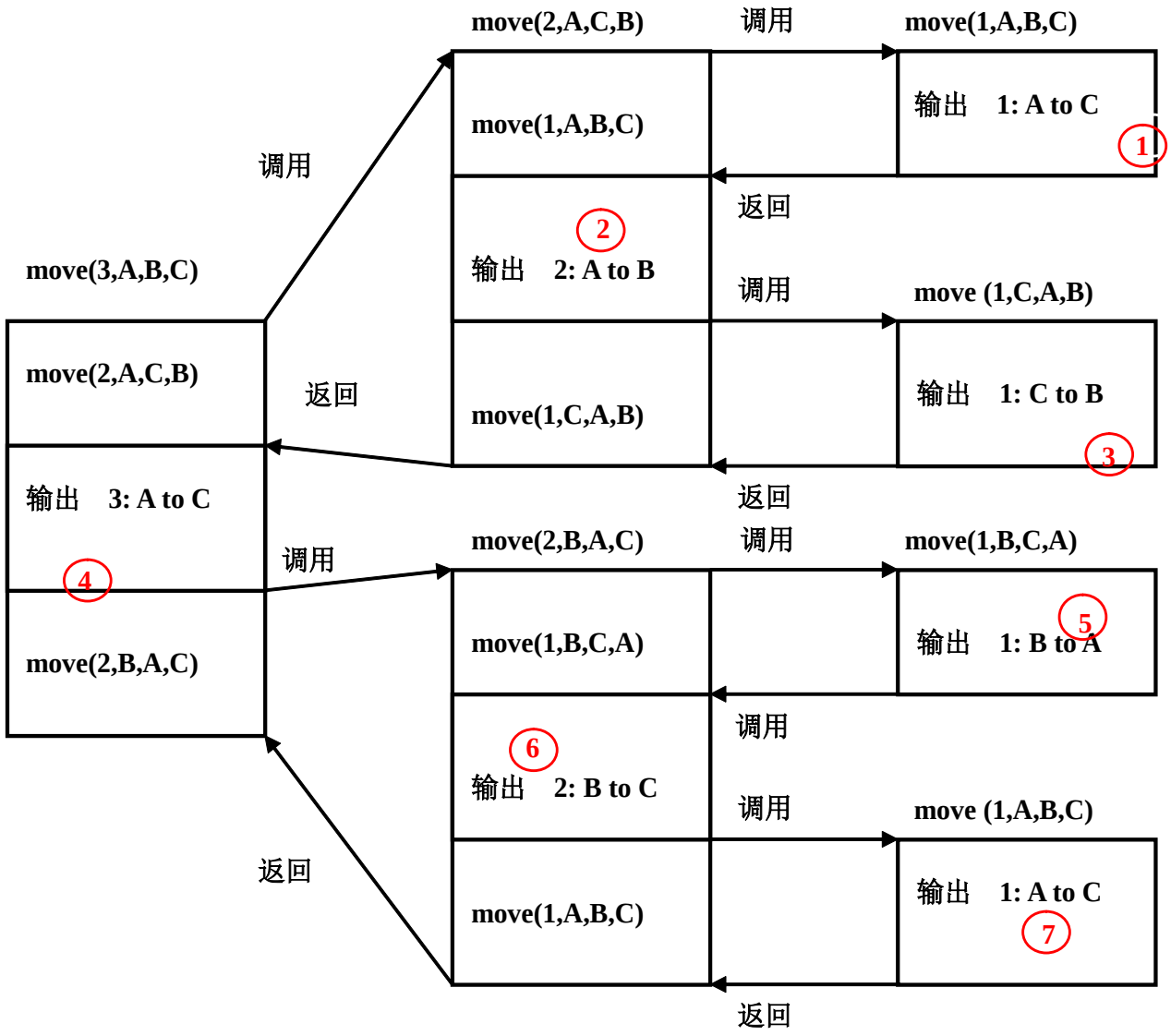
3 块盘情况分解

move (3, A, B, C)

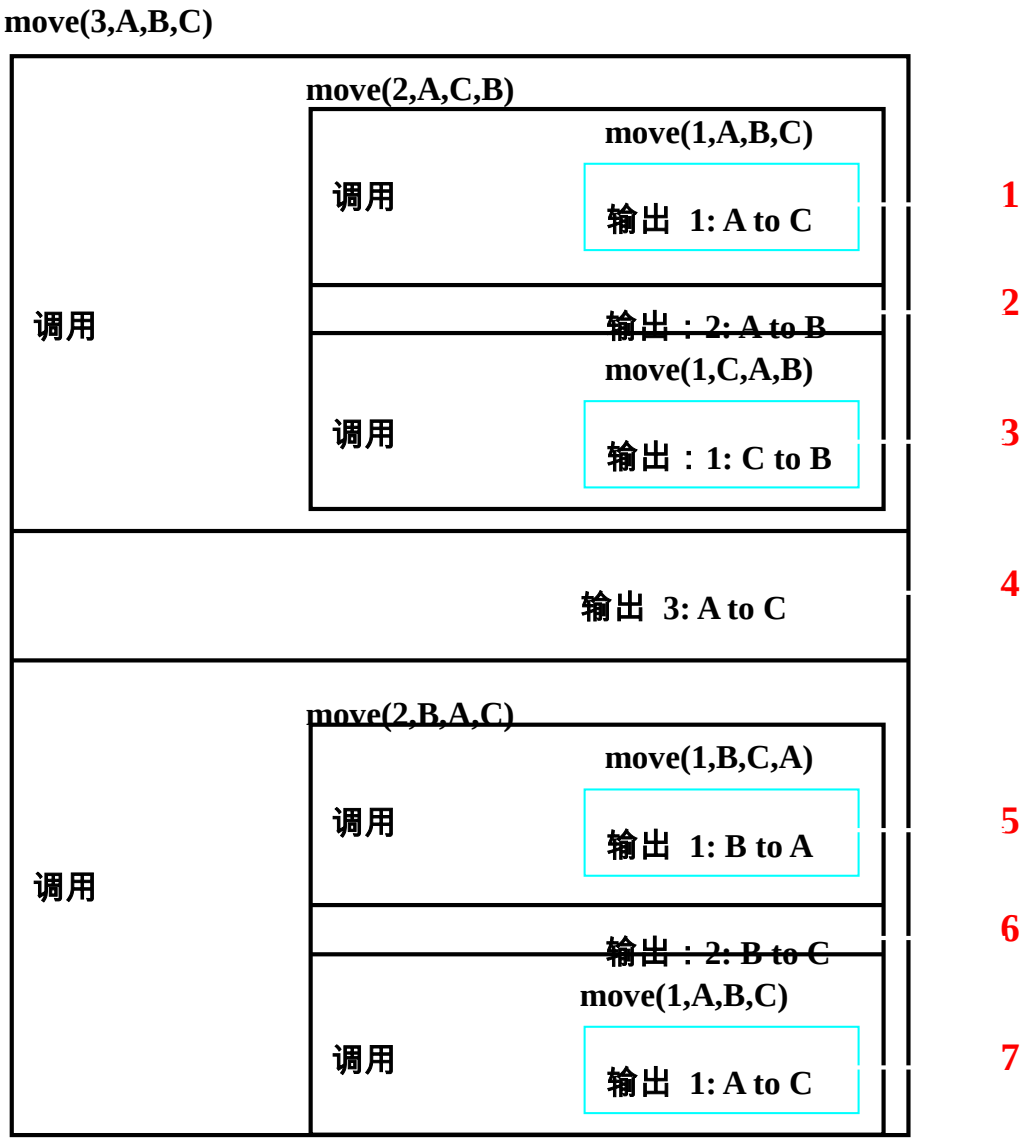
- move (2, A, C, B)
  - move (1, A, B, C)
  - move (1, A, C, B)
  - move (1, C, A, B)
- move (1, A, B, C)
- move (2, B, A, C)
  - move (1, B, C, A)
  - move (1, B, A, C)
  - move (1, A, B, C)



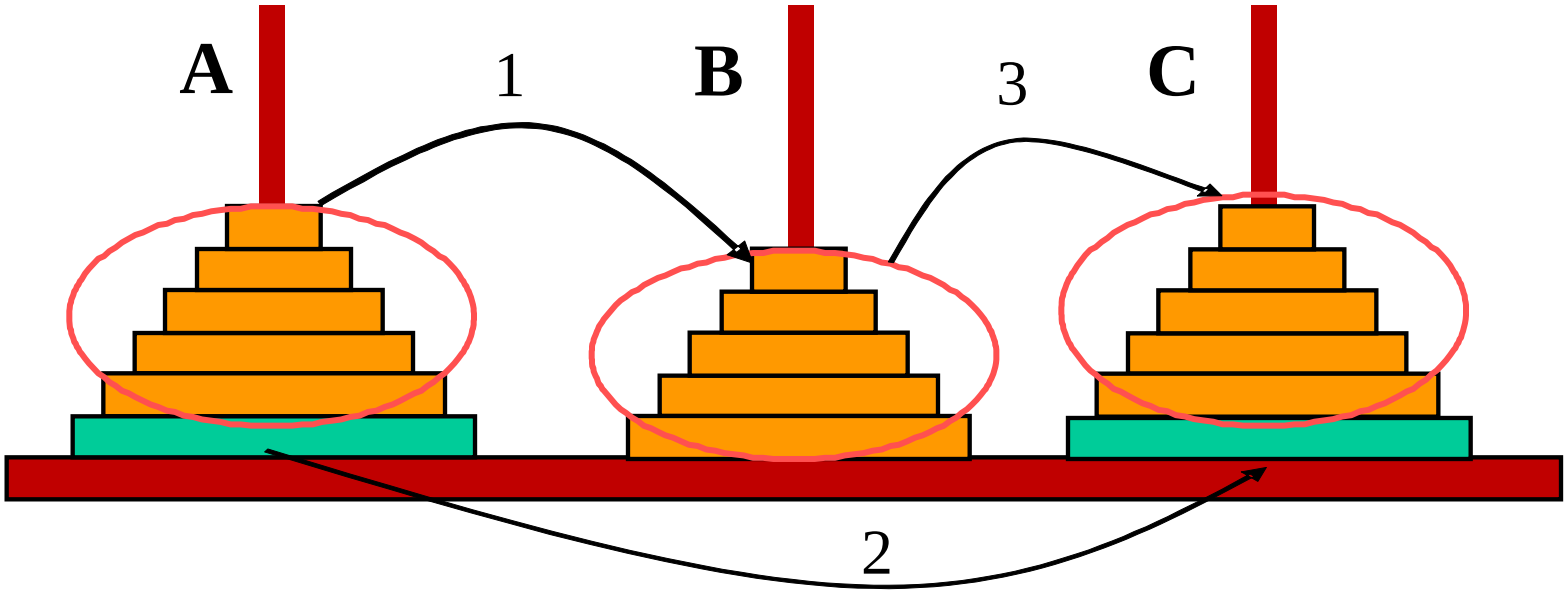
调用过程


move(2,A,C,B)
调用
move(1,A,B,C)
move(1,A,B,C)
输出 1: A to C
1
返回
2
输出 2: A to B
move(3,A,B,C)
调用
move (1,C,A,B)
move(2,A,C,B)
返回
move(1,C,A,B)
输出 1: C to B
3
输出 3: A to C
返回
move(2,B,A,C)
调用
move(1,B,C,A)
4
调用
move(1,B,C,A)
输出 1: B to A
5
move(2,B,A,C)
6
调用
输出 2: B to C
调用
move (1,A,B,C)
返回
move(1,A,B,C)
输出 1: A to C
7
返回

调用的进一步解释


move(3,A,B,C)
move(2,A,C,B)
move(1,A,B,C)
调用
输出 1: A to C
1
调用
输出：2: A to B
2
move(1,C,A,B)
调用
输出：1: C to B
3
输出 3: A to C
4
move(2,B,A,C)
move(1,B,C,A)
调用
输出 1: B to A
5
调用
输出：2: B to C
6
move(1,A,B,C)
调用
输出 1: A to C
7

N块盘的移动过程



```
#include <iostream>
using namespace std;

int step=1;                                  // 全局变量,预置1,步数
void  move(int, char, char, char);           // 原型函数
int main()                                   // 主函数
{
        int n;                               // 整型变量,n为盘数,
        cout << "请输入盘数 n=";             // 提示信息
        cin >> n;                            // 输入盘子数 n
        cout<< "在3根柱子上移"               // 输出提示信息
                << n << "只盘的步骤为:"<<endl;
        move(n,'A','B','C');
        return 0;
}
```

```
// 返回值：无
void move(int m, char p, char q, char r)
{                                   // 自定义函数体开始
    if (m==1)                       // 如果m为1,则为直接可解结点,
    {
        // 直接可解结点,输出移盘信息
        cout<<"["<<step<<"] move 1# from "<<p<<" to "<<r<<endl;
        step++;                     // 步数加1
    }
    else                            // 如果不为1,则要调用move(m-1)
    {
        move(m-1,p,r,q);    // 递归调用move(m-1)
        //直接可解结点,输出移盘信息
        cout<<"["<<step<<"] move "<<m
            <<"# from "<< p<<" to "<< r<<endl;
        step++;                             // 步数加1
        move(m-1,q,p,r);    // 递归调用move(m-1)
    }
}                                   //自定义函数体结束
```

内容

- 递推
- 递归
- **递归与递推**

递归与递推

- 递归算法是一种非常重要的算法，是求解问题的有力工具
- 递归算法的初学者需要建立起递归概念
- 与递推的比较
  - ✓ 递归：出发点不放在初始条件上，而放在求解的目标上，从所求的未知项出发逐次**调用自身**的求解过程，直到初始条件
  - ✓ 递推：从初始条件出发逐步迭代得到目标结果

递推计算阶乘序列 n!

- **递推问题**
  - 后续的运算依赖于已知的条件；当前的运算是下一步运算的基础；
  - 解法：从已知的初始条件出发，逐次去求所需要的阶乘值

初始条件 fact(1) = 1

fact(2) = 2*fact(1) = 2

fact(3) = 3*fact(2) = 6

… …

递推的程序实现

```
#include<iostream>
using namespace std;
int fact(int n)
{    int  k, result=1;   // 令 0!=1
     for (k = 1; k<=n; ++k)
              result=result*k;
     return result;
}
int main()
{            cout<<fact(10)<<endl;
             return 0;
}
```

递归计算阶乘序列 n!

- **递归问题**
  - 从所求的未知项出发逐次**调用自身**的求解过程，直到递归的边界（即初始条件）；
  - 需要给出初始条件

  初始条件： fact(1) = 1

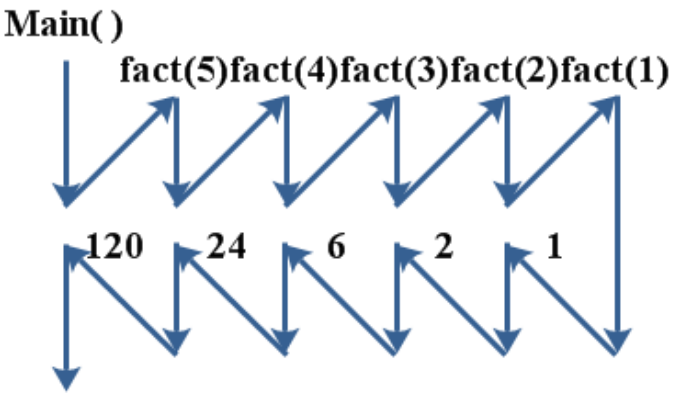
  递归（归纳）： fact(n)=fact(n-1)*n

程序实现

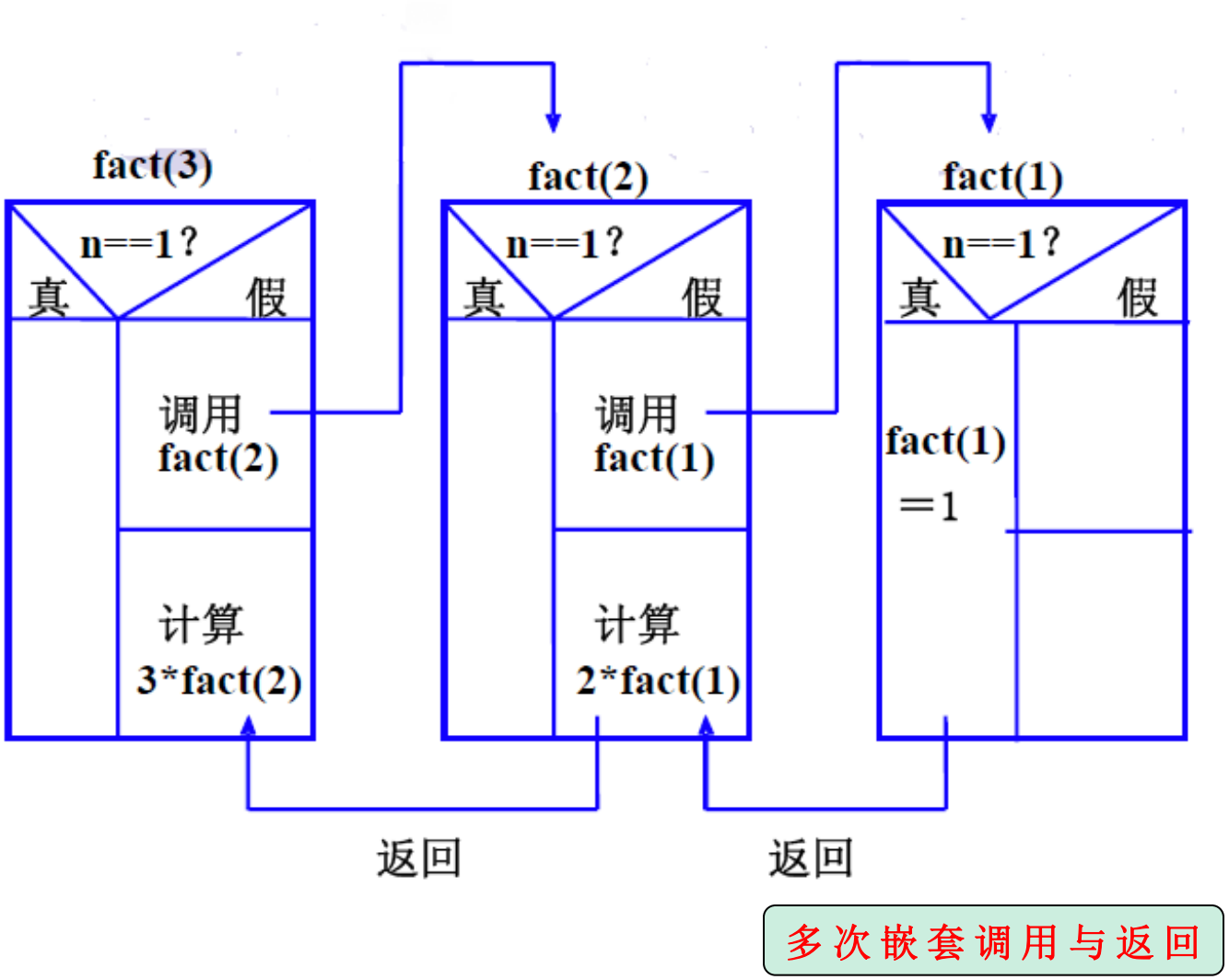
```
#include<iostream>
using namespace std;
int fact(int n){
    if (n == 1)
        return 1;
    else
        return( n * fact(n-1));
}

int main(){
        cout<<fact(5)<<endl;
        return 0;
}
```

很多次（不确定次数）的嵌套



再看调用与返回过程



多次嵌套调用与返回

另一个例子

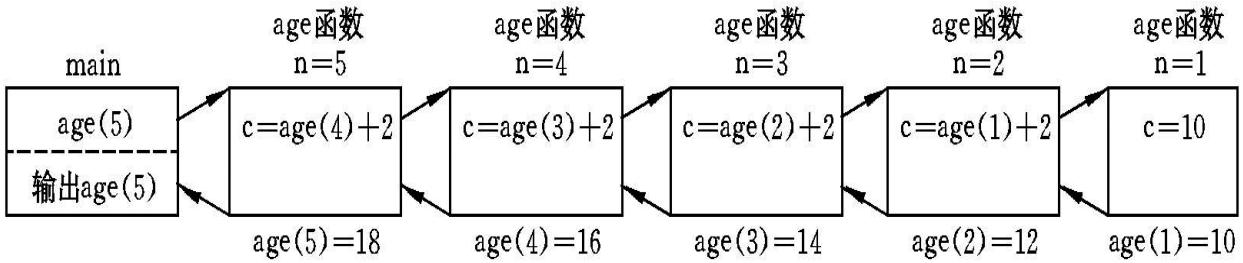
- 有5个人坐在一起，问第5个人多少岁？他说比第4个人大2岁。问第4个人岁数，他说比第3个人大2岁。问第3个人，又说比第2个人大2岁。问第2个人，说比第1个人大2岁。最后问第1个人，他说是10岁。请问第5个人多大。
  - Age[1] = 10;
  - Age[2] = age[1] + 2;
  - Age[3] = age[2] + 2;
  - Age[4] = age[3] + 2;
  - Age[5] = age[4] + 2;

递推实现

```
#include <iostream.h>
using namespace std;
int main()
{   int age[6];
    age[1] = 10;
    for (int i = 2; i <= 5; i++)
        age[i] = age[i-1] + 2;
     cout << "第5个人的年龄是："<<age[5] <<endl;
    return 0;
}
```

递归实现

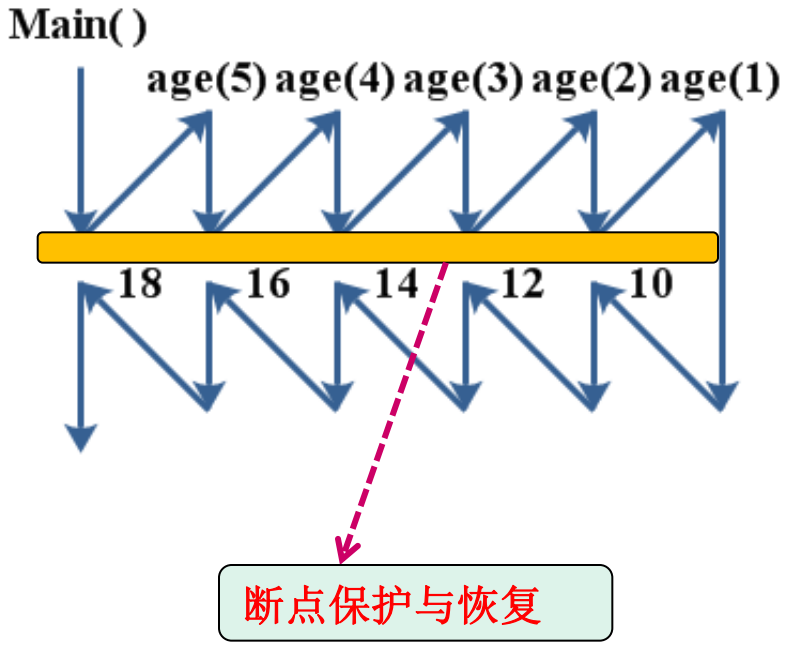
```
#include<iostream.h>
int age(int n)
{   int c;
    if(n == 1)
          c = 10;
    else
          c = age(n-1)+2;
    return(c);
}
void main()
{   cout<<“第五个人的年龄是：" <<age(5);
}
```



5 次 嵌 套 调 用 与 返 回

递归实现进一步解释

```
#include<iostream.h>
int age(int n)
{   int c;
    if(n == 1)
            c = 10;
    else
            c = age(n-1)+2;
    return(c);
}
void main()
{   cout<<“第五个人的年龄是：" <<age(5);
}
```



用递归实现递推

- 优点
  - **让程序变得简明**
  - **很多情况下设计程序更方便**
- 缺点
  - 程序本身的运行时空复杂性变高
    - 断点保护与恢复需时间
    - 断点保护需额外空间
- 递归的主要特点
  - 把关注点放在要求解的目标上
    - 找到第n次做与第n-1次做之间的关系（归纳）
    - 确定第1次的返回结果（基始）

函数参数变化有利于向基始值逼近

再看一个例子

- 递归计算正整数m和n的最大公约数
- 递归与递推的不同在哪里？

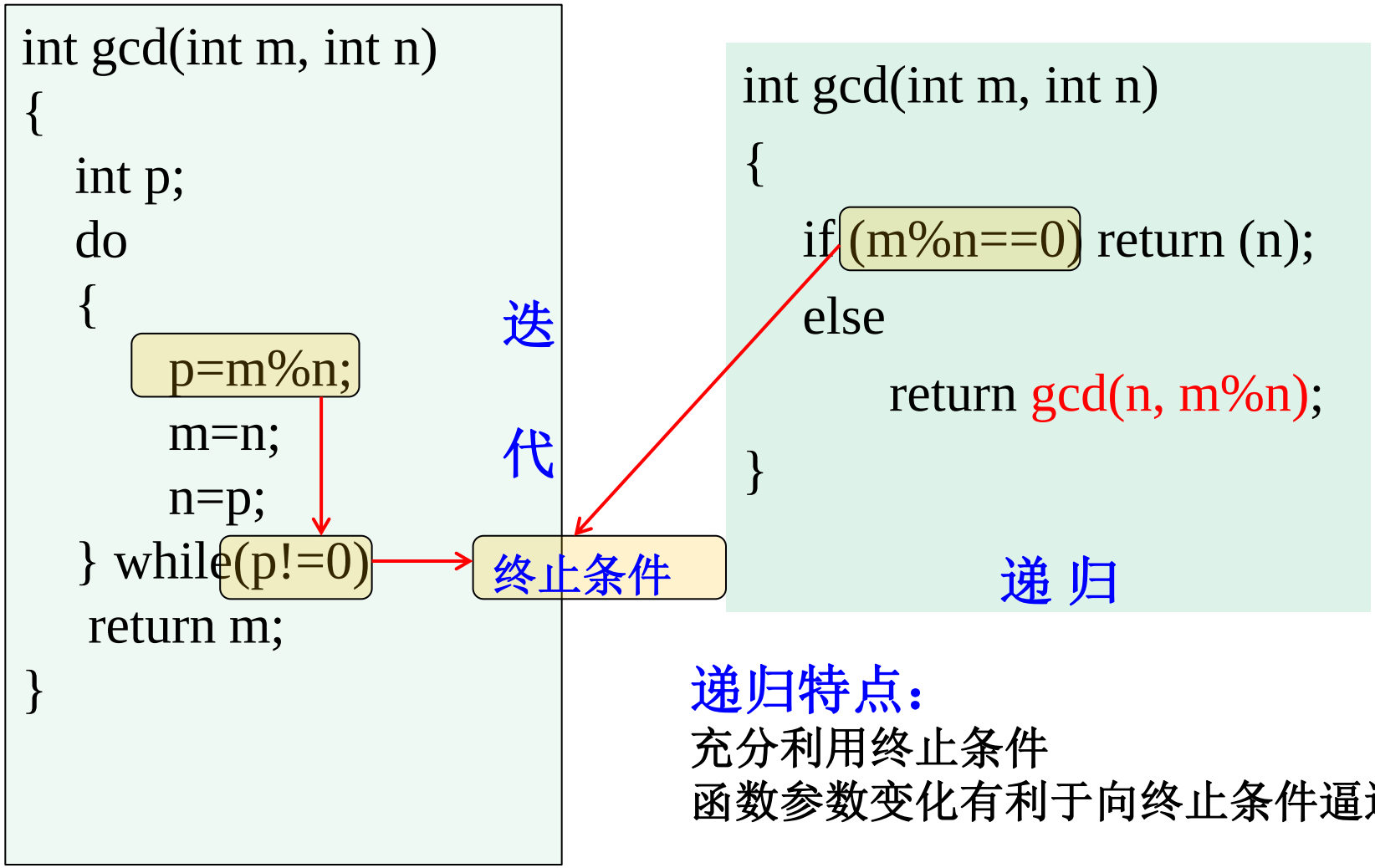
计算最大公约数的递推（迭代）

- 基本思想：
  - 假设： m > n
  - 引入临时变量： p 作为余数
  - step1: p=m%n;
  - step2: if (p==0) 结果为n
  - step3: 否则，m=n, n=p, 转 step1

迭代

- 函数实现：
  - int gcd(int m, int n)

如何递归？



递归特点：

充分利用终止条件

函数参数变化有利于向终止条件逼近

前面的例子如何用递归实现

- 斐波拉切序列
- 切片
- 分鱼
- 素数
- …