

中山大学网络空间安全学院必修课程

第6节 函数

岳 晟

网络空间安全学院 工学园2栋508

yuesh5@mail.sysu.edu.cn

2026年秋季

C Programming

哥德巴赫猜想

7-6 验证“哥德巴赫猜想”

分数 20

[查看题目详情](#) [全屏浏览](#) [切换布局](#)

数学领域著名的“哥德巴赫猜想”的大致意思是：任何一个大于2的偶数总能表示为两个素数之和。比如：24=5+19，其中5和19都是素数。本实验的任务是设计一个程序，验证20亿以内的偶数都可以分解成两个素数之和。

输入格式：

输入在一行中给出一个(2, 2 000 000 000]范围内的偶数N。

输出格式：

在一行中按照格式“N = p + q”输出N的素数分解，其中 $p \leq q$ 均为素数。又因为这样的分解不唯一（例如24还可以分解为7+17），要求必须输出所有解中p最小的解。

```
bool is_prime;  
int n;  
scanf("%d", &n);
```

```
int p = 2;  
for(; p<=n; p++){  
    is_prime = 1;  
    for (int i=2; i<=p/i; i++){  
        if (p % i == 0){  
            is_prime = 0;  
            break;  
        }  
    }  
    if(is_prime){  
        for (int i=2; i<=(n-p)/i; i++){  
            if ((n-p) % i == 0){  
                is_prime = 0;  
                break;  
            }  
        }  
        if(is_prime) break;  
    }  
    printf("%d = %d + %d\n", n, p, n-p);
```

```
int is_prime = 1;  
for (int i=2; i<=m/i; i++){  
    if (m % i == 0) {  
        is_prime = 0;  
        break;  
    }  
}
```

判断素数

用循环

```
bool is_prime;
int n;
scanf("%d", &n);

int p = 2;
for(; p<=n; p++){
    is_prime = true;
    for (int i = 2; i <= p / i; i++){
        if (p % i == 0){
            is_prime = false;
            break;
        }
    }
    if(is_prime){
        for (int i = 2; i <= (n-p) / i; i++){
            if ((n-p) % i == 0){
                is_prime = false;
                break;
            }
        }
    }
    if(is_prime) break;
}

printf("%d = %d + %d\n", n, p, n-p);
```

用循环

```
bool is_prime;
int n;
scanf("%d", &n);

int p = 2;
for(; p<=n; p++){
    is_prime = true;
    for (int i = 2; i <= p / i; i++){
        if (p % i == 0){
            is_prime = false;
            break;
        }
    }
    if(is_prime){
        for (int i = 2; i <= (n-p) / i; i++){
            if ((n-p) % i == 0){
                is_prime = false;
                break;
            }
        }
    }
    if(is_prime) break;
}

printf("%d = %d + %d\n", n, p, n-p);
```

代码重复!

用函数

```
int n;
scanf("%d", &n);

for(int p=2; p<=n; p++)
    if(is_prime(p) && is_prime(n-p))
        printf("%d = %d + %d", n, p, n-p);
```

主要内容

1. 函数的概念

2. 函数的定义、调用与声明

3. 递归

4. 头文件

5. 储存类别

主要内容

1. 函数的概念

2. 函数的定义、调用与声明

3. 递归

4. 头文件

5. 储存类别

函数是什么

函数 (function) 是实现某功能的独立代码块

我们已经用过了许多函数：

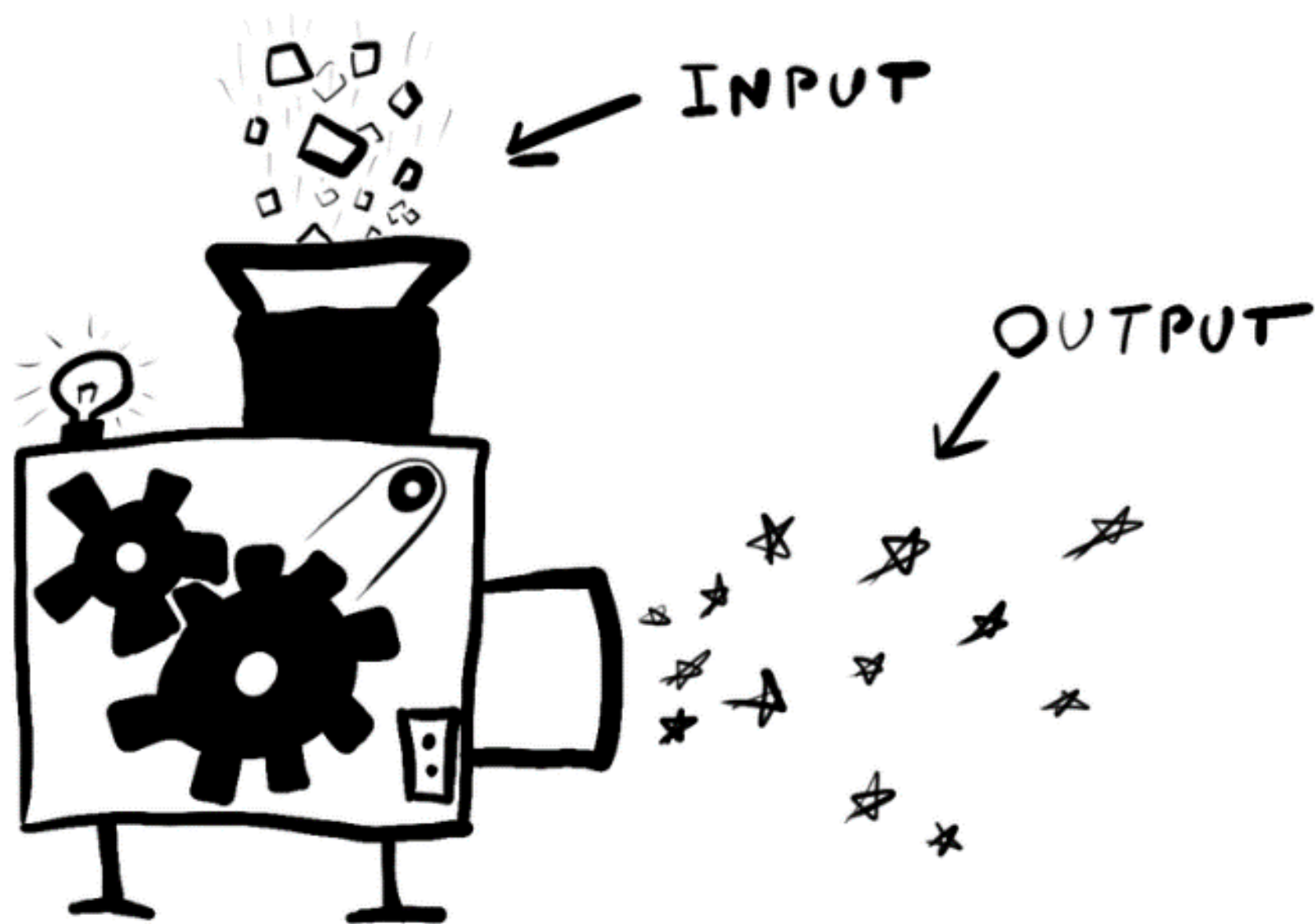
`printf()` `scanf()` `getchar()` `putchar()`

`sqrt()` `pow()` `fabs()`

`strlen()` `isdigit()` `isupper()` ...

函数是什么

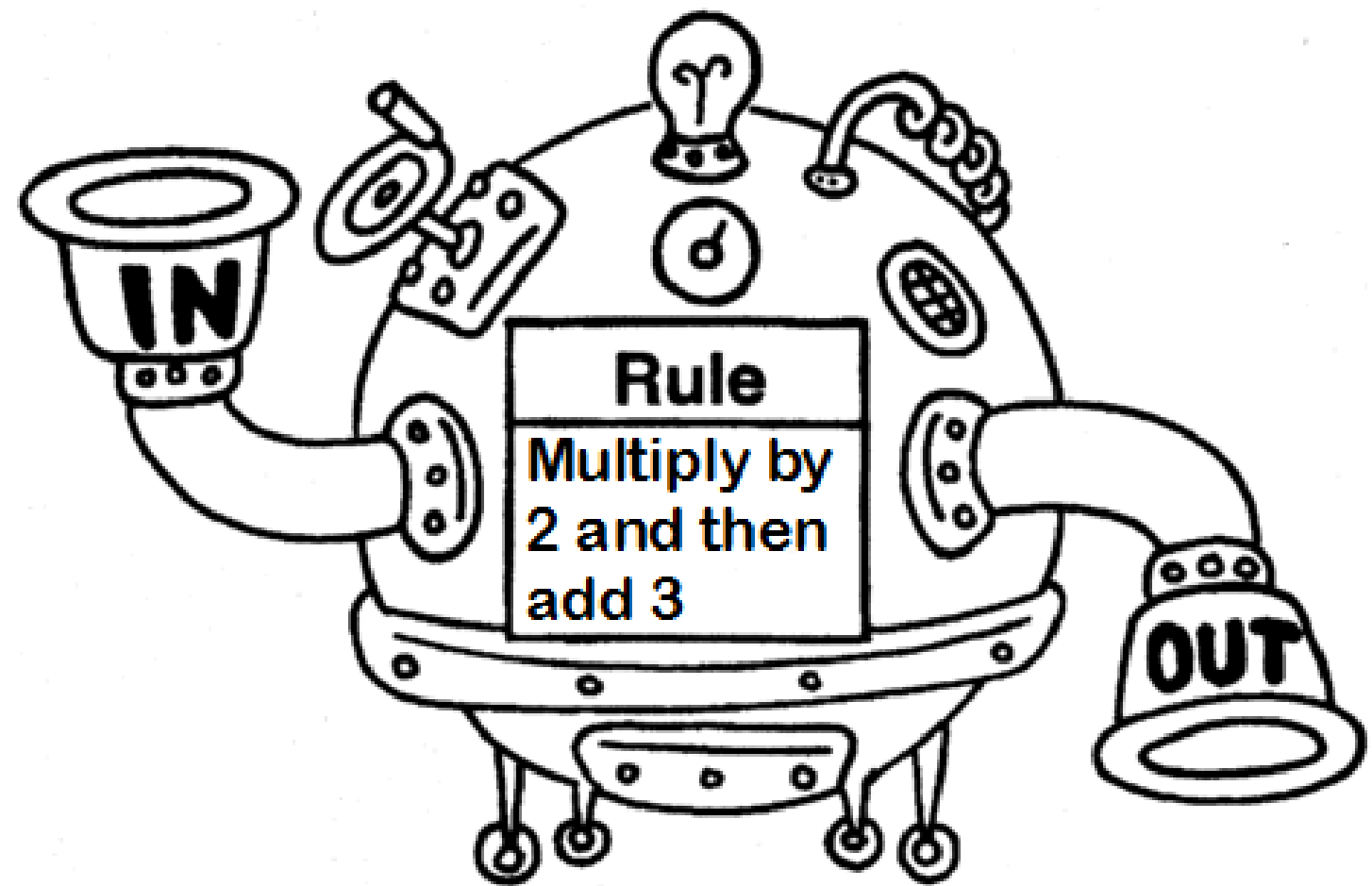
- ❑ 函数可理解为一个抽象的“黑盒”：接受“输入”，对输入进行处理并给出“输出”
- ❑ 使用时不需要关注具体的实现细节



输入		输出
<code>sqrt(4.0)</code>	----->	2.0
<code>printf("A")</code>	----->	打印"A"
<code>strlen("A")</code>	----->	1

函数让程序更加模块化

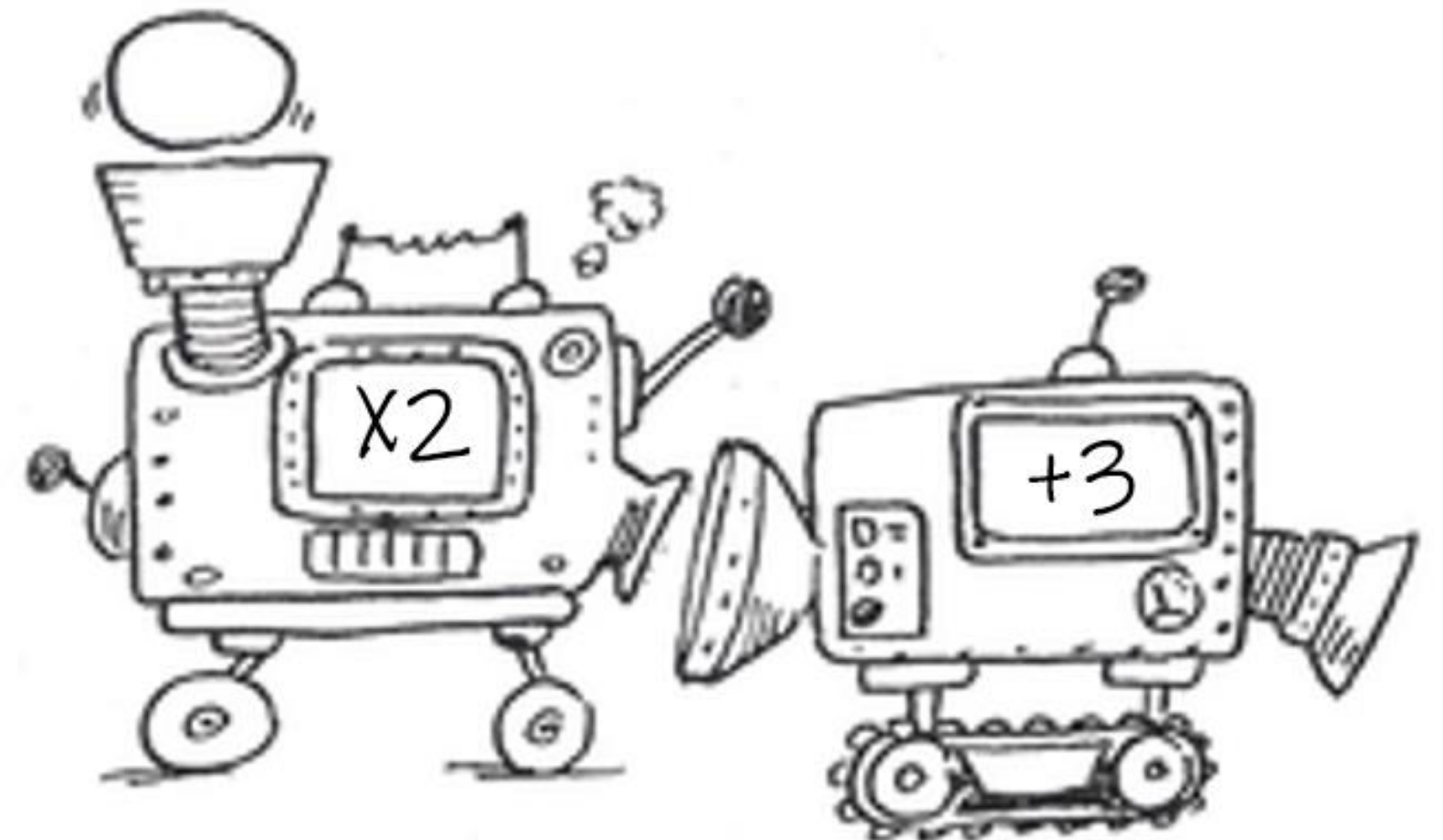
- 函数能够将一大块代码分解成若干个**模块**



一个大程序

分解

组装



分解成两个小函数

函数让程序更加模块化

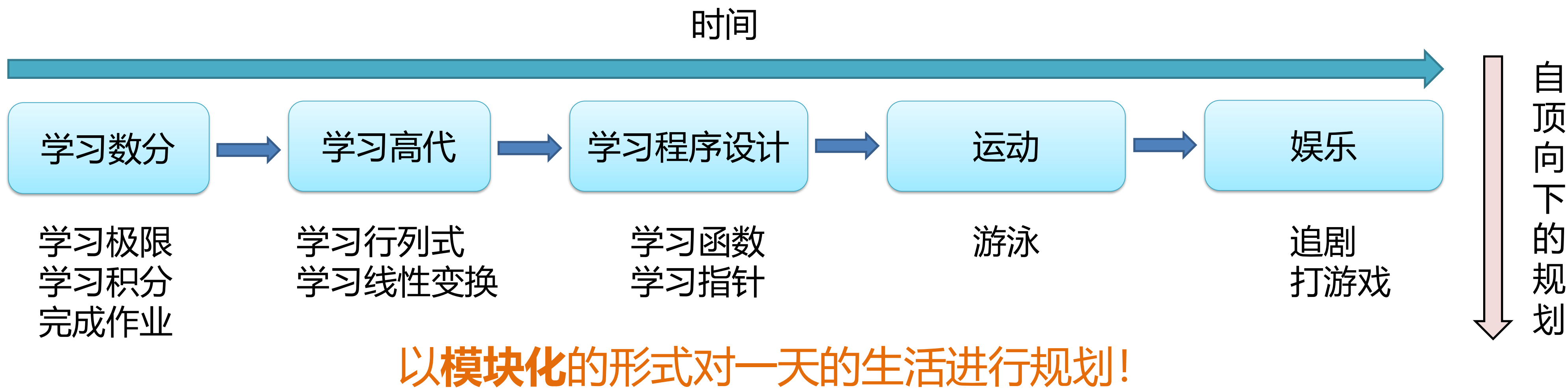
```
int n;  
scanf("%d", &n);  
  
for(int p=2; p<=n; p++)  
    if(is_prime(p) && is_prime(n-p))  
        printf("%d = %d + %d", n, p, n-p);
```

函数让程序更加模块化

```
int n;  
scanf("%d", &n);  
  
for(int p=2; p<=n; p++)  
    if(is_prime(p) && is_prime(n-p))  
        printf("%d = %d + %d", n, p, n-p);
```

函数带来哪些好处

□ 模块化使代码易于设计



□ 函数代码具备可重用性

□ 函数使代码易于理解，易于调试

主要内容

1. 函数的概念

2. 函数的定义、调用与声明

3. 递归

4. 头文件

5. 储存类别

主调函数与被调函数

主调函数：main()

```
int main(void)
{
    int n = 17;
    int r = is_prime(n);
    return 0;
}
```

调用别的函数的函数，叫主调函数。

调用
is_prime(n)



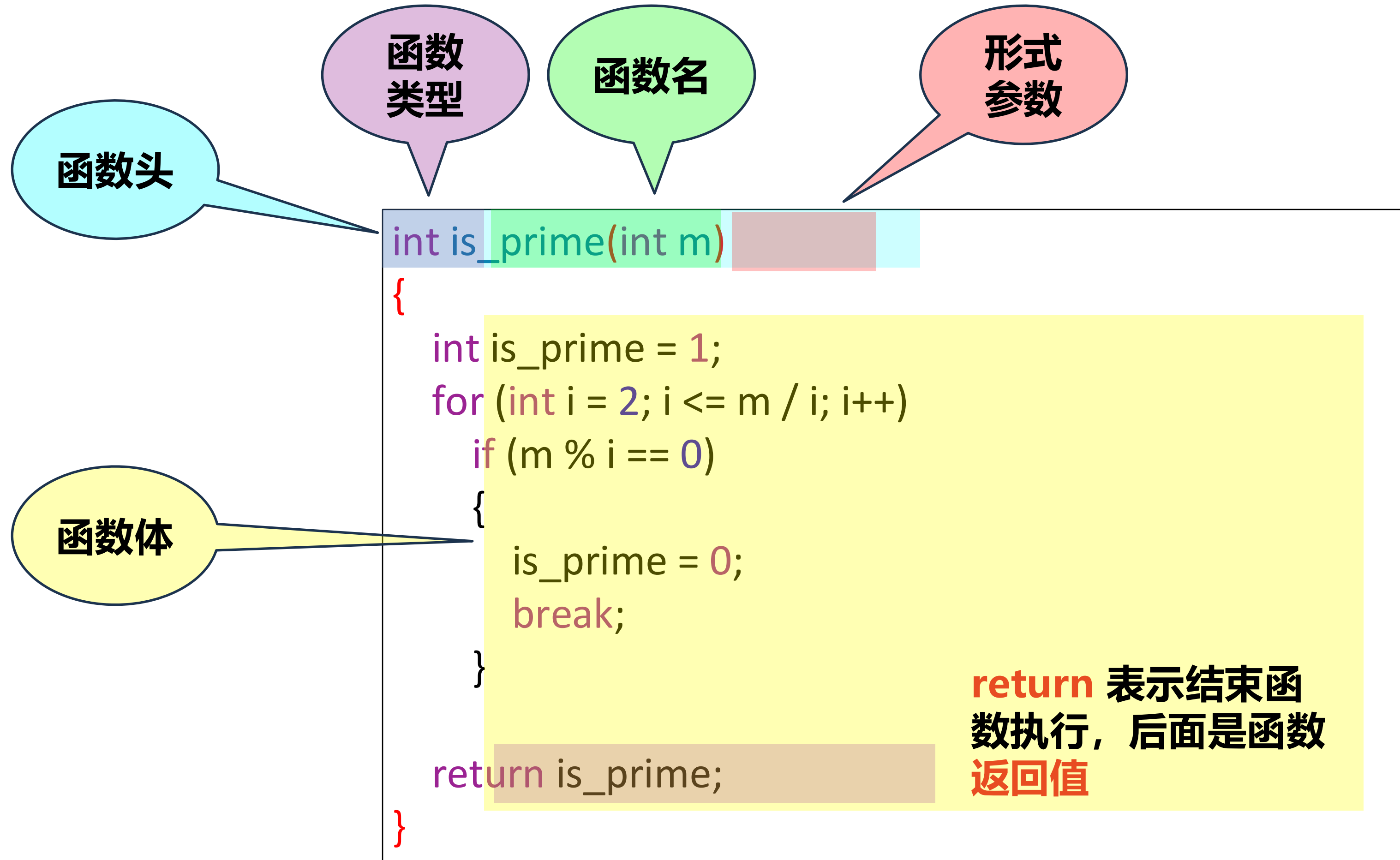
返回结果 **r**

被调函数：is_prime()

```
int is_prime(int m)
{
    ...
    return 1;
}
```

被其他函数调用的函数，叫被调函数。

函数的定义：有参函数



函数的定义：无参函数

无返回值
类型

无参数

```
void sum_1_100(void)
{
    int i;
    int sum = 0;
    for (i=1; i<=100; i++)
        sum += i;
    printf("1到100的和是%d\n", sum);
}
```

函数的定义

有参函数

类型名 函数名(形式参数表列)

{

函数体

}

无参函数

类型名 函数名(void)

{

函数体

}

1. 函数和变量一样，**必须定义类型**
2. 函数类型是**返回值的类型**
3. 函数体的**花括号**不能省略
4. 函数名必须**唯一**
5. 函数名应反映其代表的功能

函数的种类

无参数，无返回值

```
void function_name(void)
{
    ...;
}
```

有参数，无返回值

```
void function_name(parameters)
{
    ...;
}
```

无参数，有返回值

```
type function_name(void)
{
    ...;
    return value;
}
```

↗ 函数值（函数返回值）

有参数，有返回值

```
type function_name(parameters)
{
    ...;
    return value;
}
```

无返回值函数可以没有return，有返回值函数必须有return

函数的返回值

对于有返回值的函数来说：

1. return 后面**表达式**的值就是函数的返回值，返回值可以赋值给变量、用于表达式、传递给函数或丢弃
2. 一个函数中可以有一个**以上**的 return 语句
3. return 语句中的**表达式类型**应该和**函数类型一致**，否则自动类型转换为函数类型

练习：以下程序会输出什么？

```
#include <stdio.h>

int max(float x, float y) {
    float z;
    z = x > y ? x : y;
    return z;
}
```

```
int main(void) {
    float a = max(1.2, 3.3);
    printf("Max is %.1f\n", a);
    return 0;
}
```

A. 3.3

B. 1.2

C. 4.0

D. 3.0

练习：以下程序会输出什么？

```
#include <stdio.h>

int max(float x, float y) {
    float z;
    z = x > y ? x : y;
    return z;
}
```

```
int main(void) {
    float a = max(1.2, 3.3);
    printf("Max is %.1f\n", a);
    return 0;
}
```

A. 3.3

B. 1.2

C. 4.0

D. 3.0

函数的调用

函数调用：函数名(参数值1,参数值2,...) 或 函数名()

printf("%d",i)

is_prime(i)

is_prime(7+i)

getchar()

1. 圆括号()用来标识函数，即使无参数也不可省略
2. 若有参数，则需给出参数值正确的数量、顺序与类型，参数值会按顺序初始化函数参数

sum(1,100);

```
void sum(int begin, int end) {  
    int i;  
    ...;  
    printf("%d到%d的和是%d\n", begin, end, sum);  
}
```

形参与实参

形式参数/形参

```
void sum(int begin, int end)
{...;}
```

实际参数/实参

```
sum(1, 100);
```

□ 形式参数 (formal parameter)

- 函数在**定义和声明**时用的参数，需在**每个**变量前声名其类型
- 形参为**局部变量**，每次**调用函数时**才为这些变量**分配内存单元并赋值**

□ 实际参数 (actual argument)

- 主调函数拷贝给被调函数形式参数的**具体值** (可为常量、变量、表达式)
- 实参与形参必须**数量相同**、**类型兼容**
- 若实、形参类型兼容但不一致，则按初始化规则**强制转换**

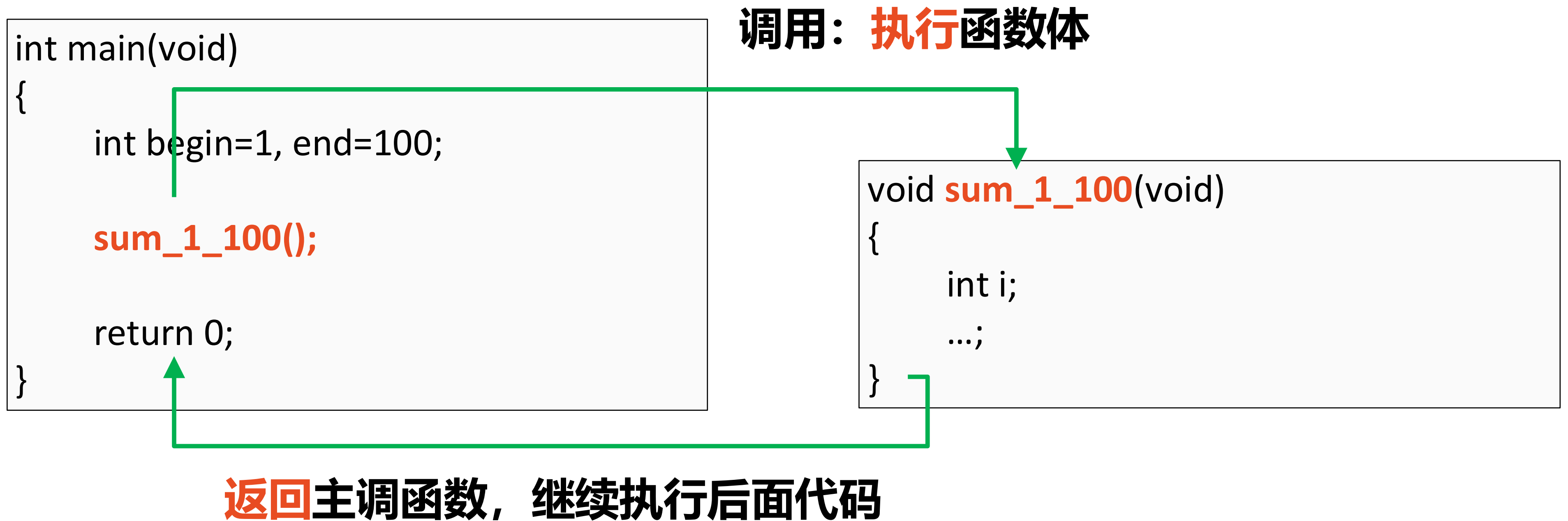
练习：以下程序会输出什么？

```
#include <stdio.h>
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```
int main(void) {
    int a = 10, b = 20;
    swap(a, b);
    printf("a=%d b=%d\n", a, b);
    return 0;
}
```

实参、形参为不同储存单元！

函数调用过程：无返回值函数



函数调用过程：有返回值函数

调用：创建形参单元，将实参**值拷贝**给形参单元，**执行函数体**

```
int main(void)
{
    int begin=1, end=100;
    int sum;

    sum = sum(1,100);

    return 0;
}
```

```
int sum(int begin, int end)
{
    int i, sum;
    ...;
    return sum;
}
```

终止函数，形参单元**释放**，**返回**sum的**值**给主调函数

函数调用：总结

- 函数是一个初等表达式 (primary expression)
 - 函数返回值的类型就是表达式的类型
 - 函数遇到 return 语句或函数块执行完毕**函数表达式完成求值**
 - **除了返回 void 类型，必须使用 return 语句返回一个值**
- 函数实参与形参
 - 实参与形参必须数量一致，**类型兼容**
 - 实参是**传值**给形参，形参为局部变量
 - 形参是传引用则实参必须是 “&左值表达式” 或 “地址” （下下周）

函数的声明

- ❑ 编译器**自上而下**分析代码
- ❑ 在看到 `swap(a, b);` 时需要知道 `swap` 函数是什么，才能知道对`swap` 函数的调用是否正确
 - **参数个数**有多少？ **参数类型**分别是什么？ **返回值类型**是什么？
- ❑ 但把所有函数都定义在 `main` 函数之前会使代码读起来很费劲

函数的声明/函数原型

函数类型 函数名(参数类型1 [参数名1], 参数类型2 [参数名2]...);

- 函数类型 函数名(参数类型1 参数名1...); 被称为**函数原型**
- 目的是为了告诉编译器函数的接口信息：**名称、参数 (数量与类型) 和返回类型**
- 函数原型一般写在**主调函数之前**
- 函数原型可以用**不同**参数名或**省略**参数名

函数原型

```
int is_prime(int);
```

```
int main(void) {  
    is_prime(7);  
}
```

根据原型
检查

```
int is_prime(int m)
```

函数头

头文件

- 每次逐个声明函数很麻烦，所以会按功能把一组函数声明放在一个**头文件里 (.h)**，方便使用
- 从另一个角度看，这分离了使用者和设计者的世界，是使用者和设计者之间共同遵守的契约（接口）
 - 作为使用者，不关心函数是如何实现，我们只关心如何使用函数，有声明就足够了
 - 作为设计者，根据函数声明，编写函数实现（例如：scanf 在不同C语言编译器中实现各不相同，但使用者都可放心使用它们）

头文件

函数的声明

```
double sin(double _X);
double cos(double _X);
double tan(double _X);
double sinh(double _X);
double cosh(double _X);
double tanh(double _X);
double asin(double _X);
double acos(double _X);
double atan(double _X);
double atan2(double _Y,double _X);
double exp(double _X);
double log(double _X);
double log10(double _X);
double pow(double _X,double _Y);
double sqrt(double _X);
double ceil(double _X);
double floor(double _X);
```

一目了然，一眼知道如何调用 😊

函数的定义

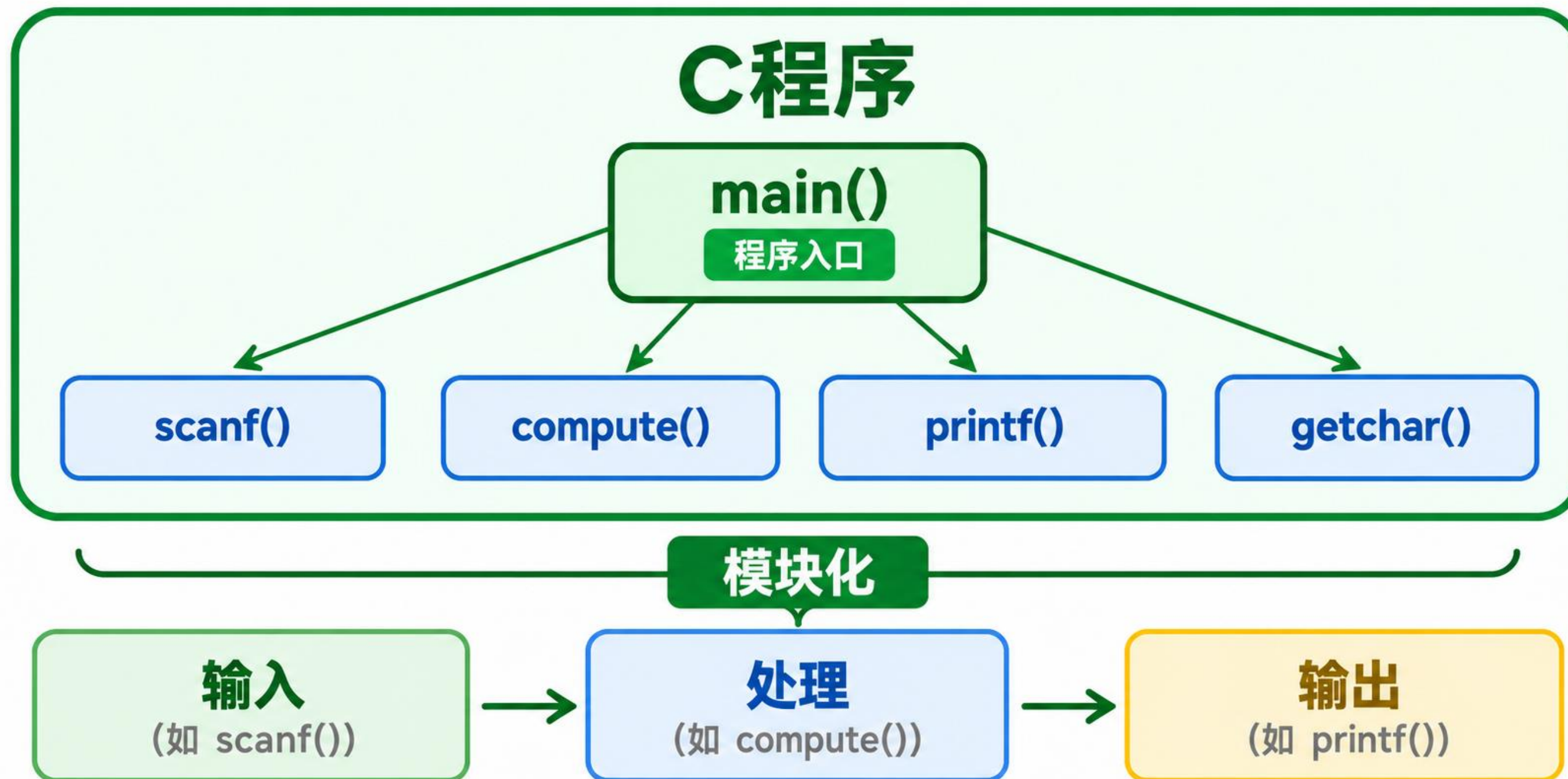
```
__mingw_ovr
__attribute__((__format__(gnu_scanf, 1, 2))) __MINGW_ATTRIB_NONNULL(1)
int scanf(const char *__format, ...)
{
    int __retval;
    __builtin_va_list __local_argv; __builtin_va_start( __local_argv, __format );
    __retval = __mingw_vfscanf( stdin, __format, __local_argv );
    __builtin_va_end( __local_argv );
    return __retval;
}

__mingw_ovr
__attribute__((__format__(gnu_scanf, 2, 3))) __MINGW_ATTRIB_NONNULL(2)
int fscanf(FILE *__stream, const char *__format, ...)
{
    int __retval;
    __builtin_va_list __local_argv; __builtin_va_start( __local_argv, __format );
    __retval = __mingw_vfscanf( __stream, __format, __local_argv );
    __builtin_va_end( __local_argv );
    return __retval;
}
```

需要在万千代码中辛苦找到函数接口 😭

C程序与函数

- ❑ C程序由若干函数构成 (**必须包含 main** 函数作为程序入口)
- ❑ 函数允许我们**模块化程序**



常用数学函数

- 在标头 `stdlib.h` 中定义
 - `int abs(int n);`
- 定义于头文件 `math.h`
 - `double fabs(double arg);`
 - `double exp(double arg);`
 - `double log(double arg);`
 - `double log10(double arg);`
 - `double pow(double base, double exponent);`
 - `double sqrt(double arg);`
 - `double sin(double arg);`
 - `cos, tan, asin, acos, atan`
 - `double ceil(double arg);`
 - `double floor(double arg);`

要求掌握!

练习：电报加密

输入一行电报文字，将字母变成其下一字母（如 'a' 变成 'b' 'Z' 变成 'A'，其它字符不变）。

样例输入： a Z

样例输出： b A

请打开 6_01_encipher.c 结合代码注释完成程序

主要内容

1. 函数的概念

2. 函数的定义、调用与声明

3. 递归

4. 头文件

5. 储存类别

递归



递归通过调用自身解决规模更小的同类问题

递归示例：阶乘

□ n 的阶乘就是所有小于等于 n 的正整数的积：

$$n! = n \times \underbrace{(n-1) \times (n-2) \times \cdots \times 1}_{(n-1)!}$$

□ n 的阶乘还可以这样定义：

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{if } n > 1 \end{cases} \rightarrow \text{fac}(n) = n * \text{fac}(n-1);$$

□ 第一种叫做**迭代定义**，第二种叫做**递归定义**，包含

1. **基本情况**，基本的事实 (终止条件)
2. **递推关系**，通过递推**拆解**递归到基本情况，给出问题解

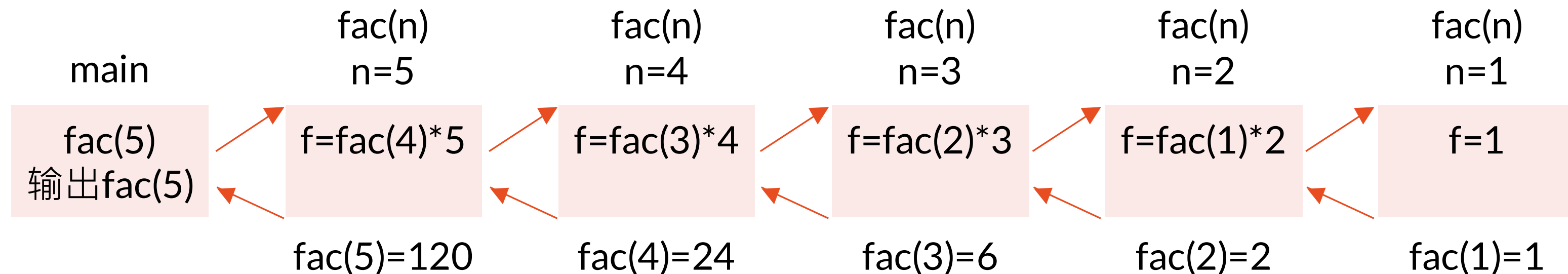
□ 在程序设计中，**递归**是一种问题求解的方法，即**函数自己调用自己**

递归示例：阶乘

```
int fac_loop(int n) {  
    int out = 1;  
    for (int i=n; i>0; i--)  
        out *= i;  
    return out;  
}
```

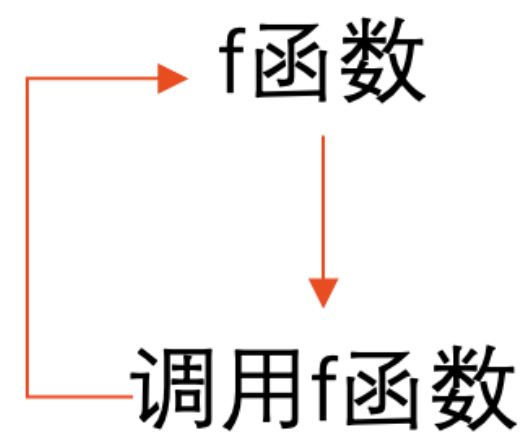
```
int fac(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n - 1);  
}
```

```
/*Facterial*/  
#include <stdio.h>  
  
int fac_loop(int);  
int fac(int);  
  
int main(void) {  
    /*请使用 0,1,2,5 测试程序*/  
    int n = 5;  
    printf("loop out %d\n", fac_loop(n));  
    printf("recursive out %d\n", fac(n));  
}
```

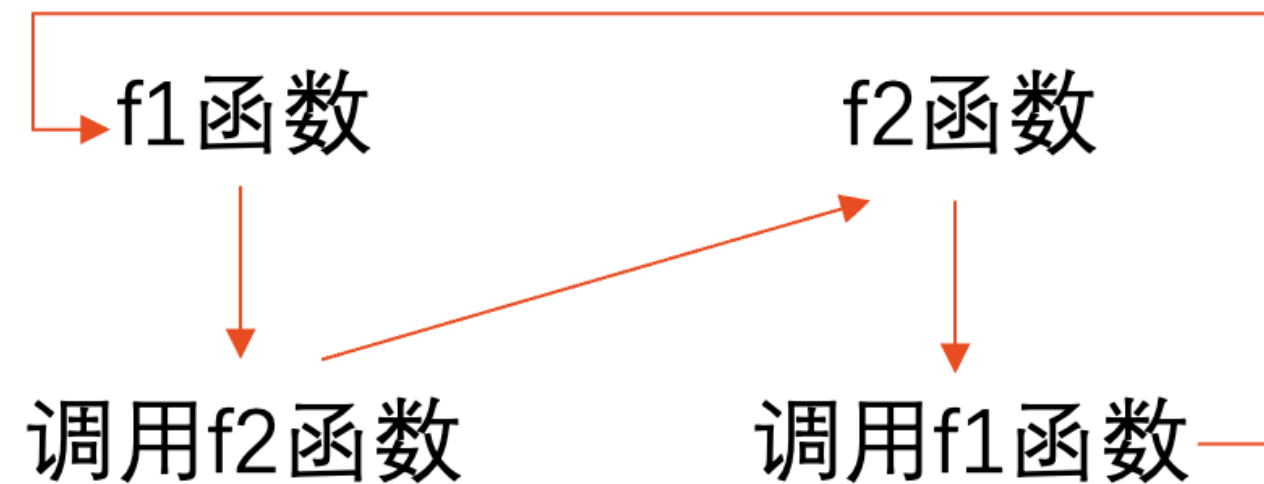


递归调用

- 在调用一个函数的过程中又出现**直接或间接地调用该函数本身**，称为函数的**递归调用**



直接递归



间接递归

- 程序中**不应出现无终止的递归调用**，而只应出现有限的、有终止的递归调用
- 这可以用 **if 语句来控制**，只有在某一条件成立时才继续执行递归调用，否则就不再继续

递归练习1：整数求和

用递归计算 $1+2+3+\dots+n$ 。

```
int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}

int main() {
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

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

递归练习2：斐波那契数列变体

题目描述

一只青蛙一次可以跳上 **1** 级台阶，也可以跳上 **2** 级。求该青蛙跳上一个 **n** 级的台阶总共有多少种跳法。

解题思路

$$F(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2 & \text{when } n = 2 \\ F(n-1) + F(n-2) & \text{when } n > 2 \end{cases}$$

输入 n	输出
1	1
2	2
3	3
5	8
10	89

尾递归

- 尾递归：递归调用处于函数末尾
- 尾递归是最简单的递归形式，**可用循环替代**

```
/* 计算Fibonacci数：递归实现 */  
long long f(int n){  
    if (n<=2) return 1;  
    return f(n-1) + f(n-2);  
}
```

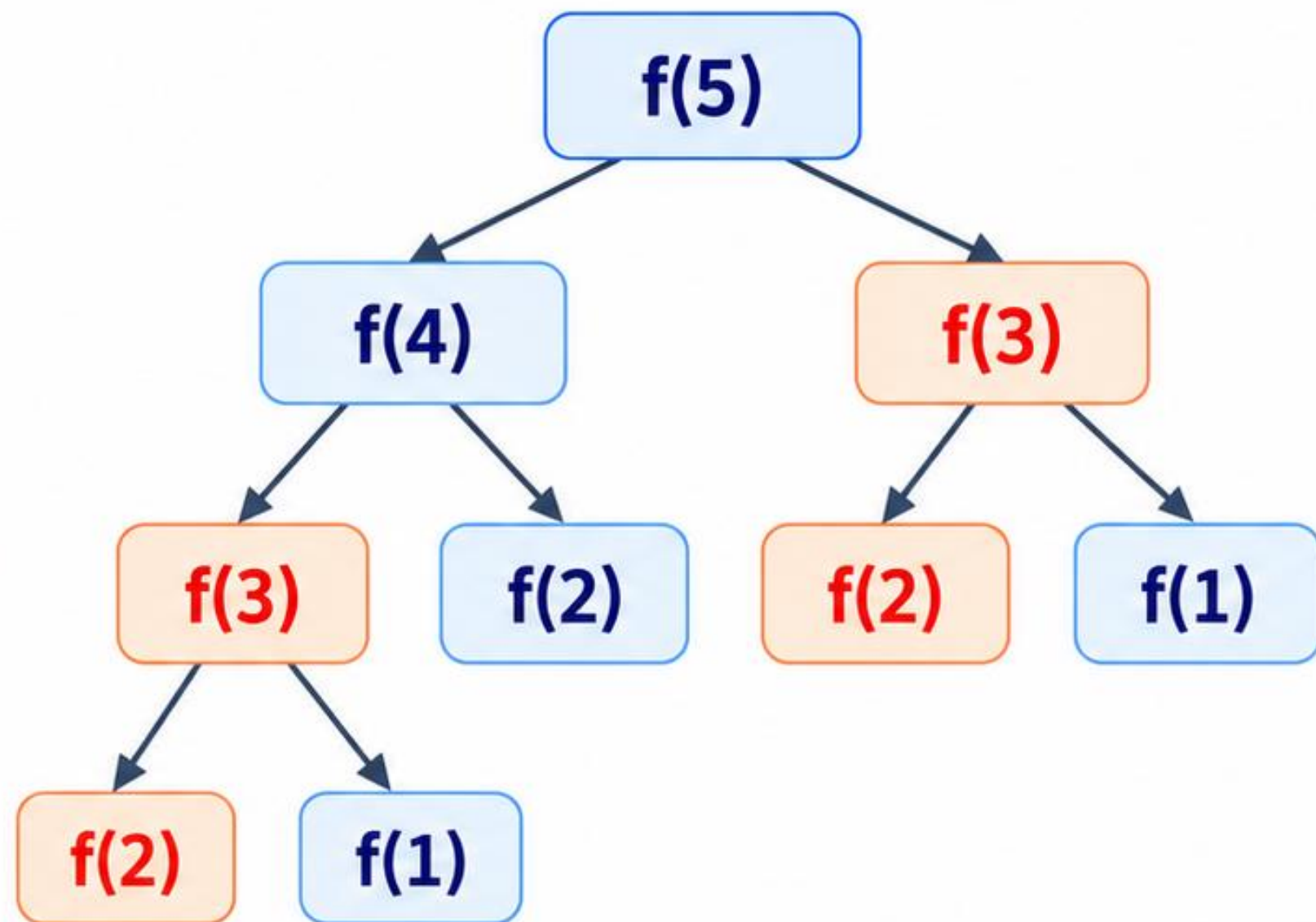
递归可能存在什么问题？

消耗内存空间！

```
/* 计算Fibonacci数：循环实现 */  
long long f(int n){  
    if (n<=2) return 1;  
    long long fn, f1=1, f2=1;  
    for (int i=1; i<=n-2; i++){  
        fn = f2 + f1;  
        f1 = f2;  
        f2 = fn;  
    }  
    return fn;  
}
```

递归的潜在开销：重复计算与内存占用

Fibonacci 递归调用示意（以 f(5) 为例）



Fibonacci 递归还会重复计算相同子问题，如 **f(3)**、**f(2)**。

每调用一次，
就新增一个
栈帧

调用栈（栈内存）



每个递归调用
都需要一个独立的
栈帧来保存：

- 参数 n
- 返回位置
(下一条要执行的指令)

递归练习3：最大公约数

□ 计算方法：辗转相除法

□ 算法定义：

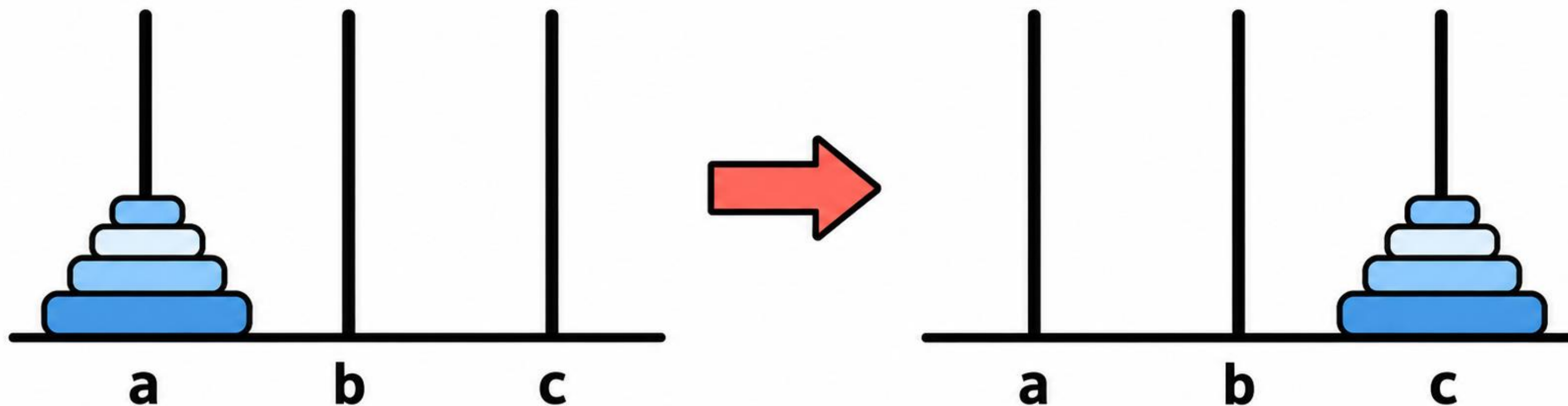
$$\text{gcd}(a, b) = \begin{cases} a & \text{if } b \% a == 0 \\ \text{gcd}(b \% a, a) & \text{if } b \% a \neq 0 \end{cases}$$

□ 例如： $\text{gcd}(98, 63) = \text{gcd}(63, 98) = \text{gcd}(35, 63) = \text{gcd}(28, 35) = \text{gcd}(7, 28) = 7$

```
int gcd(int a, int b) {  
    if (a % b == 0) return b;  
    else return gcd(b, a % b);  
}
```

递归练习4：汉诺塔 (Tower of Hanoi)

- 源于印度一个古老传说：大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始**按大小顺序重新摆放在另一根柱子上**。并且规定，任何时候，在**小圆盘上不能放大圆盘**，且在三根柱子之间**一次只能移动一个圆盘**。问应该如何操作？



递归练习4：汉诺塔 (Tower of Hanoi)

输入格式

输入一个正整数 n ，表示圆盘的数量。

输出格式

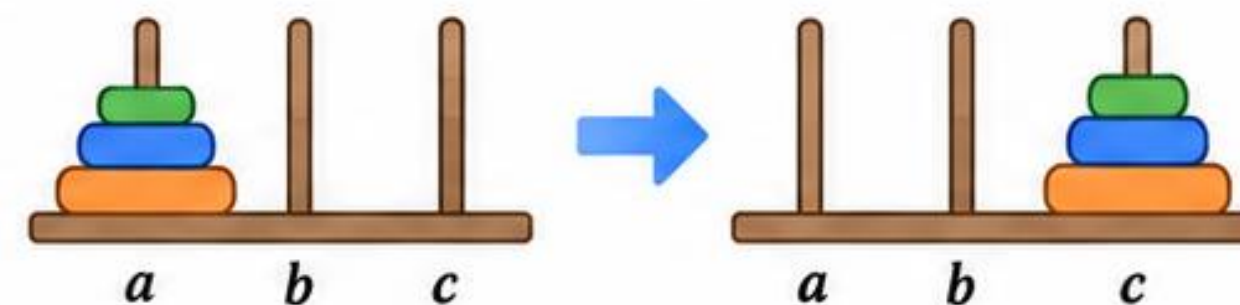
输出将 n 个圆盘从柱子 a 移动到柱子 c 的全部步骤。

每次移动占一行，格式为：起始柱 $\rightarrow$ 目标柱

```
3
a -> c
a -> b
c -> b
a -> c
b -> a
b -> c
a -> c
```

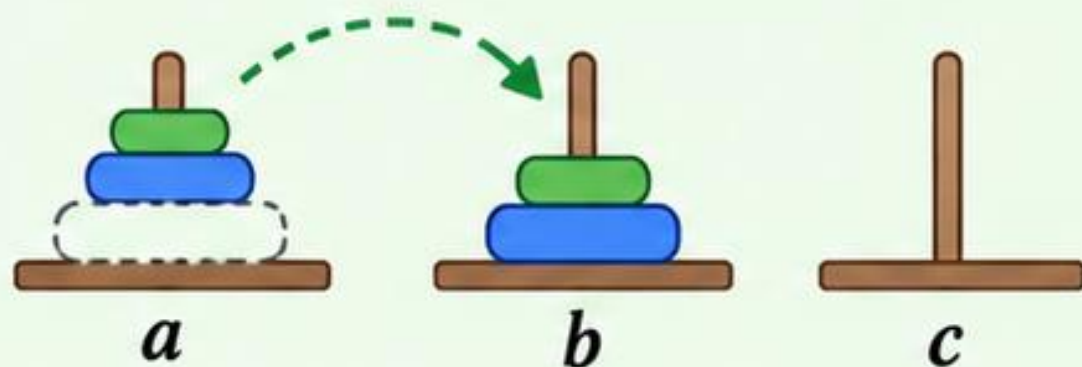
递归练习4：汉诺塔 (Tower of Hanoi)

大问题：移动 n 个圆盘



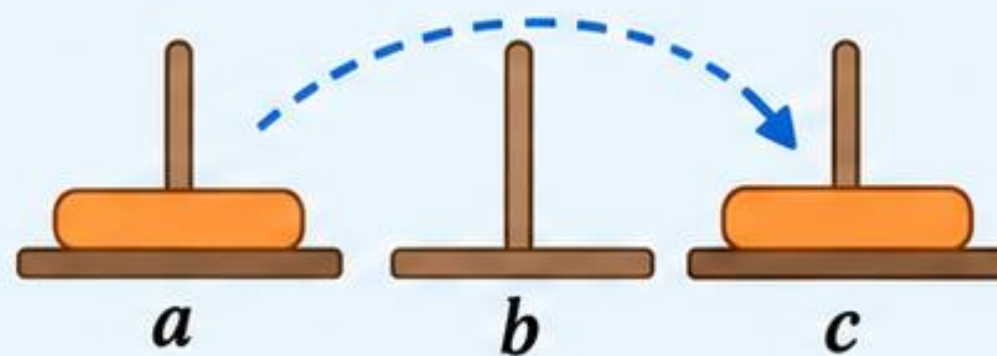
1 先处理上面的 $n-1$ 个圆盘

从 a 移到 b (借助 c)



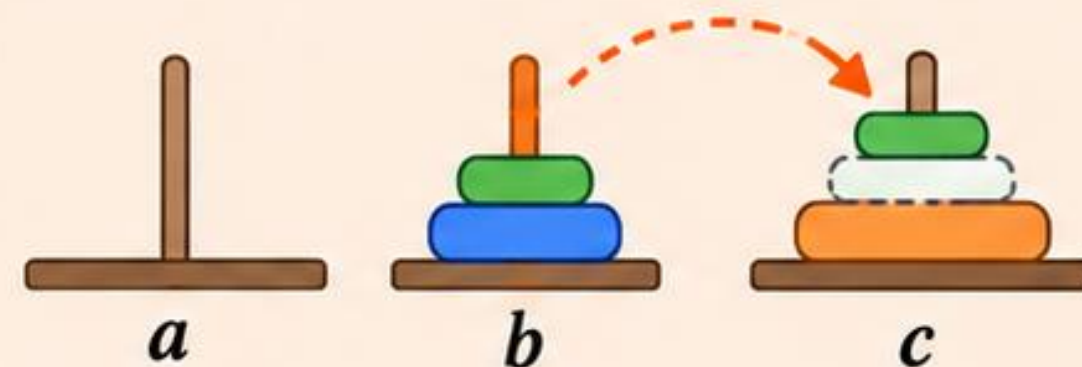
2 再处理最大的 1 个圆盘

从 a 移到 c



3 最后处理那 $n-1$ 个圆盘

从 b 移到 c (借助 a)



递归练习5：输出全排列

7-3 输出全排列

分数 10

[查看题目详情](#) [全屏浏览](#) [切换布局](#)

```
3
123
132
213
231
312
321
```

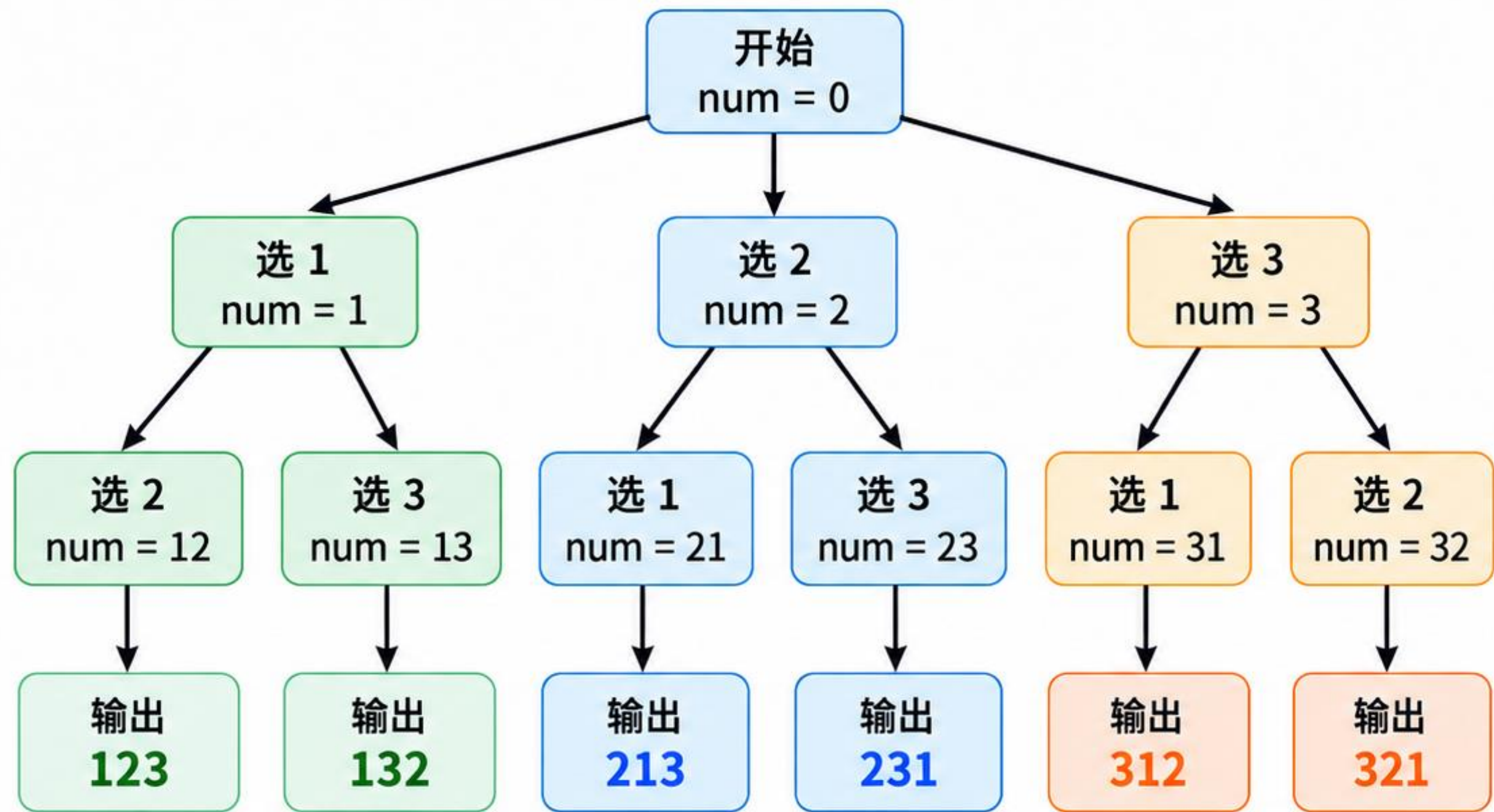
请编写程序输出前 n 个正整数的全排列 ($n < 10$)，并通过9个测试用例（即 n 从1到9）观察 n 逐步增大时程序的运行时间。

输入格式：

输入给出正整数 n (< 10)。

输出格式：

输出1到 n 的全排列。每种排列占一行，数字间无空格。排列的输出顺序为字典序，即序列 $a_1, a_2, \dots, a_n$ 排在序列 $b_1, b_2, \dots, b_n$ 之前，如果存在 k 使得 $a_1 = b_1, \dots, a_k = b_k$ 并且 $a_{k+1} < b_{k+1}$ 。



1

依次尝试 1~n

从 1 到 n，依次选择一个数字。

2

若数字未出现，就接到末尾

如果当前数字还没有使用，就把它接到 num 的末尾。

3

选满 n 位就输出

当已经选了 n 个数字时，此时的 num 就是一个全排列，输出它。

整数拼接的例子

num = 12，追加 3 → $12 * 10 + 3 = 123$

判断是否已使用的例子

判断 3 是否已用：依次看 num 的各位
例如 num = 12，依次检查 1、2，没有 3，说明 3 还未使用。



核心：

每次选一个未使用的数字，递归填写下一位。

主要内容

1. 函数的概念

2. 函数的定义、调用与声明

3. 递归

4. 头文件

5. 储存类别

头文件

- ❑ 为了确保函数**开发者**和**使用者**使用相同的声明和定义，C语言使用头文件(.h) 书写这些申明与定义
- ❑ 使用 **#include** 编译预处理指令将头文件插入即将编译的源代码中

1. 将 exercise.c 修改为:

```
#include <stdio.h>
#include "func.h"

int main(void) {
...
}
```

func.h 头文件

```
#pragma once
int fac_loop(int n);
int fac_rec(int n);
int jumping(int n);
int gcd(int a, int b);
...
```

func.c 原文件

```
#include <stdio.h>
#include "func.h"

int fac_loop(int n){
...
}
...
```

2. 将 **#include "func.h"** 添加入 main.c (exercise.c)

3. 添加 func.c 源文件，并重新编译运行

头文件

- 头文件通常包含：
 - 共享的函数声明
 - 全局变量声明与初始化
 - 定义的预处理符号或宏
- 头文件通过 `#include` 编译预处理指令添加到源代码文件中
 - 例如：源代码中需要使用内置库的求绝对值这个函数
 - 第一种方法，使用标准的头文件：`#include <math.h>`
 - 第二种方法，自己给出声明函数：`double fabs (double);`
- 注意：
 - 自定义的头文件使用"`"`号；
 - 头文件第一行使用 `#pragma once`，指令确保编译仅展开一次

主要内容

1. 函数的概念

2. 函数的定义、调用与声明

3. 递归

4. 头文件

5. 储存类别

变量属性

变量属性

作用域

变量在哪里可以被访问?
空间范围

```
void f(void) {  
    int x = 10;  
}  
  
int main(void) {  
    printf("%d", x);  
}
```

main中能否访问x?

生存期

变量在什么时候存在?
时间范围

```
void f(void) {  
    int x = 10;  
}  
  
int main(void) {  
    f();  
}
```

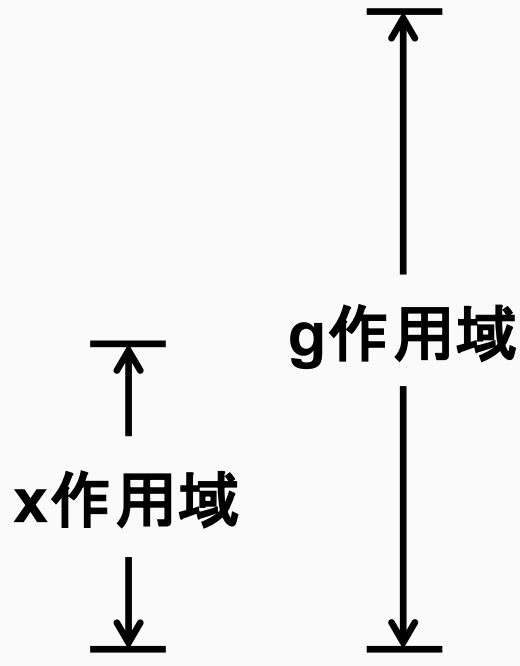
f()调用结束后, x是否还存在?

局部变量与全局变量

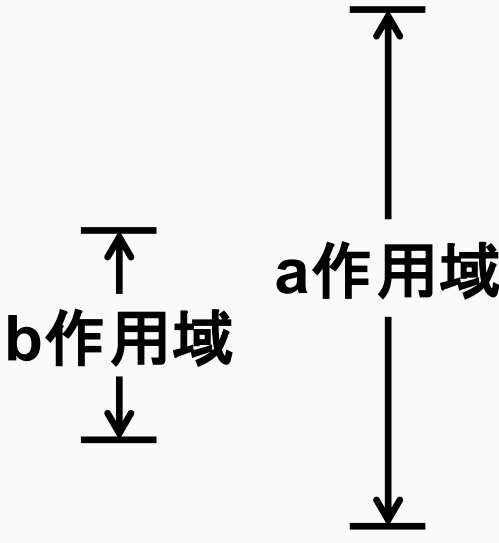
名称	定义位置	默认作用域范围
局部变量	函数或代码块内部	当前函数/代码块
全局变量	所有函数之外	从声明/定义处到文件末尾

```
int g;//全局变量

int main() {
    int x;//局部变量
    return 0;
}
```



```
int main() {
    int a = 10;//局部变量
    if (a > 0) {
        int b = 20;//局部变量
    }
}
```



局部变量与全局变量的生存期

```
int g; //全局变量
```

```
int main() {  
    int x; //局部变量  
    return 0;  
}
```



```
int main() {
```

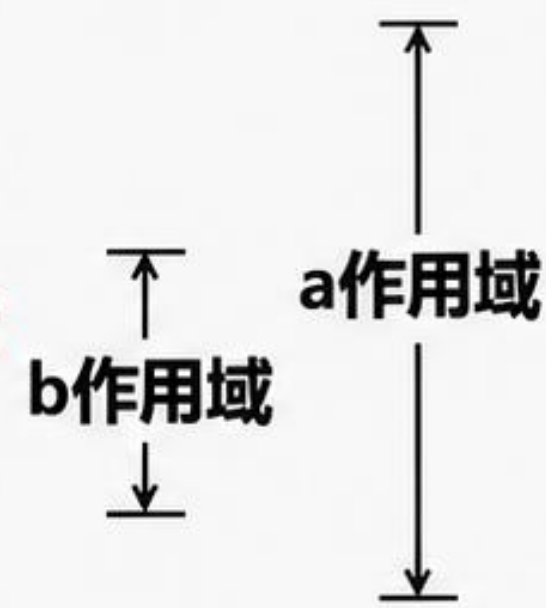
```
    int a = 10; //局部变量
```

```
    if (a > 0) {
```

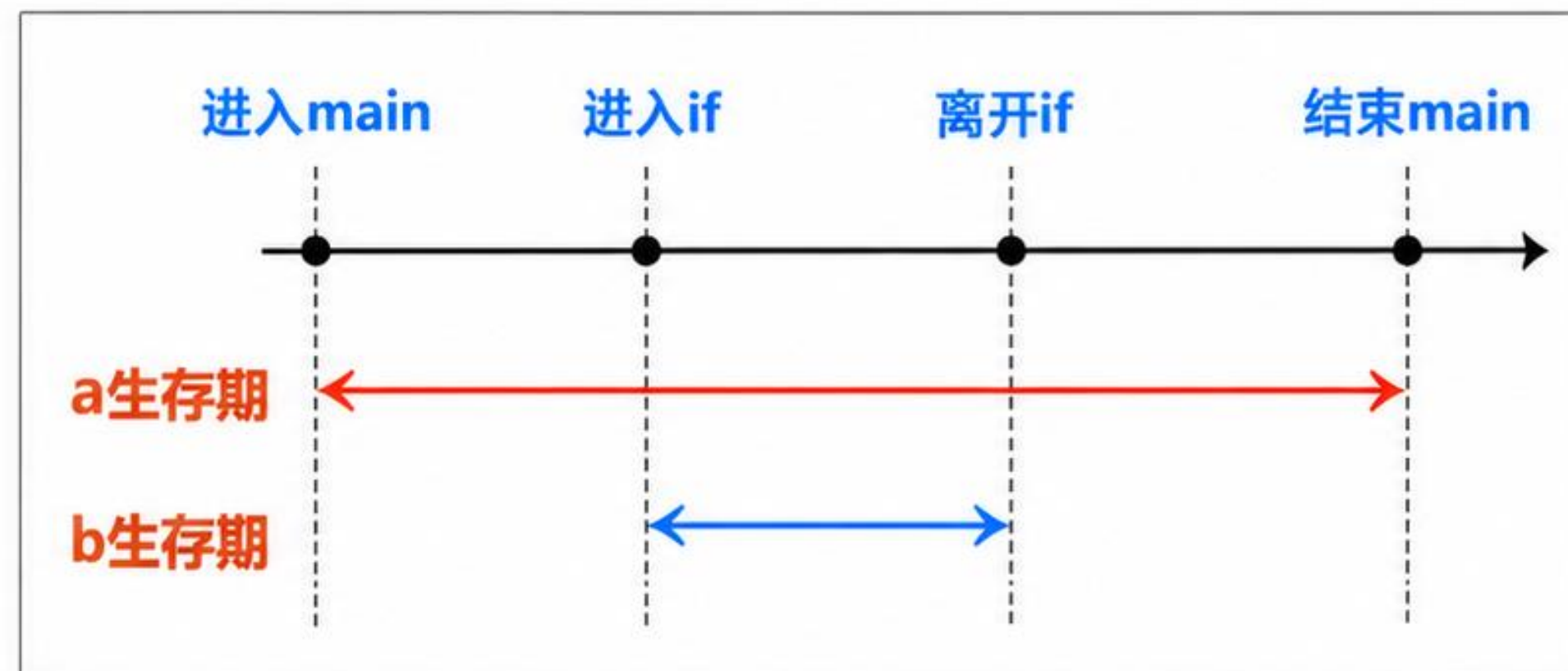
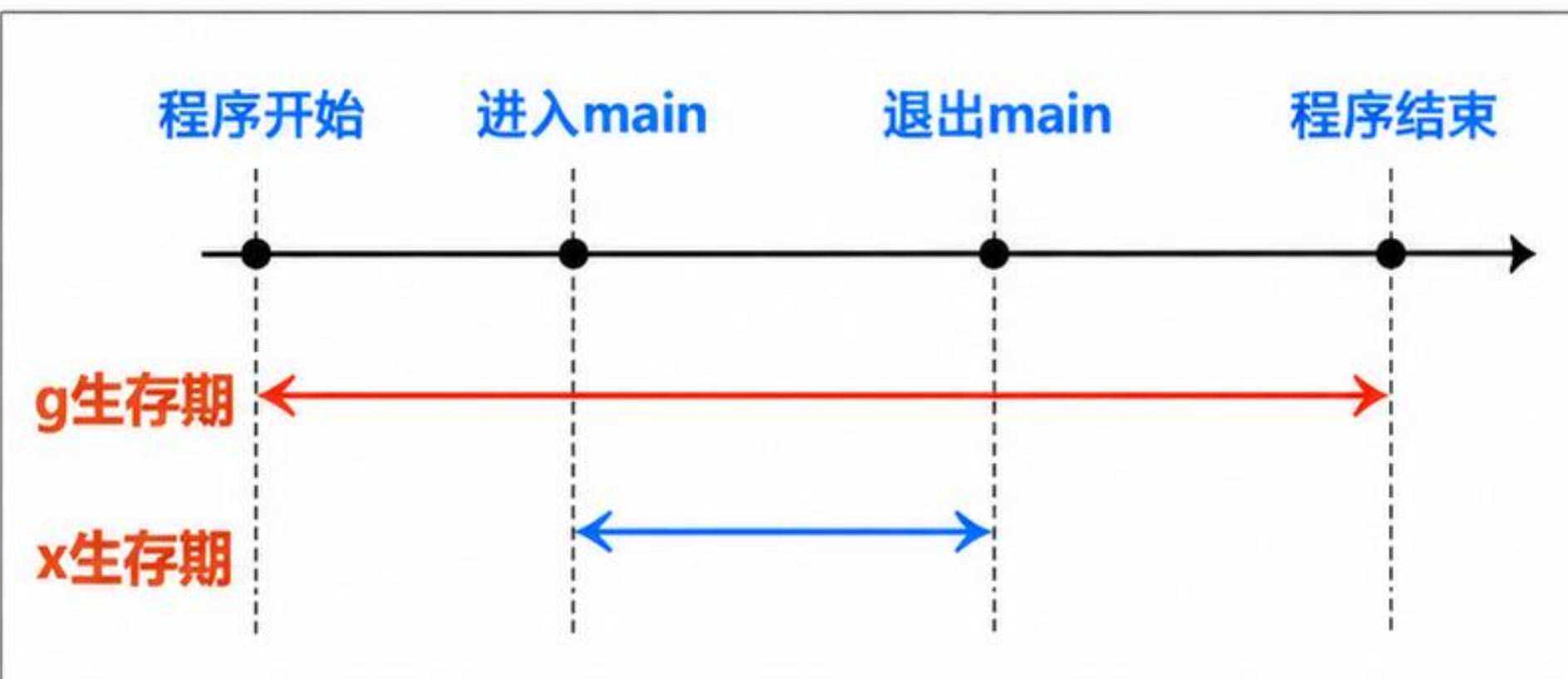
```
        int b = 20; //局部变量
```

```
    }
```

```
}
```



生存期



动态储存与静态储存

□ 从变量的生存期的角度，储存方式分为**动态储存**和**静态储存**

动态存储

```
void f(void)
{
    int x;
}
```



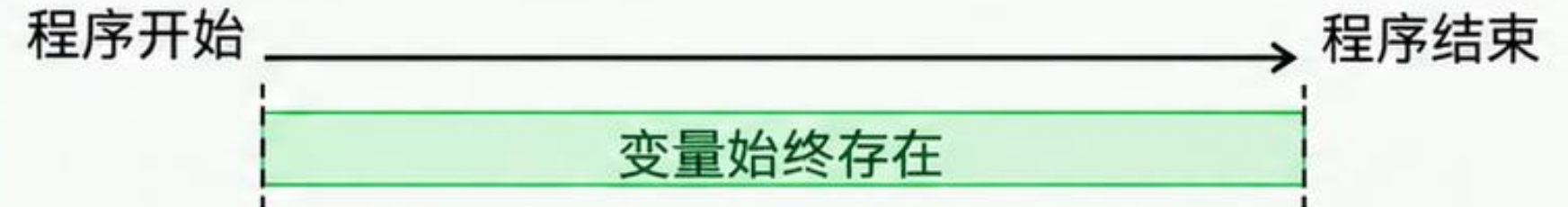
- 调用函数时产生
- 函数结束时消失

典型对象：形参、局部变量

静态存储

```
int g;
```

```
void f(void)
{
    static int x;
}
```



- 程序运行期间一直存在
- 程序结束时才释放

典型对象：全局变量、static 变量

动态存储 = 临时存在，静态存储 = 全程保留。

动态储存与静态储存

□ 数据可存放在内存的**静态存储区**或**动态存储区**中

□ 静态存储区

■ 存放：全局变量、static变量

■ 特点：程序开始时分配，程序结束时释放

□ 动态存储区

■ 存放：函数形参、auto局部变量、函数调用现场和返回地址等

■ 特点：函数调用时分配、函数结束时释放；同一函数多次调用时，局部变量的存储地址可能不同



存储类别

- ❑ 在C中，每一个变量和函数都有两个属性：**数据类型**和**存储类别**
- ❑ **存储类别**指出数据在内存中存储的方式，如静态存储或动态存储，影响变量的**生存期**和**作用域**
- ❑ C的存储类别包括4种

存储类别	关键字	典型对象	存储方式
自动变量	auto	函数形参、普通局部变量	动态存储
静态变量	static	全局变量、静态局部变量	静态存储
寄存器变量	register	局部变量	动态存储
外部变量	extern	全局变量、其他文件中定义的变量和函数	静态存储

局部变量的储存类别

自动变量

□ 自动变量

- 说明符：**auto**
- 定义位置：只能在**函数或代码块**内部
- 作用域：所在函数或代码块
- 生存期：进入所在块时产生，退出时消失
- 存储方式：动态存储

□ 普通局部变量默认就是auto变量，**auto通常省略**

```
void f(void) {  
    int x = 10;           // auto变量  
    auto int y = 20;      // 等价于：int y = 20;  
}
```

寄存器变量

□ 寄存器变量

- 说明符: **register**
- 定义位置: 只能在**函数或代码块**内部
- 作用域: 所在函数或代码块
- 生存期: 进入所在块时产生, 退出时消失
- 存储方式: 存放在**CPU的寄存器**中 (无内存地址)
- 特点: 访问速度快, 适用于频繁使用的局部变量

□ 现在的编译器能够自动识别频繁使用的变量, 并将其放在寄存器中, 通常不需要显式使用register

```
void f(void) {  
    register int i;  
    int sum = 0;  
    for (i = 0; i < 1000000; i++)  
        sum += i;  
}
```

静态局部变量

□ 静态局部变量

- 说明符: **static**
- 定义位置: 在函数或代码块内部
- 作用域: 所在函数或代码块
- 生存期: 程序**整个运行期间**
- 初始化: 程序整个运行期间
- 特点: 函数返回后值保留, 下次调用继续使用

□ static修饰局部变量改变的是生存期, 不是作用域

示例代码

```
void f(void) {  
    static int cnt = 0; // 只初始化一次  
    cnt++;  
    printf("%d\n", cnt);  
}
```

多次调用: 变量值会继续保留

第 1 次调用
cnt = 1



第 2 次调用
cnt = 2



第 3 次调用
cnt = 3

仍是局部变量: 只能在所在函数或代码块中访问, 其他函数不能引用。

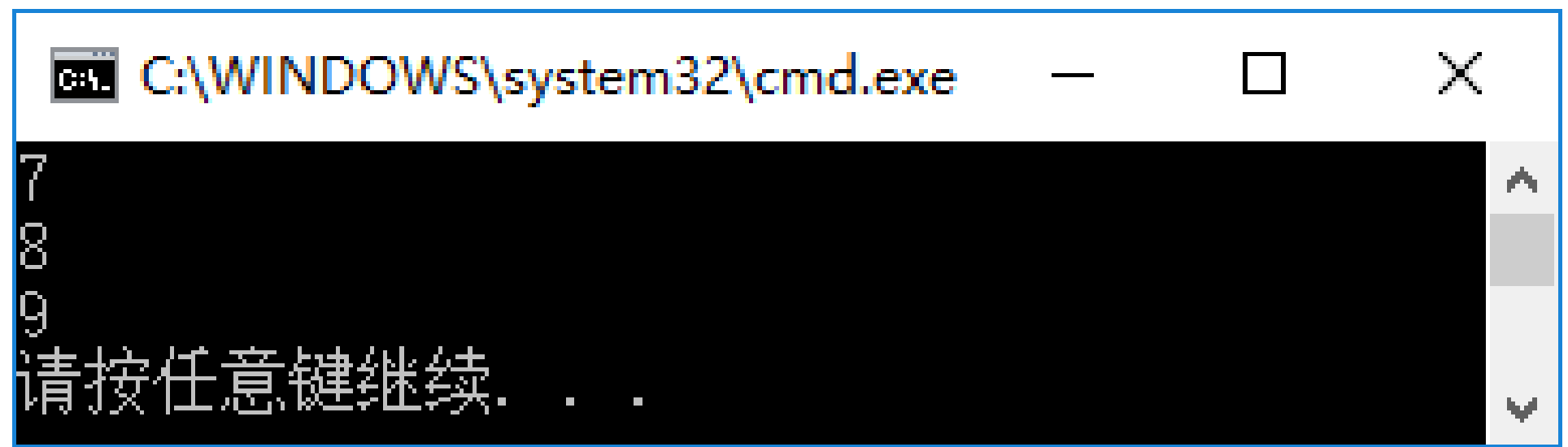
练习1：以下程序会输出什么？

```
#include <stdio.h>
int main(void) {
    int f(int);
    int a = 2, i;
    for(i=0; i<3; i++)
        printf("%d\n", f(a));
    return 0;
}

int f(int a) {
    auto int b = 0;
    static int c = 3;
    b = b + 1;
    c = c + 1;
    return a+b+c;
}
```

静态变量与自动变量的值的比较分析

第几次调用	调用时初值		调用结束时的值		
	b	c	b	c	a+b+c
第1次	0	3	1	4	7
第2次	0	4	1	5	8
第3次	0	5	1	6	9

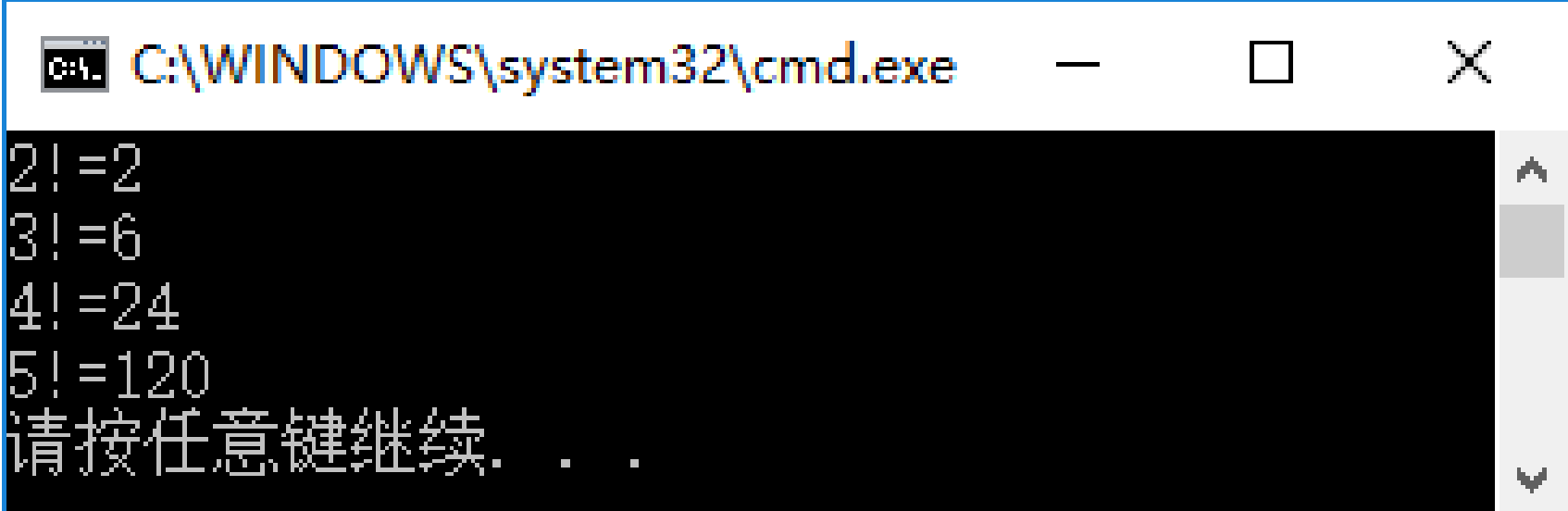


练习1：以下程序会输出什么？

```
#include <stdio.h>

int main() {
    int fac(int n);
    int i;
    for(i=1; i<=5; i++)
        printf("%d!=%d\n", i, fac(i));
    return 0;
}

int fac(int n) {
    static int f = 1;
    f = f * n;
    return f;
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text: '2!=2', '3!=6', '4!=24', '5!=120', and '请按任意键继续. . .'. The text is displayed in a monospaced font on a black background.

三种局部变量

自动局部变量 (auto)

- 说明符: **auto** (通常省略)
- 作用域: 所在函数/代码块
- 生存期: 调用期间存在
- 存储位置: 动态存储区

静态局部变量 (static)

- 说明符: **static**
- 作用域: 所在函数/代码块
- 生存期: 程序整个运行期间
- 存储位置: 静态存储区

寄存器局部变量 (register)

- 说明符: **register**
- 作用域: 所在函数/代码块
- 生存期: 调用期间存在
- 存储位置: CPU寄存器 (建议)

现代编译器常会自动优化, register 实际上较少显式使用。

3种局部变量的存储位置不同: **auto** 在动态存储区, **static** 在静态存储区, **register** 建议放在寄存器中。

使用建议: 若非必要, 不要多用静态局部变量。

原因: 长期占用内存, 不如动态存储节省空间; 调用次数多时, 也容易弄不清当前值。

全局变量的储存类别

全局/外部变量

- ❑ 全局变量是**定义在所有函数之外**的变量
- ❑ **作用域**：从**定义处**开始，到本文件末尾
- ❑ **生存期**：程序**整个运行期间**
- ❑ **存储方式**：静态存储区
- ❑ **特点**：可被其作用域内的多个函数引用

示例代码

```
int count = 0;    // 全局变量

void f(void) {
    count++;
}

int main(void) {
    f();
    return 0;
}
```

全局变量：程序运行期间始终存在



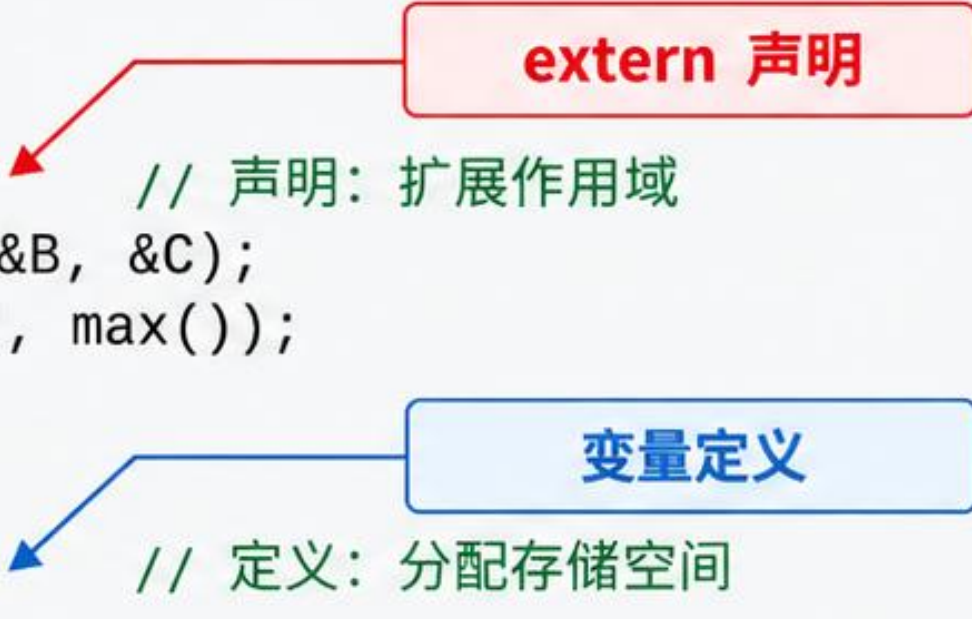
用extern拓展全局变量作用域

- **extern**可用于声明一个已在别处定义的变量或函数，使其可以在当前位置被引用，但本身不分配新的存储空间

代码节选

```
int max(void);
int main(void) {
    extern int A, B, C;    // 声明：扩展作用域
    scanf("%d%d%d", &A, &B, &C);
    printf("max is %d\n", max());
}

int A, B, C;              // 定义：分配存储空间
int max(void) { ... }
```



extern：把“可使用的位置”提前到声明处

文件开头

extern 声明
可在此处使用

变量定义
真正分配空间

文件末尾

可见范围被提前



- 提倡将全局变量的定义放在引用它的所有函数之前

用extern拓展全局变量作用域

- **extern**也可将**其他文件**中已定义的全局变量的可见范围**扩展到当前文件**，使多文件能够共享同一全局变量

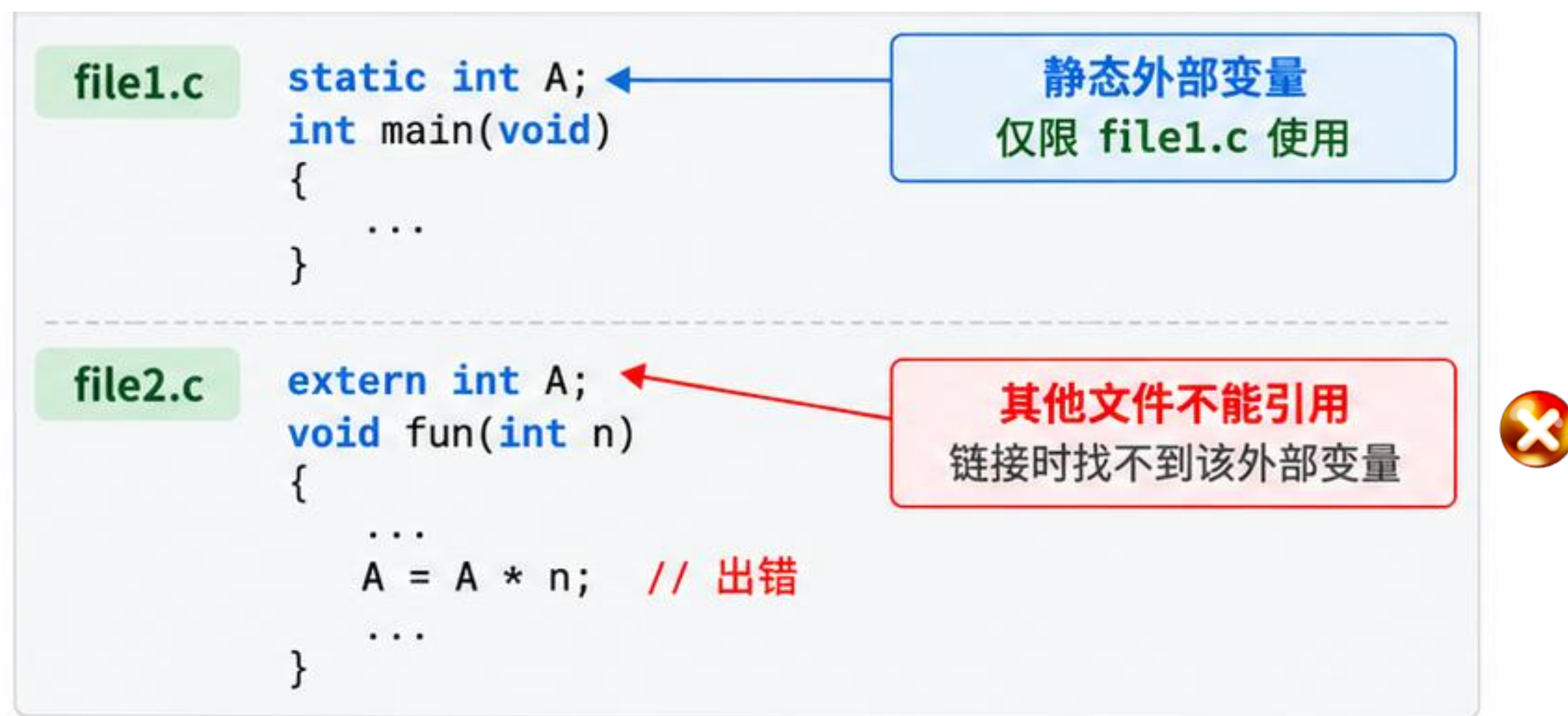
```
// file1.c
#include <stdio.h>
int A;
int main() {
    int power(int);
    int b = 3, c, d, m;
    printf("enter the number and its power:\n");
    scanf("%d%d", &A, &m);
    c = A * b;
    printf("%d*d=%d\n", A, b, c);
    d = power(m);
    printf("%d**%d=%d\n", A, m, d);
    return 0;
}
```

```
// file2.c
extern A;
int power(int n) {
    int i, y = 1;
    for(i=1; i<=n; i++)
        y *= A;
    return y;
}
```

```
C:\WINDOWS\system32\cmd.exe
enter the number a and its power m:
13, 3
13*3=39
13**3=2197
请按任意键继续. . .
```

用static限制全局变量作用域

- ❑ 在定义全局变量时加上**static**，可以把它限制为**仅在本文件**中使用
- ❑ 变量**仍具有全程生命期**，但不能被其他文件引用



- ❑ static修饰全局变量限制作用域，不改变生存期

static对局部变量与全局变量的作用

- ❑ static修饰局部变量：**改变生存期**
- ❑ static修饰全局变量：**限制文件作用域**

static 局部变量

说明	定义在函数或代码块内部
改变	生存期
作用	分配在静态存储区，程序运行期间 始终存在
不变	作用域仍限于所在函数或代码块

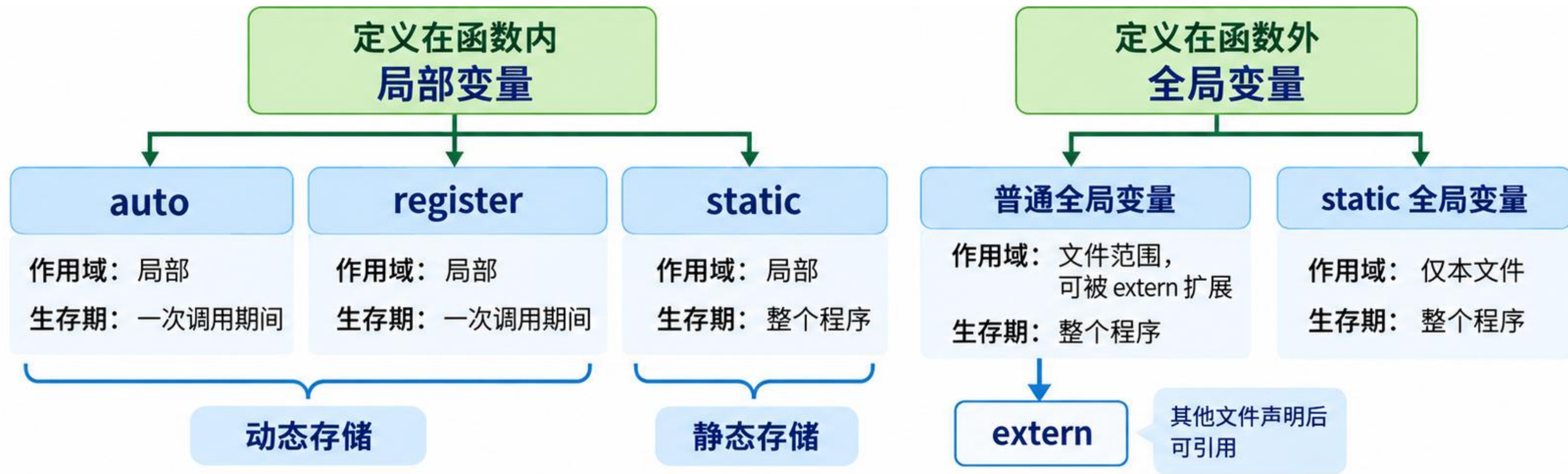
```
void f(void) {  
    static int cnt = 0;  
    cnt++;  
}
```

static 全局变量

说明	定义在所有函数之外
改变	作用域 / 跨文件可见性
作用	只限本文件使用 ，其他文件不能引用
不变	生存期仍是整个程序运行期间

```
static int A;  
  
int main(void) {  
    ...  
}
```

储存类别总结



局部 / 全局 主要说明变量在哪里定义、哪里可以访问；**动态 / 静态** 主要说明变量存在多久。

补充

内部函数与外部函数

□ 根据函数是否被其他源文件调用，可分为内部函数和外部函数

内部函数 (static 函数)

- 写法: **static** int fun(int a, int b);
- 作用: 只能被**本文件中的**函数调用
- 适合: 只供**本模块内部使用的**辅助函数

```
static int fun(int a, int b)
{
    return a > b ? a : b;
}
```

外部函数 (extern 函数)

- 写法: **extern** int fun(int a, int b);
- 作用: 可以被**其他文件**调用
- 说明: 函数默认具有**外部属性**，extern 通常省略

```
extern int fun(int a, int b)
{
    return a > b ? a : b;
}
```



作业

- 阅读《C程序设计》第7.1-7.6、7.8-7.11节
- 完成 PTA 本周习题

谢谢