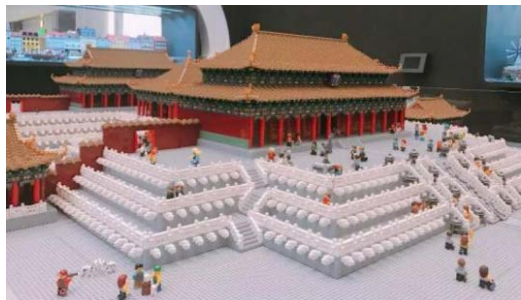


第五章 (Chapter 5)

函数

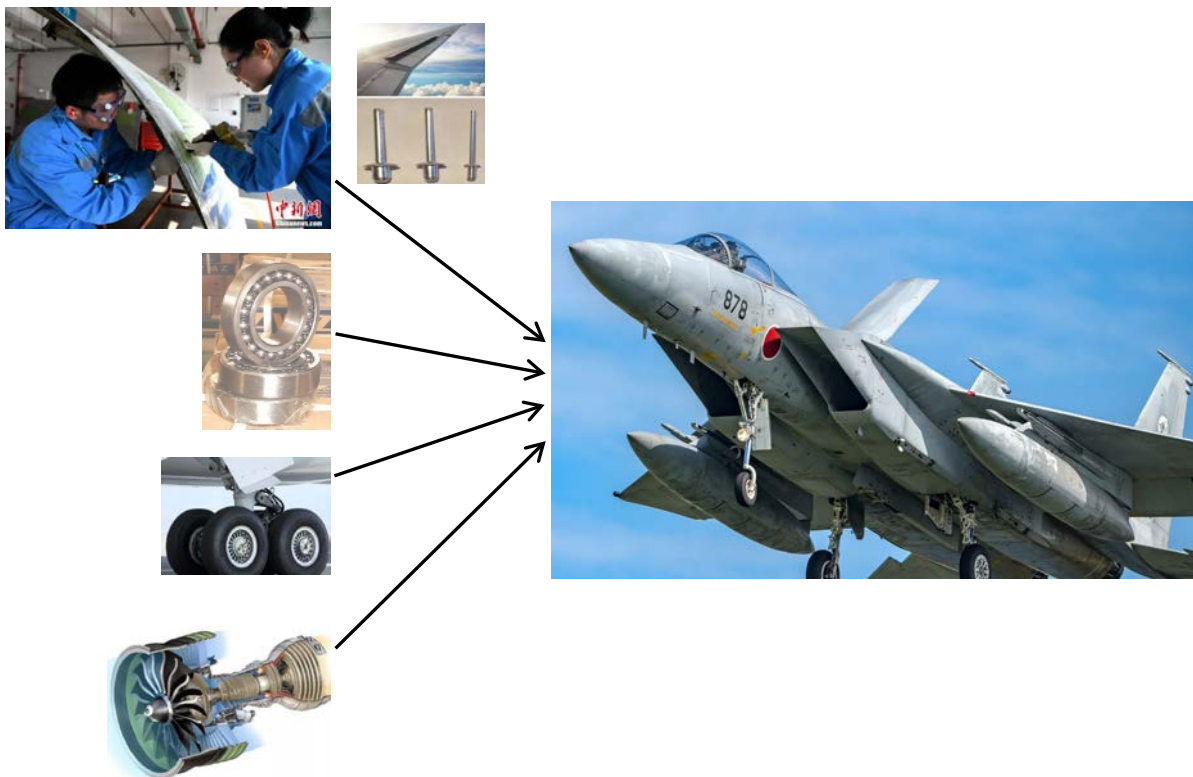
function



北京航空航天大学

李莹

2024 Fall



函数 function

Make interface **easy** to use correctly, **hard** to use incorrectly.

——Scott Meyers

接口应该这样设计，当正确使用时编程很容易，而错误使用时编程很困难。

为什么使用函数？

例4-27 输入yyyymmdd，查询是星期几

D Deadline 的艺术 (复活版)

时间限制: 1000ms 内存限制: 65536kb

通过率: 619/981 (63.10%) 正确率: 619/2936 (21.08%)

$$D = \left\lfloor \frac{c}{4} \right\rfloor - 2c + y + \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + d - 1$$
$$W = D \bmod 7$$

2024Fall
航C4-D

存在问题:

- 代码较长，可读性不好、可维护性差。
- 代码无法重复使用、编程效率低。

```
#include <stdio.h>
int main()
{
    int c, y, m, w, d, longday = 1;
    char ch;

    printf("Query what day a certain date is\n");
    printf("Note: the format of the day is like 20220929\n");
    printf("The input is between 101 and 99991231\n");
    printf("Input date (or -1 to quit):\n\n");

    while(1)
    {
        int in_flag = scanf("%d", &longday);

        if(longday == -1 || in_flag <= 0)
            break;

        if(!(longday >= 101 && longday <= 99991231))
        {
            printf("Wrong input format, try again!\n\n");
            continue;
        }

        y = longday/10000;
        m = (longday%10000)/100;
        d = longday%100;

        if(m<3)
        {
            y--;
            m += 12;
        }

        c = y/100;
        y = y%100;
        w = (c/4 -2*c + y + y/4 + (26*(m+1))/10 + d - 1)%7;
        if(w<0)
            w += 7;

        printf("The day is: ");
        switch(w)
        {
            case 0:
                printf("Sun");
                break;
            case 1:
                printf("Mon");
                break;
            case 2:
                printf("Tue");
                break;
            case 3:
                printf("Wed");
                break;
            case 4:
                printf("Thu");
                break;
            case 5:
                printf("Fri");
                break;
            case 6:
                printf("Sat");
                break;
        }
        printf("\n\n");
    }
    return 0;
}
```

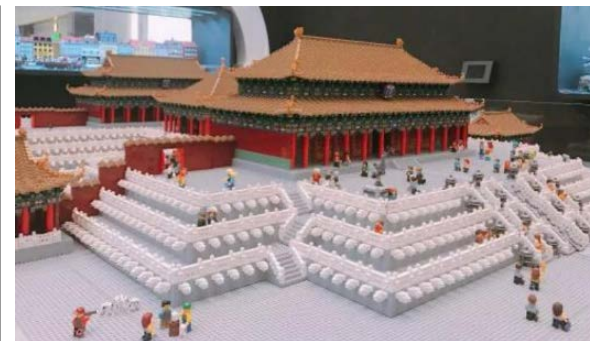
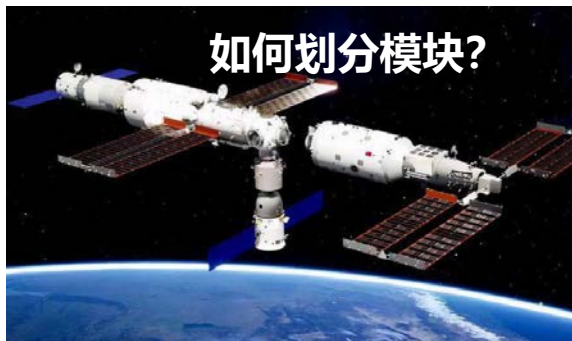
一个并不算长的程序，
读起来比较麻烦？

30行左右的一个函数
比较好（正常阅读字
号和间距时，代码长
度占满一屏）。这个
代码70行，不太好！

第五章 函数

学习要点

1. 常用的标准库函数
2. 自定义函数
3. 函数原型、函数定义
4. 函数调用与返回
5. 函数原型与实参的类型转换
6. 局部变量、全局变量
7. 变量的存储类、作用域
8. 模块化编程的思想
9. 递归函数简介
10. 良好的软件工程思想与高性能编程的辩证关系

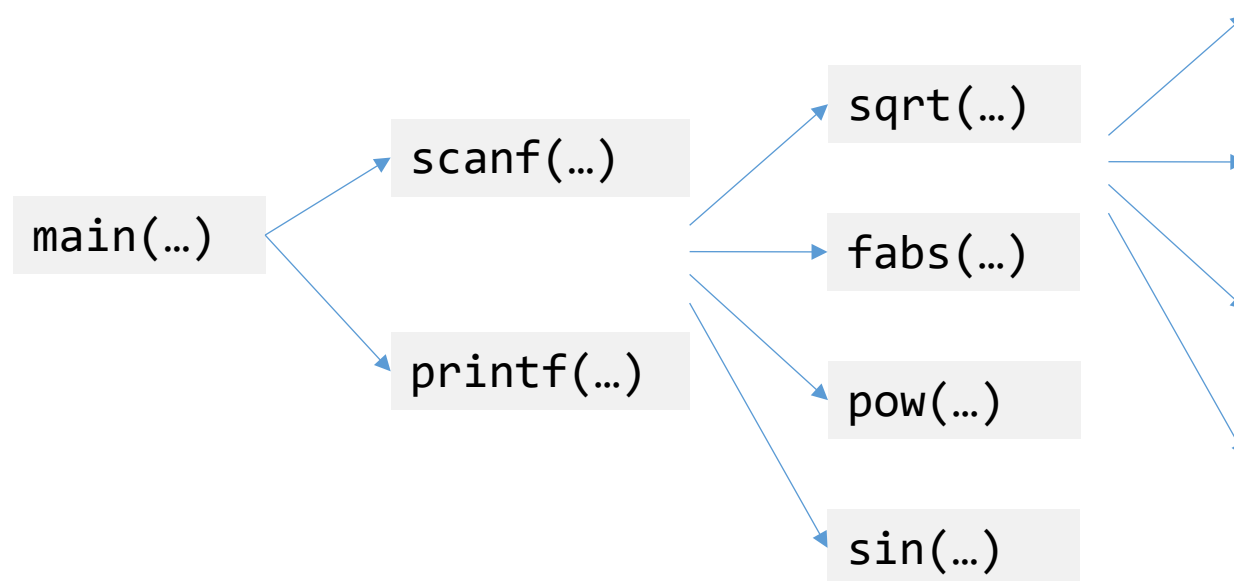


积木搭的房子、飞机，虽然很像，但还是假的。函数设计的真正目的是让软件快速开发，有效运行，如，电脑运行、空间站翱翔，飞机驰骋！

什么是函数?

```
#include <stdio.h>
reType1 function1(...);
reType2 function2(...);
...
reTypeX functionX(...);
int main()
{
    ...
    scanf(...);
    ...
    function1(...)
    function2(...)
    pow(...);
    functionX(...)
    ...

    printf(...);
    return 0;
}
```



- 函数是一个具有特定功能的若干行代码集合。
- C语言程序设计的主要工作就是设计函数。
- C语言程序设计是函数式程序设计。

5.1 函数

例5-1：利用数学库函数 pow 计算 x 的 y 次方 (x, y 为整数) (power)

```
double ans = 1.0;
int n, days, base=1;
scanf("%d%d", &base, &days);

// 用库函数pow
for (n = 1; n <= days; n++)
{
    ans = pow(1.01, n); // day_n = day_(n-1) * (1+r) = base*(1+r)^n
    printf("%d*pow(1.01,%d) = %.10f\n", base, n, base*ans);
}
```

```
double ans = 1.0;
int n, days, base=1;
scanf("%d%d", &base, &days);

// 不用库函数，自己写循环实现
for (n = 1; n <= days; n++)
{
    ans = ans * 1.01; // 回想part4的复利计算公式
    printf("%d*pow(1.01,%d) = %.10f\n", base, n, base*ans);
}
```

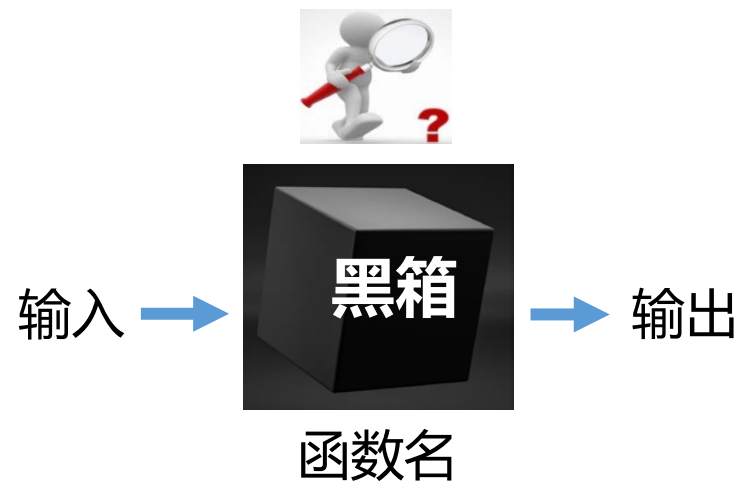
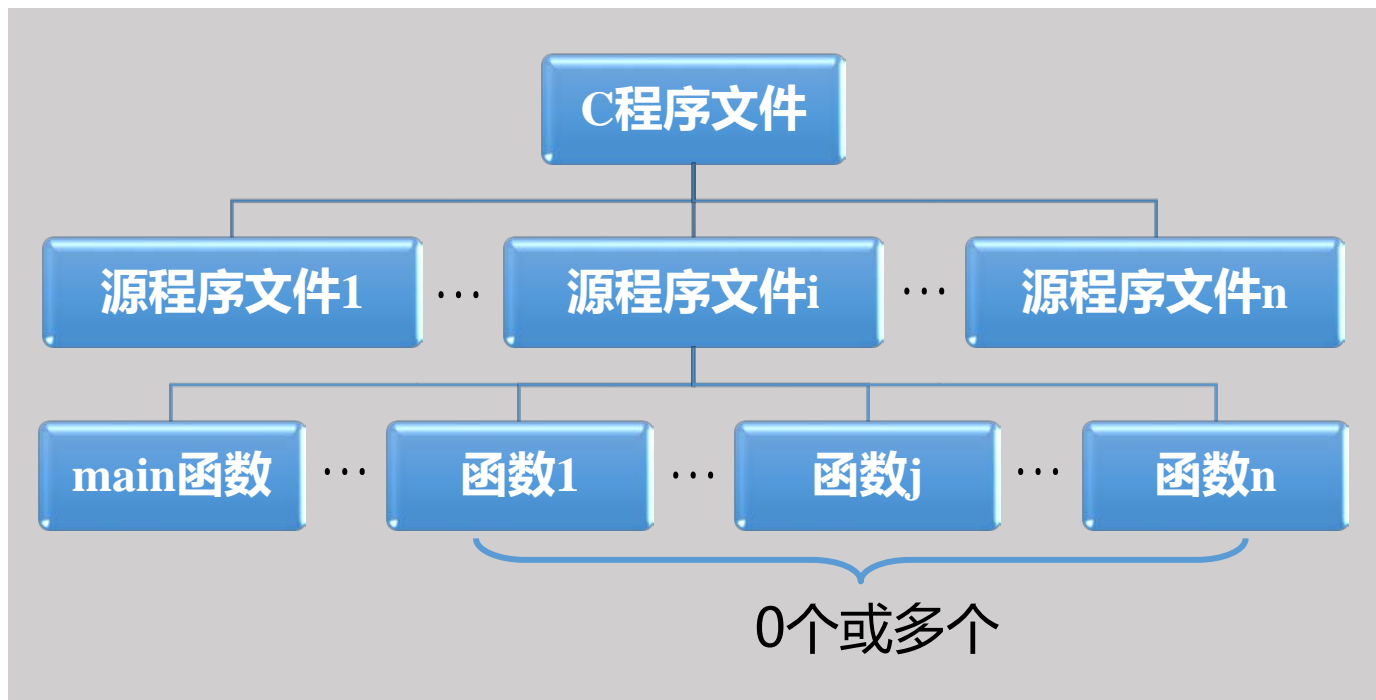
```
// 数学库函数pow()原型
double pow(double x, double y);
```

函数的一些基本概念：

- 函数是什么？
- 函数如何定义？
- 函数如何调用？
- 函数如何执行？
- 函数如何返回？

函数含义

- **从使用角度**：函数是一个黑盒。使用时，只需了解“函数名称、输入数据、输出结果、实现功能”等信息，不必关心具体的设计方法，如`pow(x, y)`, `sin(x)`等。
- **从结构角度**：函数是一段具有特定功能的、按固定顺序执行的、可重用语句块
- **从程序角度**：程序是函数集合体，每个函数都是一个独立模块。



```
for (n = 1; n <= N; n++)  
{  
    s = pow(x, n); // 调用库函数  
    .....  
}
```

5.2 定义函数

例5-2：自定义函数pow计算 x 的 y 次方 (x, y 为整数)

```
#include <stdio.h>
long long LL_pow(int, int); ← 函数声明

int main()
{
    int x, y, N;
    scanf("%d%d", &x, &N);

    for (y = 1; y <= N; y++)
        printf("pow(%d, %d) = %lld\n", x, y, LL_pow(x,y));

    return 0;
}

long long LL_pow(int x, int y) ← 函数定义 (实现)
{
    long long ans = 1;
    int i;
    for (i = 1; i <= y; i++)
        ans *= x;
    return ans;
}
```

白箱：自定义函数实现

函数类型

- **自定义函数**：用户根据特定需求自己实现，不用包含头文件，但需要用户自行定义函数
- **标准函数（库函数）**：系统自带，可直接调用，但要用include命令引入包含其函数声明的头文件

```
#include <stdio.h>
#include <math.h> ← 函数声明位置

int main()
{
    int x, y, N;
    scanf("%d%d", &x, &N);

    for( y = 1; y <= N; y++ )
        printf("pow(%d,%d) = %.0f\n", x, y, pow(x,y));

    return 0;
}
```

黑箱：库函数

定义函数

例5-2：自定义函数pow计算 x 的 y 次方 (x, y 为整数)

```
#include <stdio.h>
long long LL_pow(int, int); ← 函数声明

int main()
{
    int x, y, N;
    scanf("%d%d", &x, &N);

    for (y = 1; y <= N; y++)
        printf("pow(%d, %d) = %lld\n", x, y, LL_pow(x, y));

    return 0;
}

long long LL_pow(int x, int y) ← 函数定义 (实现)
{
    long long ans = 1;
    int i;
    for (i = 1; i <= y; i++)
        ans *= x;
    return ans;
}
```

白箱：自定义函数实现

函数定义

- **含义**：用于函数实现的一段独立代码，包含函数头和函数体两部分
- **函数头**：指定了函数的返回值类型、函数名称和形式参数列表等信息
- **函数体**：用花括号{}括起来的若干条语句，它实现了函数的具体功能

返回值类型 函数名(形式参数列表)

{

函数体;

}

函数头

函数头

函数头指定函数的返回值类型、函数名称和形式参数列表等信息

- (1) **返回值类型**: 是**被调函数**向**主调函数** (如 `main`) 返回值的数据类型, 一般和 `return` 语句返回值的类型相同。如果函数没有返回值, 则设置为 `void`。
- (2) **函数名**: 有意义的标识符, 详细的命名规则参见第二讲的变量命名规则。
- (3) **形式参数列表**: 用逗号分隔的 <数据类型> <形参名称> 对, 说明参数数量、顺序和类型。函数没有形参时, 函数名 `f` 后的括号内容为空或 `void`, 即 `f()` 或 `f(void)`。

```
long long LL_pow(int x, int y)
{
    long long ans = 1;
    int i;
    for (i = 1; i <= y; i++)
        ans *= x;
    return ans;
}
```

类型应相同

返回值类型 函数名(形参列表)

```
{
    函数体;
}
```

`return` 返回值类型要和函数定义的返回值类型一致 (不一致, 会发生类型转换, 以函数定义的类型为准)。

函数声明（函数原型，函数接口）

例5-2：自定义函数 pow 计算 x 的 y 次方（x, y 为整数）

```
#include <stdio.h>
long long LL_pow(int, int); ← 函数声明

int main()
{
    int x, y, N;
    scanf("%d%d", &x, &N);

    for (y = 1; y <= N; y++)
        printf("pow(%d, %d) = %lld\n", x, y, LL_pow(x,y));

    return 0;
}

long long LL_pow(int x, int y)
{
    long long ans = 1;
    int i;
    for (i = 1; i <= y; i++)
        ans *= x;
    return ans;
}
```

白箱：自定义
函数实现

函数声明（≈函数原型）具有重要作用：函数声明是把函数原型写到程序中，起到声明的作用。

- **含义：**用于函数描述的一条语句，包括**函数名、形参类型、个数和顺序、返回值类型**等主要信息，便于用户正确调用函数（正确的函数接口）。
- **作用：**将函数主要信息告知编译器，使其能判断对该函数调用是否正确。
- **位置：****函数声明要放在函数调用前**。函数声明不能独立存在，必须提前定义好函数（可以在别的文件里定义函数）。
- **格式：**函数头+分号（形参变量名可省略）
 - ◆ 格式一：long long LL_pow(int x, int y);
 - ◆ 格式二：long long LL_pow(int, int);

函数定义方式


- **方式一**：先声明、保证正确调用、单独定义
- **方式二**：先定义、后面随处可调用（函数定义中的函数头也同时作为函数原型）

```
#include <stdio.h>
long long LL_pow(int, int);

int main()
{
    int x, y, N;
    scanf("%d%d", &x, &N);
    for (y = 1; y <= N; y++)
        printf("pow(%d, %d) = %lld\n", x, y, LL_pow(x, y));

    return 0;
}

long long LL_pow(int x, int y)
{
    long long ans = 1;
    int i;
    for (i = 1; i <= y; i++)
        ans *= x;
    return ans;
}
```

 **推荐风格**

函数声明

函数调用

函数定义 (实现)

先声明（接口清晰），**后定义**（可能交给他人完成），**突出main的整体和全局地位，强调函数的独立功能地位但又不喧宾夺主。**

main函数是C程序唯一入口，其它函数直接或间接被main函数调用后才能被执行

VS

```
#include <stdio.h>

long long LL_pow(int x, int y)
{
    long long ans = 1;
    int i;
    for (i = 1; i <= y; i++)
        ans *= x;
    return ans;
}

int main()
{
    int x, y, N;
    scanf("%d%d", &x, &N);
    for (y = 1; y <= N; y++)
        printf("pow(%d, %d) = %lld\n", x, y, LL_pow(x, y));

    return 0;
}
```

函数定义且“声明”

先定义细节，后按需调用，头重脚轻、削弱了main函数主体地位

5.3 函数调用与返回

例5-2：自定义函数 pow 计算 x 的 y 次方 (x, y 为整数)

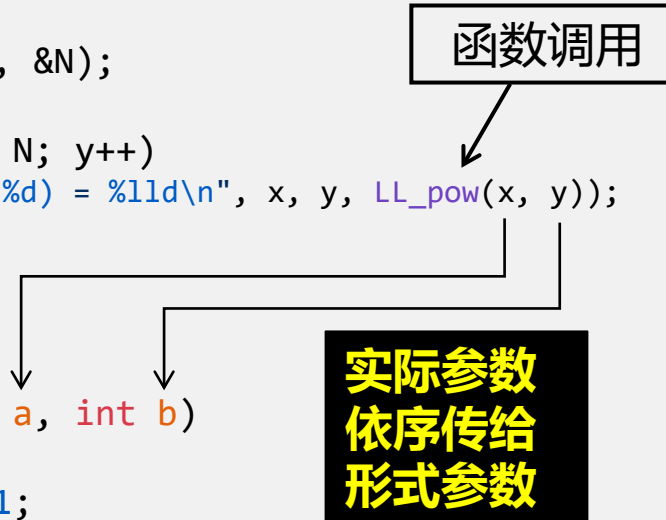
```
#include <stdio.h>
long long LL_pow(int, int);

int main()
{
    int x, y, N;
    scanf("%d%d", &x, &N);

    for (y = 1; y <= N; y++)
        printf("pow(%d, %d) = %lld\n", x, y, LL_pow(x, y));

    return 0;
}

long long LL_pow(int a, int b)
{
    long long ans = 1;
    int i;
    for (i = 1; i <= b; i++)
        ans *= a;
    return ans;
}
```



函数调用

- **含义**：用于执行已经定义好的函数
- **格式**：函数名(实参列表)
- **作用**：将实参按序传递给形参
- **位置**：形参和实参的数量、顺序、类型三者应保持一致
- **用法**：除main函数外的函数必须通过调用才能被执行。几种调用方式：
 - ◆ 单独的函数调用语句
 - ◆ 出现在表达式中
 - ◆ 函数嵌套调用

5.3 函数调用与返回

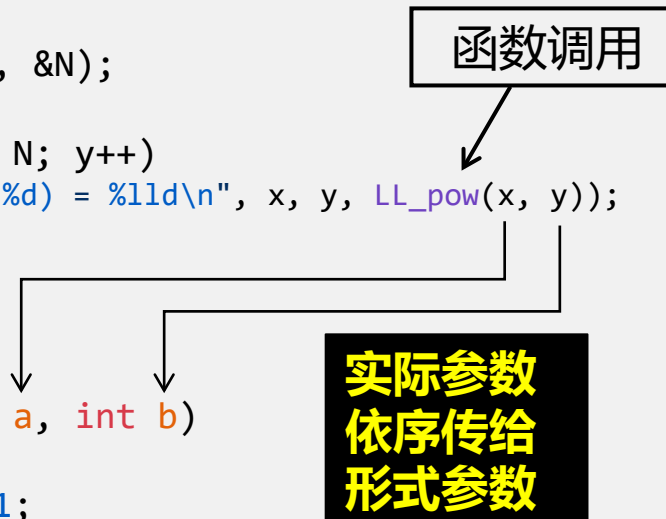
```
#include <stdio.h>
long long LL_pow(int, int);

int main()
{
    int x, y, N;
    scanf("%d%d", &x, &N);

    for (y = 1; y <= N; y++)
        printf("pow(%d, %d) = %lld\n", x, y, LL_pow(x, y));

    return 0;
}

long long LL_pow(int a, int b)
{
    long long ans = 1;
    int i;
    for (i = 1; i <= b; i++)
        ans *= a;
    return ans;
}
```



函数调用

实际参数
依序传给
形式参数

函数调用现场

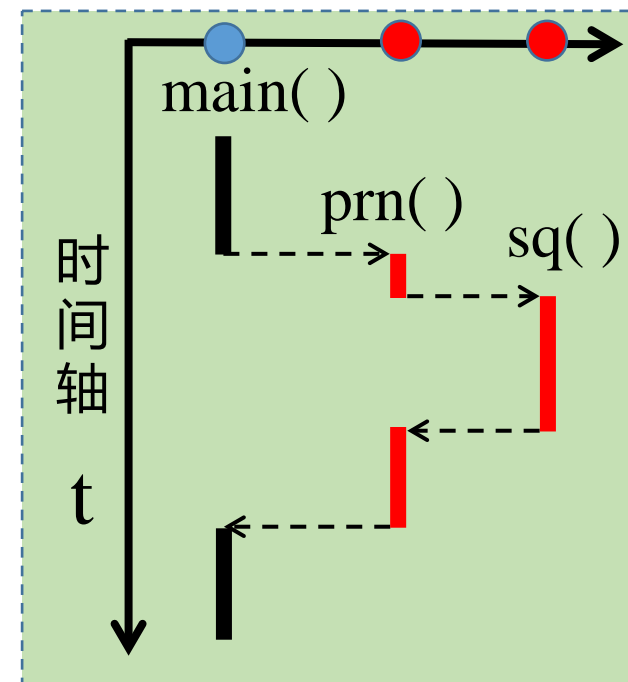
- 1 件事情：完成一件专业的事情
- 2 个对象：被调函数、主调函数
- 3 类数据：传递值、接收值、返回值
- 4 个动作：调用、跳转、执行、返回

函数调用流程

程序通过函数名实现函数调用，执行已经定义好的函数

- **调用：**主调函数执行到函数名，调用被调函数（调用时可将实参传递给形参）
- **跳转：**跳转到被调函数定义的位置
- **执行：**按顺序执行被调函数中的语句
- **返回：**被调函数结束后，返回到主调函数的调用位置继续向下执行（返回时可以通过return语句带一个返回值）

```
#include <stdio.h>
void prn();
int sq(int);
int main()
{
    prn();
    return 0;
}
void prn()
{
    int x;
    scanf("%d", &x);
    printf("%d\n", sq(x));
}
int sq(int x)
{
    return x*x;
}
```



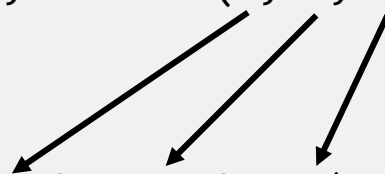
- 从哪离开，回到哪里
- 各司其职、独立完成
- 关注结果、忽略细节
- 分工有序、协同合作

函数传递值

例5-3：求三个数的最大值

```
#include <stdio.h>
int maximum(int, int, int);
int main()
{
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    printf("%d\n", maximum(a, b, c));
    return 0;
}

int maximum(int x, int y, int z)
{
    int max = x;
    if (y > max)
        max = y;
    if (z > max)
        max = z;
    return max;
}
```



实参按序传给形参

函数调用时——传递值

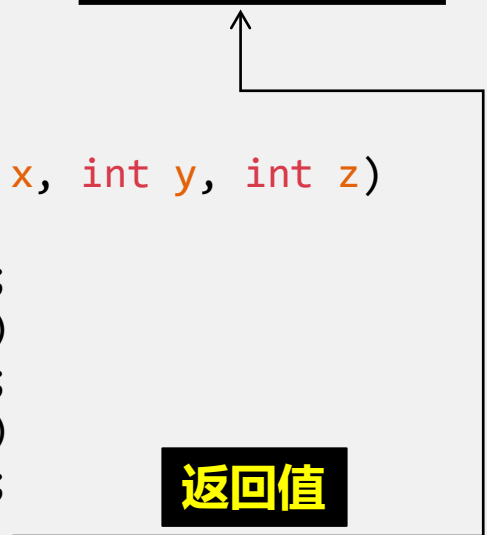
- **形参**：形式参数，函数定义中的参数，在函数中也相当于局部变量。
- **实参**：实际参数，函数调用中的参数，必须有确定的值。
- 形参和实参的**数量和顺序**要严格一致。
- 形参和实参的**类型**原则上要**保持一致**（不一致时会发生**类型转换**，以形参类型为准，这可能导致精度损失）。
- 函数调用时实参传递给形参是单向的，不能把形参反向传递给实参。

函数返回值

例5-3：求三个数的最大值

```
#include <stdio.h>
int maximum(int, int, int);
int main()
{
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    printf("%d\n", maximum(a, b, c));
    return 0;
}

int maximum(int x, int y, int z)
{
    int max = x;
    if (y > max)
        max = y;
    if (z > max)
        max = z;
    return max;
}
```



函数调用后——返回值

- return 语句使被调函数立刻返回到主调函数的调用位置
 - ◆ return 附带一个返回值，则返回值类型和函数定义的返回值类型原则上要保持一致（不一致会发生类型转换，以函数定义为准）；
 - ◆ return 不带返回值，则函数定义的返回值类型应为 void。
- 一个函数可以有 multiple return 语句，但每次函数调用只能执行一条 return 语句。
- main 函数执行到 return 语句，程序直接退出（返回 0 表示正常退出，非 0 表示异常退出）。

函数返回值

return语句三种功能

- 返回函数的计算结果，如①
- 返回函数的执行状态，一般是**具有逻辑判断功能的值**，如②、③
- 无返回值，强制退出当前函数，如④

带有返回值的标准函数示例

- 根据返回值设计条件表达式
- scanf 返回 int 类型，如 while(scanf(..) != EOF)
- isdigit 返回 int 类型，它具有逻辑判断能力，如if(isdigit(c))

```
int max(int x, int y)
{
    int max = x;
    if(y > max)
        max = y;
    return max;
}
```

① 返回计算结果

```
int is_even(int x)
{
    return x%2 == 0;
}
```



```
int is_even(int x)
{
    if (x%2 == 0)
        return 1;
    else
        return 0;
}
```

②



```
int isLeapYear(int y)
{
    return ( (y%4==0 && y%100!=0) || y%400 == 0 );
}
```

③

返回逻辑状态

```
void prnLeap(int y)
{
    if( isLeapYear(y) )
    {
        printf("%d is a leap year", y);
        return ;
    }
    printf("%d is not a leap year", y);
}
```

④ 离开函数

函数嵌套案例

实例

函数嵌套调用

- 函数不能嵌套定义，但可以嵌套调用（C语言对嵌套深度没有理论限制，但受计算环境影响，而且嵌套会影响效率）。
- 函数间是独立的，只存在调用和被调用关系。
- 函数执行顺序只与调用顺序有关，与函数声明和函数定义的位置无关，但函数声明要在函数调用前。
- 使被调函数返回调用位置的方法
 - ◆ 执行到函数结束的右花括号
 - ◆ 执行语句 `return`;
 - ◆ 执行语句 `return 返回值`（返回值可以是常量、变量或者表达式）

```
#include <stdio.h>
void inputError();
int input();
int isLeapYear(int y);
void prnLeap(int y);

int main()
{
    int year = input();
    prnLeap(year);
    return 0;
}

void prnLeap(int y)
{
    if( isLeapYear(y) )
    {
        printf("%d is a leap year", y);
        return ;
    }
    printf("%d is not a leap year", y);
}

int input()
{
    int y = 0;
    scanf("%d", &y);
    if (y > 0)
        return y;
    inputError();
    return 0;
}

void inputError()
{
    printf("Invalid input.");
}

int isLeapYear(int y)
{
    return ((y%4==0 && y%100!=0) || y%400==0);
}
```

函数嵌套案例

例5-3：求三个数的最大值

```
#include <stdio.h>
int maximum(int, int, int);

int main()
{
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    printf("%d\n", maximum(a, b, c));
    return 0;
}

int maximum(int x, int y, int z)
{
    int max = x;
    if (y > max)
        max = y;
    if (z > max)
        max = z;
    return max;
}
```



```
#include <stdio.h>
int max(int, int); //函数声明

int main()
{
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    printf("%d", max(max(a, b), c)); //函数嵌套调用
    return 0;
}

int max(int x, int y) //函数定义
{
    return (y > x) ? y : x;
}
```

```
// max()进一步的应用：找出数组a[N]的最大值
max_a = max(a[0], a[1]);
for (i = 2; i < N; i++)
    max_a = max(max_a, a[i]);
```

这是一行有趣的代码！

函数应用案例

例5-4：求两个整数的gcd

```
#include <stdio.h>
int main()
{
    int a, b, r;

    scanf("%d%d", &a, &b);
    if (b == 0)
    {
        printf("gcd is: %d\n", a < 0 ? -a : a);
        return 0;
    }
    while ((r = a % b) != 0)
    {
        a = b;
        b = r;           // gcd(a, b) ← gcd(b, a%b)
    }

    printf("gcd is: %d\n", b < 0 ? -b : b);
    return 0;
}
```

- 所有代码堆积在main函数中
- 程序结构不清晰
- 代码不可复用



```
#include <stdio.h>
int gcd(int, int);
int main()
{
    int a, b, r;
    scanf("%d%d", &a, &b);
    r = gcd(a, b);
    printf("gcd is: %d\n", r);
    return 0;
}

int gcd(int a, int b)
{
    int r;
    if (b == 0)
        return a < 0 ? -a : a;
    while ((r = a % b) != 0)
    {
        a = b;
        b = r;
    }
    return b < 0 ? -b : b;
}
```

把具有特定功能的程序
写成函数，作为标准模
块，作为基本素材、模
版，随身“携带”，随
时可能用到！
学习完递归函数后，会
有更巧妙的写法！

- 特定功能的代码封装成自定义函数
- main函数只负责输入、调用、输出
- 程序结构清晰

函数应用案例

马青公式由英国天文学教授约翰·马青(John Machin , 1686 –1751)于1706年发现, 他利用这个公式计算到了100位的圆周率。

例5-5: 求pi (精确到小数点后x位, x是一个正整数, 如x=10)

$$\pi = 16 \cdot \left(\frac{1}{5} - \frac{1}{3 \cdot 5^3} + \frac{1}{5 \cdot 5^5} - \frac{1}{7 \cdot 5^7} + \dots \right) - 4 \cdot \left(\frac{1}{239} - \frac{1}{3 \cdot 239^3} + \frac{1}{5 \cdot 239^5} - \frac{1}{7 \cdot 239^7} + \dots \right)$$

$$\pi = 16 \arctan \frac{1}{5} - 4 \arctan \frac{1}{239}$$

分析:

$$\pi = 16 \times A - 4 \times B$$

$$A \text{ 的通项 } \frac{(-1)^i}{(2i+1) \cdot 5^{(2i+1)}}, \quad B \text{ 的通项 } \frac{(-1)^i}{(2i+1) \cdot 239^{(2i+1)}}$$

函数应用案例

```
int i, j, k, sign = -1;
double n, d, s1 = 0.0, s2 = 0.0, e = 1e-10;
```

```
for (i = 0, d = 1; 16.0*d > e; i++)
{
    sign = (i % 2 == 0 ? 1 : -1);
    k = 2 * i + 1;
    for (j = 0, n = 1.0; j < k; j++)
        n *= 5;
    d = 1.0 / (n * k);
    s1 += d * sign;
}
```

```
for (i = 0, d = 1; 4.0*d > e; i++)
{
    sign = (i % 2 == 0 ? 1 : -1);
    k = 2 * i + 1;
    for (j = 0, n = 1.0; j < k; j++)
        n *= 239;
    d = 1.0 / (n * k);
    s2 += d * sign;
}
```

```
printf("\nPai is: %.20f\n", 16 * s1 - 4 * s2);
```

非函数
版本

- 代码冗余度高 (两段代码几乎相同)
- 可维护性、可扩展性差, 若公式或参数有变, 两个地方都要进行相应修改, 容易遗漏

$$\pi = 16 \times A - 4 \times B$$

$$A \text{ 的通项 } \frac{(-1)^i}{(2i+1) \cdot 5^{(2i+1)}}, B \text{ 的通项 } \frac{(-1)^i}{(2i+1) \cdot 239^{(2i+1)}}$$

```
double item(double v, double eps);
int main()
{
    double s1, s2, e = 1e-10;
    s1 = item(5.0, e/16.0);
    s2 = item(239.0, e/4.0);
    printf("\nPai is: %.20f\n", 16*s1 - 4*s2);
    return 0;
}
```

函数调用

```
double item(double v, double eps)
{
    double d, n, sum = 0.0;
    int i, j, k, sign = -1;

    for(i=0, n=1.0/v, d = 1; d > eps; i++)
    {
        sign = -sign;
        k = 2*i + 1;
        n *= v*v;
        d = 1.0 / (n*k);
        sum += d*sign;
    }
    return sum;
}
```

函数
版本

- 函数item复用 (一次定义, 多次使用)
- 可变更性强, 参数易修改
- 可扩展性好, 若实现方式有变, 只修改函数item中的实现即可

函数应用案例：性能的进一步优化

$$\pi = 16 \times A - 4 \times B$$

$$A \text{ 的通项 } \frac{(-1)^i}{(2i+1) \cdot 5^{(2i+1)}}, B \text{ 的通项 } \frac{(-1)^i}{(2i+1) \cdot 239^{(2i+1)}}$$

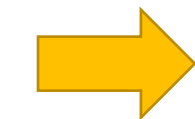
```
int i, j, k, sign = -1;
double n, d, s1 = 0.0, s2 = 0.0, e = 1e-10;

for (i = 0, d = 1; 16.0*d > e; i++)
{
    sign = (i % 2 == 0 ? 1 : -1);
    k = 2 * i + 1;
    for (j = 0, n = 1.0; j < k; j++)
        n *= 5;
    d = 1.0 / (n * k);
    s1 += d * sign;
}

for (i = 0, d = 1; 4.0*d > e; i++)
{
    sign = (i % 2 == 0 ? 1 : -1);
    k = 2 * i + 1;
    for (j = 0, n = 1.0; j < k; j++)
        n *= 239;
    d = 1.0 / (n * k);
    s2 += d * sign;
}

printf("\nPai is: %.20f\n", 16 * s1 - 4 * s2);
```

非函数
版本



将二层循
环优化成
一层循环

效率提高n倍

```
double item(double v, double eps);
int main()
{
    double s1, s2, e = 1e-10;
    s1 = item(5.0, e/16.0);
    s2 = item(239.0, e/4.0);
    printf("\nPai is: %.20f\n", 16*s1 - 4*s2);
    return 0;
}
```

```
double item(double v, double eps)
{
    double d, n, sum = 0.0;
    int i, j, k, sign = -1;

    for(i=0, n=1.0/v, d = 1; d > eps; i++)
    {
        sign = -sign;
        k = 2*i + 1;
        n *= v*v; // v^{2i+1} = v^{2i-1}*v*v
        d = 1.0 / (n*k);
        sum += d*sign;
    }

    return sum;
}
```

功能复用



请以后写这样的程序！

函数应用案例：设计高效且优美的算法

例5-6：验证哥德巴赫猜想。输入一个大于 6 的偶数 s ，验证它等于两个素数之和。

比赛排名 更新中，上次更新于 2022-10-02 11:03:53

2022Fall C4

« < > »

排名	用户	得分	罚时	A	B	C	D	E	F	G	H	I	J
				1272/1273	1241/1267	1196/1248	1145/1234	537/864	382/558	6/140	2/25	16/90	7/53

C4-2022级程序设计基础第四次上机

简介

题目

排名

我的提交

A B C D E F G H I J

E 航小萱挑战哥德巴赫猜想

时间限制：1000ms 内存限制：65536kb

通过率：537/864 (62.15%) 正确率：537/2496 (21.51%)

题目描述

哥德巴赫猜想被誉为是数学界最难证明的猜想之一，许多著名的科学家对此都望而生畏，

函数应用案例：设计高效且优美的算法

例5-6：验证哥德巴赫猜想。输入一个大于 6 的偶数 s ，验证它等于两个素数之和。

- 分析：**
- 对不大于 $s/2$ 的整数 a ，验证 a 和 $s-a$ 是否为素数，（问题的核心是判断素数！）
 - 设计函数 `isPrime(x)` 判断 x 是否为素数，分别调用 `isPrime(a)` 和 `isPrime(s-a)`，
（一次定义，多次使用，功能复用）

判断 x 是否为素数，函数 `isPrime(x)`？

素数：只能被1和其自身整除的正整数。

函数应用案例：设计高效且优美的算法

函数 isPrime(x) 判断 x 是否为素数？

素数：只能被1和其自身整除的正整数。

方法1： 2~x/2 的任何数都不能整除x，则x为素数。

```
#include <stdio.h>
int isPrime(int);
int main()
{
    int x;
    scanf("%d", &x); //x为大于1的整数

    printf(isPrime(x) ? "prime": "Not prime");
    return 0;
}
int isPrime(int x)
{
    int i, prime_flag = 1;
    for(i=2; i <= x/2; i++)
    {
        if(x%i == 0)
        {
            prime_flag = 0;
            break;
        }
    }
    return prime_flag; //素数返回1，合数返回0
}
```

方法2： 大于2的奇数才可能是素数；奇数只可能被奇数整除。3 ~ x/2 的任何奇数都不能整除x，则x为素数。

```
int isPrime(int x)
{
    int i, prime_flag = 1;
    if( x>2 && (x%2 == 0) ) //x为大于2的偶数
        prime_flag = 0;
    else // x为大于3的奇数的情况
    {
        // 仅判断x是否被奇数整除
        for(i=3; i <= x/2; i += 2)
        {
            if(x%i == 0)
            {
                prime_flag = 0;
                break;
            }
        }
    }
    return prime_flag; // x为2，也是返回1
}
```

方法3： 方法1 或 方法2 中的循环区间只需为 3 ~ sqrt(x)即可。为什么？

```
// 不用 i <= sqrt(x)
for(i=3; i*i <= x; i += 2)
```

用任何一种方法都能通过C4-E（因为数据不大）。方法3非常高效！是否还有更快的方法？

函数应用案例：设计高效且优美的算法

函数 isPrime(x) 判断 x 是否为素数？

方法1：2 ~ x/2 的任何数都不能整除x，则x为素数。 (步长为1)

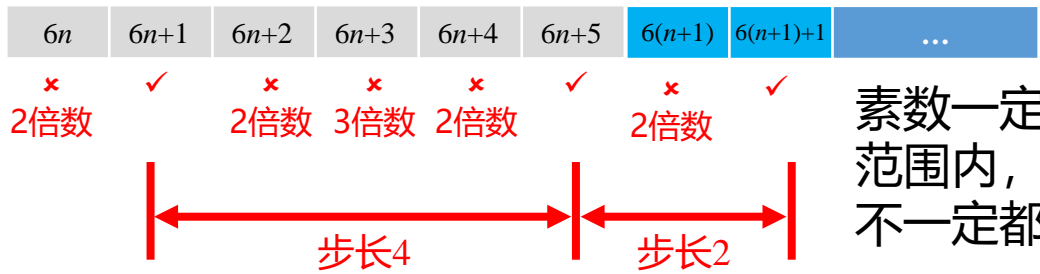
方法2：大于2的奇数才可能是素数；奇数只可能被奇数整除。3 ~ x/2 的任何奇数都不能整除x，则x为素数。 (步长为2)

方法3：区间控制为 3 ~ sqrt(x)。 (循环上限从 x/2 到 sqrt(x))

方法4：2 ~ sqrt(x) 之间的任何素数都不能整除 x，则 x 为素数。2 ~ sqrt(x) 之间可能的素数有哪些？
(2, 3, 5, 7; 大于7的素数时，步长至少为 4-2-4-2)

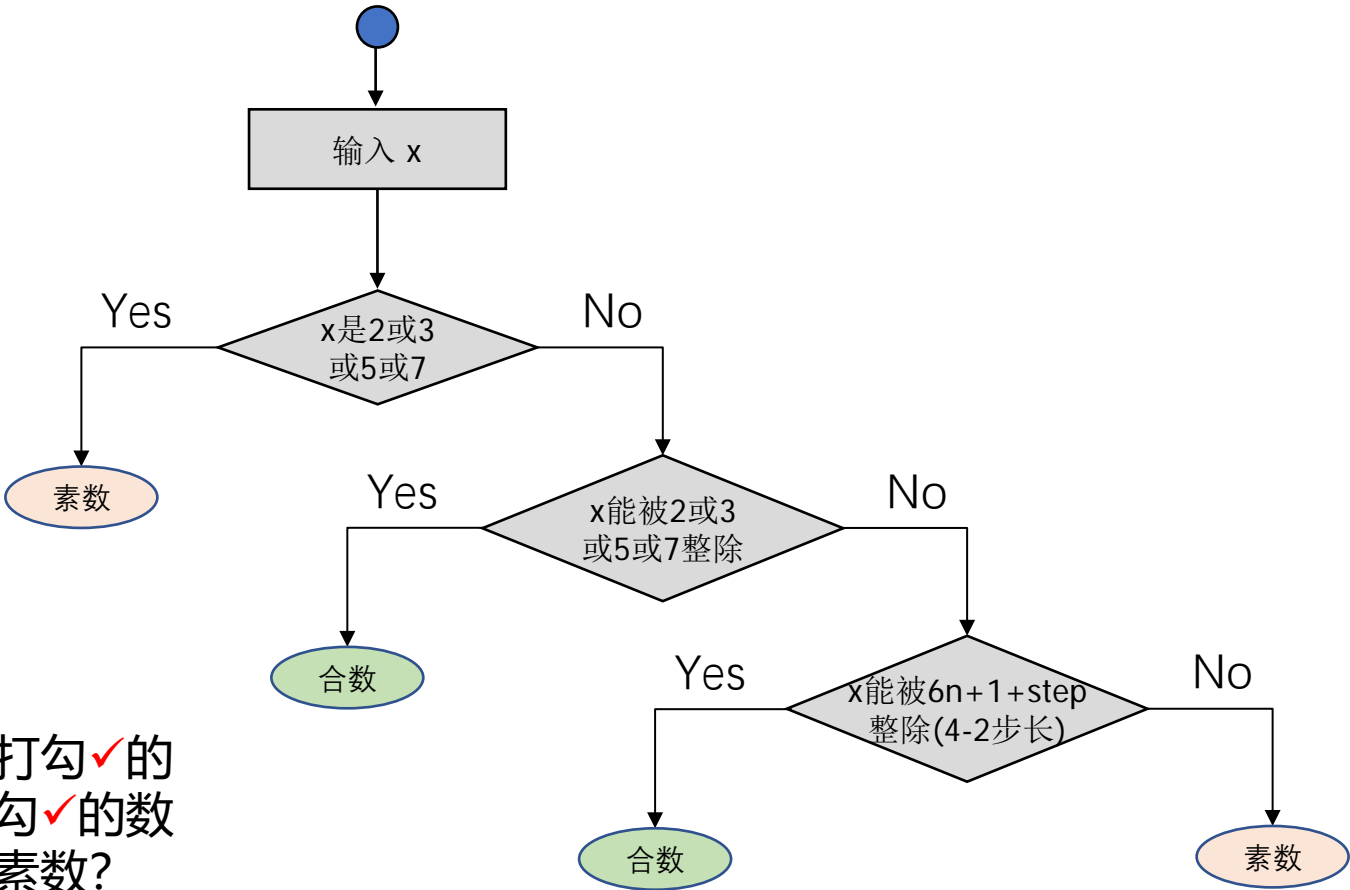
$$\{6, 7, 8, 9, \dots, \infty\} \Leftrightarrow \bigcup_{n=1}^{+\infty} \{6n, 6n+1, 6n+2, 6n+3, 6n+4, 6n+5\}$$

✓：可能的素数



素数一定在打勾✓的范围内，打勾✓的数不一定是素数？

素数：只能被1和其自身整除的正整数。

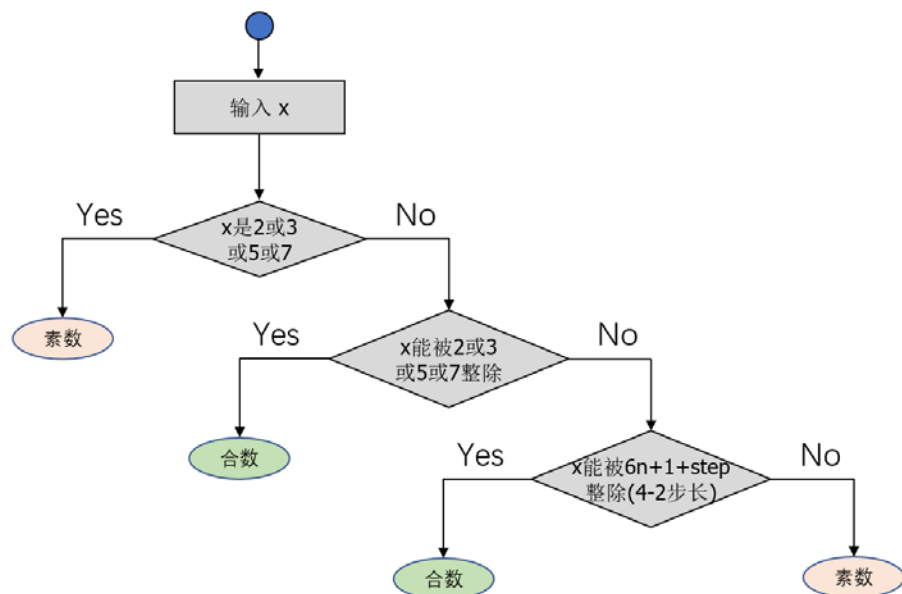


函数应用案例：设计高效且优美的算法

函数 isPrime(x) 判断 x 是否为素数？

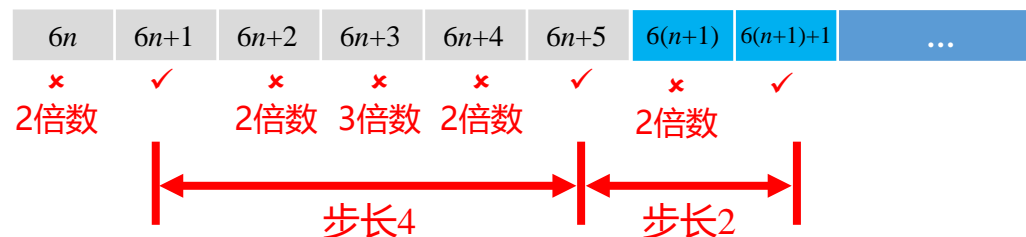
素数：只能被1和其自身整除的正整数。

方法4：2 ~ sqrt(x) 之间的任何素数都不能整除 x，则 x 为素数。2 ~ sqrt(x) 之间可能的素数有哪些？
(2, 3, 5, 7; 大于7的素数时，步长至少为 4-2-4-2)



$$\{6, 7, 8, 9, \dots, \infty\} \Leftrightarrow \bigcup_{n=1}^{+\infty} \{6n, 6n+1, 6n+2, 6n+3, 6n+4, 6n+5\}$$

✓：可能的素数



```
int isPrime(int x) // 跟第一次印刷的书上有些不同，书上有点错误（书上例5-4）
{
    int i, step;
    if (x == 2 || x == 3 || x == 5 || x == 7) // 特判，找出前几个素数
        return 1; // 素数返回1
    else if (x % 2 == 0 || x % 3 == 0 || x % 5 == 0 || x % 7 == 0)
        return 0; // 能被任意一个素数整除（先处理前4个素数），则是合数
    else
    {
        step = 4;
        for (i = 7 + step; i * i <= x; i += step)
        {
            // 能被任意一个素数整除（处理i = (6n+1 + step)，step为4-2切换），则是合数
            if (x % i == 0)
                return 0;
            step = 6 - step;
        }
        return 1;
    }
}
```

5.4 局部变量与全局变量

- **变量空间域和时间域**

- ◆ **空间域**：变量的**访问区域**（作用范围 **scope**），由变量在**程序中的定义位置**决定（也称为作用域）
- ◆ **时间域**：变量的**生命周期**（存在时间 **duration**），由变量在**内存中的存储位置**决定
- ◆ 存在，但可能不允许访问

- **局部变量和全局变量**

- ◆ 根据空间域，变量分为局部变量和全局变量
- ◆ 局部变量是在函数内部或语句块中定义的变量，它的空间域仅限于被定义的函数内部或语句块
- ◆ 全局变量是在所有函数（包括main函数）之外定义的变量，它的空间域是从定义开始到本程序文件结束
- ◆ 局部变量和全局变量重名时，局部变量起作用
- ◆ **当全局变量（数组）没有初始化时，系统自动赋值为 0**

```
int a;  
void fun();  
int main() ①  
{  
    int a = 5;  
    fun();  
    printf("1st a: %d\n", a);  
    return 0;  
}  
void fun()  
{  
    printf("2nd a: %d\n", a);  
}
```

```
int a = 100;  
void fun(int);  
int main() ②  
{  
    int a = 5;  
    fun(a);  
    printf("1st a: %d\n", a);  
    return 0;  
}  
void fun(int a)  
{  
    printf("2nd a: %d\n", a);  
}
```

变量空间域：变量在程序里能访问的位置

```
#include <stdio.h>
int x;
int test();
int main()
{
    printf("A: %d\n", x);
    int x = 5;
    printf("B: %d\n", x);
    x = test();
    {
        int x = 7;
        printf("C: %d\n", x);
    }
    printf("D: %d\n", x);
    return 0;
}
int test()
{
    int x = 3;
    return x;
}
```

- 根据定义变量的**语句块**确定变量空间域
- 语句块是指被一对 **{ }** 括起来的若干条语句
- 嵌套作用域的变量重名时，空间域小的变量起作用

变量时间域：变量在程序里生存（存在）的时间

```
#include <stdio.h>
int x;
int test();
int main()
{
    printf("A: %d\n", x);
    int x = 5;
    printf("B: %d\n", x);
    x = test();
    {
        int x = 7;
        printf("C: %d\n", x);
    }
    printf("D: %d\n", x);
    return 0;
}
int test()
{
    int x = 3;
    return x;
}
```

- 通过**存储类型**确定变量的时间域
- 存储类型决定变量的存储位置和存在状态，常用的有4种
 - **auto**：默认省略，变量存储在**动态内存**，其时间域是从定义变量的函数或语句块开始执行时刻到执行结束时刻
 - **register**：变量存储在寄存器【*】
 - **static**：变量被存储在静态存储区域【*】
 - **extern**：引用被存储在另一个文件中的全局变量【*】
- 带存储类型的变量定义格式

存储类型 数据类型 变量名;

5.4 局部变量和全局变量

局部变量与全局变量特征简介 (都是基本概念, 请仔细阅读教材)

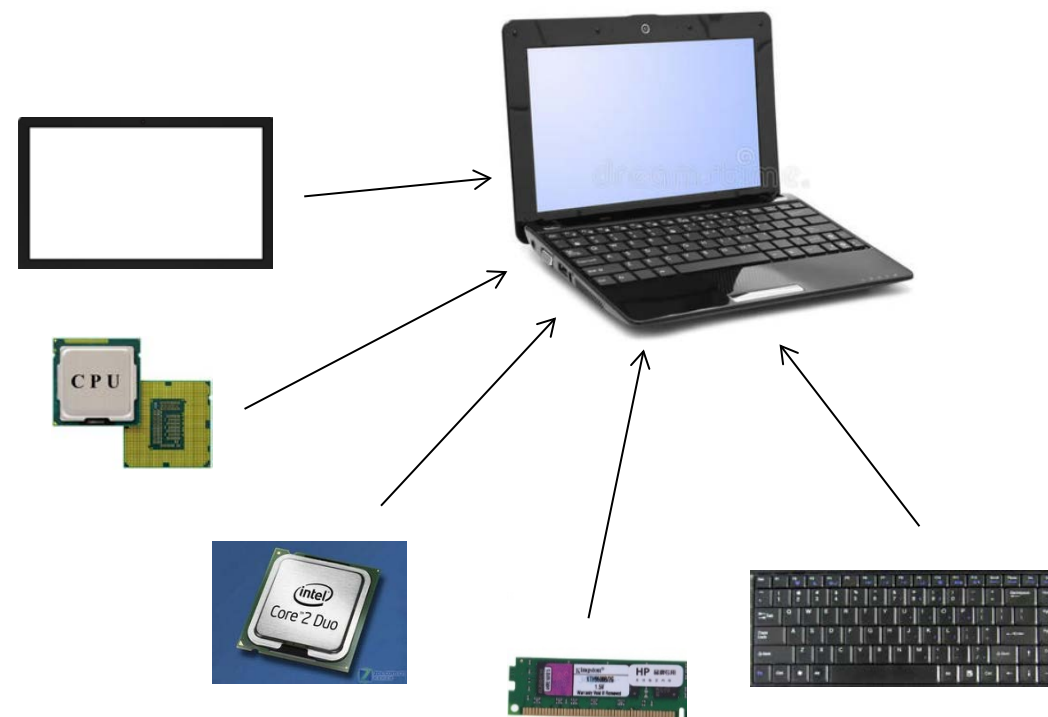
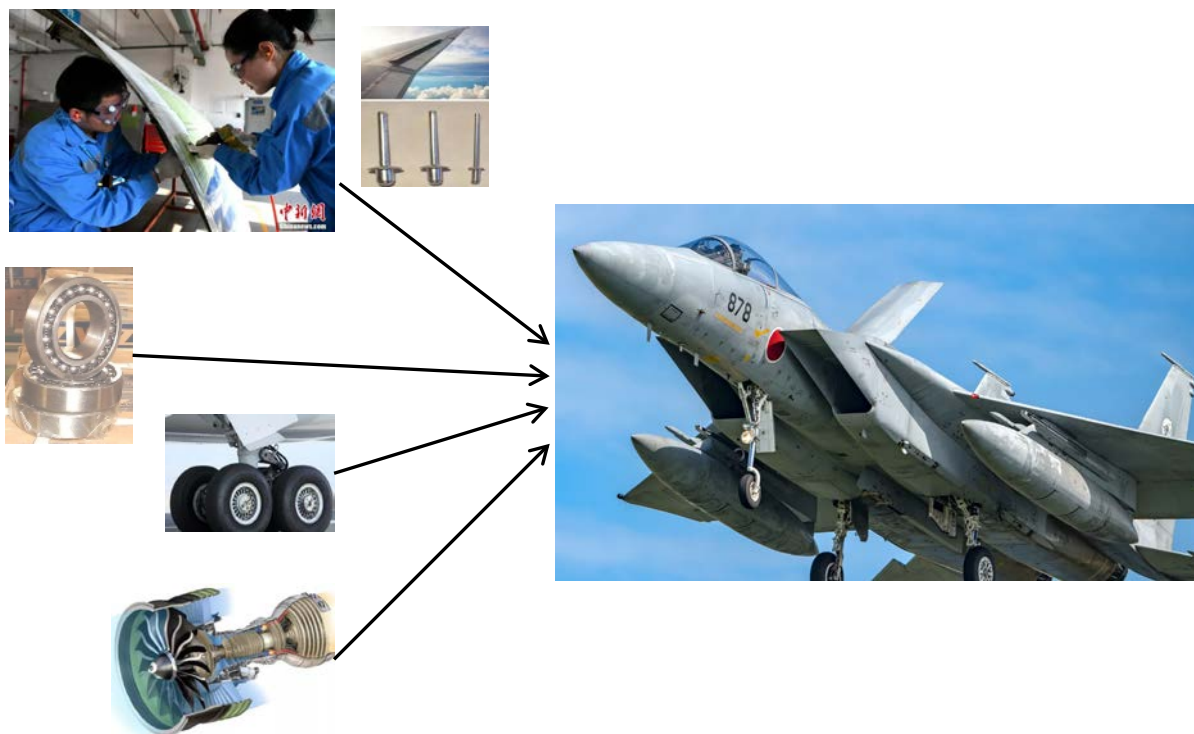
特征 \ 变量类型	局部变量	全局变量
生命周期	从函数调用开始到函数执行结束	程序整个运行期间 (main函数调用之前全局变量就分配了空间)
访问区域	从变量定义位置到所在块结束的 }	默认全局都可以访问
初始化	默认初值不确定, 使用时应该小心赋初值问题	默认为0 (int 为 0, char 为 '\0')
数据 (数组) 大小	对于小数组, 比如几万个元素以内的数组, 可以定义在函数内, 是局部数组	对于大数组, 比如数百万甚至更多个元素的数组, 一般定义在函数函数之外, 是全局数组

- **为什么要使用函数?**
- **函数设计思想是什么?**

5.5 模块化编程思想

要开发和维护大程序，最好的办法是从容易管理的小块和小组件开始

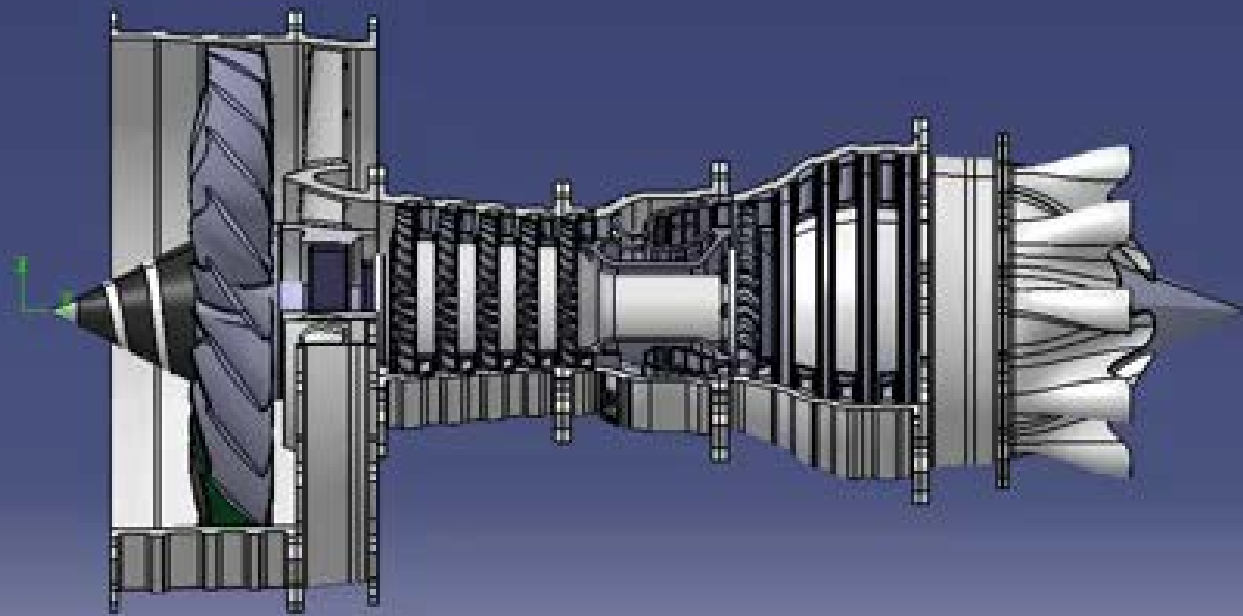
- ◆ 造飞机（铆钉、轴承、发动机、.....）
- ◆ 造电脑（键盘、电池、显示器、内存、CPU、.....）



“分而治之，各个击破” (divide and conquer)

什么是模块化

按功能划分，按接口装配



- 风扇
- 核心机
- 低压涡轮
- 尾喷管



模块是如何组装成产品的？—— 按功能划分，按接口装配

用函数编程，像用积木搭飞机？但积木搭的飞机终究还是假的飞机
(使用函数进行模块化的复杂度远远超过了积木所能表达的含义)。
积木分解出来的是模块（积木的方法仍然有借鉴意义），问题分解出来的是函数！

模块化编程的意义

真正的飞机是这样设计出来的！



软件设计比飞机设计还要复杂！

- 每个函数是一个小部分，具有独立功能。
- 整体到局部：每个函数具有**好的接口**，组装时能标准化处理，对每个函数分别设计，函数与函数之间的设计不相关（**松耦合、高内聚**），可以由很多程序员分别做，只要按接口标准来设计与开发即可。
- 局部回整体：最终，函数通过**接口**组合成一个大程序，完成整体工作。

接口设计原则：Make interface **easy** to use correctly, **hard** to use incorrectly.

接口应该这样设计，当正确使用时编程很容易，而错误使用时编程很困难。

使用函数的模块化编程思想简介——特点与优点

复用（拿来主义）：
减少重复劳动，减少犯错风险

抽象（思想）：复杂问题模块化、概念化、标准化，逐步细化
简化（设计）：大事化小，繁事化简；分而治之，各个击破
封装（生产）：数据与信息隐藏，安全，方便，简单
组合（装配）：小功能程序聚合成大功能的程序

```
int i, j, k, sign = -1;
double n, d, s1 = 0.0, s2 = 0.0, eps, e = 1e-10;
eps = e / 16.0;
for (i = 0, d = 1; d > eps; i++)
{
    sign = i % 2 == 0 ? 1 : -1;
    k = 2 * i + 1;
    for (j = 0, n = 1.0; j < k; j++)
        n *= j;
    d = 1.0 / (n * k);
    s1 += d * sign;
}
eps = e / 4.0;
for (i = 0, d = 1; d > eps; i++)
{
    sign = i % 2 == 0 ? 1 : -1;
    k = 2 * i + 1;
    for (j = 0, n = 1.0; j < k; j++)
        n *= j;
    d = 1.0 / (n * k);
    s2 += d * sign;
}
printf("\nPai is: %.20f\n", 16 * s1 - 4 * s2);
```

非函数版本

```
double item(double v, double eps);
int main()
{
    double s1, s2, e = 1e-10;
    s1 = item(5.0, e/16.0);
    s2 = item(25.0, e/4.0);
    printf("\nPai is: %.20f\n", 16*s1 - 4*s2);
    return 0;
}

double item(double v, double eps)
{
    double d, n, sum=0.0;
    int i, j, k, sign;
    for (i=0, n=1.0/v, d = 1; d > eps; i++)
    {
        sign = i % 2 == 0 ? 1 : -1;
        k = 2 * i + 1;
        n *= v * v;
        d = 1.0 / (n * k);
        sum += d * sign;
    }
    return sum;
}
```

函数调用

函数版本

```
int main()
{
    ...
    此处省略10000行
}
```

VS

```
int main()
{
    input();
    process_1();
    ...
    process_n();
    output();
}
```

input()
{ ... }

process_1()
{ ... }

process_n()
{ ... }

output()
{ ... }

subprocess()
{ ... }



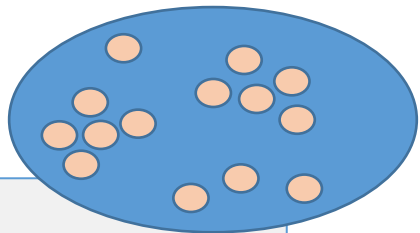
所有代码都写在main函数里，必然熵增！

使用函数的根本意义在于：**对抗熵增**——降低代码的混乱程度。

为什么使用函数?

例5-5: 用马青公式求pi

小分子太多,
混乱: 熵增!



```
int i, j, k, sign = -1;
double n, d, s1 = 0.0, s2 = 0.0, e = 1e-10;
```

```
for (i = 0, d = 1; 16.0*d > e; i++)
{
    sign = (i % 2 == 0 ? 1 : -1);
    k = 2 * i + 1;
    for (j = 0, n = 1.0; j < k; j++)
        n *= 5;
    d = 1.0 / (n * k);
    s1 += d * sign;
}
```

```
for (i = 0, d = 1; 4.0*d > e; i++)
{
    sign = (i % 2 == 0 ? 1 : -1);
    k = 2 * i + 1;
    for (j = 0, n = 1.0; j < k; j++)
        n *= 239;
    d = 1.0 / (n * k);
    s2 += d * sign;
}
```

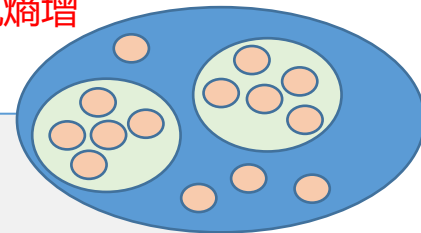
```
printf("\nPai is: %.20f\n", 16 * s1 - 4 * s2);
```

非函数
版本



1. 代码复用 (一次定义, 多次使用)
2. 扩展性好 (实现有变, 只修改函数定义)
3. 可变性强 (修改参数实现不同计算)

对抗熵增



```
double item(double v, double eps);
int main()
{
    double s1, s2, e = 1e-10;
    s1 = item(5.0, e/16.0);
    s2 = item(239.0, e/4.0);
    printf("\nPai is: %.20f\n", 16*s1 - 4*s2);
    return 0;
}
```

```
double item(double v, double eps)
{
    double d, n, sum=0.0;
    int i, j, k, sign = -1;

    for(i=0, n=1.0/v, d = 1; d > eps; i++)
    {
        sign = -sign;
        k = 2*i + 1;
        n *= v*v;
        d = 1.0 / (n*k);
        sum += d*sign;
    }

    return sum;
}
```

函数
版本

函数调用

为什么使用函数？

2021Fall, E4

C 输出距离

时间限制: 1000ms 内存限制: 65536kb

通过率: 1130/1163 (97.16%) 正确率: 1130/1387 (81.47%)

题目背景

自定义一个函数，计算两个三维坐标点的距离。请大家体会“利用函数实现模块化编程，可以提高程序开发效率”。

题目描述

输入 4 个三维坐标点，按要求顺序输出它们两两之间的距离。

输入格式

共 4 行。第 i 行三个整数 x_i, y_i, z_i ，表示第 i 个三维坐标点，数字之间用一个空格隔开。
 $-100 \leq x_i, y_i, z_i \leq 100$ 。

输出格式

共 6 行，每行分别输出：坐标点 2 与坐标点 4、坐标点 1 与坐标点 3、坐标点 2 与坐标点 3、坐标点 3 与坐标点 4、坐标点 1 与坐标点 2、坐标点 1 与坐标点 4 的距离。结果保留 2 位小数。

输入样例

```
20 21 10
21 10 17
10 17 20
17 20 21
```

输入：4个三维坐标点

输出样例

```
11.49
14.70
13.38
7.68
13.08
11.45
```

输出：

```
Dist(2, 4)
Dist(1, 3)
Dist(2, 3)
Dist(3, 4)
Dist(1, 2)
Dist(1, 4)
```

```
double x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4;
double xsq, ysq, zsq, m1, m2, m3, m4, m5, m6;
```

```
scanf("%lf%lf%lf", &x1, &y1, &z1);
scanf("%lf%lf%lf", &x2, &y2, &z2);
scanf("%lf%lf%lf", &x3, &y3, &z3);
scanf("%lf%lf%lf", &x4, &y4, &z4);
```

```
xsq = (x2 - x4) * (x2 - x4);
ysq = (y2 - y4) * (y2 - y4);
zsq = (z2 - z4) * (z2 - z4);
m1 = sqrt(xsq + ysq + zsq);
```



```
xsq = (x1 - x3) * (x1 - x3);
ysq = (y1 - y3) * (y1 - y3);
zsq = (z1 - z3) * (z1 - z3);
m2 = sqrt(xsq + ysq + zsq);
```



```
xsq = (x2 - x3) * (x2 - x3);
ysq = (y2 - y3) * (y2 - y3);
zsq = (z2 - z3) * (z2 - z3);
m3 = sqrt(xsq + ysq + zsq);
```



```
xsq = (x4 - x3) * (x4 - x3);
ysq = (y4 - y3) * (y4 - y3);
zsq = (z4 - z3) * (z4 - z3);
m4 = sqrt(xsq + ysq + zsq);
```



```
xsq = (x1 - x2) * (x1 - x2);
ysq = (y1 - y2) * (y1 - y2);
zsq = (z1 - z2) * (z1 - z2);
m5 = sqrt(xsq + ysq + zsq);
```



```
xsq = (x1 - x4) * (x1 - x4);
ysq = (y1 - y4) * (y1 - y4);
zsq = (z1 - z4) * (z1 - z4);
m6 = sqrt(xsq + ysq + zsq);
```



```
printf("%.2f\n", m1);
printf("%.2f\n", m2);
printf("%.2f\n", m3);
printf("%.2f\n", m4);
printf("%.2f\n", m5);
printf("%.2f", m6);
```

代码都写在main函数里，很多相同特征的小分子。

为什么使用函数？

2021Fall, E4

C 输出距离

时间限制: 1000ms 内存限制: 65536kb

通过率: 1130/1163 (97.16%) 正确率: 1130/1387 (81.47%)

题目背景

自定义一个函数，计算两个三维坐标点的距离。请大家体会“利用函数实现模块化编程，可以提高程序开发效率”。

题目描述

输入 4 个三维坐标点，按要求顺序输出它们两两之间的距离。

输入格式

共 4 行。第 i 行三个整数 x_i, y_i, z_i ，表示第 i 个三维坐标点，数字之间用一个空格隔开。
 $-100 \leq x_i, y_i, z_i \leq 100$ 。

输出格式

共 6 行，每行分别输出：坐标点 2 与坐标点 4、坐标点 1 与坐标点 3、坐标点 2 与坐标点 3、坐标点 3 与坐标点 4、坐标点 1 与坐标点 2、坐标点 1 与坐标点 4 的距离。结果保留 2 位小数。

输入样例

```
20 21 10
21 10 17
10 17 20
17 20 21
```

输入：4个三维坐标点

输出样例

```
11.49
14.70
13.38
7.68
13.08
11.45
```

输出：

```
Dist(2, 4)
Dist(1, 3)
Dist(2, 3)
Dist(3, 4)
Dist(1, 2)
Dist(1, 4)
```

弱函数版本

6 个组合都写，
代码冗余，
重复劳动，
工作量大，
容易出错！

```
double x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4;
double xsq, ysq, zsq, m1, m2, m3, m4, m5, m6;
```

```
scanf("%lf%lf%lf", &x1, &y1, &z1);
scanf("%lf%lf%lf", &x2, &y2, &z2);
scanf("%lf%lf%lf", &x3, &y3, &z3);
scanf("%lf%lf%lf", &x4, &y4, &z4);
```

```
xsq = (x2 - x4) * (x2 - x4);
ysq = (y2 - y4) * (y2 - y4);
zsq = (z2 - z4) * (z2 - z4);
m1 = sqrt(xsq + ysq + zsq);
```

```
xsq = (x1 - x3) * (x1 - x3);
ysq = (y1 - y3) * (y1 - y3);
zsq = (z1 - z3) * (z1 - z3);
m2 = sqrt(xsq + ysq + zsq);
```

```
xsq = (x2 - x3) * (x2 - x3);
ysq = (y2 - y3) * (y2 - y3);
zsq = (z2 - z3) * (z2 - z3);
m3 = sqrt(xsq + ysq + zsq);
```

```
xsq = (x4 - x3) * (x4 - x3);
ysq = (y4 - y3) * (y4 - y3);
zsq = (z4 - z3) * (z4 - z3);
m4 = sqrt(xsq + ysq + zsq);
```

```
xsq = (x1 - x2) * (x1 - x2);
ysq = (y1 - y2) * (y1 - y2);
zsq = (z1 - z2) * (z1 - z2);
m5 = sqrt(xsq + ysq + zsq);
```

```
xsq = (x1 - x4) * (x1 - x4);
ysq = (y1 - y4) * (y1 - y4);
zsq = (z1 - z4) * (z1 - z4);
m6 = sqrt(xsq + ysq + zsq);
```

```
printf("%.2f\n", m1);
printf("%.2f\n", m2);
printf("%.2f\n", m3);
printf("%.2f\n", m4);
printf("%.2f\n", m5);
printf("%.2f\n", m6);
```

为什么使用函数？

2021Fall, E4

C 输出距离

时间限制: 1000ms 内存限制: 65536kb

通过率: 1130/1163 (97.16%) 正确率: 1130/1387 (81.47%)

题目背景

自定义一个函数，计算两个三维坐标点的距离。请大家体会“利用函数实现模块化编程，可以提高程序开发效率”。

题目描述

输入 4 个三维坐标点，按要求顺序输出它们两两之间的距离。

输入格式

共 4 行。第 i 行三个整数 x_i, y_i, z_i ，表示第 i 个三维坐标点，数字之间用一个空格隔开。
 $-100 \leq x_i, y_i, z_i \leq 100$ 。

输出格式

共 6 行，每行分别输出：坐标点 2 与坐标点 4、坐标点 1 与坐标点 3、坐标点 2 与坐标点 3、坐标点 3 与坐标点 4、坐标点 1 与坐标点 2、坐标点 1 与坐标点 4 的距离。结果保留 2 位小数。

输入样例

```
20 21 10
21 10 17
10 17 20
17 20 21
```

输入：4个三维坐标点

输出样例

```
11.49
14.70
13.38
7.68
13.08
11.45
```

输出：

```
Dist(2, 4)
Dist(1, 3)
Dist(2, 3)
Dist(3, 4)
Dist(1, 2)
Dist(1, 4)
```

```
double x[5], y[5], z[5];
```

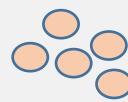
```
double sq(int x)
{
    return x*x;
}
```

```
void getDist(int i, int j)
{
    double d, xsq, ysq, zsq;
    xsq = sq(x[i]-x[j]);
    ysq = sq(y[i]-y[j]);
    zsq = sq(z[i]-z[j]);
    d = sqrt(xsq + ysq + zsq);
    printf("%.2f\n", d);
}
```

```
int main()
{
    for(int i=1; i<=4; i++)
        scanf("%lf%lf%lf", &x[i], &y[i], &z[i]);
    getDist(2, 4);
    getDist(1, 3);
    getDist(2, 3);
    getDist(3, 4);
    getDist(1, 2);
    getDist(1, 4);
    return 0;
}
```

强函数版本

1 个大分子，每次只“复制”大分子，以整体进行管理



参数决定组合，工作量小，简单清晰，不易出错！

如何进行函数分解

例5-7: 查询某天 (如20241025) 是星期几 (用函数实现, Zeller公式 $w = (C/4 - 2C + Y + Y/4 + 26(M+1)/10 + D - 1) \% 7$)

```
1  #include <stdio.h>
2  void printWeek(int);
3  int getWeek(int);
4  int main()
5  {
6      int longday = 1, w;
7      // printf("Query what day a certain date is\n");
8      // printf("Note: the format of the day is like 20120101\n");
9      // printf("The input is between 101 and 99991231\n\n");
10
11     scanf("%d", &longday);
12
13     if(!(longday >= 101 && longday <= 99991231))
14     {
15         printf("Wrong input format, try again!\n");
16         return 0;
17     }
18     w = getWeek(longday);
19     printWeek(w);
20
21     return 0;
22 }
```

```
23 int getWeek(int day)
24 {
25     // c: century-1, y: year, m:month, w:week, d:day
26     int c, y, m, d, w;
27     y = day/10000;
28     m = (day%10000)/100;
29     d = day%100;
30     if(m<3)
31     {
32         y = y-1;
33         m = m+12;
34     }
35     c = y/100;
36     y = y%100;
37     // Zeller formula
38     w = (y + y/4 + c/4 - 2*c + (26*(m+1))/10 + d - 1)%7;
39     if (w<0)
40         w+=7;
41     return w;
42 }
```

```
43 void printWeek(int w)
44 {
45     switch(w)
46     {
47         case 0:
48             printf("Sun\n");
49             break;
50         case 1:
51             printf("Mon\n");
52             break;
53         case 2:
54             printf("Tue\n");
55             break;
56         case 3:
57             printf("Wed\n");
58             break;
59         case 4:
60             printf("Thu\n");
61             break;
62         case 5:
63             printf("Fri\n");
64             break;
65         case 6:
66             printf("Sat\n");
67             break;
68     }
69 }
```

问题抽象, 事情简化
功能封装, 层次清晰

三个函数 (main也是一个函数),
每个独立实现, 甚至交给三个人独立去完成 (按接口要求即可) !

函数重在接口

例5-7: 查询某天 (如20241025) 是星期几 (用函数实现, Zeller公式 $w = (C/4 - 2C + Y + Y/4 + 26(M+1)/10 + D - 1) \% 7$)

```
1  #include <stdio.h>
2  void printWeek(int);
3  int getWeek(int);
4  int main()
5  {
6      int longday = 1, w;
7      // printf("Query what day a certain date is\n");
8      // printf("Note: the format of the day is like 20120101\n");
9      // printf("The input is between 101 and 99991231\n\n");
10
11     scanf("%d", &longday);
12
13     if(!(longday >= 101 && longday <= 99991231))
14     {
15         printf("Wrong input format, try again!\n");
16         return 0;
17     }
18     w = getWeek(longday);
19     printWeek(w);
20
21     return 0;
22 }
```

```
23 int getWeek(int day)
24 {
25     // c: century-1, y: year, m: month, w: week, d: day
26     int c, y, m, d, w;
27     y = day/10000;
28     m = (day%10000)/100;
29     d = day%100;
30     if(m<3)
31     {
32         y = y-1;
33         m = m+12;
34     }
35     c = y/100;
36     y = y%100;
37     // Zeller formula
38     w = (y + y/4 + c/4 - 2*c + (26*(m+1))/10 + d - 1)%7;
39     if (w<0)
40         w+=7;
41     return w;
42 }
```

```
43 void printWeek(int w)
44 {
45     switch(w)
46     {
47     case 0:
48         printf("Sun\n");
49         break;
50     case 1:
51         printf("Mon\n");
52         break;
53     case 2:
54         printf("Tue\n");
55         break;
56     case 3:
57         printf("Wed\n");
58         break;
59     case 4:
60         printf("Thu\n");
61         break;
62     case 5:
63         printf("Fri\n");
64         break;
65     case 6:
66         printf("Sat\n");
67         break;
68     }
69 }
```

单一功能, 责任明确

year

month

day

longday

这里, 接口设计是 `int getWeek(int day);`
这个接口不太好! 这里要求输入 `yyyymmdd` 的格式, 如果有一天实际输入改变为 `yyyy mm dd` (这种输入更常见、更通用), 则接口需要重新设计 (对应的函数实现也要修改)。函数的通用性不好。

函数能否进一步细分？

```
23 int getWeek(int day)
24 {
25     // c: century-1, y: year, m:month, w:week, d:day
26     int c, y, m, d, w;
27     y = day/10000;
28     m = (day%10000)/100;
29     d = day%100;
30     if(m<3)
31     {
32         y = y-1;
33         m = m+12;
34     }
35     c = y/100;
36     y = y%100;
37     // Zeller formula
38     w = (y + y/4 + c/4 -2*c + (26*(m+1))/10 + d - 1)%7;
39     if (w<0)
40         w+=7;
41     return w;
42 }
```

一个函数，代码量少，效率高

等价



多个函数，再分别用函数求得 century, year, month, 多个函数调用开销大，但结构好像更清晰些？

```
int getWeek(int day)
{
    // c: century-1, y: year, m:month, w:week, d:day
    int c, y, m, d, w;
    c = getCentury(day);
    y = getYear(day);
    m = getMonth(day);
    d = day%100;
    // Zeller formula
    w = (y + y/4 + c/4 -2*c + (26*(m+1))/10 + d - 1)%7;
    if (w<0)
        w+=7;
    return w;
}
```

```
int getCentury(int day)
{
    int c, y, m;
    y = day/10000;
    m = (day%10000)/100;
    if (m<3)
        y = y-1;
    c = y/100;
    return c;
}
```

```
int getYear(int day)
{
    int c, y, m;
    y = day/10000;
    m = (day%10000)/100;
    if(m<3)
        y = y-1;
    y = y%100;
    return y;
}
```

```
int getMonth(int day)
{
    int m;
    m = (day%10000)/100;
    if(m<3)
        m = m+12;
    return m;
}
```

结构清晰or代码量少？
鱼与熊掌可否兼得？

事实上，本分解非常糟糕！getWeek没必要再进行函数分解，分解的几个函数有相同运算，很冗余，内聚性不好。且getWeek的调用和数据容错没有发生任何改变。getWeek的接口值得再设计！

函数重在接口（庖丁解牛——函数接口设计很重要！）

```
#include <stdio.h>
void printWeek(int);
int getWeekDay(int, int, int);
int main()
{
    int yyyy, mm, dd, w;
    printf("Please input the date.\n");
    printf("format is #### ## ## or #####\n\n");
    while(~scanf("%4d%2d%2d", &yyyy, &mm, &dd))
    {
        w = getWeekDay(yyyy, mm, dd);
        printWeek(w);
    }
    return 0;
}
```

不好的接口

int getWeek(int day);

✗

更好的接口

int getWeekDay(int year, int month, int day);

✓

接口 getWeekDay 能处理输入为 yyyy mm dd 的格式，也能处理输入为 yyymmdd 的格式。yyyy mm dd 的输入更常见。接口是给调用者设计的，要为调用者提供方便！

在main函数里定义三个变量 yyyy mm dd，输入格式为 scanf("%4d%4d%4d", &yyyy, &mm, &dd); 然后直接调用接口 w = getWeekDay(yyyy, mm, dd); 输入的日期是连着的8位数 #####，或分开的年月日 #### ## ##，上述函数均能有效解决！

任尔东西南北风，我自岿然不动！（接受不同输入，很友好！）

函数重在接口

接口的实现

```
int getWeekDay(int year, int month, int day)
{
    int century, weekday;
    if(month < 3)
    {
        year--;
        month += 12;
    }
    century = year / 100;
    year %= 100;
    // Zeller formula
    weekday = (year + year/4 + century/4 - 2*century + (26*(month+1))/10 + day - 1) % 7;
    if (weekday < 0)
    {
        weekday += 7;
    }
    return weekday;
}
```

这样的函数设计，不仅仅用于本例，后面的应用还能大放异彩！平时在OJ上提交的程序，个人应该永远保留！

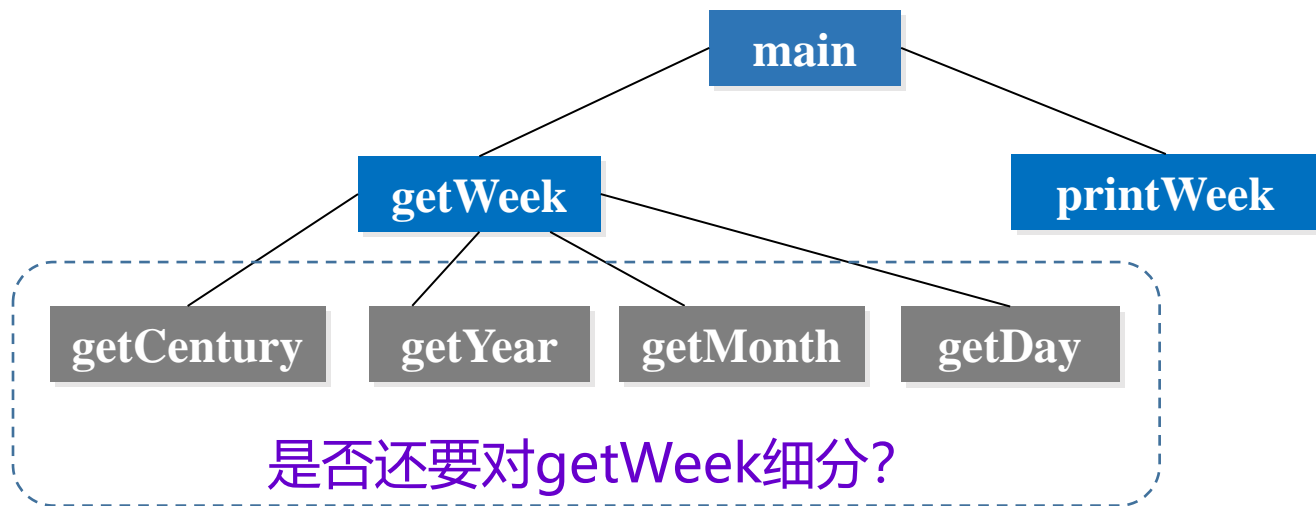
```
#include <stdio.h>
void printWeek(int);
int getWeekDay(int, int, int);
int main()
{
    int yyyy, mm, dd, w;
    printf("Please input the date.\n");
    printf("format is #### ## ## or #####\n\n");
    while(~scanf("%4d%2d%2d", &yyyy, &mm, &dd))
    {
        w = getWeekDay(yyyy, mm, dd);
        printWeek(w);
    }
    return 0;
}
```

轻松应对
不同输入

接口简洁：Make interface easy to use correctly, hard to use incorrectly!

单一责任：防止“多做之过”，实现时内部不要有多余操作！

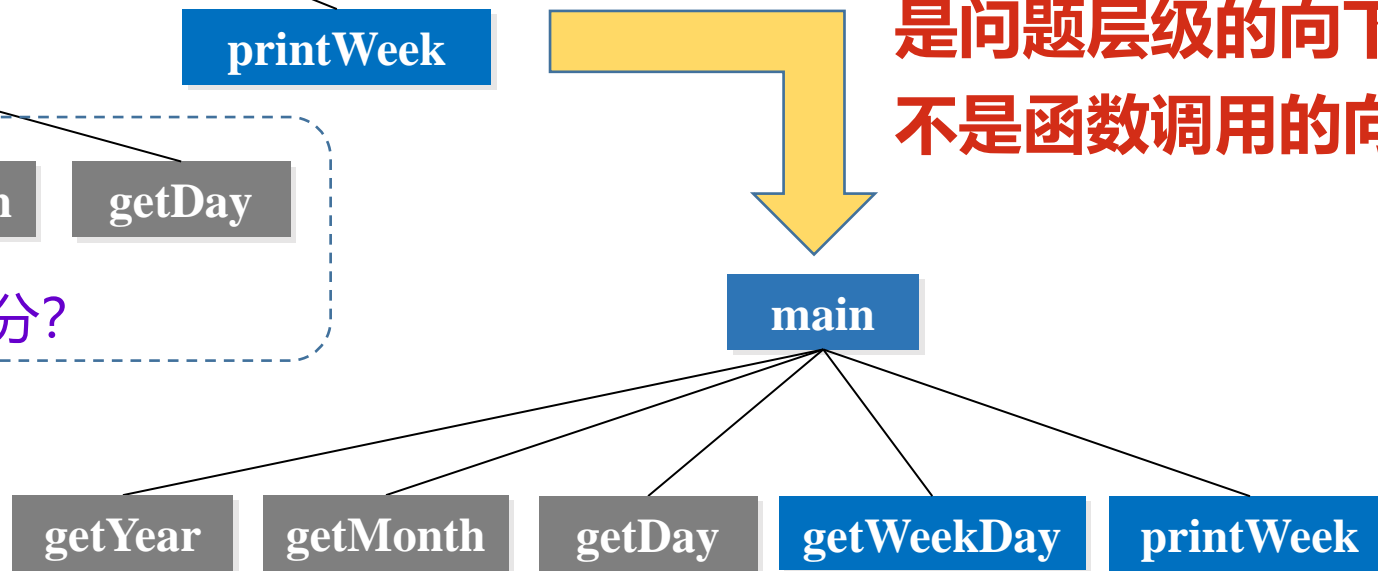
自顶向下的模块化思想



```
int getWeek(int day)
{
    int c, y, m, d, w;
    c = getCentury(day);
    y = getYear(day);
    m = getMonth(day);
    d = day%100;
    // Zeller formula
    ...
    return w;
}
```



自顶向下：
是问题层级的向下！
不是函数调用的向下！

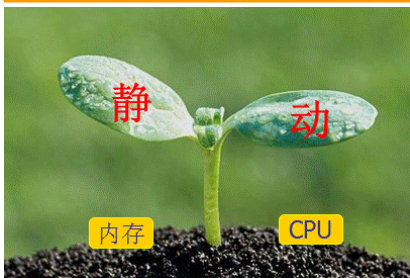


```
int getWeekDay(int year, int month, int day)
{
    ...
    // Zeller formula
    ...
    return weekday;
}
```

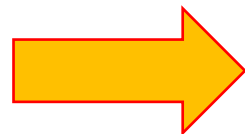


函数的意义

```
#include <stdio.h>
int main()
{
    // your code here
    return 0;
}
```



程序萌芽



解耦与封装



程序长成

```
int getWeek(int day)
{
    int c, y, m, d, w;
    c = getCentury(day);
    y = getYear(day);
    m = getMonth(day);
    d = day%100;
    // Zeller formula
    ...
    return w;
}
```



函数：面向过程语言进行解耦问题与封装实现的主要工具，没有之一！

解耦与封装是个技术问题，更是哲学问题

从调用的角度来看，模块化实现的是功能解耦
从实现的角度来看，模块化实现的是功能封装

```
int getWeekDay(int yy, int mm, int dd)
{
    ...
    // Zeller formula
    ...
    return weekday;
}
```



例5-8：设计万年历（非函数实现版本）

```

1 #include <stdio.h>
2 int main()
3 {
4     int year, month, day, weekday, monthDays, i, n, leap=0;
5     printf("\n请输入某年年份: ");
6     scanf("%d",&year);
7
8     n = year - 1900; //求元旦是星期几
9     n = (n+(n-1)/4+1)%7;
10    weekday = n;
11
12    if((year%4==0 && year%100!=0)||year%400==0)//判断是否是闰年
13        leap = 1;
14
15    printf("\n\n\t\t\t%d\n\n", year);
16    for(month=1; month<=12; month=month+1) //打印12个月的月历
17    {
18        printf("\n%d月份\n",month);
19        printf("-----");
20        printf("星期日 星期一 星期二 星期三 星期四 星期五 星期六");
21        printf("\n");
22        for(i=0; i<weekday; i=i+1) //找当月1日的打印位置
23            printf("    ");
24
25        if(month == 4 || month == 6 || month == 9 || month == 12)
26            monthDays = 30;
27        else if(month == 2)
28        {
29            if(leap == 1)
30                monthDays = 29;
31            else
32                monthDays = 28;
33        }
34        else
35            monthDays = 31;
36
37        for(day=1; day<=monthDays; day++) //打印当月日期
38        {
39            printf("    %2d    ", day);
40            weekday++;
41            if(weekday==7) //打满一星期应换行
42            {
43                weekday = 0;
44                printf("\n");
45            }
46        }
47        if(weekday)
48            printf("\n"); //打完一月应换行
49    }
50
51    return 0;
52 }

```

运行程序

贴图版（防拷贝）

非函数版本

代码略长, 可读性、可维护性、可扩展性, 都不太好!

```

D:\a1ac\example\chap5>c5-8_CalendarMain

请输入某年年份：2024

2024

1月份
-----
星期日  星期一  星期二  星期三  星期四  星期五  星期六
    1      2      3      4      5      6
    7      8      9     10     11     12     13
   14     15     16     17     18     19     20
   21     22     23     24     25     26     27
   28     29     30     31

.
.
.

10月份
-----
星期日  星期一  星期二  星期三  星期四  星期五  星期六
                                1      2      3      4      5
    6      7      8      9     10     11     12
   13     14     15     16     17     18     19
   20     21     22     23     24     25     26
   27     28     29     30     31

11月份
-----
星期日  星期一  星期二  星期三  星期四  星期五  星期六
                                1      2
    3      4      5      6      7      8      9
   10     11     12     13     14     15     16
   17     18     19     20     21     22     23
   24     25     26     27     28     29     30

```

这五个地方用函数来实现会更清晰，更易于维护！

```
int main() // main函数清清爽爽
{
    int year, weekday;
    printf("\ninput year: ");
    scanf("%d", &year);

    printYearCalender(year); // 打印年历

    return 0;
}
```

作业：讲完后面几页后，把这五个地方用五个函数来实现！

```

1 #include <stdio.h>
2 int main()
3 {
4     int year, month, day, weekday, monthDays, i, n, leap=0;
5     printf("\n请输入某年年份: ");
6     scanf("%d",&year);
7
8     n = year - 1900; //求元旦是星期几
9     n = (n+(n-1)/4+1)%7;
10    weekday = n;
11
12    if((year%4==0 && year%100!=0)||year%400==0) //判断是否是闰年
13        leap = 1;
14
15    printf("\n\n\t\t\t\t\t%d\n\n", year);
16    for(month=1; month<=12; month=month+1) //打印12个月的月历
17    {
18        if(month == 4 || month == 6 || month == 9 || month == 11)
19            monthDays = 30;
20        else if(month == 2)
21        {
22            if(leap == 1)
23                monthDays = 29;
24            else
25                monthDays = 28;
26        }
27        else
28            monthDays = 31;
29
30        printf("\n%d月份\n",month);
31        printf("-----\n");
32        printf("星期日 星期一 星期二 星期三 星期四 星期五 星期六\n");
33        printf("-----\n");
34        for(i=0; i<weekday; i=i+1) //找当月1日的打印位置
35            printf("    ");
36
37        for(day=1; day<=monthDays; day++) //打印当月日期
38        {
39            printf("    %2d    ", day);
40            weekday++;
41            if(weekday==7) //打满一星期应换行
42            {
43                weekday = 0;
44                printf("\n");
45            }
46        }
47        if(weekday)
48            printf("\n"); //打完一月应换行
49    }
50
51    return 0;
52 }

```

*例5-9：设计万年历（函数实现）

```
void printYearCalender(int year);           // 打印年历
void printMonthCalendar(int month, int days, int weekday);
int getWeekDay(int year, int month, int day); // 查询某天是星期几
int isLeap(int year); // 判断是否是闰年
int getDaysOfMonth(int year, int month);    // 获得月天数

int main() // main函数清清爽爽
{
    int year, weekday;

    printf("\ninput year: ");
    scanf("%d", &year);

    printYearCalender(year); // 打印年历

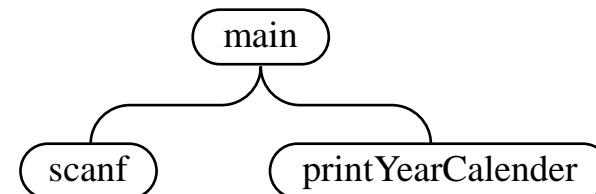
    return 0;
}
```

Terminal output for month 1:

Sun	Mon	Tue	Wed	Thu	Fri	Sat
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

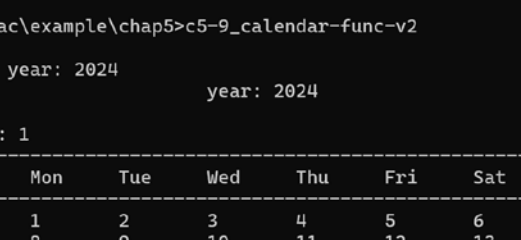
Terminal output for month 2:

Sun	Mon	Tue	Wed	Thu	Fri	Sat
4	5	6	7	1	2	3
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		



主程序就两行：输入和输出

例5-9：设计万年历（函数实现）



```
D:\a1ac\example\chap5>c5-9_calendar-func-v2

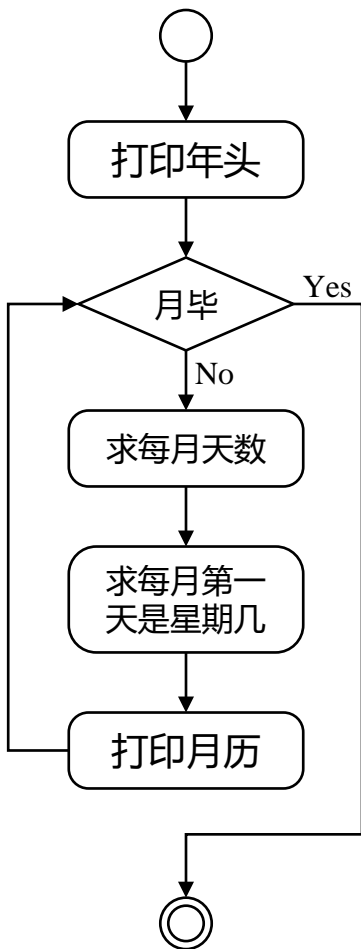
input year: 2024

year: 2024

month: 1
-----
Sun      Mon      Tue      Wed      Thu      Fri      Sat
-----
7         1         2         3         4         5         6
14        8        9        10       11       12       13
21       15       16       17       18       19       20
28       22       23       24       25       26       27

month: 2
-----
Sun      Mon      Tue      Wed      Thu      Fri      Sat
-----
         1         2         3
4         5         6         7         8         9        10
11        12        13        14        15        16        17
18        19        20        21        22        23        24
25        26        27        28        29
```

•



```
void printYearCalender(int year)
{
    int days, month, weekday;
    printf("\t\t\tyear: %d\n", year);
    for(month = 1; month <= 12; month++)
    {
        days = getDaysOfMonth(year, month); // 每月多少天?
        weekday = getWeekDay(year, month, 1); // 月初星期几?
        printMonthCalendar(month, days, weekday); //打印月历
    }
}
```

解题思路：

(1) 逐月打印一年的所有天。

不难，但是很繁琐！
能少写点就太好了！
代码的复用是关键！

例5-9：设计万年历（函数实现）

```
void printYearCalendar(int year)
{
    int days, month, weekday;
    printf("\t\t\tyear: %d\n", year);
    for(month = 1; month <= 12; month++)
    {
        days = getDaysOfMonth(year, month); // 返回值：某年某月有多少天
        weekday = getWeekDay(year, month, 1); // 返回值：每个月的第一天是星期几
        printMonthCalendar(month, days, weekday); // 打印月历，依序打印每个月，从每月的第一天开始
    }
}

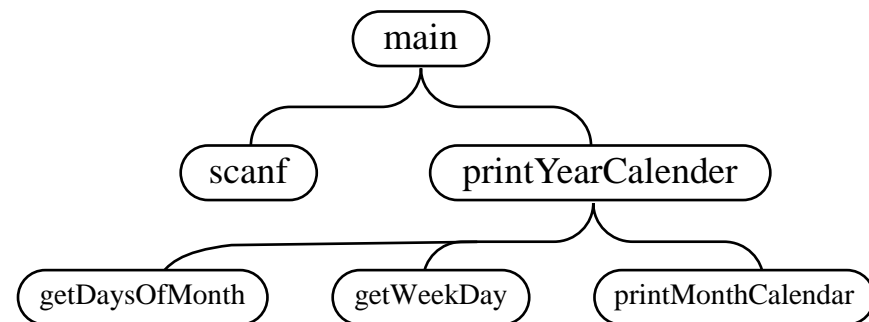
void printMonthCalendar(int month, int days, int weekday)
{
    int monthday, i;
    printf("\nmonth: %d\n", month); // ..... // 打印月头
    printf("-----\n"); // 打印星期表格头
    printf("Sun\tMon\tTue\tWed\tThu\tFri\tSat\n");
    printf("-----\n");

    for (i = 1; i <= days + weekday; i++)
        i <= weekday ? putchar('\t') : printf((i % 7) ? "%d\t" : "%d\n", i - weekday);

    putchar('\n');
}

int getDaysOfMonth(int year, int month) //返回值：某年某月有多少天
{
    .....
}

int getWeekDay(int year, int month, int day)
{
    .....
}
```



我的手机版

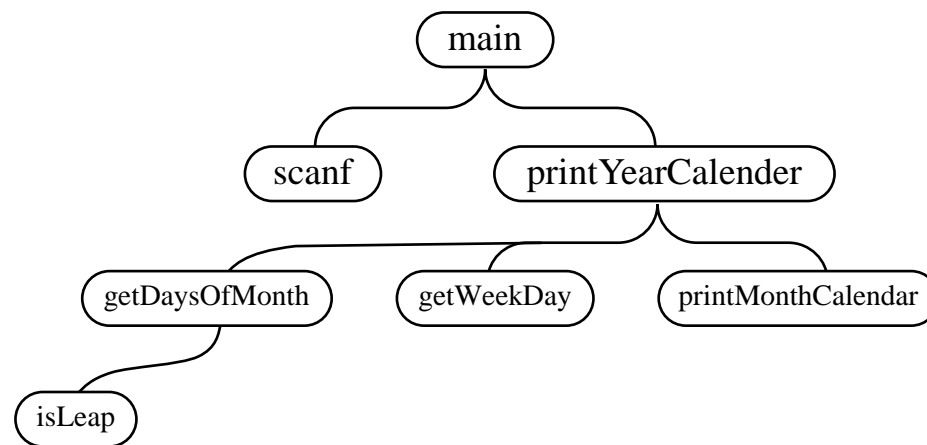
我的C编程版

month: 10						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

例5-9：设计万年历（函数实现）

```
int getDaysOfMonth(int year, int month) //返回值: 某年某月有多少天
{
    int days = 0;
    switch(month)
    {
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:
            days = 31;
            break;
        case 4: case 6: case 9: case 11:
            days = 30;
            break;
        case 2:
            days = isLeap(year) ? 29 : 28;
            break;
    }
    return days;
}

int isLeap(int year) // 代码复用
{
    return (year%4 == 0) && (year%100 != 0) || (year%400 == 0);
    // 逢四则闰, 百年不闰; 四百再闰
}
```



每月的天数

1	2	3	4	5	6	7	8	9	10	11	12
31	28 29	31	30	31	30	31	31	30	31	30	31

例5-9：设计万年历（函数实现）

代码复用（例5-7，Zeller公式）

```
int getWeekDay(int year, int month, int day)
{
    int century, weekday;
    if(month < 3)
    {
        year--;
        month += 12;
    }
    century = year / 100;
    year %= 100;
    // Zeller formula
    weekday = (year + year/4 + century/4 - 2*century + (26*(month+1))/10 + day - 1) % 7;
    if (weekday < 0)
    {
        weekday += 7;
    }
    return weekday;
}
```

← 函数重在接口！函数要实现复用！

求某年year的某月month的第一天是星期几：

weekday = getWeekDay(year, month, 1);
如，输入 2024 8 1，直接算出2024年8月1日是星期四。

直接调用已经写好的 **zeller 公式**，以前写的直接拿来用！

$$W = \left(\left\lfloor \frac{C}{4} \right\rfloor - 2C + Y + \left\lfloor \frac{Y}{4} \right\rfloor + \left\lfloor \frac{26(M+1)}{10} \right\rfloor + D - 1 \right) \bmod 7$$

Where

W : the day of week. (0 = Sunday, 1 = Monday, ..., 5 = Friday, 6 = Saturday)

C : the zero-based century. (= $\lfloor \text{year}/100 \rfloor = \text{century} - 1$)

Y : the year of the century. (= $\begin{cases} \text{year} \bmod 100, & M = 3, 4, \dots, 12, \\ (\text{year} - 1) \bmod 100, & M = 13, 14. \end{cases}$)

M : the month. (3 = March, 4 = April, 5 = May, ..., 14 = February)

D : the day of the month.

NOTE: In this formula January and February are counted as months 13 and 14 of the previous year. E.g. if it is 2010/02/02, the formula counts the date as 2009/14/02.

例5-9：设计万年历（函数实现）

- 主函数简简单单，清清爽爽。
- 每个函数实现特定功能，实现更清晰，更易于维护！
- 利用了以前写的函数，减少重复劳动，如 `getWeekDay()`，`isLeap()`等。

```
void printYearCalender(int year);           // 打印年历
void printMonthCalendar(int month, int days, int weekday);
int getWeekDay(int year, int month, int day); // 查询某天是星期几
int isLeap(int year); // 判断是否是闰年
int getDaysOfMonth(int year, int month);    // 获得月天数

int main() // main函数清清爽爽
{
    int year, weekday;

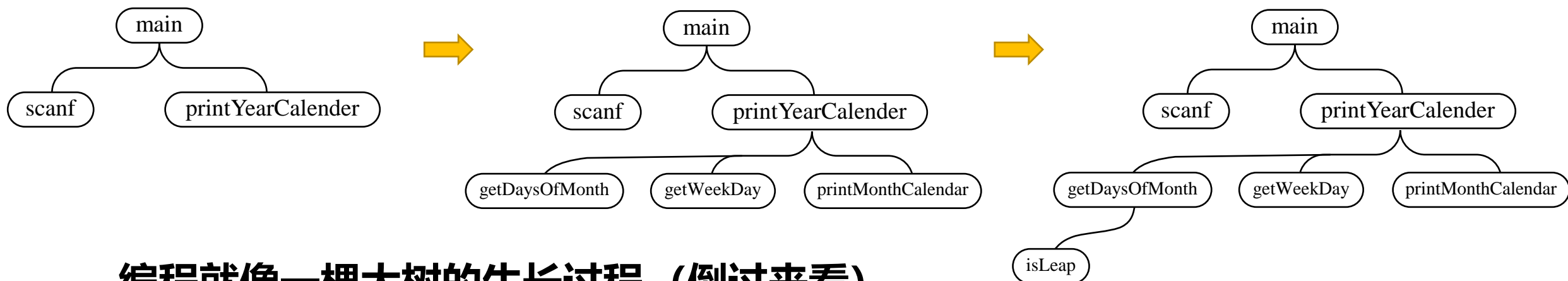
    printf("\ninput year: ");
    scanf("%d", &year);

    printYearCalender(year); // 打印年历

    return 0;
}
```

代码复用，开发省力；
单一责任，接口简洁；
意义明确，格式优雅；
调试方便，扩展便捷。

函数的意义



编程就像一棵大树的生长过程（倒过来看）



程序员/媛
是园丁



不是码农
不是猿

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



萌芽

解耦与封装



长成

程序不是“画”写出来的，是“长”出来的

5.6 递归函数

一看就会，一写就废。

只可意会，不可言传。

递归函数简介

Fibonacci数列（斐波纳契数列）

小张在年初买回一对幼鸽，问， y 年后小张家里的鸽子有多少对（ $1 \leq y \leq 20$ ）？

设一对幼鸽半年后成熟，并繁殖出一对新幼鸽，此后该成熟鸽子每季度繁殖一对幼鸽。新幼鸽半年后也按此规律繁殖。这里不考虑鸽子衰老、死亡的情况。



递归函数简介

Fibonacci数列（斐波纳契数列）

5年后小明家共有多少对鸽子（不考虑鸽子死亡问题）？

第 y 年的开始，对应第 $n = 4*(y-1)+1$ 季度的开始：

1, 1, 2, 3, 5, 8, ...

$$f(1) = f(2) = 1$$

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

$$f(4) = f(3) + f(2) = 2 + 1 = 3$$

$$f(5) = f(4) + f(3) = 3 + 2 = 5$$

...

$$f(n-2) = x$$

$$f(n-1) = y$$

$$f(n) = ?$$

第 n 季度的鸽子数量：

$$f(n) = f(n-1) + f(n-2)$$

鸽子家园

年	季度	日期	鸽子名称（对）
1	1	1.1	a
	2	4.1	a
	3	7.1	a a1
	4	10.1	a a2 a1
2	5	1.1	a a3 a2 a1 a11
	6	4.1	a a4 a3 a2 a21 a1...
	7	7.1	...
			...

递归函数简介

- 一般的函数：按层次方式调用（有显式表达式），如：
- 递归函数(recursive function)：**直接调用自己或通过另一函数间接调用自己的函数**（没有显式表达式），例如
- 递归是计算科学中的重要问题。

$$s = f(n) = a \times n^2 + c$$

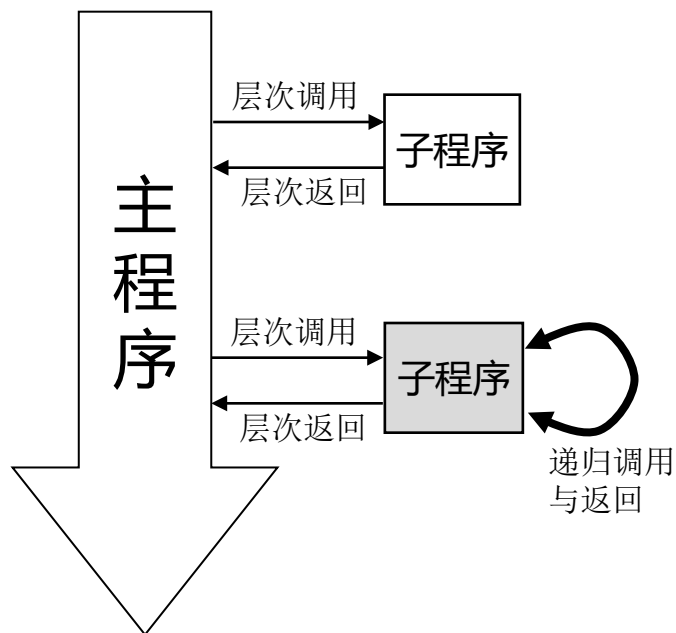
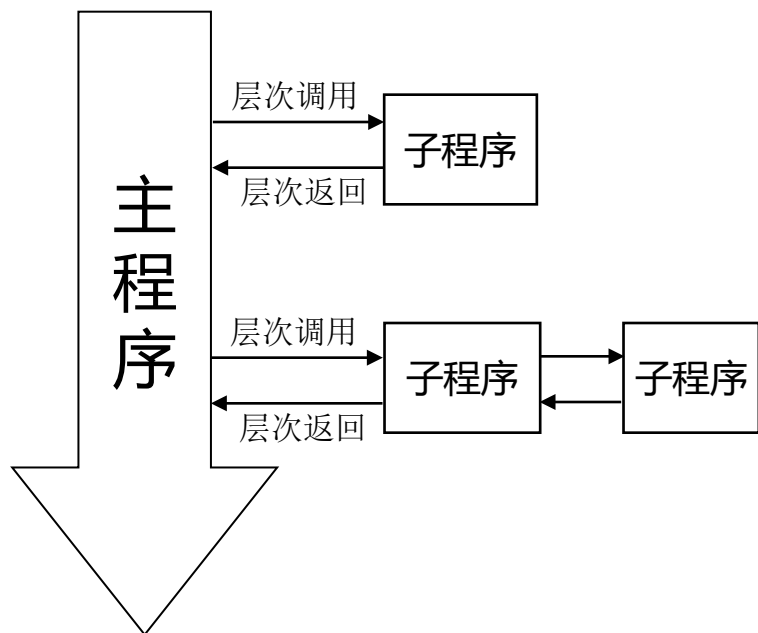
```
int f(int n)
{
    return a*n*n+c;
}
```

$$f(n) = n! \quad // \text{显示表达式}$$

$$f(n) = n \times f(n-1) \quad // \text{隐式表达式?}$$

$$f(n) = n! \\ = n \times (n-1)! = n \times f(n-1)$$

```
int f(int n)
{
    return n*f(n-1);
}
```



递归的概念与思想

1. **基本情况**: 调用递归函数解决问题时, 函数通常只能解决最简单的情况 (称为基本情况)。基本情况的功能调用只是简单地返回一个结果 (或函数调用结束)。
2. **递归调用**: 对复杂问题调用函数时, 函数将问题分成两个概念性部分, 函数中**能够直接处理的部分**和函数中**不能处理的新问题**。
 - ① 不能处理的新问题部分模拟原复杂问题, 但复杂度减小 (问题简化或缩小)。
 - ② 新问题与原问题相似, 函数启动 (调用) 自己的最新副本来处理此新问题, 称为递归调用 (recursive call) 或递归步骤 (recursion step)。
 - ③ 递归步骤可能包括关键字 return, 其结果与函数中需要处理的部分组合, 形成的结果返回原调用者 (回溯)。

例5-10: 求非负长整数的阶乘 $n!$ ($n \geq 0$)

$f(n) = n!$
 $0! = 1, 1! = 1$
 $f(n) = n! = n \times (n-1)! = n \times f(n-1)$
 $f(n-1) = (n-1) \times f(n-2)$
...
 $f(2) = 2 \times f(1)$
 $f(1) = 1$
 $f(0) = 1$

```
long long f(int n)
{
    if (n <= 1)
        return 1;
    return n*f(n-1);
}
```

递归实现

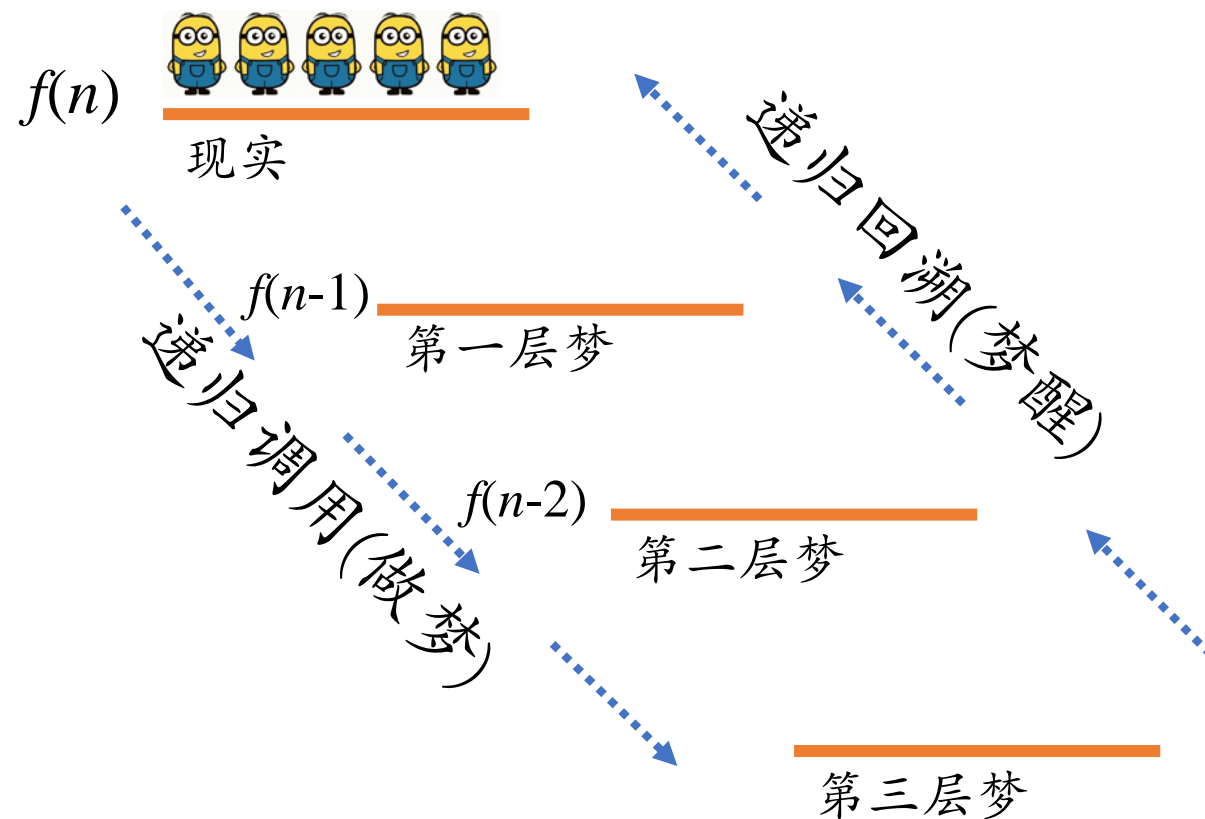
```
long long g(int n)
{
    long long ans = 1;
    for(int i=1; i<=n; i++)
        ans *= i;
    return ans;
}
```

非递归实现

递归原理：《盗梦空间》



盗梦空间：递归的梦

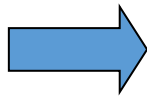


递归函数实例1：阶乘

例5-10：求非负长整数的阶乘 $n!$ ($n \geq 0$)

$f(n) = n!$
 $0! = 1, 1! = 1$
 $f(n) = n! = n \times (n-1)! = n \times f(n-1)$
 $f(n-1) = (n-1) \times f(n-2)$
...
 $f(2) = 2 \times f(1)$
 $f(1) = 1$
 $f(0) = 1$

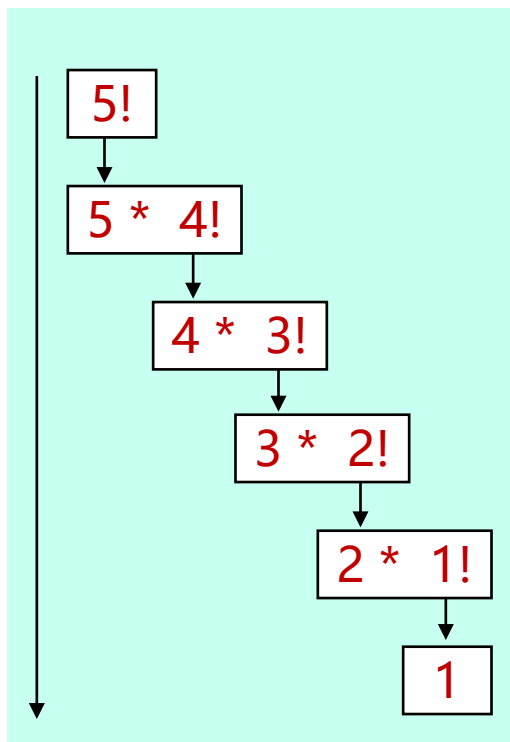
```
int f(n)    // n=3
{
    if (n <= 1) // base case
        return 1;
    return n*f(n-1);
}
```



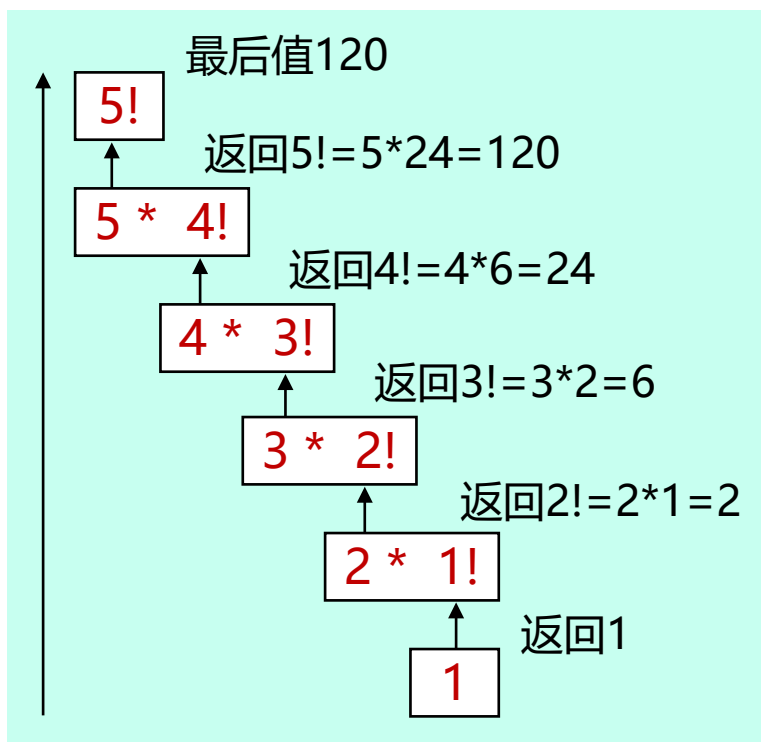
```
f(n)    // n=3
{
    if (n <= 1)    // base case
        return 1;
    return
        n*f(n-1); // m=n-1=2
    {
        if (m <= 1) // base case
            return 1;
        return
            m*f(m-1); // k = m-1 = 1
            {
                if (k <= 1) // base case
                    return 1;
                return k*f(k-1);
            }
    }
}
```

以 $n = 3$ 为例进行递归调用与返回的过程分析

递归函数实例1：阶乘



a) 处理递归调用



b) 每个递归调用向调用者返回的值

注意：
递归中的基本情况不能省略，否则会导致无穷递归。

```
#include <stdio.h>

unsigned long long f( int );

int main()
{
    int i;
    for(i = 1; i <= 10; i++ )
        printf("%2d != %llu\n", i, f(i));

    return 0;
}

unsigned long long f(int n)
{
    if ( n <= 1 )
        return 1;

    return ( n * f(n-1) );
}
```

递归函数实例 2: Fibonacci数

- 神奇的 Fibonacci 数 (斐波纳契数列)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$f(0) = 0, f(1) = 1$$

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

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

$$f(4) = f(3) + f(2) = 2 + 1 = 3$$

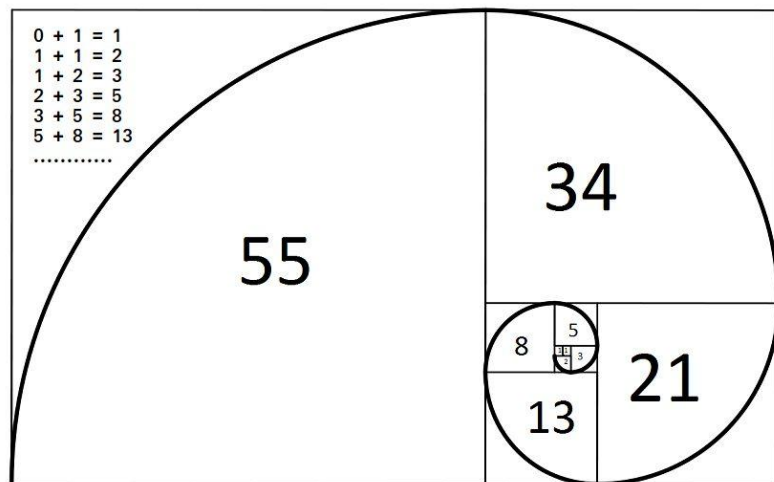
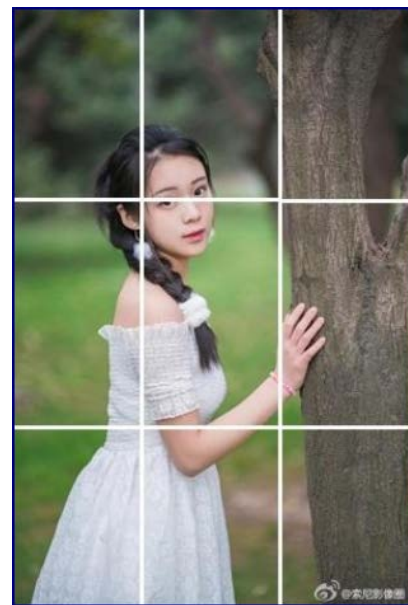
...

$$f(n) = f(n-1) + f(n-2)$$

$$f(n-1)/f(n) \rightarrow 0.618 \quad (n \rightarrow \infty)$$

- 编程, 数学, 与生活中的美:

- 照相取景时人的位置比、色差比
- 房子、门、窗、明信片、书、相片的长宽比
- 动物界、植物界的很多比例符合黄金分割比

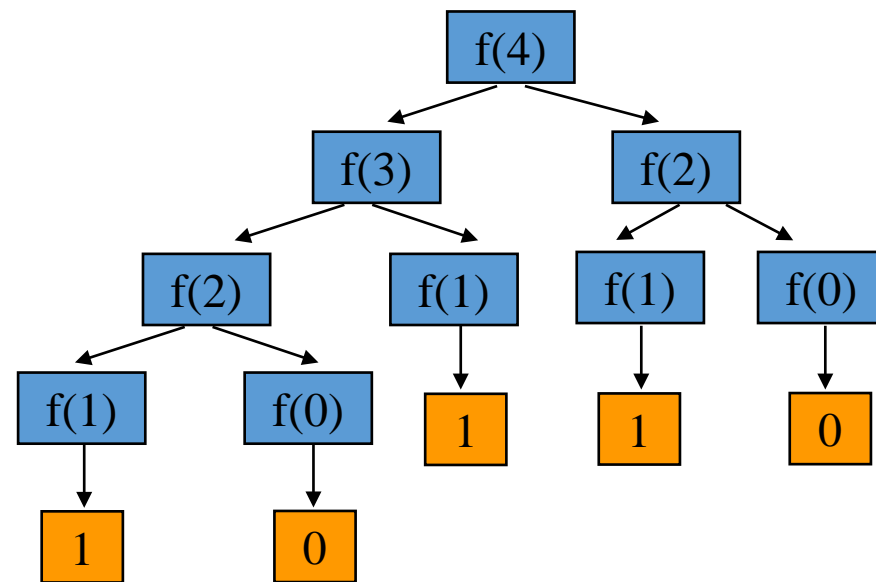


递归函数实例 2: Fibonacci数

例5-11: 求 Fibonacci 数列的值

- Fibonacci 数列递归调用的过程 (以 $f(4)$ 为例)
- Fibonacci 函数中的每一层递归对调用数有“加倍”的效果, 第 n 个 Fibonacci 数的递归调用次数是指数函数(exponential function), $a^n (a > 1)$, 这种问题能让最强大的计算机望而生畏。尽量避免使用计算时间为指数函数的算法。

$$f(n) = f(n - 1) + f(n - 2)$$



递归函数实例 2: Fibonacci数

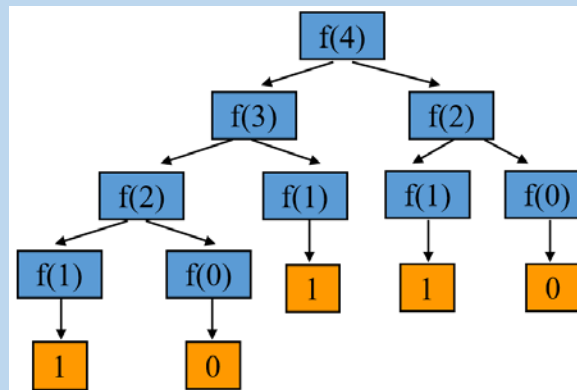
例5-11: 求 Fibonacci 数列的值

$$f(n) = f(n-1) + f(n-2)$$

```
unsigned long long fib(int n )
{
    if ( ( n == 0 ) || ( n == 1 ) )
        return n;

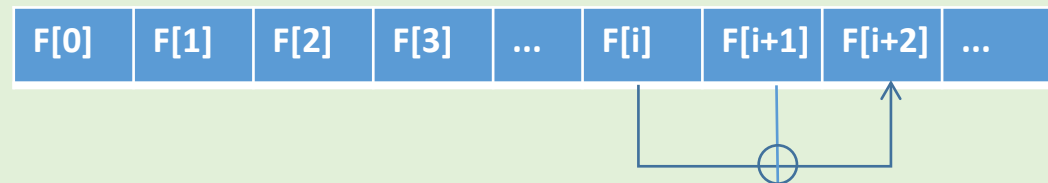
    return fib(n-1) + fib(n-2);
}
```

递归实现:
简洁易懂,
程序好写;
计算很慢!



```
#define N 100
unsigned long long F[101] = {0, 1};
void fib_loop(int n)
{
    int i;
    for(i=2; i<=N; i++)
        F[i] = F[i-1] + F[i-2];
}
```

递推实现:
略增空间,
计算神速!



gcd的循环实现（非递归）

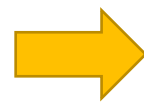
例5-12：求整数 a 和 b 的最大公约数
($a > 0, b \geq 0$)。

辗转相除算法 $\text{gcd}(a, b)$:

1. $\text{gcd}(a, b)$: if b is 0, $\text{gcd}(a, b)$ is a , stop.
2. $\text{gcd}(a, b)$: if $a \% b$ is 0, $\text{gcd}(a, b)$ is b , stop.
3. let $r \leftarrow a \% b$, $a \leftarrow b$, $b \leftarrow r$, go to step 2.

【 $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$ 】

非递归实现【参考例4-16】



```
#include<stdio.h>
int gcd(int, int);
int main()
{
    int a, b, g;
    scanf("%d%d", &a, &b);
    g = gcd(a, b);
    printf("%d\n", g);
    return 0;
}

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

递归函数实例 3: gcd的递归实现

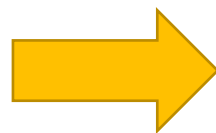
例5-12: 求整数 a 和 b 的最大公约数 ($a > 0, b \geq 0$)

辗转相除算法 $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$:

1. $\text{gcd}(a, b)$: if b is 0, $\text{gcd}(a, b)$ is a , stop.
2. $\text{gcd}(a, b)$: if $a \% b$ is 0, $\text{gcd}(a, b)$ is b , stop.
3. let $r \leftarrow a \% b, a \leftarrow b, b \leftarrow r$, go to step 2.



1. $\text{gcd}(a, b)$: if b is 0, $\text{gcd}(a, b)$ is a , stop;
else, next step;
 2. let $r \leftarrow a \% b, a \leftarrow b, b \leftarrow r$, go to step 1.
- 【 $\text{gcd}(a, b) = \text{gcd}(b, a \% b) = \text{gcd}(a', b')$ 】**



```
#include<stdio.h>
int gcd(int, int);

int main()
{
    int a, b, g;
    scanf("%d%d", &a, &b);
    g = gcd(a, b);
    printf("%d\n", g);
    return 0;
}

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

递归的实现如此优美!

递归函数实例 4：组合数

例5-13：求组合数 (m 个数里选 n 个组合)

$c(m, n)$, or C_m^n , 其中: $m \geq n \geq 0$

分析:

组合恒等式

$$C(m, n) = \frac{m!}{n!(m-n)!}$$

非递归实现: 三个求阶乘运算, 循环

$$C(m, n) = C(m-1, n) + C(m-1, n-1)$$

基本情况

$$C(m, n) = \begin{cases} 1, & n = 0 \\ 1, & m = n \\ 0, & m < n \\ m, & n = 1 \end{cases}$$

递归实现: 递归与基本情况

```
#include <stdio.h>
int comb_num(int, int);
int main()
{
    int m, n, comb;
    scanf("%d%d", &m, &n);
    comb = comb_num(m, n);
    printf("%d\n", comb);
    return 0;
}

int comb_num(int m, int n)
{
    if (n == 0 || m == n)
        return 1;
    if (m < n)
        return 0;
    if (n == 1)
        return m;
    return comb_num(m-1, n) + comb_num(m-1, n-1);
}
```

思考题:

如何用非递归实现求 $c(m, n)$?

提示: 提前把 $k!$ 存在数组 T 的第 k 项 (参考Fib数的非递归实现), 通过打表 (查表), 可快速计算 $c(m, n)$ 。

优点: 节省时间。

缺点: 需要额外空间。

组合数的非递归实现

例5-13-a: 求组合数 (查表法, 打表法)

$c(m, n)$, or C_m^n , 其中: $m \geq n \geq 0$

分析:

组合恒等式

$$C(m, n) = \frac{m!}{n! (m - n)!}$$

非递归实现: 三个
求阶乘运算, 循环

$$C(m, n) = C(m - 1, n) + C(m - 1, n - 1)$$

基本情况

$$C(m, n) = \begin{cases} 1, & n = 0 \\ 1, & m = n \\ 0, & m < n \\ m, & n = 1 \end{cases}$$

递归实现: 递归与基本情况

```
#include<stdio.h>
long long T[15];
long long comb_num(int, int);

int main()
{
    int m, n, comb, i;
    T[1] = 1;
    for(i=2; i<=15; i++)
        T[i] = i*T[i-1];    // 存储i的阶乘, i!

    printf("input m, n (m>=n): ");
    scanf("%d%d", &m, &n);
    printf("%lld\n", comb_num(m, n));
    return 0;
}

long long comb_num(int m, int n)
{
    if ( m < n )
        return 0;
    if ( 1 == n )
        return m;
    if ( n == m || 0 == n )
        return 1;
    return T[m]/(T[n]*T[m-n]);    // 打表法
}
```

*递归函数实例 5：阿克曼数

例5-14：求阿克曼函数

$$ack(0, n) = n + 1$$

$$ack(m, 0) = ack(m - 1, 1)$$

$$ack(m, n) = ack(m - 1, ack(m, n - 1))$$

不易求得显示函数，能得到隐函数（写成跟自己相似的表达式），则可以方便地写递归程序实现。
若用循环，必须获得显式公式。

$ack(3, 12) = 32765$, $ack(3, 13)$ 非常大，递归爆栈？

$ack(4, 0) = 13$, $ack(4, 1)$ 也不易求出来？

```
#include<stdio.h>
int ack(int, int);
int main()
{
    int m, n, r;
    scanf("%d%d", &m, &n);
    r = ack(m, n);
    printf("ack num is: %d\n", r);
    return 0;
}

int ack(int m, int n)
{
    if ( m == 0 )
        return n+1;
    if ( n == 0 )
        return ack(m-1, 1);
    return ack(m-1, ack(m, n-1));
}
```

只可意会，不可言传

递归函数实例 6：汉诺塔

汉诺塔 Hanoi Tower (demo)

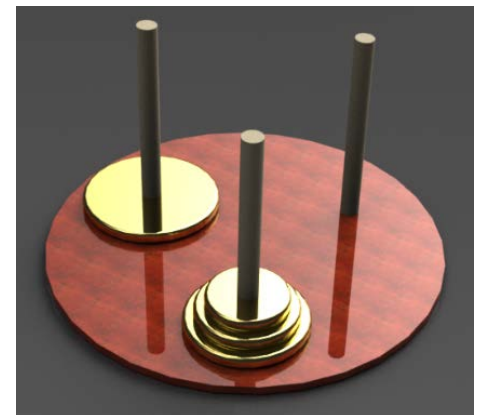
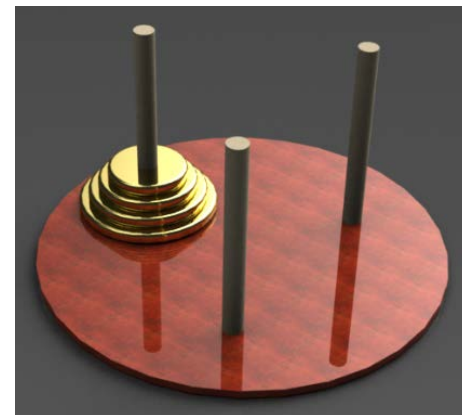
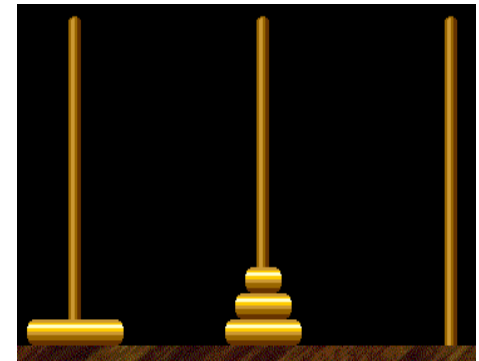
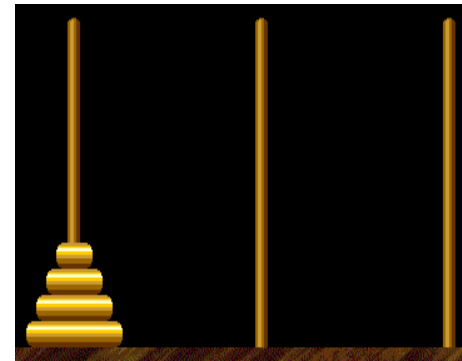
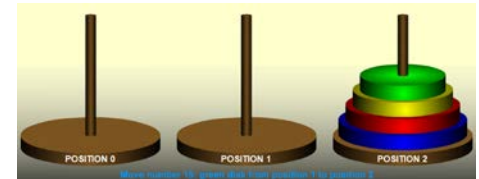
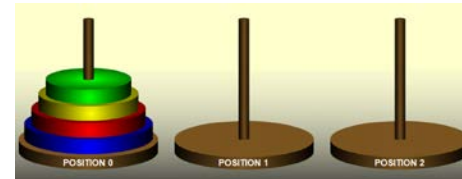
源自印度神话.....

上帝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大小顺序摞着64片黄金圆盘。

上帝命令婆罗门把所有圆盘重新摆放在另一根柱子上（从下往上按大小顺序）。并且规定，只能在三根柱子之间移动圆盘，一次只能移动一个圆盘，任何时候在小圆盘上不能放大圆盘。

有预言说，这件事完成时宇宙会在一瞬间闪电式毁灭。也有人相信婆罗门至今还在一刻不停地搬动着圆盘。

汉诺塔与宇宙寿命：如果移动一个圆盘需要1秒钟的话，等到64个圆盘全部重新摞在另一根柱子上（宇宙毁灭）是什么时候呢？



函数重在接口，递归重在调用



$$f(n) = f(n-1) + f(n-2)$$

```
unsigned long fib( unsigned long n )
{
    if ( ( n == 0 ) || ( n == 1 ) )
        return n;

    return fib(n-1) + fib(n-2);
}
```

1. 所有的梦境是在做梦前，由**造梦师**设计的，并且在做梦的过程中不可以改；
2. 造梦师关心的是**预设场景**，而不是**梦境故事**（后面会自动完成）；
3. 带着**正确的人员进入正确的梦境入口**，那么最终目的就一定能完成。



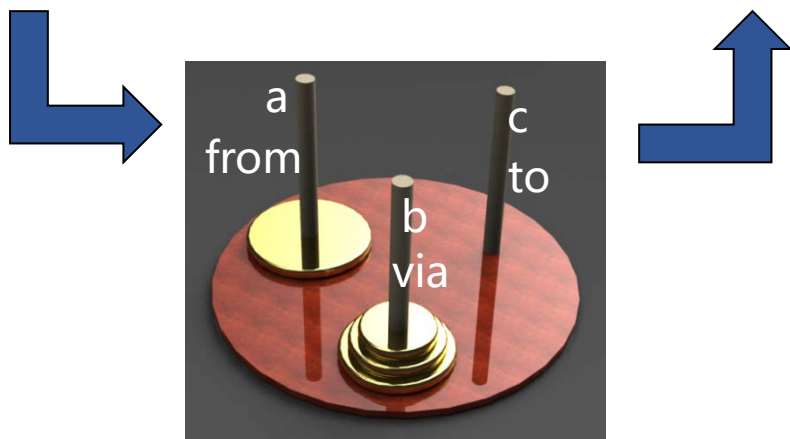
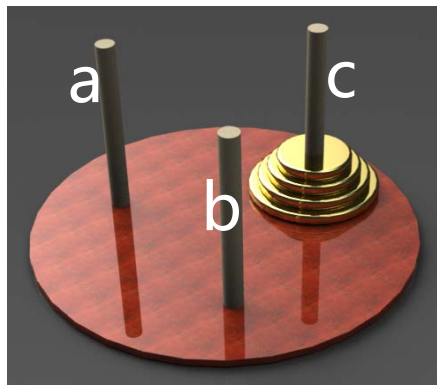
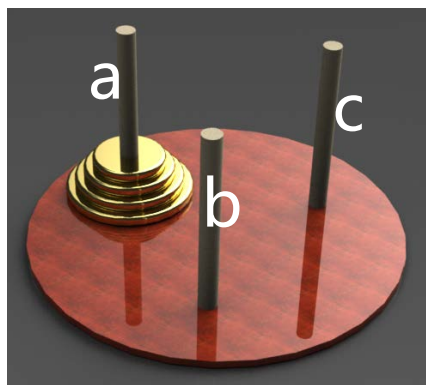
如何破解 “一看就会，一写就废” ？

程序员就是造梦师

预设场景：接口设计、基本情况、分解实现、递归调用；
梦境故事：递归过程（这个一定要忘记！）

递归函数实例 6: 汉诺塔

例5-15: 汉诺塔 Hanoi Tower



函数重在接口，递归重在调用

`void hanoi(int n, char a, char b, char c);` // 接口清晰
`void move(int i, char from, char to);`

这个 n 的意思是有 n 个盘子的 hanoi 塔。
本函数把 n 个盘子从柱子 a 通过 b 挪到 c 上。

`int main()`
{

这个 i 的意思是第 i 号盘。本函数把第 i 号盘子
从柱子 from 直接挪到 to 上。

```
    int num;  
    char a = 'A', b = 'B', c = 'C';  
    scanf("%d", &num);  
    hanoi(num, a, b, c);  
    return 0;  
}
```

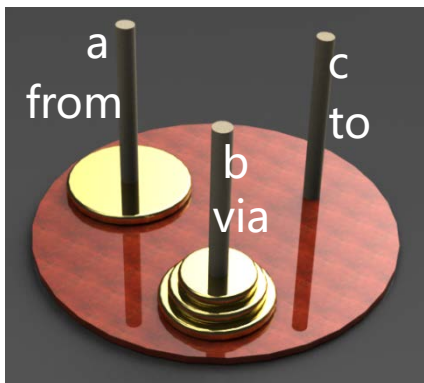
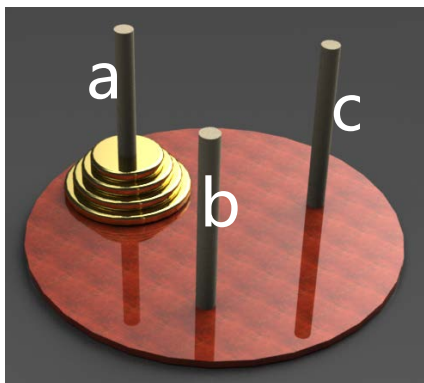
```
void hanoi(int n, char from, char via, char to)  
{
```

```
    if (n == 1)  
    {  
        move(n, from, to);  
        return;  
    }  
    hanoi(n - 1, from, to, via); // 递归: 挪上面的 n-1 个盘子  
    move(n, from, to);           // 移动第 n 个盘子  
    hanoi(n - 1, via, from, to); // 递归: 挪 n-1 个盘子  
}
```

```
void move(int i, char from, char to)  
{  
    printf("Disk %d, %c --> %c\n", i, from, to);  
}
```

递归函数实例 6：汉诺塔

例5-15：汉诺塔 Hanoi Tower



```
void hanoi(int n, char from, char via, char to)
{
    if (n == 1)
    {
        move(n, from, to);
        return;
    }
    hanoi(n - 1, from, to, via); // 递归：挪上面的n-1个盘子
    move(n, from, to);           // 移动第n个盘子
    hanoi(n - 1, via, from, to); // 递归：挪n-1个盘子
}
```

保证正确调用就行，
递归的执行过程交给
计算机去完成！

函数重在接口，递归重在调用

递归函数实例 6：汉诺塔

婆罗门能否让宇宙毁灭？

```
int main()
{
    .....
    hanoi(num, a, b, c);
    return 0;
}

void hanoi(int n, char from, char via, char to)
{
    if (n == 1)
    {
        move(n, from, to);
        return;
    }
    hanoi(n - 1, from, to, via); // 递归调用
    move(n, from, to);
    hanoi(n - 1, via, from, to); // 递归调用
}

void move(int i, char from, char to)
{
    printf("Disk %d, %c --> %c\n", i, from, to);
}
```



递归函数实例 6：汉诺塔

婆罗门能否让宇宙毁灭？

```
int main()
{
    .....
    hanoi(num, a, b, c);
    return 0;
}

void hanoi(int n, char from, char via, char to)
{
    if (n == 1)
    {
        move(n, from, to);
        return;
    }
    hanoi(n - 1, from, to, via); // 递归调用
    move(n, from, to);
    hanoi(n - 1, via, from, to); // 递归调用
}

void move(int i, char from, char to)
{
    printf("Disk %d, %c --> %c\n", i, from, to);
}
```



$$\begin{aligned}T(n) &= T(n-1) + T(n-1) + 1 \\&= 2T(n-1) + 1 \\&= 2(2T(n-2) + 1) + 1 \\&= 2 \cdot 2T(n-2) + 2 + 1 \\&= \dots \rightarrow 2^n\end{aligned}$$

若每秒移动一次

$$n = 64$$

$$T(64) = 2^n = 18446744073709551616 \text{ second}$$

$$\frac{2^n}{(365 \cdot 24 \cdot 60 \cdot 60)} \approx 584,942,417,355 \text{ year}$$

(5800 多亿年)

若每秒移动十亿次： $T(64) \approx 585 \text{ year}$

递归函数实例 7：找座位

例5-16：座位选择（教材上的例5-17）

看书自学

循环与递归

例：求整数 a 和 b 的最大公约数 ($a > 0, b \geq 0$)

```
int gcd(int a, int b)
{
    int r;
    if (b == 0)
        return a;
    while((r = a%b) != 0)
    {
        a = b;
        b = r;
    } // 循环结束后, b 是最大公约数
    return b;
}
```

非递归实现

```
int gcd(int a, int b)
{
    if (b == 0) // 达到基本情况, 递归结束
        return a;
    return gcd(b, a%b);
}
```

递归实现

循环与递归方法的对比

	循环	递归
依赖的控制结构	重复（循环）	选择（条件）
重复方式	使用重复结构	重复函数调用来实现重复
终止条件	循环条件失败时	遇到基本情况
循环方式	不断改变循环控制条件，直到循环条件失败	不断产生最初问题的简化版本（副本），直到达到基本情况
无限情况	循环条件永远不能变为false，则发生无限循环	递归永远无法回到基本情况，则出现无穷递归

使用递归的优缺点

- 缺点:

- ◆ 重复函数调用的开销很大, 将占用很长的处理器时间和大量的内存空间。

- 优点:

- ◆ 能更自然地反映问题, 使程序更容易理解和调试。
- ◆ 在没有明显的迭代方案时使用递归方法比较容易。

$$\text{gcd}(a, b) = \text{gcd}(b, a \% b)$$

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

- 能用递归解决的问题也能用迭代（非递归）方法解决。

递归程序练习

把前面的 7 个递归程序亲自编程实践

1. 阶乘 factorial (递归与非递归)
2. 斐波纳契数 fibonacci (递归与非递归)
3. 最大公约数 gcd (递归与非递归)
4. 组合数 comb (递归与非递归)
5. 阿克曼函数 ack
6. 汉诺塔 hanoi tower
7. 座位选择

*5.7 标准库函数【本部分自行阅读】

- C语言的编译系统以标准库函数的方式实现了大量常用的功能。
- 标准函数的函数原型在相应的标准库头文件（*.h）中说明；函数的定义以目标码的形式保存在函数库中（用户使用时，用#include 引用相应的.h文件）。
- 常用的标准库头文件（*.h）附后。

```
#include <stdio.h>
#include <math.h>

int main()
{
    double a=3, b=4;

    c = sqrt(a+b);

    printf("%.2f\n", c);

    return 0;
}
```

*随机函数的小应用：投单骰子游戏

- rand() 函数产生的并不是真正的随机数，而是**伪随机数** (pseudo-random number)，它是按一定规则（算法）来产生一系列看似随机的数，这种过程是可以重复的。
- 为了在每次重新执行程序时 rand 产生不同的随机序列，就需要为随机数产生器（算法）赋予不同的初始状态，这称为**随机化过程** (randomizing)。
 - ▣ 函数 srand(unsigned) 能设定随机函数 rand 的初始状态，由 srand(unsigned) 的参数决定。
 - ▣ 调用 **srand** 函数又称为设置随机函数的**种子**。
 - ▣ srand 函数的参数通常设置为**系统时钟**。

```
// c-5-17-1.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int point;
    // srand( time( 0 ) );

    point = 1 + rand()%6;

    printf("Dice is: %d\n", point);

    printf((point%2) ? "lose": "win");//单数输，双数赢

    return 0;
}
```



**每次都是你赢。
你作弊!**

*随机函数的小应用：投单骰子游戏

分析：骰子的点数为 1~6

$0 \leq \text{rand}() \leq \text{RAND_MAX}$

$\Rightarrow 0 \leq \text{rand}() \% 6 \leq 5$

$\Rightarrow 1 \leq 1 + \text{rand}() \% 6 \leq 6$

%常用于比例缩放(scaling)，这里6称为比例因子(scaling factor)，1称为平移因子(moving factor)。

说明：时间函数time(0)返回当前时钟日历时间的秒数（从格林尼治标准时间1970年1月1日0时起到现在的秒数）。

作业1：输赢规则不固定，由玩家押点决定，即玩家可押单（双）数，则投出单（双）数时玩家赢，该怎么实现？

```
// c-5-17-2.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int point;
    srand( time( 0 ) );

    point = 1 + rand()%6;

    printf("Dice is: %d\n", point);

    printf((point%2) ? "lose": "win"); //单数输，双数赢

    return 0;
}
```



这下公平了吧！

**投双骰子游戏

作业2：投双骰子游戏

规则：

1. 投双骰子，若首次投掷的点数总和是7或11，则玩家赢；
2. 若首次投掷的点数之和是2、3或12，则玩家输（即庄家赢）；
3. 若首次投掷的点数之和是4、5、6、8、9或10，则这个和是玩家的Point，为了分出胜负，玩家需连续地掷骰子，若点数与Point相同，则玩家赢；若掷到的和是7，则玩家输；其他情况，继续投骰子。

(请参照单骰子游戏，完成该游戏的编程)



一个不好玩



来，投双骰子

本章小结

- 熟练使用常用的标准库函数
- 知道常用的标准库函数
- 能够自定义函数
- 理解函数原型、函数定义的含义
- 理解函数原型与实参的类型转换
- 熟练掌握函数调用与返回
- 理解函数之间传递信息的机制
- 掌握局部变量、全局变量的用法，理解其存储类、作用域的含义
- 了解递归原理，会使用简单的递归函数
- 初步认识模块化编程的思想、良好的软件工程思想与高性能程序的辩证关系

本章小结——函数的设计原则

Make interface easy to use correctly, hard to use incorrectly.

——Scott Meyers

直译1：让接口在被正确调用时好用，被错误调用时难用。

直译2：用易用鼓励接口被正确调用，用难用预防接口被错误调用。

意译1：接口应该这样设计，当正确使用时编程很容易，而错误使用时编程很困难。

意译2：接口设计之道，正用则顺之，误用则逆之。

意译3：接口设计之道：正用则顺，误用则逆。

.....

【课后读物】 C89的标准库头文件

头文件	宏及函数的功能类别
assert.h	运行时的断言检查
ctype.h	字符类型和映射
errno.h	错误信息及处理
float.h	对浮点数的限制
limits.h	编译系统实现时的限制
locale.h	建立与修改本地环境
math.h	数学函数库
setjmp.h	非局部跳转

头文件	宏及函数的功能类别
signal.h	事件信号处理
stdarg.h	变长参数表处理
stddef.h	公用的宏和类型
stdio.h	数据输入输出
stdlib.h	不属于其它类别的常用函数
string.h	字符串处理
time.h	日期和时间

【课后读物】 标准库函数：I/O (include <stdio.h>)

常用的I/O函数（主要文本的 I/O）

函数原型	功能说明	备注
int getchar()	返回从标准输入文件中读入的一个字符	读到文件尾时返回EOF
int putchar(int c)	向标准输出文件中写入一个字符	函数出错时返回EOF
char *gets(char *buf)	从标准输入文件中读入的一行字符，并将其保存在buf所指向的存储区	当读到文件尾部或函数出错时返回NULL，否则返回存储区buf
char *fgets(char *buf, int n, FILE *fp)	从指定文件中读入的一行不超过n-1个字符的字符串，并将其保存在buf所指向的存储区	当读到文件尾部或函数出错时返回NULL，否则返回存储区buf
int puts(const char *string)	向标准输出文件中写入的一行字符，并将字符末尾处的结束符'\0'替换成换行符'\n'	函数出错时返回EOF
int scanf(const char *format [,argument]...)	根据format中的格式规定从标准输入文件中读入数据，并保存到由argument指定的存储单元中	当读到文件尾部或函数出错时返回EOF，否则返回读入数据的个数
int printf(const char *format [,argument]...)	根据format中的格式规定将argument指定的数据写入标准输出文件中	返回输出的字符数，函数出错时返回负值

【课后读物】 标准库函数： I/O (include <stdio.h>)

常用的I/O函数（本学期主要学习部分文本的 I/O，二进制方式的I/O以后再学习）

printf() 函数的进一步说明:

函数原型: int printf(const char *format[, argument] ...);

说明: format是一个字符串（格式说明）， argument 是数据参数（可选）。format中可包含普通的字符串，也可包含由%引导的说明字段（与后面的参数按顺序一一对应，说明对应参数的类型与输出格式）。语法示例: **%[flags] [width] [.precision] type**

标志	作用	默认效果	注释
-	数据在字段宽度内左对齐	右对齐	
+	在有符号的数据类型前加符号	只对负数加符号	
0	数据在字段宽度内加前导0	不加前导0	
#	对类型o,x,X, 分别加前缀0,0x和0X	不加前缀	对类型c,d,i,u,s无效
	对类型e,E,f,g,G,强制输出小数点	只有小数时输出小数点	

```
double x=1.23456;  
int a = 789;  
printf("%.3f, %+08d, %+-8d!", x, a, a);
```

➡

1.235, +0000789, +789 !

【课后读物】 标准库函数： 字符判断 (include <ctype.h>)

函数原型	函数功能	函数原型	函数功能
int isalnum(int c)	c是否是字母或数字	int isprint(int c)	c是否是可打印字符(0x20~0x7e)
int isalpha(int c)	c是否是字母(a~z,A~Z)	int ispunct(int c)	c是否是符号，可打印但非字母数字
int iscntrl(int c)	c是否是控制符(0x00~0x1f,0x7f)	int isspace(int c)	c是否是空白符(0x09~0x0D,0x20)
int isdigit(int c)	c是否是数字	int isupper(int c)	c是否是大写字母(A~Z)
int isgraph(int c)	c是否是可打印字符(空格除外)	int isxdigit(int c)	c是否是16进制数字 (0~9, a~f, A~F)
int islower(int c)	c是否是小写字母(a~z)		

isprint()	isgraph()	isxdigit()	isdigit()	0~9	
			A~F,a~f		
		isalnum()	isalpha()	isupper()	A~Z
				islowwe()	a~z
			isdigit()		0~9
	ispunct()	!@#\$%^&*()_-= ~;'"<>?/~`			
	空格符(' ',0x20)				
isspace()	空格符(0x20), 换页符('\f'),换行符('\n'),回车符('\r'),水平制表符('\t'),垂直制表符('\v')				
iscntrl()	控制字符(0x00~0x1f,0x7f)				

【课后读物】 标准库函数： 字符判断 (include <ctype.h>)

```
// 判断一个字符是否为数字型字符
int isdigit(int c)
{
    return c >= '0' && c <= '9';
}
```

应用实例

```
// 判断一个字符是否为小写字母
int islower(int c)
{
    return c >= 'a' && c <= 'z';
}
```

```
int toupper(int c); // 小写字母to大写
int tolower(int c); // 大写字母to小写
.....
```

```
#include <stdio.h>
#include <ctype.h>

int main()
{
    char a = '6';
    printf("%d\n%d\n%c\n", isdigit(a), a, a);

    a = 'x';
    printf("%d\n%d\n%c\n", isdigit(a), a, a);

    a = 56;
    printf("%d\n%d\n%c\n", isdigit(a), a, a);

    return 0;
}
```

输出为:

```
1
54
6
0
120
x
1
56
8
```

【课后读物】 标准库函数：字符串处理 (include <string.h>)

字符串处理跟数组密切相关

函数原型	函数功能	函数原型	函数功能
char *strcat(char *dst, char *src)	将src追加到dst之后	int strcmp(char *s1, char *s2)	比较字符串s1和s2
char *strncat(char *dst, char *src, size_t n)	将src中的前n个字符追加到dst之后	int strncmp(char *s1, char *s2, size_t n)	比较字符串s1和s2的前n个字符
char *strcpy(char *dst, char *src)	将src复制到dst中	char *strchr(char *str, int c)	在str中查找c首次出现的位置
char *strncpy(char *dst, char *src, size_t n)	将src中的前n个字符复制到dst中	char *strrchr(char *str, int c)	在str中查找c最后一次出现的位置
size_t strlen(char *str)	返回字符串str的长度	char *strstr(char *str, char *s1)	在str中查找s1首次出现的位置

【课后读物】 标准库函数：数学计算 (include <math.h>)

数学库 math.h 的参数和返回类型都是double

函数原型	函数功能	函数原型	函数功能
double sqrt(double x);	x的平方根	double asin(double x);	反正弦函数, x以弧度为单位
double sin(double x);	正弦函数, x以弧度为单位	double acos(double x);	反余弦函数, x以弧度为单位
double cos(double x);	余弦函数, x以弧度为单位	double log(double x);	x的自然对数
double tan(double x);	正切函数, x以弧度为单位	double log10(double x);	x的常用对数
double atan(double x);	反正切函数, 返回值以弧度为单位	double exp(double x);	指数函数 e^x
double atan2(double y, double x);	反正切函数, 返回值以弧度为单位, 根据x和y的符号确定象限	double fabs(double x);	x的绝对值

【课后读物】 标准库函数：通用数据处理 (include <stdlib.h>)

求整数绝对值、随机数、快速排序等

函数原型	函数功能	函数原型	函数功能
int abs(int n);	n的绝对值	int rand();	生成伪随机数
double atof(char *s);	将s转换为double类型的数	void srand(unsigned int s);	设置随机数的种子
int atoi(char *s);	将s转换为int类型的数	void exit(int status);	终止程序运行
void *bsearch(void *key,void *base,size_t num,size_t width,int (*f)(const void *e1, const void *e2));	以二分查找的方式在排序数组base中查找元素key	void qsort(void *base,size_t num,size_t width,int (*f)(const void *e1, const void *e2));	以快速排序方式对数组base进行排序

【课后读物】 标准库函数：通用数据处理 (include <stdlib.h>)

求整数绝对值、随机数、快速排序等

例：

函数原型	函数功能
<code>double atof(char *s);</code>	将s转换为double类型的数

```
double x = atof("1.2345");
```

```
printf("%f\n", x);
```

局部变量、全局变量的小结与进一步对比分析（自学）

局部变量

- 只能在函数内部访问（函数A的变量x与函数B的变量x即便同名，也是两个完全不同的对象），体现了函数封装的特征，使得程序容易管理、数据安全等。
- 局部（自动）变量随函数的调用而存在，函数返回后将消失，从一次调用到下一次调用之间不保持值，每次调用函数时都重新初始化（动态分配）。
- 变量定义时未初始化，则没有确定的数值，读取前应赋值（经常容易忘记！）。
- 约定俗成：循环控制变量常用 i、j、k 等，循环次数变量常用 m、n 等（不是C语言的要求，而是编程人员的习惯，或团队的要求等）。
- 形式参数是局部变量。

全局变量

- 程序运行时始终存在，定义（或声明）后的位置都能访问。
- 若未初始化，则默认为0。
- 常用于定义常量（如数组大小M、N等，系统配置数据等），或多个函数交换数据时的共同变量。
- 在函数中尽量避免或减少对全局变量的修改（便于程序调试和维护）。

一个文件中，全局变量与函数的局部变量可以同名，但不建议这么做（变量命名习惯？）。

```
#include <stdio.h>
void useLocal(void );
void useGlobal(void );
void useLocal2(int );
int x = 10; // 全局变量

int main()
{
    int x = 5; // 局部变量
    printf("%d\n", x);
    useLocal();
    useGlobal();
    useLocal2();

    return 0;
}

void useLocal( void )
{
    int x = 25;
    printf("%d\n", ++x);
}

void useGlobal( void )
{
    x *= 10;
    printf("%d\n", x);
}

void useLocal2(int x)
{
    printf("%d\n", ++x);
}
```

局部变量的补充说明（自学）

C89 标准规定，一个函数中所有局部变量必须集中定义在函数体中第一个执行语句的前面，而不能穿插在执行语句之间。

遵从 C89 标准的编译系统会报错

较老版本的 CB, DevC 会提示仅支持 C99 模式。

```
// int i;  
for(int i=1; i<5; i++)  
    printf("%d, ", i);
```

```
#include <stdio.h>  
int main()  
{  
    int x;  
    scanf("%d", &x);  
    printf("%d\n", x);  
  
    double y;  
    scanf("%lf", &y);  
    printf("%f\n", y);  
  
    return 0;  
}
```

支持 C99 的编译系统接受这样的书写。现在的绝大多数编译器都支持（一般不会遇到问题）？

按 C99 风格，写程序会更灵活些。按 C89 风格，易于对变量集中管理；也会更保险些（现在的编译器基本都对旧的标准兼容；按 C89 风格跟老的程序兼容性好。）

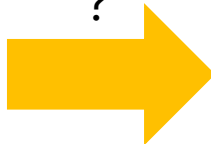
static修饰的静态局部变量【*】

- static将局部变量的生命周期延长到程序执行结束
- 静态局部变量只被初始化一次，空间域无效后时间域仍有效，且保留最近一次的值
- 静态局部变量的空间域只能在定义该变量的函数或语句块内起作用

【例】求阶乘 n!

```
int fun(int);  
int main()           ①  
{  
    int i;  
    for(i=1; i<=10; i++)  
        printf("%d!=%d\n", i, fun(i));  
    return 0;  
}  
int fun(int a)  
{  
    int m = 1, j;  
    for(j=1; j<=a; j++)  
        m = m * j;  
    return m;  
}
```

哪个效率高？



```
int fun(int);           ②  
int main()  
{  
    int i;  
    for(i=1; i<=10; i++)  
        printf("%d! = %d\n", i, fun(i));  
    return 0;  
}  
int fun(int a)  
{  
    static int m = 1;  
    m = m * a;  
    return m;  
}
```



效率更高

main函数内部可以访问m吗？

答：不可以！

5.4 局部变量和全局变量

外部变量与静态变量区别简介* (自学)

作用对象 \ 关键字	extern	static
局部变量	无此用法	变量一次定义，不再释放，随函数第一次调用“永生”，直至程序运行结束
全局变量	使用其他文件中定义的全局变量，需要extern进行声明（注意，非定义，通常放在头文件中）	全局变量本来就是“永生”，static一个全局变量的唯一结果是仅限本文件使用（较少使用）
函数	默认extern，函数默认外连接，有些“大型”代码中明确在函数声明时写上extern，突出说明此函数是其他文件内定义的	本函数仅限本文件调用，消除可能的函数命名冲突
static用于局部变量，决定了其生命周期； static/extern用于函数和全局变量，决定了它们是本文件可见还是其他文件可见。		