

# 第五讲

## 函数

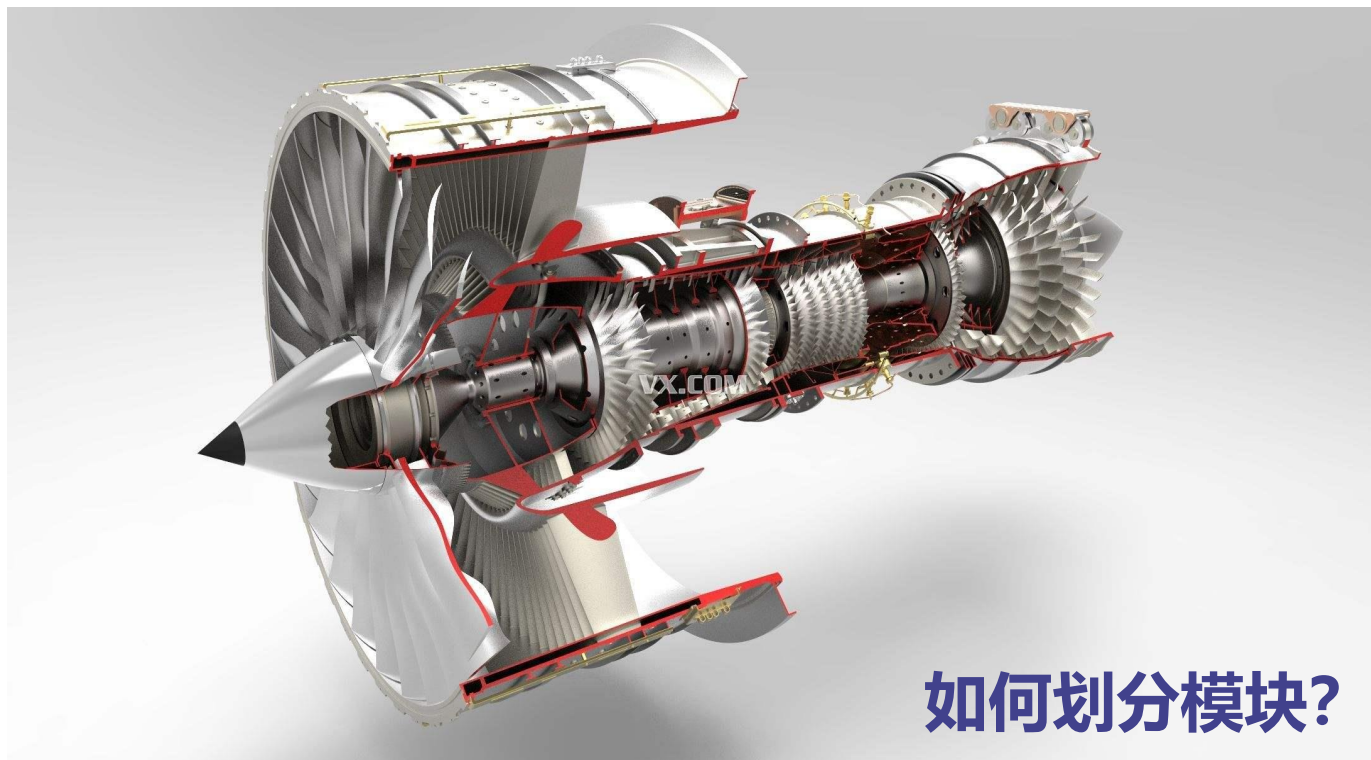
function



# 第5章 函数

## 学习要点

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



如何划分模块?

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，能让发动机能工作，飞机能飞行！

**什么是函数？**

## 5.1 函数

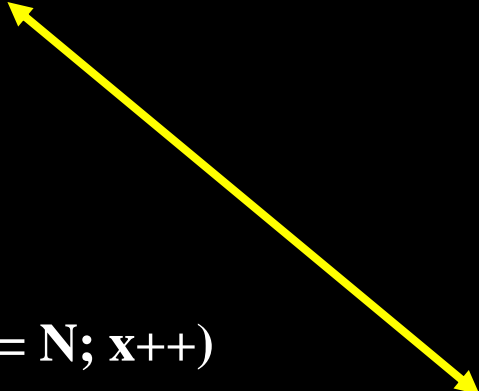
### 【例a5-1】 利用数学标准函数pow计算 3的 x 次方 (x为整数)

```
double pow(double x, double y);//标准函数pow()原型
```

```
#include<stdio.h>
#include<math.h>
#define N 30

int main(){
    int x;
    for(x = 1; x<= N; x++)
        printf("3^%d = %.0f\n", x, pow(3,x));

    return 0;
}
```

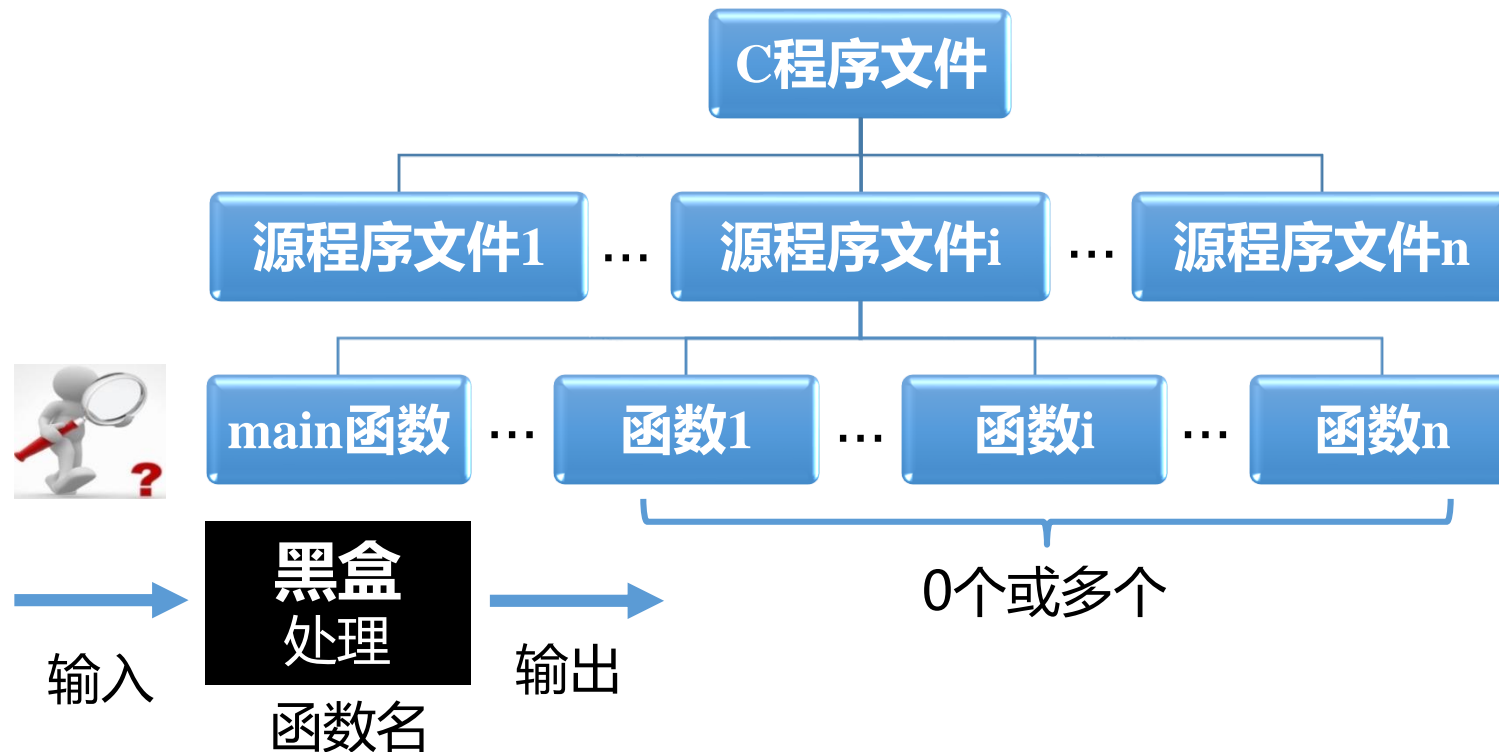


#### 函数八问

- 函数是什么?
- 函数头文件是什么?
- 函数如何定义?
- 函数如何使用?
- 函数如何执行?
- 函数如何输入?
- 函数如何输出?
- 函数main是什么?

# 函数含义

- **从概念角度**：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。
- **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。



**如何定义函数？**

## 5.2 定义函数

### 【例a5-1】 利用数学标准函数pow计算 3 的 x 次方 (x为整数)

```
double pow(double x,double y);//标准函数pow原型
```

```
#include<stdio.h>
#include<math.h>
#define N 40

int main()
{
    int x;
    for(x = 1; x<= N; x++)
        printf("3^%d = %.0f\n", x, pow(3,x));
    return 0;
}
```

标准函数不能满足实际需求怎么办?

答: 自定义函数

测试: ①当N =40时, 结果是多少?

②当N =700时, 结果是多少?

提示: double表示最多15-16位有效数字

double表示最大值约为 $1.79e+308$

( $1.79e+308 \approx 3^{646}$ )

```
3^30 = 205891132094649
3^31 = 617673396283947
3^32 = 1853020188851841
3^33 = 5559060566555523
3^34 = 16677181699666570
3^35 = 50031545098999704
3^36 = 150094635296999140
3^37 = 450283905890997380
3^38 = 1350851717672992000
3^39 = 4052555153018976300
3^40 = 12157665459056929000
```

N=40

部分输出

```
3^647 = 1
3^648 = 1
3^649 = 1
3^650 = 1
3^651 = 1
3^652 = 1
3^653 = 1
3^654 = 1
3^655 = 1
3^656 = 1
3^657 = 1
```

N=700

# 自定义函数

## 【例a5-1-1】自定义函数计算3的x次方（x为整数）

```
#include<stdio.h>
#define N 40
long long u_pow(int, int);
int main(){
    int x;
    for(x = 1; x<= N; x++)
        printf("3^%d = %lld\n", x, u_pow(3,x));
    return 0;
}
long long u_pow(int y, int x){
    long long power = 1;    int i;
    for(i = 1; i <= x; i++)
        power *= y;
    return power;
}
```

函数声明

函数调用

自定义函数

```
#include<stdio.h>
#include<math.h>
#define N 40

int main(){
    int x;
    for(x = 1; x<= N; x++)
        printf("3^%d = %.0f\n", x, pow(3,x));
    return 0;
}
```

函数声明

函数调用

标准函数

### 函数类型

- **标准函数（库函数）**：由C语言定义，可直接调用，但要用include命令引入包含其函数声明的头文件
- **自定义函数**：用户根据特定需求自己实现，不用包含头文件，但需要先定义（或声明）再调用



# 自定义函数

## 【例a5-1-1】自定义函数计算 3 的 x 次方 (x为整数)

```
#include<stdio.h>
#define N 40
long long u_pow(int, int);
int main(){
    int x;
    for(x = 1; x <= N; x++)
        printf("3^%d = %lld\n", x, u_pow(3,x));
    return 0;
}
```

函数声明

函数调用

函数定义

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

当N=40时,  
结果是多少?

### 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

- **函数名**：有意义的标识符，详细的命名规则请复习PPT 2.2节（视频2.1）的知识。
- **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)。
- **返回值类型**：是被调函数向主调函数返回值的数据类型，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。

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）

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

返回值类型 函数名(形参列表)

```
{  
    函数体;  
}
```

# 函数声明（函数原型）

## 【例a5-1-1】自定义函数计算 3 的 x 次方（x为整数）

```
#include<stdio.h>
#define N 40
long long u_pow(int, int);
int main(){
    int x;
    for(x = 1; x<= N; x++){
        printf("3^%d = %lld\n", x, u_pow(3,x));
    }
    return 0;
}

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

函数声明

函数调用

函数定义

函数声明（~函数原型）具有重要作用：

- **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
- **格式**：函数头+分号（形参变量名可省略）
  - ◆ 格式一：long long u\_pow(int y, int x);
  - ◆ 格式二：long long u\_pow(int, int);
- **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定义和函数调用前。函数声明不能独立存在，必须和函数定义配套使用。

# 函数定义方式

- **方式一**：先声明、后调用、再定义
- **方式二**：先定义、后调用（函数定义中的函数头也同时作为函数原型）

```
#include<stdio.h>
#include<math.h>
#define N 40
long long u_pow(int, int);
int main(){
    int x, y;
    for(x = 1; x<= N; x++)
        printf("3^%d = %lld\n", x, u_pow(3,x));
    return 0;
}
long long u_pow(int y, int x){
    long long power = 1;
    int i;
    for(i = 1; i <= x; i++)
        power *= y;
    return power;
}
```



推荐风格

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

```
#include<stdio.h>
#include<math.h>
#define N 40
long long u_pow(int y, int x){
    long long power = 1;
    int i;
    for(i = 1; i <= x; i++)
        power *= y;
    return power;
}
int main()
{
    int x, y;
    for(x = 1; x<= N; x++)
        printf("3^%d = %lld\n", x, u_pow(3,x));
    return 0;
}
```

方式一：先整体声明，后定义细节，快速定位到程序的主体main函数

VS

方式二：先定义细节，后按需调用，头重脚轻、削弱了main函数主体地位

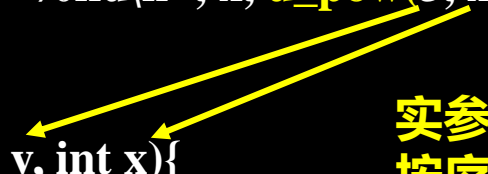
# 函数调用

## 【例a5-1-1】自定义函数计算 3 的 x 次方 (x为整数)

```
#include<stdio.h>
#define N 40
long long u_pow(int, int);

int main(){
    int x;
    for(x = 1; x<= N; x++)
        printf("3^%d = %lld\n", x, u_pow(3, x));
    return 0;
}

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



The diagram consists of two yellow arrows pointing from the arguments '3' and 'x' in the function call `u_pow(3, x)` within the `main` function to the parameters `y` and `x` in the function definition `u_pow(int y, int x)`. This illustrates the flow of actual arguments to formal parameters.

实参  
按序  
传给  
形参

### 函数调用

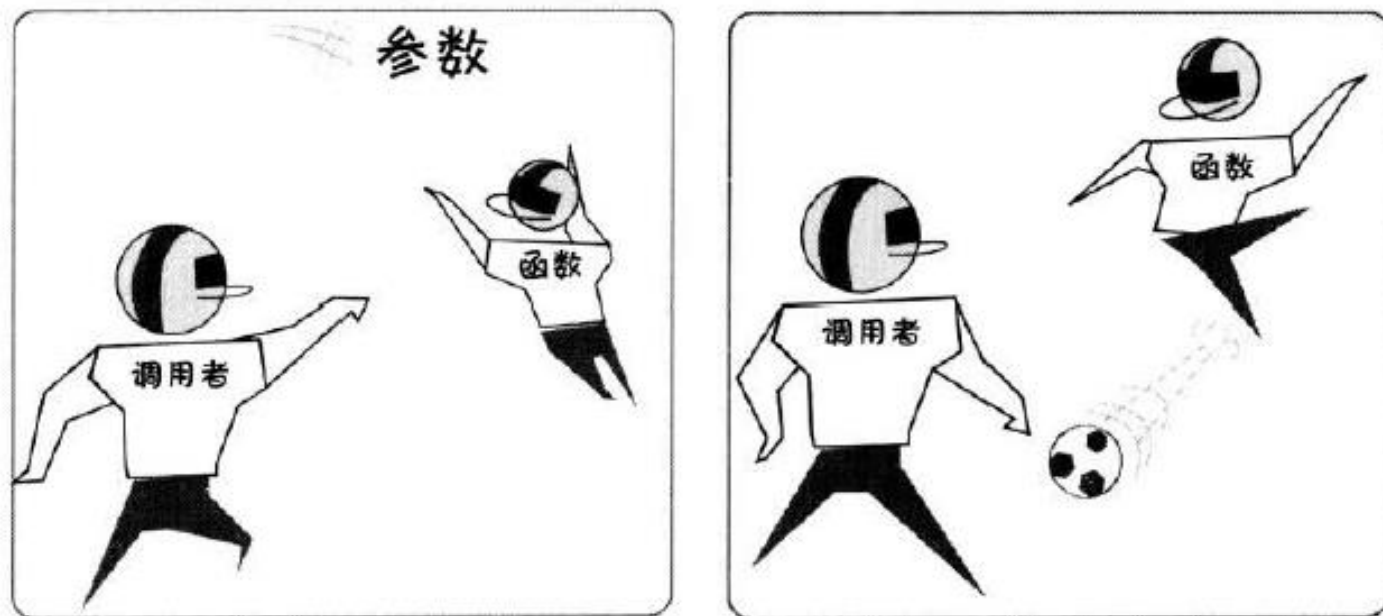
- **含义**：用于执行已经定义好的函数
- **格式**：函数名(实参列表)
- **作用**：将实参按序传递给形参
- **位置**：形参和实参的数量、顺序、类型三者应保持一致；
- **用法**：除main函数外的函数必须通过调用才能被执行
  - ◆ 单独的函数调用语句
  - ◆ 出现在表达式中
  - ◆ 函数嵌套调用

**如何调用函数？**

## 5.3 函数调用

### • 函数调用现场

- ◆ **两个对象**：被调函数、主调函数
- ◆ **三类数据**：传递值、接收值、返回值
- ◆ **四个动作**：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）

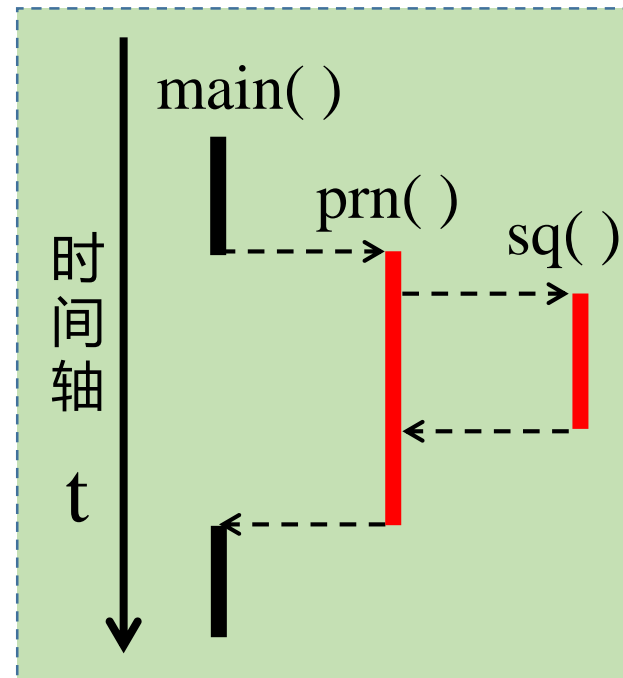
```
#include <stdio.h>

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;
}
```



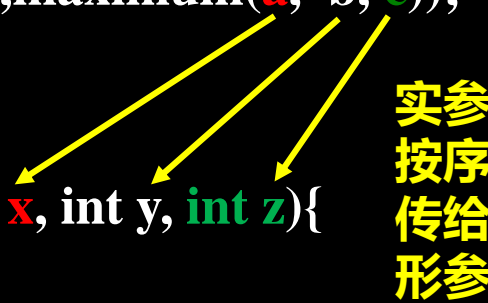
- 从哪离开，回到哪里
- 各司其职、独立完成
- 关注结果、忽略细节
- 分工有序、协同合作



# 函数传递值

## 【例a5-2】求最大值

```
#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**：也称为**形式参数**，函数定义中参数，在函数中也相当于局部变量。
- **实参**：也称为**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，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a5-2】求最大值

```
#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。
- 一个函数可以有多条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

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

①

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



②

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



```
void isLeapYear(int y){  
    if((y%4==0 && y%100!=0) || y%400==0){  
        printf("%d is a leap year", y);  
        return ;  
    }  
    printf("%d is not a leap year", leap);  
}
```

③

## 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；
- 返回空，强制退出当前函数，如③；

## 带有返回值的标准函数

- 根据返回值设计条件表达式；
- scanf返回int类型，则while(scanf(..)!=EOF)
- gets返回char \*类型，则while(gets(..)!=NULL)
- strcmp返回int类型，则if(strcmp(str1,str2)==0)
- isdigit返回int类型，它具有逻辑判断能力，则if(isdigit(c))

# 函数嵌套案例

```
#include <stdio.h>
void isLeapYear(int leap);
int input();
void inputError();
int main(){
    int year;
    year = input();
    if(year > 0) isLeapYear(year);
    return 0;
}
void isLeapYear(int leap){
    if((leap%4==0 && leap%100!=0)||leap%400==0){
        printf("%d is a leap year", leap);
        return;
    }
    printf("%d is not a leap year", leap);
}
int input(){
    int y = 0;
    scanf("%d",&y);
    if (y > 0)    return y;
    inputError();
    return 0;
}
void inputError(){
    printf("Invalid input.");
}
```

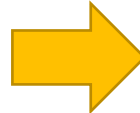
## 函数嵌套调用

- 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
  - ◆ 执行到达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)

# 函数应用案例

## 【例5-3】求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;
    }
    printf("gcd is: %d\n", b<0 ? -b : b);
    return 0;
}
```



```
#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;
}
```

- 所有代码堆积在main函数中
- 程序结构不清晰
- 代码不可复用

- 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

## 【例5-4】求水仙花数（3位）

```
#include <stdio.h>
int main()
{
    short a, b, c, i;
    for(i = 100; i < 1000; i++){
        a = i/100;
        b = (i%100)/10;
        c = i%10;
        if(i == a*a*a + b*b*b + c*c*c)
            printf("%5d\n",i);
    }
    return 0;
}
```

- 不易扩展
- 需求更改后要重写



```
#include <stdio.h>
short isDaffodil(short);

int main()
{
    short i;
    for(i = 100; i < 1000; i++)
        if(isDaffodil(i))
            printf("%5d\n",i);
    return 0;
}

short isDaffodil(short x)
{
    short a, b, c;
    a = x/100;
    b = (x%100)/10;
    c = x%10;
    return (x == a*a*a + b*b*b + c*c*c);
}
```

- 可扩展性好，如4位水仙花数
- 只修改函数实现，保持调用方法不变

# 函数应用案例

**【例5-5】求pi（精确到小数点后x位，x是一个正整数，如x=10）**

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

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

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。

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, 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 *= 5;
    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 *= 239;
    d = 1.0/(n*k);
    s2 += d*sign;
}
```

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

- 代码冗余度高（两段代码几乎相同）
- 可维护性差，参数修改易出错



将二层循环  
优化成  
一层循环

```
#include <stdio.h>
double item(double, double);
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;
    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;
}
```



效率提高n倍

- 函数item可重复调用（一次定义，多次使用）
- 可变更性强，参数易修改



# 函数应用案例

## 【例5-6】已知本月天数，某天是星期几，求下月第k天是星期几

```
#include <stdio.h>
int main()
{
    short x, y, n, k, m;
    //we choose 2020/04/01, Wed.
    x = 1; y = 3; n = 30;
    printf("day of next moth:");
    scanf("%d",&k);
    m = (n - x + k + y) % 7;
    printf("the weekday is: %d\n",m);
    return 0;
}
```



```
#include <stdio.h>
int weekDay(int, int, int, int);
int main()
{
    short x, y, n, k, m;
    x = 1; y = 3; n = 30;
    printf("day of next moth:");
    scanf("%d",&k);
    m = weekDay(x, y, n, k);
    printf("the weekday is: %d\n",m);
    return 0;
}
int weekDay(int x, int y, int n, int k)
{
    return (n - x + k + y) % 7;
}
```

- 代码耦合性较强
- 不易模块化编程

- 即便只有一条语句，也值得定义一个函数
- 增加代码可读性、可维护性

再学变量

## 5.4 局部变量和全局变量

### • 变量空间域和时间域

- ◆ **空间域**：指变量的**访问区域**（作用范围），它由变量在程序中的**定义位置**决定
- ◆ **时间域**：指变量的**生命周期**（存在时间），它由变量在内存中的**存储位置**决定

### • 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**

```
#include <stdio.h>
int a, b;
void fun();
int main(){
    int a = 5, b = 10;
    fun();
    printf("%d,%d\n",a, b);
    return 0;
}
void fun(){
    a = 100, b = 200;
}
```

①

```
#include <stdio.h>
int a, b;
void fun(int, int);
int main(){
    a = 5, b = 10;
    fun(a,b);
    printf("%d,%d\n",a, b);
    return 0;
}
void fun(int a, int b){
    a = 100, b = 200;
}
```

②

# 局部变量空间域

## • 局部变量空间域

- ◆ C语言通过**定义局部变量的语句块**确定局部变量空间域
- ◆ 语句块是指被一对 **{ }** 括起来的若干条语句
- ◆ 找到定义局部变量语句块的结束位置 **}**，是确定局部变量空间域的关键
- ◆ 变量重名时，空间域小的变量起作用
- ◆ 空间域大的变量可以在不大于它的范围内访问，反之不行

局部变量只能在定义它的语句块内被访问

```
#include <stdio.h>
int test();
int main()
{
    int x = 5;
    x = test();
    {
        int x = 7;
    }
    printf("%d",x);
    return 0;
}

int test(){
    int x = 3;
    return x;
}
```

# 完整的变量定义格式

## • 变量时间域

- ◆ 时间域是指变量在内存中的存储时间
- ◆ C语言通过**存储类型**确定的变量时间域
- ◆ 存储类型决定变量的存储位置，一共有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

存储类型 数据类型 变量名;

```
#include <stdio.h>
int fun(int);
int main()
{
    int i;
    for(i=1; i<=5; 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;
}
```

# 静态局部变量

## • static修饰局部变量

- ◆ 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
- ◆ 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

```
#include <stdio.h>
int fun(int);
int main(){
    int i;
    for(i=1;i<=5;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;
}
```

①



```
#include <stdio.h>
int fun(int);
int fun(int a){
    static int m = 1;
    m = m * a;
    return m;
}
int main(){
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i, fun(i));
    return 0;
}
```

②



效率更高

main函数可以  
访问m吗?  
答：不可以

# 局部变量和全局变量应用

## 【例5-8】自定义函数实现两个整数交换

```
#include <stdio.h>
void swap(int, int);
int main(){
    int a = 5, b = 6;
    swap(a, b);
    printf("a=%d,b=%d", a, b);
    return 0;
}
void swap(int x, int y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

①

```
#include <stdio.h>
int a, b;
void swap(int, int);
int main(){
    a = 5, b = 6;
    swap(a, b);
    printf("a=%d,b=%d", a, b);
    return 0;
}
void swap(int x, int y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

②

```
#include <stdio.h>
int a, b;
void swap();
int main(){
    a = 5, b = 6;
    swap();
    printf("a=%d,b=%d", a, b);
    return 0;
}
void swap(){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

③

输出结果: a=5, b=6  
错误原因: 形参是它所在函数的局部变量, 函数执行后  
形参被销毁

思考问题: 是否能够通过return语句同时返回多个形参值? 值传递? 地址传递?

# 5.4 局部变量和全局变量

局部变量与全局变量特征简介

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}	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	用于数组时，最大约100KB级	用于数组时，可达100MB级



# 5.4 局部变量和全局变量

外部变量与静态变量区别简介

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。		

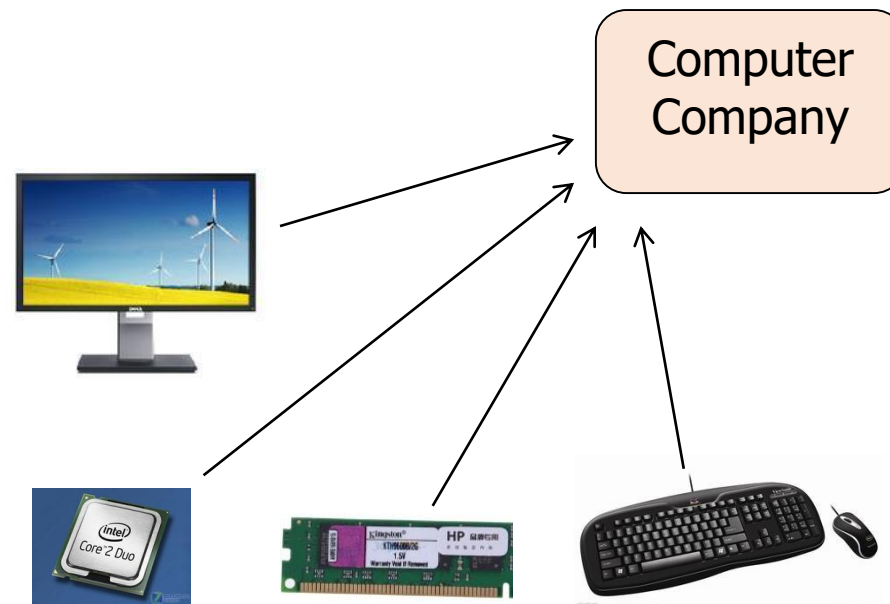
# 为什么要使用函数

# 使用函数的模块化编程思想简介

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

“分而治之，各个击破” (divide and conquer)



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。  
积木分解出来的是模块，函数分解出来的是问题！

# 使用函数的模块化编程思想简介—特点与优点

- **抽象**：复杂、零散问题模型化、概念化、标准化，整体规划，逐步细化
- **简化**：大事化小，繁事化简；分而治之，各个击破
- **封装**：数据与信息隐藏，安全，方便，简单
- **组合**：小功能程序聚合成大功能的程序
- **复用**：减少重复劳动，减少犯错风险

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

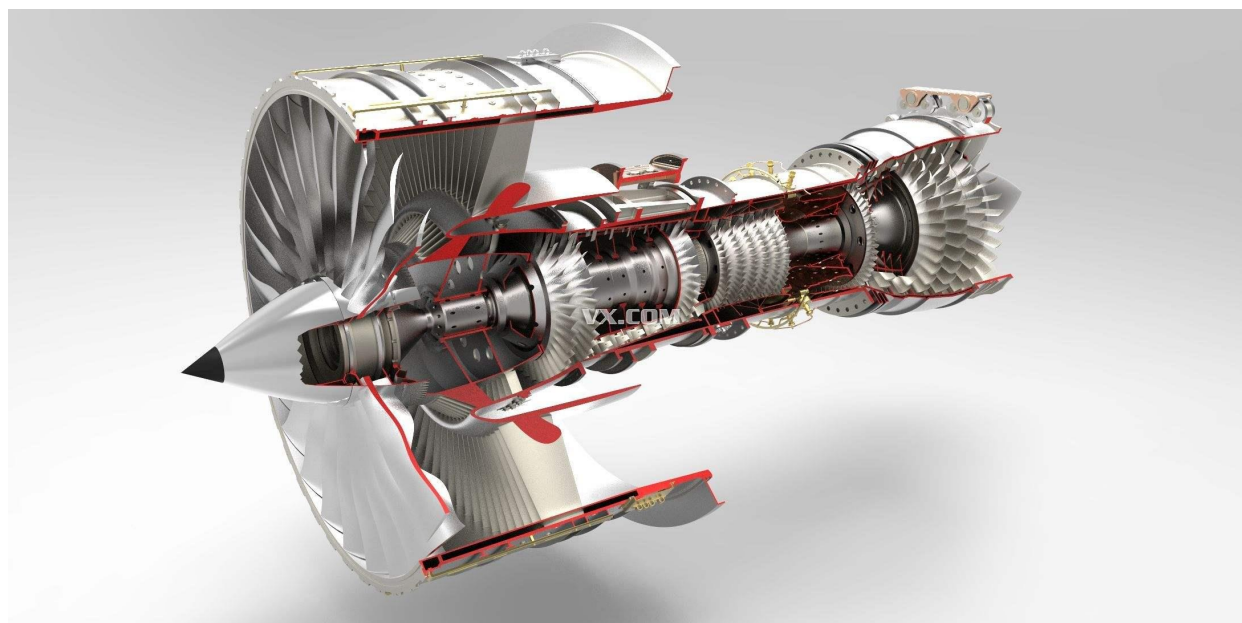
VS

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

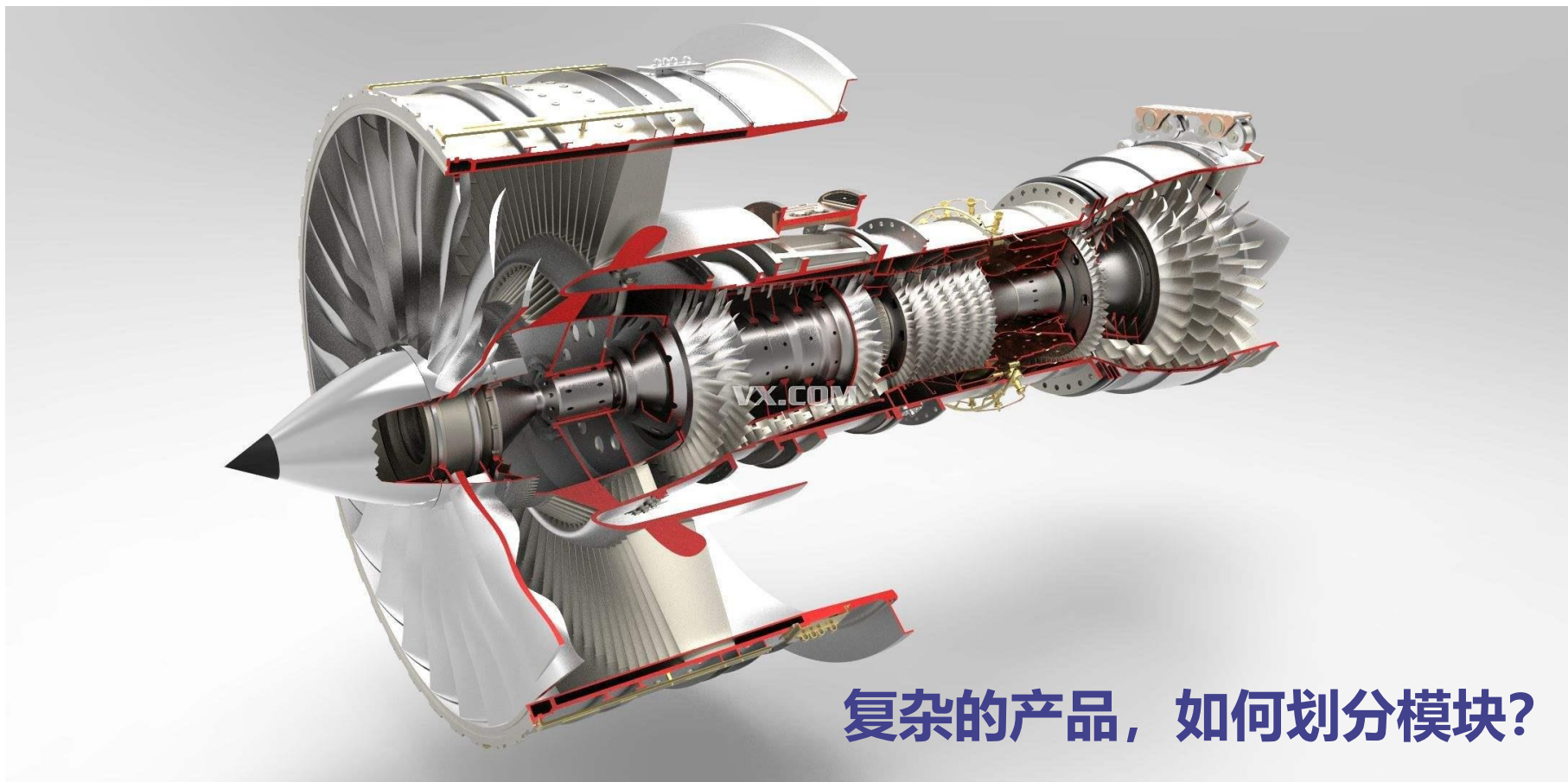
The diagram illustrates the modular programming approach. A central box contains the `main()` function code. To its right, four separate boxes represent individual functions: `input()`, `Process_1()`, `Process_n()`, and `output()`. Arrows indicate the flow of control: a black arrow points from `input();` to the `input()` box; red arrows point from `process_1();` and `process_n();` to their respective boxes; and a black arrow points from `output();` to the `output()` box. This visualizes how a large monolithic function is decomposed into smaller, reusable modules.

## 5.5 模块化编程思想

# 正确理解模块化



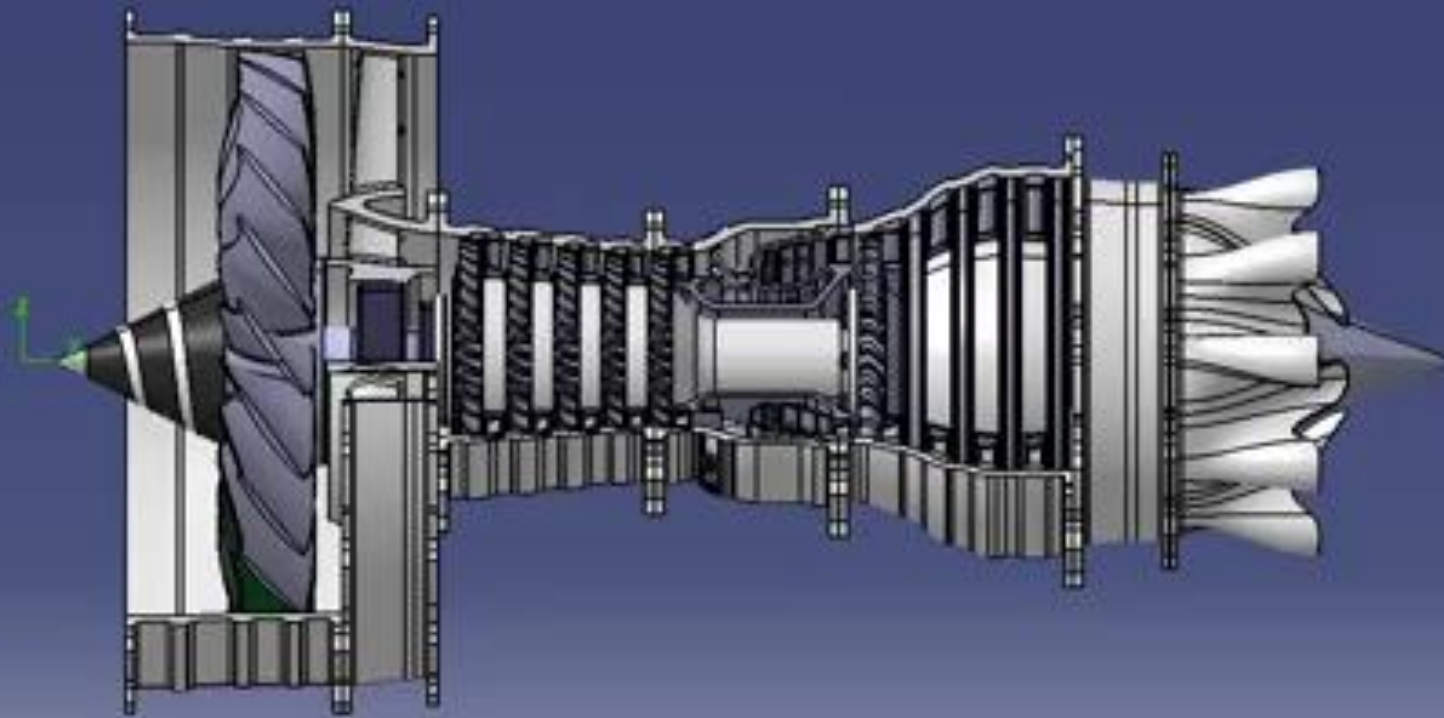
# 什么是模块化



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，能让发动机能运转，飞机能飞行！

# 什么是模块化

- 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



# 模块化编程的意义

真正的飞机是这样造出来的！



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!



# 为什么使用函数？

## 【例 a4-6】

### 查询某天是星期几



#### 存在问题：

- 层次不清晰、代码不易读、可维护性差。
- 代码无法重复使用、编程效率低。

一个并不算长的程序，读起来比较麻烦？

一个程序片段（函数）多长比较好？通常20多行，正常阅读字号和间距时，代码长度占满一屏就比较好。50多行长的函数“不太好”！

```
1 #include <stdio.h>
2 int main()
3 {
4     int c, y, m, w, d, longday = 1;
5     printf("Query what day a certain date is\n");
6     printf("Note: the format of the day is like 20120101\n");
7     printf("The input is between 101 and 99991231\n");
8     while(1) {
9         printf("\nInput date (or -1 to quit): ");
10        scanf("%d", &longday);
11        if(longday == -1)
12            break;
13        if(!(longday >= 101 && longday <= 99991231)) {
14            printf("Wrong input format, try again!\n");
15            continue;
16        }
17        y = longday/10000;
18        m = (longday%10000)/100;
19        d = longday%100;
20        if(m<3) {
21            y = y-1;
22            m = m+12;
23        }
24        c = y/100;
25        y = y%100;
26        // Zeller formula
27        w = (y + y/4 + c/4 - 2*c + (26*(m+1))/10 + d - 1)%7;
28        if(w<0)
29            w+=7;
30        printf("The day is: ");
31        switch(w) {
32            case 0:
33                printf("Sun\n");
34                break;
35            case 1:
36                printf("Mon\n");
37                break;
38            case 2:
39                printf("Tue\n");
40                break;
41            case 3:
42                printf("Wed\n");
43                break;
44            case 4:
45                printf("Thu\n");
46                break;
47            case 5:
48                printf("Fri\n");
49                break;
50            case 6:
51                printf("Sat\n");
52                break;
53        }
54        return 0;
55    }
```

# 为什么使用函数?

【例 a4-6】查询某天 (如20191107) 是星期几 (用函数实现)

```
1 #include <stdio.h>
2 void printWeek(int);
3 int getWeek(int);
4 int main()
5 {
6     // c: century-1, y: year, m:month, w:week, d:day
7     int longday = 1, w;
8     printf("Query what day a certain date is\n");
9     printf("Note: the format of the day is like 20120101\n");
10    printf("The input is between 101 and 99991231\n\n");
11    while(1)
12    {
13        printf("\nInput date (or -1 to quit): ");
14        scanf("%d", &longday);
15        if(longday == -1)
16            break;
17        if(!(longday >= 101 && longday <= 99991231))
18        {
19            printf("Wrong input format, try again!\n");
20            continue;
21        }
22        w = getWeek(longday);
23        printWeek(w);
24    }
25    return 0;
26 }
```

```
28 int getWeek(int day)
29 {
30     int c, y, m, d, w;
31     y = day/10000;
32     m = (day%10000)/100;
33     d = day%100;
34     if(m<3)
35     {
36         y = y-1;
37         m = m+12;
38     }
39     c = y/100;
40     y = y%100;
41     // Zeller formula
42     w = (y + y/4 + c/4 - 2*c + (26*(m+1))/10 + d - 1)%7;
43     if (w<0)
44         w+=7;
45     return w;
46 }
```

```
48 void printWeek(int w)
49 {
50     switch(w)
51     {
52     case 0:
53         printf("Sun\n");
54         break;
55     case 1:
56         printf("Mon\n");
57         break;
58     case 2:
59         printf("Tue\n");
60         break;
61     case 3:
62         printf("Wed\n");
63         break;
64     case 4:
65         printf("Thu\n");
66         break;
67     case 5:
68         printf("Fri\n");
69         break;
70     case 6:
71         printf("Sat\n");
72         break;
73     }
74 }
```

代码复用  
层次清晰

# 为什么使用函数?

```
28 int getWeek(int day)
29 {
30     int c, y, m, d, w;
31     y = day/10000;
32     m = (day%10000)/100;
33     d = day%100;
34     if(m<3)
35     {
36         y = y-1;
37         m = m+12;
38     }
39     c = y/100;
40     y = y%100;
41     // Zeller formula
42     w = (y + y/4 + c/4 - 2*c + (26*(m+1))/10 + d - 1)%7;
43     if (w<0)
44         w+=7;
45     return w;
46 }
```

一个函数，代码量少，效率高

结构清晰or代码量少?  
鱼与熊掌可否兼得?

等价



```
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;
}
```

多个函数，调用开销大；  
但结构清晰。

事实上，本分解并不优美！甚至  
非常糟糕！如果getWeek一定要  
再分，有更好的分解办法。

```
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;
}
```

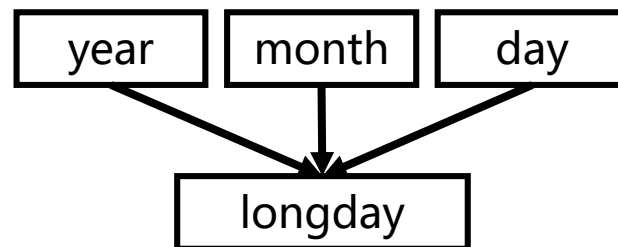
# 函数重在接口

【例 a4-6】查询某天（如20191107）是星期几（用函数实现）

```
1 #include <stdio.h>
2 void printWeek(int);
3 int getWeek(int);
4 int main()
5 {
6     // c: century-1, y: year, m:month, w:week, d:day
7     int longday = 1, w;
8     printf("Query what day a certain date is\n");
9     printf("Note: the format of the day is like 20120101\n");
10    printf("The input is between 101 and 99991231\n\n");
11    while(1)
12    {
13        printf("\nInput date (or -1 to quit): ");
14        scanf("%d", &longday);
15        if(longday == -1)
16            break;
17        if(!(longday >= 101 && longday <= 99991231))
18        {
19            printf("Wrong input format, try again!\n");
20            continue;
21        }
22        w = getWeek(longday);
23        printWeek(w);
24    }
25    return 0;
26 }
```

```
28 int getWeek(int day)
29 {
30     int c, y, m, d, w;
31     y = day/10000;
32     m = (day%10000)/100;
33     d = day%100;
34     if(m<3)
35     {
36         y = y-1;
37         m = m+12;
38     }
39     c = y/100;
40     y = y%100;
41     // Zeller formula
42     w = (y + y/4 + c/4 - 2*c + (26*(m+1))/10 + d - 1)%7;
43     if (w<0)
44         w+=7;
45     return w;
46 }
```

```
48 void printWeek(int w)
49 {
50     switch(w)
51     {
52     case 0:
53         printf("Sun\n");
54         break;
55     case 1:
56         printf("Mon\n");
57         break;
58     case 2:
59         printf("Tue\n");
60         break;
61     case 3:
62         printf("Wed\n");
63         break;
64     case 4:
65         printf("Thu\n");
66         break;
67     case 5:
68         printf("Fri\n");
69         break;
70     case 6:
71         printf("Sat\n");
72         break;
73     }
74 }
```



代码复用  
层次清晰

# 函数重在接口（庖丁解牛——函数接口设计很重要！）

```
1 #include <stdio.h>
2 void printWeek(int);
3 int getWeek(int);
4 int main()
5 {
6     // c: century-1, y: year, m: month, w: week, d: day
7     int longday = 1, w;
8     printf("Query what day a certain date is\n");
9     printf("Note: the format of the day is like 20120101\n");
10    printf("The input is between 101 and 99991231\n\n");
11    while(1)
12    {
13        printf("\nInput date (or -1 to quit): ");
14        scanf("%d", &longday);
15        if(longday == -1)
16            break;
17        if(!(longday >= 101 && longday <= 99991231))
18        {
19            printf("Wrong input format, try again!\n");
20            continue;
21        }
22        w = getWeek(longday);
23        printWeek(w);
24    }
25    return 0;
26 }
```

任尔东西南北风，  
我自岿然不动！

`int getWeek(int day);`

VS

✗

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

✓

哪个接口更友好？  
接口是给调用者设计的！  
效率怎么办？宏函数：

`int getYear(int longday);  
int getMonth(int longday);  
int getDay(int longday);`

`#define YEAR(ld) ld / 10000  
#define MONTH(ld) (ld % 10000) / 100  
#define DAY(ld) ld % 100`

`w = getWeekDay(YEAR(longday), MONTH(longday),  
DAY(longday));`

友好与效率可以兼得！



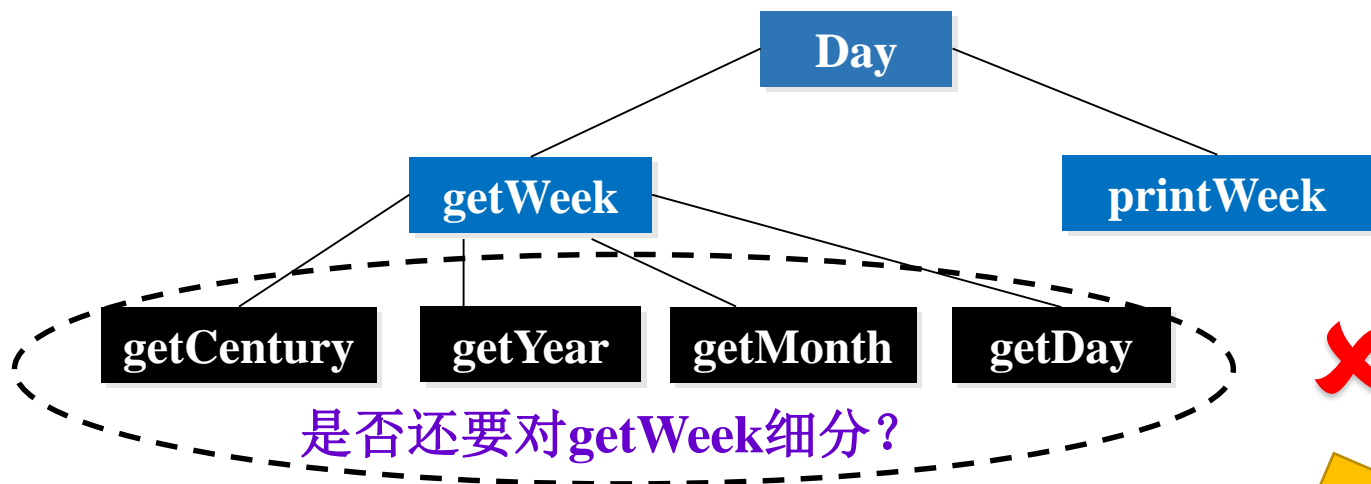
# 函数重在接口

```
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;
}
```

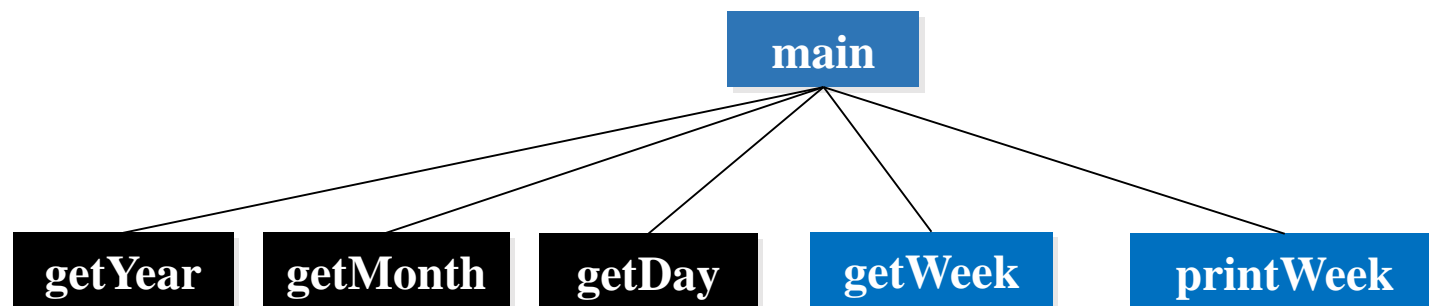
接口简洁: Make interface easy to use correctly, hard to use incorrectly!

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

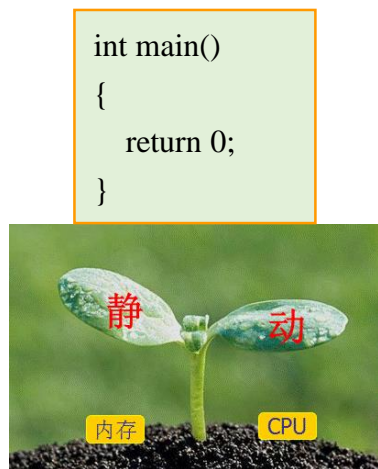


自顶向下，是问题层级的向下！  
不是函数调用的向下！



详见：C11 问题的求解方法

# 函数的意义



程序萌芽



程序长成

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



# 打印万年历

```
#include<stdio.h>
int main()
{
    int year, month, day, weekday, monthDays, leap, i, n;
    printf("\n请输入某年年份: ");
    scanf("%d",&year);
    n = year - 1900;//求元旦是星期几
    n = n+(n-1)/4+1;
    n = n%7;
    weekday = n;
    leap = 0;
    if((year%4==0 && year%100!=0)||year%400==0)//判断是否是闰年
        leap = 1;
    printf("\n\n\n          %d年\n",year); //打印年份
    for(month=1; month<=12; month=month+1) //打印12个月的月历
    {
        printf("\n%d月份\n",month);
        printf("-----\n");
        printf("星期日 星期一 星期二 星期三 星期四 星期五 星期六\n");
        printf("-----\n");
        for(i=0; i<weekday; i=i+1)//找当月1日的打印位置
            printf(" ");
        if(month==4 || month== 6 || month==9 || month==11)
            monthDays = 30;
        else if(month==2)
        {
            if(leap == 1)
                monthDays = 29;
            else
                monthDays = 28;
        }
        else
            monthDays = 31;
        for(day=1; day<=monthDays; day=day+1)//打印当月日期
        {
            printf("  %2d  ",day);
            weekday = weekday+1;
            if(weekday==7) //打满一星期应换行
            {
                weekday = 0;
                printf("\n");
            }
        }
        printf("\n");//打完一月应换行
    }
    return 0;
}
```

非函数实现



命令提示符

请输入某年年份: 2020

2020年

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	

2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

3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				

4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		

5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

# 万年历--函数实现

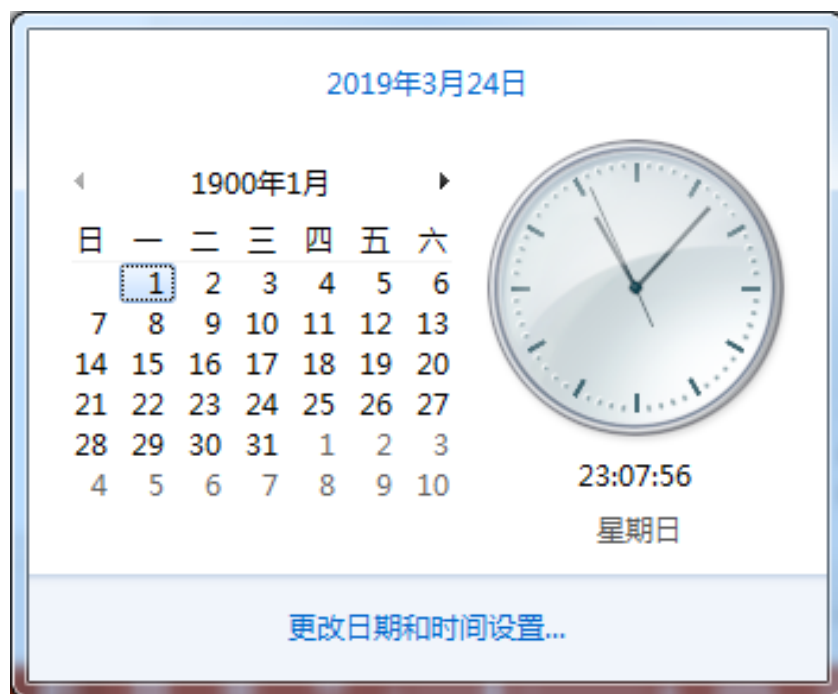
## 思考题

【例】万年历（函数实现）

解题思路：

(1) 先确定元旦是星期几；

(2) 逐月打印一年的所有天。



不难，但是不能错！  
调好了就别动了——专心做一件事！  
算法的封装是关键！

已知：

- ✓ 1900年1月1日是星期一；
- ✓  $365 \% 7 = 1$ ，平年多一天；
- ✓  $366 \% 7 = 2$ ，闰年多两天；

假设 $y$ 年的元旦为星期 $n$ ；

$$n = \left\{ (y - 1900) + \left\lfloor \frac{y - 1900 - 1}{4} \right\rfloor + 1 \right\} \% 7$$



平闰年多的  
基础天数



闰年多的额  
外天数



基数为  
星期一

# 更好的解决方案：代码复用

```
int getWeekDay(int year, int month, int day)  ← 函数重在接口!  
{  
    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;  
}
```

求元旦是星期几: `weekday = getWeekDay(year, 1, 1);`

# 万年历--函数实现

## • 思考题

### 【例】万年历（函数实现）

#### 解题思路：

(1) 先确定元旦是星期几

(2) 逐月打印一年的所有天

命令提示符

请输入某年年份: 2020

2020年

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	

2月份

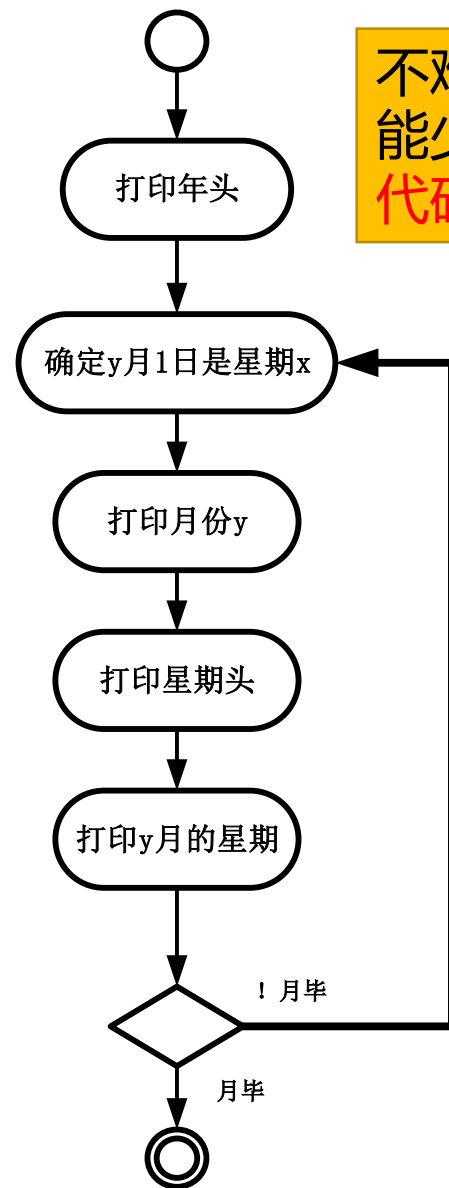
星期日	星期一	星期二	星期三	星期四	星期五	星期六
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

3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				

4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



不难，但是很繁琐！  
能少写点就太好了！  
代码的复用是关键！

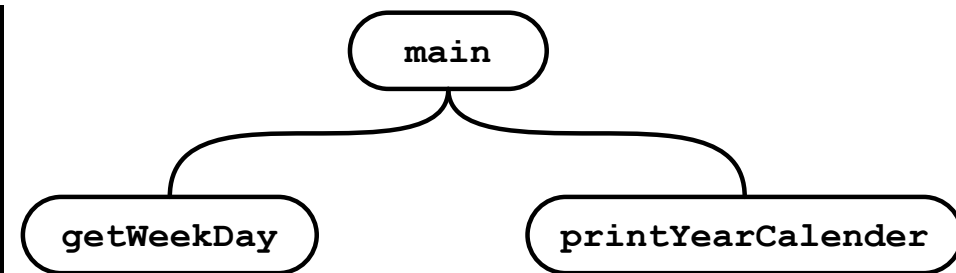
# 万年历--函数实现

```
#include<stdio.h>

// 求元旦是星期几，函数实现算法封装
int getWeekDay(int year, int month, int day); //求某年某月某日是星期几

// 打印日历，函数实现代码复用
int isLeap(int year); //判断是否是闰年
int getDaysOfMonth(int year, int month); // 获得月天数
int printYearCalender(int year, int weekday); // 打印年历
int printMonthCalendar(int month, int days, int weekday); // 打印月历
int printFirstWeekCalendar(int weekday); // 打印头周历
int printMiddleWeekCalendar(int monthday); // 打印中间周历
int printLastWeekCalendar(int monthday, int days); // 打印尾周历

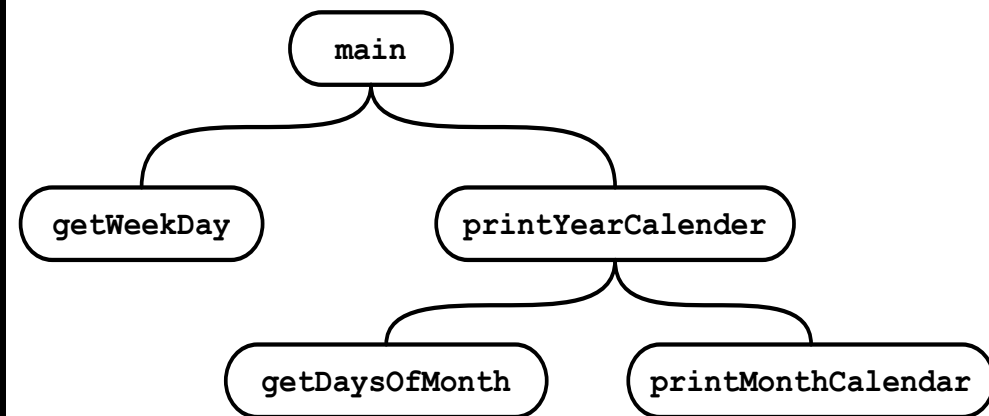
int main()
{
    int year, weekday;
    printf("\n请输入某年年份: ");
    scanf("%d", &year);
    weekday = getWeekDay(year, 1, 1); //求元旦是星期几
    printYearCalender(year, weekday); //打印年历, main函数清清爽爽
    return 0;
}
```



# 万年历--函数实现

```
int printYearCalender(int year, int weekday)
{
    int days, month;
    printf("\t\t\t%d年\n", year);
    for(month = 1; month <= 12; month++)
    {
        days = getDaysOfMonth(year, month);
        weekday = printMonthCalendar(month, days, weekday);
    }
    return weekday;
}

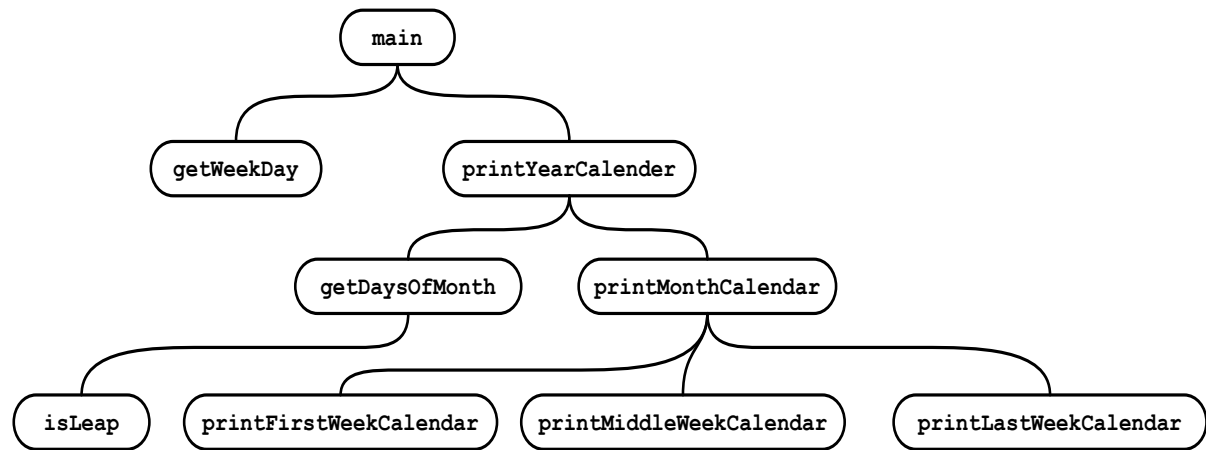
int printMonthCalendar(int month, int days, int weekday)
{
    int monthday;
    printf("\n%d月份\n", month);
    printf("-----\n");
    printf("星期日\t星期一\t星期二\t星期三\t星期四\t星期五\t星期六\n");
    printf("-----\n");
    monthday = printFirstWeekCalendar(weekday);
    while (days - monthday >= 7)
    {
        monthday = printMiddleWeekCalendar(monthday);
    }
    weekday = printLastWeekCalendar(monthday, days);
    return weekday;
}
```



# 万年历--函数实现

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

int getDaysOfMonth(int year, int month)
{
    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;
}
```



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

# 万年历--函数实现

```
int printFirstWeekCalendar(int weekday)
{
    int i, monthday = 1;
    if(weekday == 0) return 1;
    for(i = 1; i < weekday; i++)
        putchar('\t');

    for(i = weekday; i <= 6; i++)
        printf("\t%d", monthday++);

    printf("\n");
    return monthday;
}

int printMiddleWeekCalendar(int monthday)
{
    int i;
    printf("%d", monthday++);
    for(i = 1; i <= 6; i++)
        printf("\t%d", monthday++);

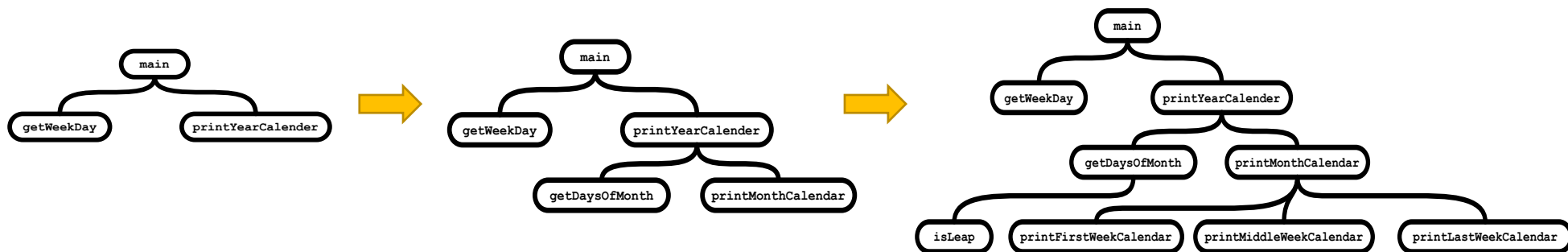
    printf("\n");
    return monthday;
}
```

```
int printLastWeekCalendar(int monthday, int days)
{
    int i, weekday = 0;
    if(monthday > days) return 0;
    printf("%d", monthday++);
    weekday++;
    while(monthday <= days){
        printf("\t%d", monthday++);
        weekday++;
    }
    printf("\n");
    return weekday % 7;
}
```

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。

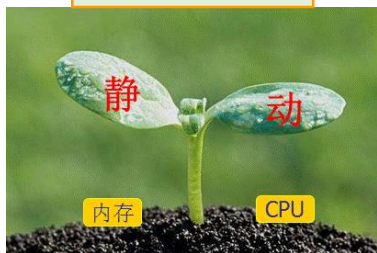


# 函数的意义



多像一棵生长的大树（反过来看）

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



程序萌芽

解耦与封装



程序长成

## 5.6 递归函数

---

**一看就会，一写就废**

# 递归函数简介

## Fibonacci数列（斐波纳契数列）

小明喜欢养鸽子，但高中时学业太重，没有时间养。拿到\*\*大学的录取通知书后，有时间了，养了一对鸽子。四年（大学毕业）时，家里的鸽子有多少对？

（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。新幼鸽也按此规律繁殖。这里假设鸽子寿命很长.....）



# 递归函数简介

- Fibonacci数列（斐波纳契数列）

小明年初买回一对幼鸽，半年后幼鸽成熟，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。新幼鸽半年后也成熟并开始繁殖，且每季度繁殖一对幼鸽。……。问，2年后小明家共有多少对鸽子？5年后呢（设鸽子寿命为10年？）？

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

$f(1) = 1$

$f(2) = 1$

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

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

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

...

$f(n) = ? \qquad f(n) = f(n - 1) + f(n - 2)$

鸽子家园

日期	对
1.1	a
4.1	a
7.1	a    a1
10.1	a    a2    a1
1.1	a    a3    a2    a1    a11
4.1	a    a4    a3    a2    a21    a1...
...	

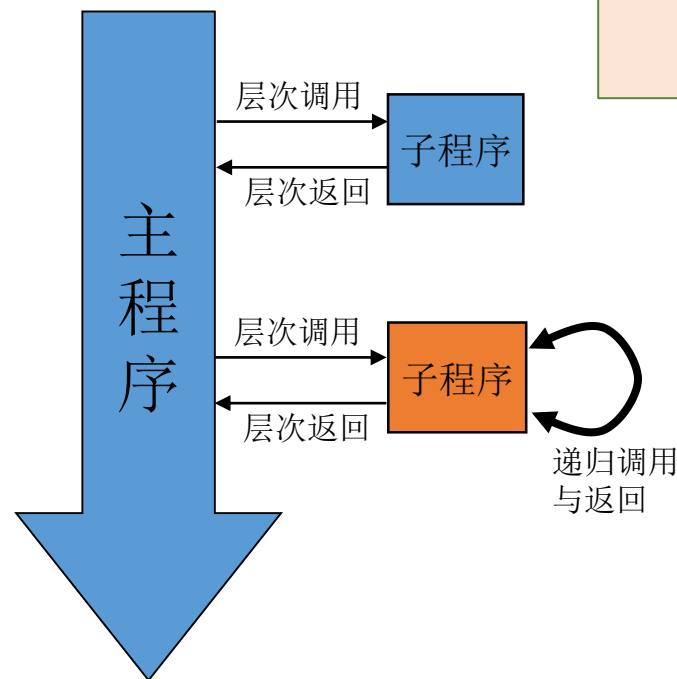
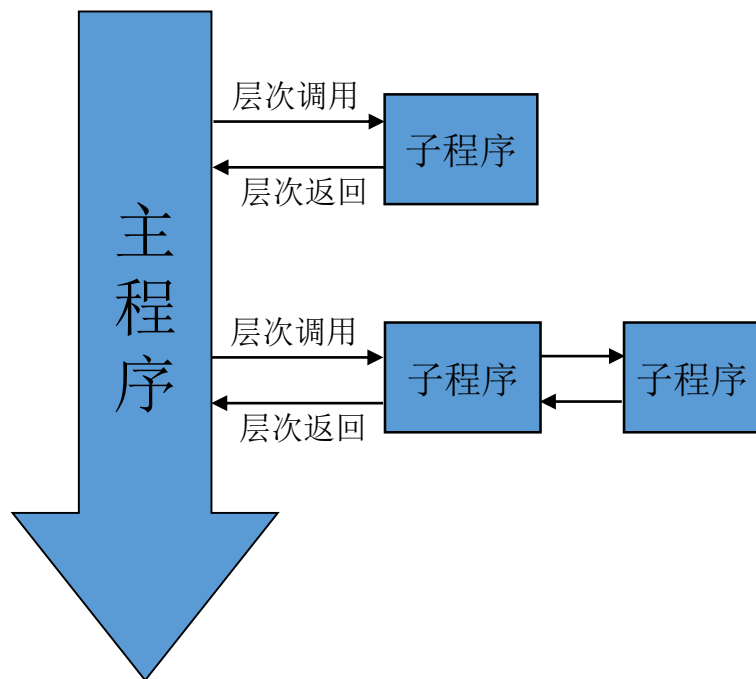
# 递归函数简介

- 一般的函数：按层次方式调用（有显式表达式），如： $s = f(n) = a \cdot n^2 + c$
- 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

$f(n) = n!$  // 显示表达式

$f(n) = n \cdot f(n-1)$  // 隐式表达式？

- 递归是计算机科学中的重要问题。



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

# 递归的概念与思想

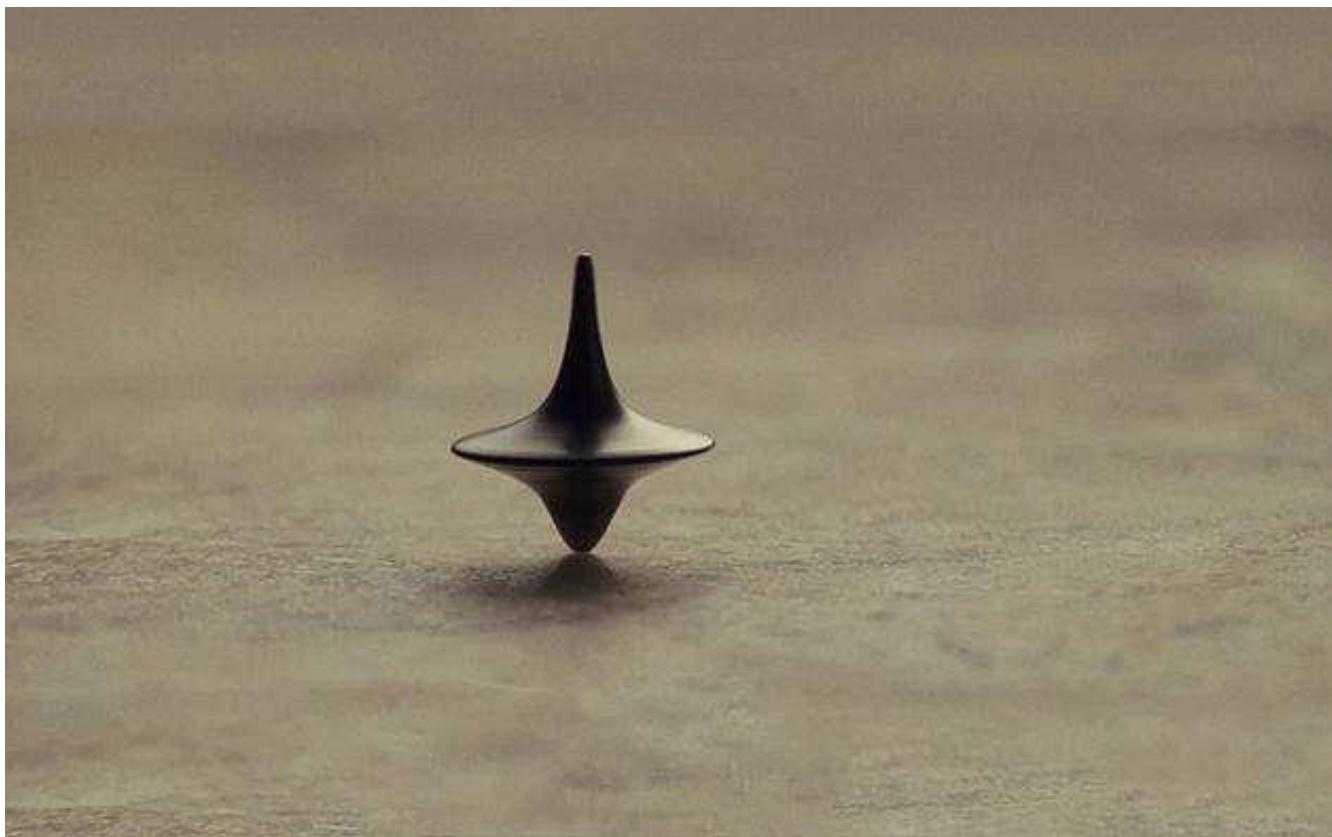
1. 调用递归函数解决问题时，函数只能解决最简单的情况（称为基本情况）。
2. **基本情况**的函数调用只是简单地返回一个结果。
3. 对复杂问题调用函数时，函数将问题分成两个概念性部分：函数中能够处理的部分和函数中不能处理的新问题。
4. **不能处理的新问题部分模拟原复杂问题，但复杂度减小（问题简化或缩小）**。
5. 新问题与原问题相似，函数启动（调用）自己的最新副本来处理此新问题，称为递归调用(recursive call)或递归步骤(recursion step)。
6. 递归步骤可能包括关键字return，其结果与函数中需要处理的部分组合，形成的结果返回原调用者（回溯）。

例：求非负长整数的阶乘  $n!$   
( $n \geq 0$ )

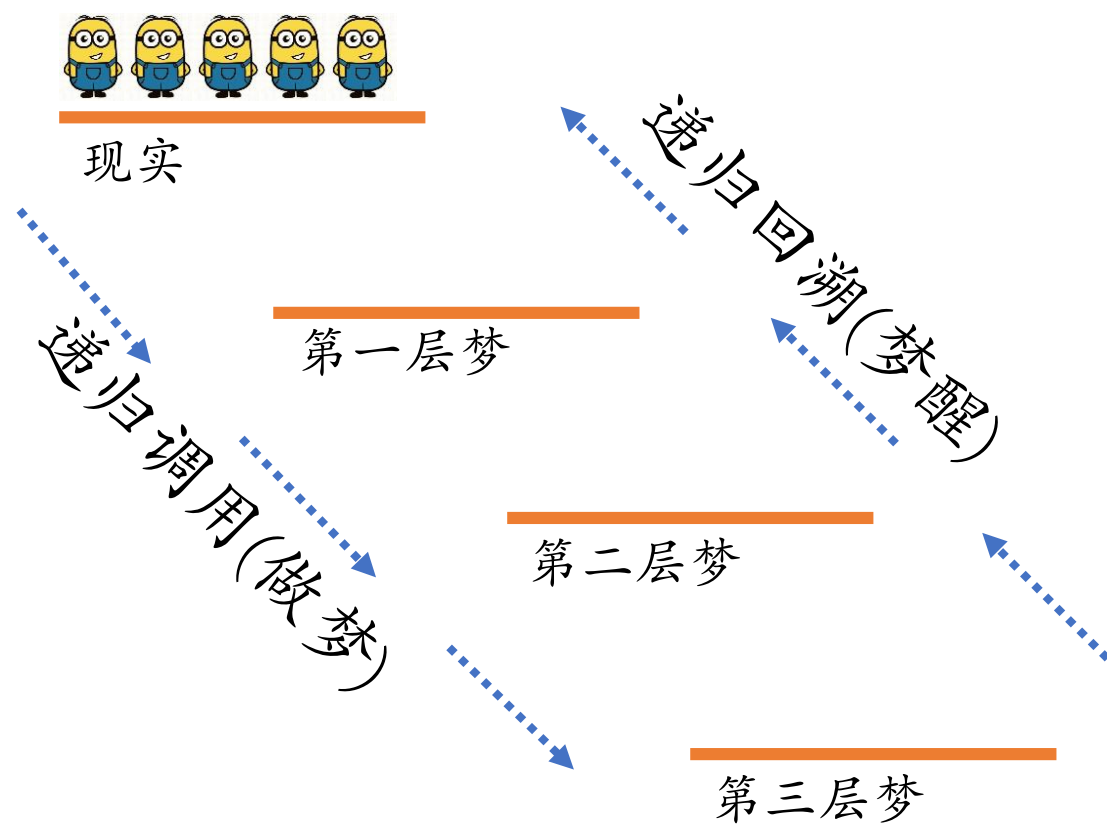
$f(n) = n!$   
 $0! = 1, 1! = 1$   
 $f(n) = n! = n * (n-1)! = n * f(n-1)$   
 $f(n-1) = (n-1) * f(n-2)$   
...  
 $f(2) = 2 * f(1)$   
 $f(1) = 1$   
 $f(0) = 1$

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

# 递归原理：《盗梦空间》



盗梦空间：递归的梦



# 递归调用与返回实例

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

以  $n = 3$  为例进行程序剖析

$f(n) = n!$

$f(3) = ?$

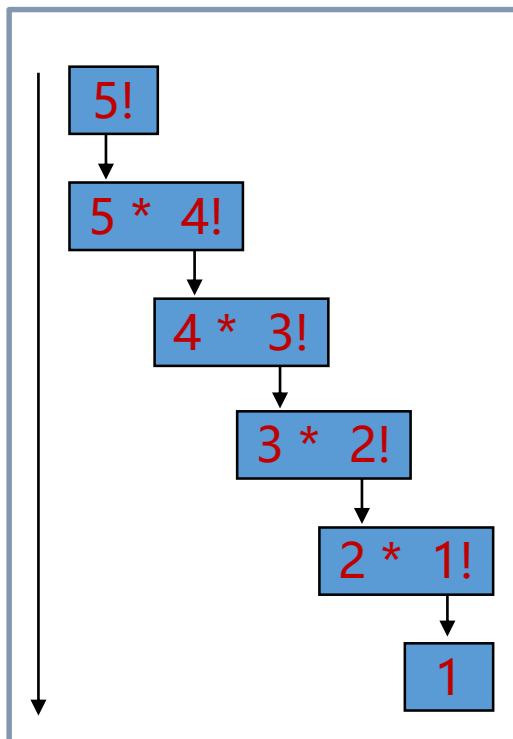
```
f(n) // n=3
{
    if (n <= 1) // base case
        return 1;
    return n * f(n-1);
}
```



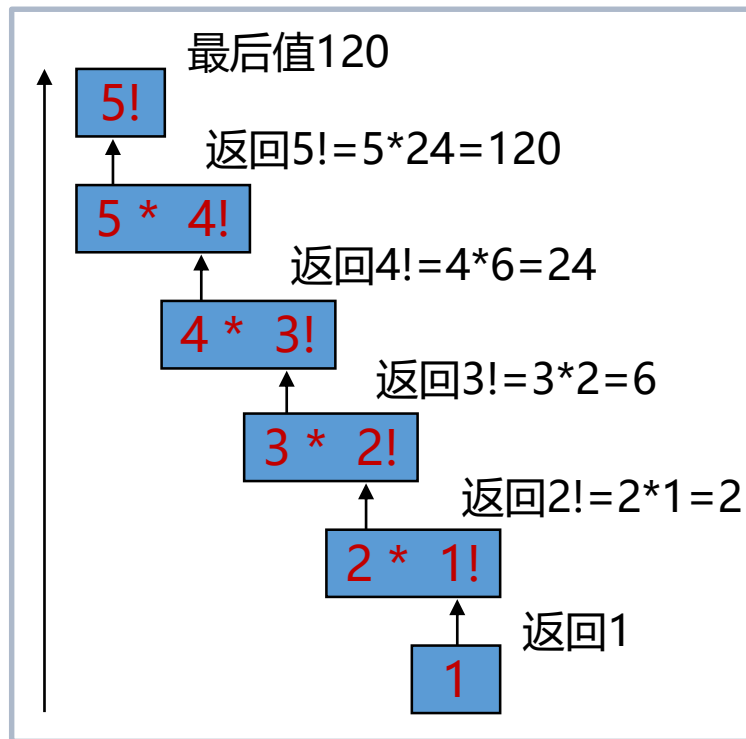
```
f(n) // n=3
{
    if (n <= 1) // base case
        return 1;
    return
        n*f(n-1); // m=n-1=2
    {
        if (m <= 1) // base case
            return 1;
        return
            m * f(m-1); // k = m-1 = 1
            {
                if (k <= 1) // base case
                    return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



# 递归调用与返回实例 【a5-8】



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 f( unsigned long );

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

    return 0;
}

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

# 递归调用与返回实例

## 神奇的 Fibonacci 数 (斐波纳契数列)

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

$$f(0) = 0$$

$$f(1) = 1$$

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

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

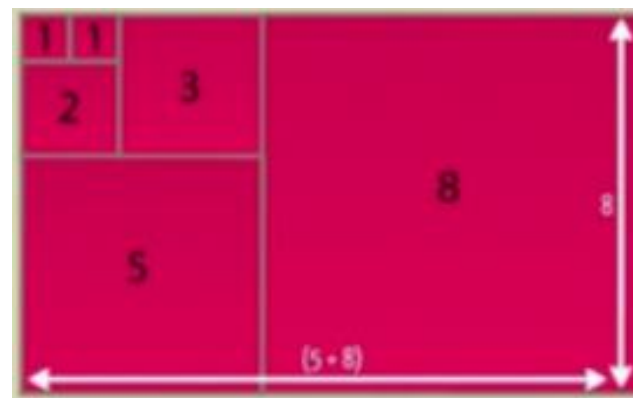
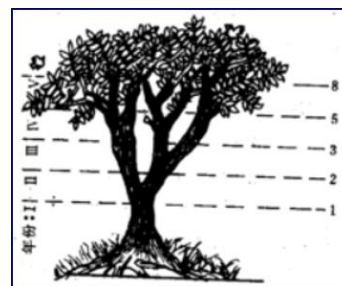
$$f(4) = f(3) + f(2) = 2 + 1 = 3$$

...

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

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

- 房子、门、窗、明信片、书、相片的长宽比
- 植物界的很多美丽形态
- 照相取景时人的位置比、色差比

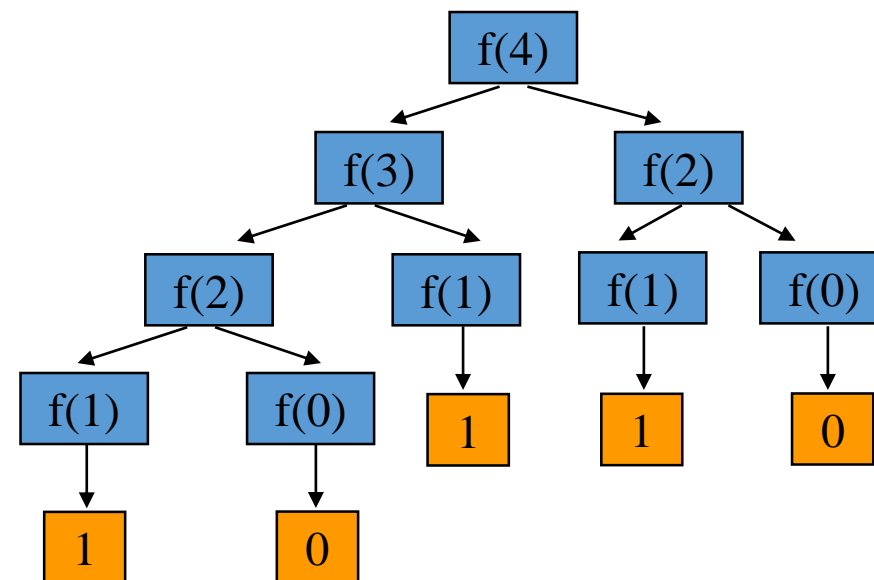


# 递归调用与返回实例【a5-9】

## 【a5-9】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。

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



# gcd的循环实现

**【例4-2-1】最大公约数** 输入两个整数a和b，用辗转相除法求它们的最大公约数。

**辗转相除算法gcd(a, b):**

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

**【gcd(a, b) = gcd(b, a%b)】**

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

    return 0;
}
```

# gcd的递归实现

## 【a5-10】求最大公约数

假设 $a > 0, b \geq 0$

辗转相除算法gcd(a, b):

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

【gcd(a, b) = gcd(b, a % b)】



1. if b is 0, gcd(a, b) is a, stop;  
else go to step 2;
2. let  $a \leftarrow b, b \leftarrow a \% b$ , go to step 1.

【gcd(a, b) = gcd(b, a % b)】



```
#include<stdio.h>

int gcd(int, int);

int main()
{
    int a, b;
    printf("Input two integers: ");
    scanf("%d%d", &a, &b);
    printf("%d\n", gcd(a, b));

    return 0;
}

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

# 递归调用与返回实例

## 【a5-11】求组合数

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

分析:

$C(m, n)$

非递归实现: 三次  
求阶乘, 循环

$$= \frac{m!}{n! (m-n)!} \quad \text{组合恒等式}$$

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

基本情况

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

```
#include<stdio.h>
int comb_num(int, int);
int main()
{
    int m, n;
    printf("Input two integers: ");
    scanf("%d%d", &m, &n);

    printf("%d\n", comb_num(m, n));
    return 0;
}

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

# 递归调用与返回实例

## 【a5-12】求阿克曼函数

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

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

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

```
#include<stdio.h>
int ack(int, int);
int main()
{
    int m, n, r;
    printf("Input two integers (>=0): ");
    scanf("%d%d", &m, &n);

    printf("ack num is: %d\n", ack(m, n));

    return 0;
}

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

# 递归调用与返回实例

## 【例】汉诺塔 Hanoi Tower

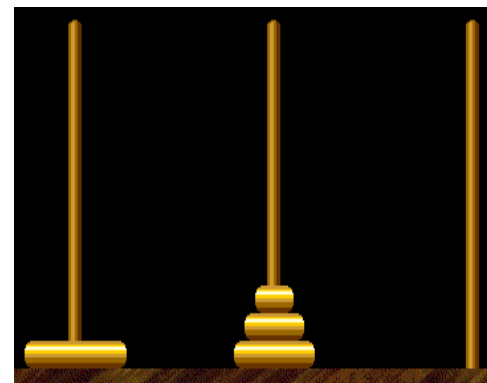
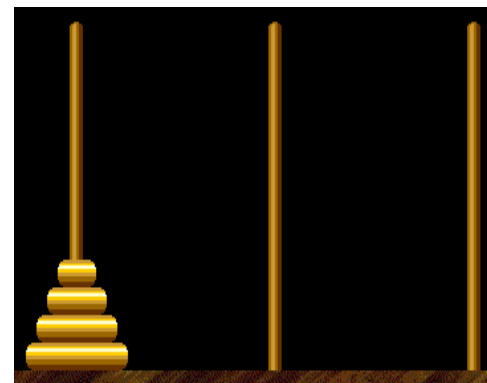
源自印度神话.....

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。

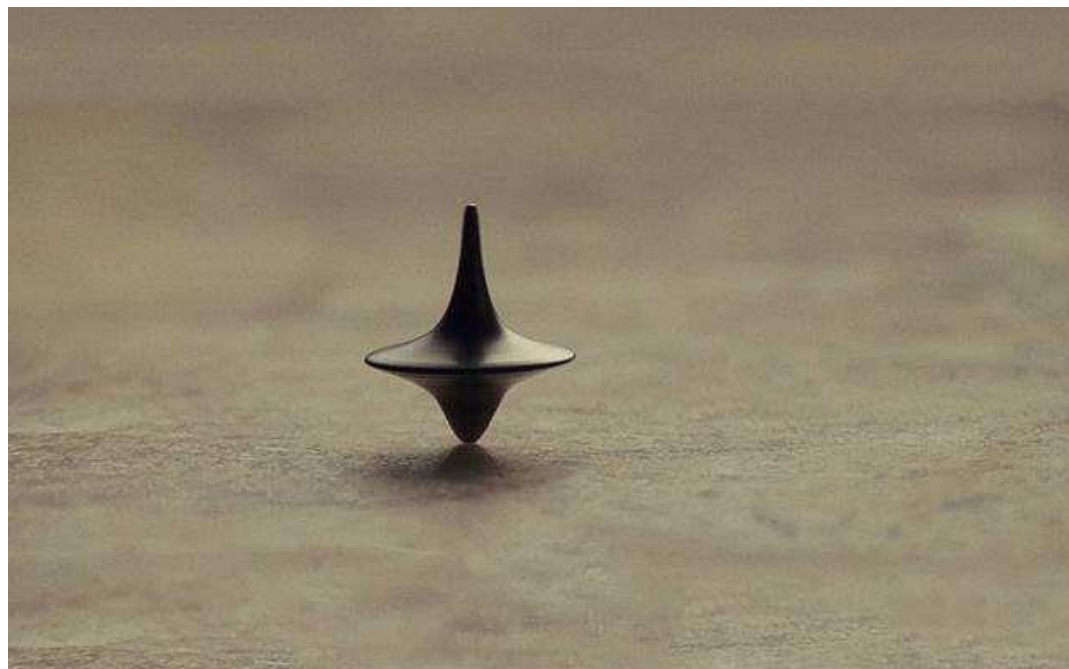
有预言说，这件事完成时宇宙会在一瞬间闪电式毁灭。也有人相信婆罗门至今还在一刻不停地搬动着圆盘。

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



盗梦空间：递归的梦

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

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” ？

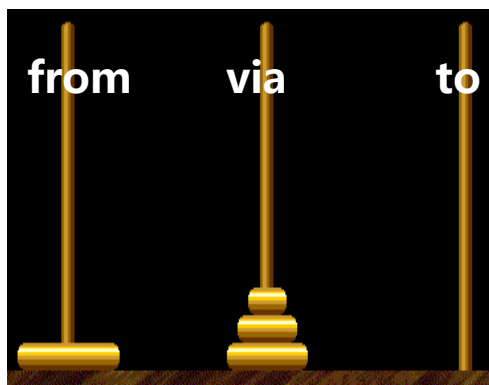
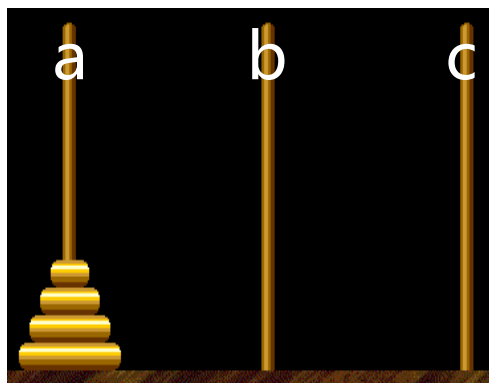
**程序员就是造梦师**

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！）

# 递归调用与返回实例

## 【a5-13】 汉诺塔 Hanoi Tower



函数重在接口，递归重在调用

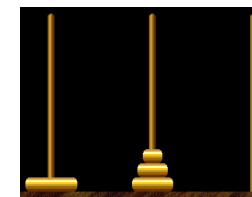
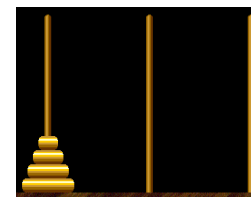
```
void hanoi(int n, char a, char b, char c);  
    // 把n个盘子从柱子 a 通过 b 挪到 c 上  
void move(int n, char a, char c); // 把第n号盘子从柱子a挪到c上
```

```
int main()  
{  
    ...  
    hanoi(num, a, b, c);  
}  
void move(int n, char from, char to)  // 这个n的意思是n号盘  
{  
    printf("move %d from %c to %c\n", n, from, to);  
}  
void hanoi(int n, char from, char via, char to)  // 这个n的意思是n层hanoi塔  
{  
    if(n == 1) {  
        move(n, from, to);  
        return;  
    }  
    hanoi(n-1, from, to, via); // 把n-1个盘子从from通过  
    move(n, from, to);  
    hanoi(n-1, via, from, to);  
}
```

# 递归调用与返回实例

## 婆罗门能否让宇宙毁灭? 【a5-13】

```
int main()
{
    ...
    hanoi(num, a, b, c);    ...
}
void move(int n, char from, char to)
{
    printf("move %d from %c to %c\n", n, from, to);
}
void hanoi(int n, char from, char via, char to)
{
    if(n == 1) {
        move(n, from, to);
        return;
    }
    hanoi(n-1, from, to, via);
    move(n, from, to);
    hanoi(n-1, via, 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

$$n = 64$$

$$2^n = 18446744073709551616$$

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

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

# 递归与迭代

例：分别用迭代和递归方法求阶乘n!

迭代

```
unsigned long factorial = 1;
int num, i;
scanf("%d", &num);
for(i = num; i >= 1; i--)
    factorial *= i;
```

递归

```
unsigned long factorial(unsigned long);
int main()
{ int num; unsigned long f;
  scanf("%d", &num);
  f = factorial(num);
  return 0;
}
unsigned long factorial(unsigned long n)
{ if (n <= 1) // base case
    return 1;
  return n * factorial(n-1);
}
```

迭代与递归方法的对比

	迭代	递归
依赖的控制结构	重复（循环）	选择（条件）
重复方式	显式使用重复结构	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。
- 能用递归解决的问题也能用迭代（非递归）方法解决。

# 本章小结

- 熟练使用常用的标准库函数
- 能够自定义函数
- 理解函数原型、函数定义的含义
- 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
- 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
- 知道常用的标准库函数
- 了解递归函数简介
- 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

# 课后作业

- 根据课件和教材内容复习
- 教材章节后相关习题
- 通过OJ平台做练习赛
- 预习第6章
- 上机实践题：
  - ◆ 把本课件和书上讲到的所有例程输入计算机，运行并观察、分析与体会输出结果。
  - ◆ 编程练习课后习题内容。

# 课堂作业

1、用递归函数实现求阶乘 $n!$ ，见右边的程序，问：递归函数  $f$  调用多少次？

```
// 求 n!  
unsigned long f(unsigned long n) {  
    if ( n <= 1 )  
        return 1;  
    return ( n * f(n-1) );  
}
```

2、用递归函数实现汉诺塔移动，见右边的程序，问：递归函数 `hanoi` 调用多少次？

```
// 汉诺塔移动  
void hanoi(int n, char x, char y, char z) {  
    if(n == 1) {  
        printf("%c --> %c\n", x, z);  
        return;  
    }  
    hanoi(n-1, x, z, y);  
    printf("%c --> %c\n", x, z);  
    hanoi(n-1, y, x, z);  
}
```