

# 第7章

## 函数

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

```
void two();  
void three();
```

```
int main()  
{  
    printf("I'm in main.\n");  
    two();  
  
    return 0;  
}
```

```
void two()  
{  
    printf("I'm in two.\n");  
    three();  
}
```

```
void three()  
{  
    printf("I'm in three.\n");  
}
```

# 关于函数

- 程序由多个函数组成
- 程序的执行总是从main函数开始
- 所有函数都是平行、独立的，一个函数不属于另一个

# 函数定义

void, int...

标识符

逗号分隔，可以为空

<返回值类型> <函数名>(<参数列表>)

{

<函数体>

}

可以为空

```
int min(int a, int b)
{
    return a < b ? a : b;
}
```

# 调用过程

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

被调函数

```
int min(int a, int b)
{
    return a < b ? a : b;
}
```

```
int main()
{
    int x, y, c;

    scanf("%d%d", &x, &y);
    c = min(x, y);

    printf("%d\n", c);

    return 0;
}
```

主调函数



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

```
int min(int a, int b)
{
    return a < b ? a : b;
}
```

```
int main()
{
    int x, y, c;

    scanf("%d%d", &x, &y);
    c = min(x, y);

    printf("%d\n", c);

    return 0;
}
```

值传递



求最大公约数

```
int gcd(int u, int v)
{
    int tmp;

    while (v != 0) {
        tmp = u % v;
        u = v;
        v = tmp;
    }

    return u;
}
```

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

```
int main()
```

```
{
```

```
    int x, y, z;
```

```
    x = 145;
```

```
    y = 25;
```

```
    z = gcd(x, y); /* 1 */
```

```
    printf("GCD(%d, %d) = %d\n", x, y, z);
```

```
    x = 16;
```

```
    y = 24;
```

```
    printf("GCD(%d, %d) = %d\n", x, y, gcd(x, y)); /* 2 */
```

```
    z = gcd(x, x + y); /* 3 */
```

```
    printf("GCD(%d, %d) = %d\n", x, x + y, z);
```

```
    return 0;
```

```
}
```

## 函数可以多次调用

# 函数返回值

# 返回值的形式

可以为空

return <表达式>;

# 返回值作用

- 从被调函数退出，返回到调用前位置
- 可以返回1个值（也可以不返回值）

# 多个return

- 执行到哪一个哪个起作用



# 返回值类型

- 由函数定义决定

```
int min(int x, int y);
```

```
float max(float x, float y);
```

```
double sin(double x);
```

# 函数的调用

# 函数调用形式

函数名(参数列表);

```
y = sin(x);
```

```
z = min(x, y);
```

```
n = printf("Hello, %d", x);
```

# 说明

- 函数调用语句也有“值”和“类型”
- 调用完成后，返回主调函数执行下一条语句

# 调用之前.....

- 函数必须存在
- 必须声明

返回值类型 函数名(参数列表类型);

声明在前， 定义在后

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

声明

```
int min(int a, int b);
```

```
int main()
```

```
{
```

```
    int x, y, c;
```

```
    scanf("%d%d", &x, &y);
```

```
    c = min(x, y);
```

```
    printf("%d\n", c);
```

```
    return 0;
```

```
}
```

定义

```
int min(int a, int b)
```

```
{
```

```
    return a < b ? a : b;
```

```
}
```

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

```
int min(int, int);
```

```
int main()
```

```
{
```

```
    int x, y, c;
```

```
    scanf("%d%d", &x, &y);
```

```
    c = min(x, y);
```

```
    printf("%d\n", c);
```

```
    return 0;
```

```
}
```

```
int min(int a, int b)
```

```
{
```

```
    return a < b ? a : b;
```

```
}
```

声明

只需类型即可

定义



声明定义在一起

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

```
int min(int a, int b)
{
    return a < b ? a : b;
}
```

声明+定义

```
int main()
{
    int x, y, c;

    scanf("%d%d", &x, &y);
    c = min(x, y);

    printf("%d\n", c);

    return 0;
}
```

# printf是如何声明的?

include 背后做了什么……

# 参数的值传递

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

```
void swap(int x, int y)
{
    int tmp;

    tmp = x;
    x = y;
    y = x;
}
```

能交换成功吗?

```
int main()
{
    int a = 10;
    int b = 20;

    printf("a = %d, b = %d\n", a, b);

    swap(a, b);

    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

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

```
void swap(int x, int y)
{
    int tmp;

    tmp = x;
    x = y;
    y = x;
}
```

能交换成功吗?

不能

```
int main()
{
    int a = 10;
    int b = 20;

    printf("a = %d, b = %d\n", a, b);

    swap(a, b);

    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

# 嵌套调用

main函数

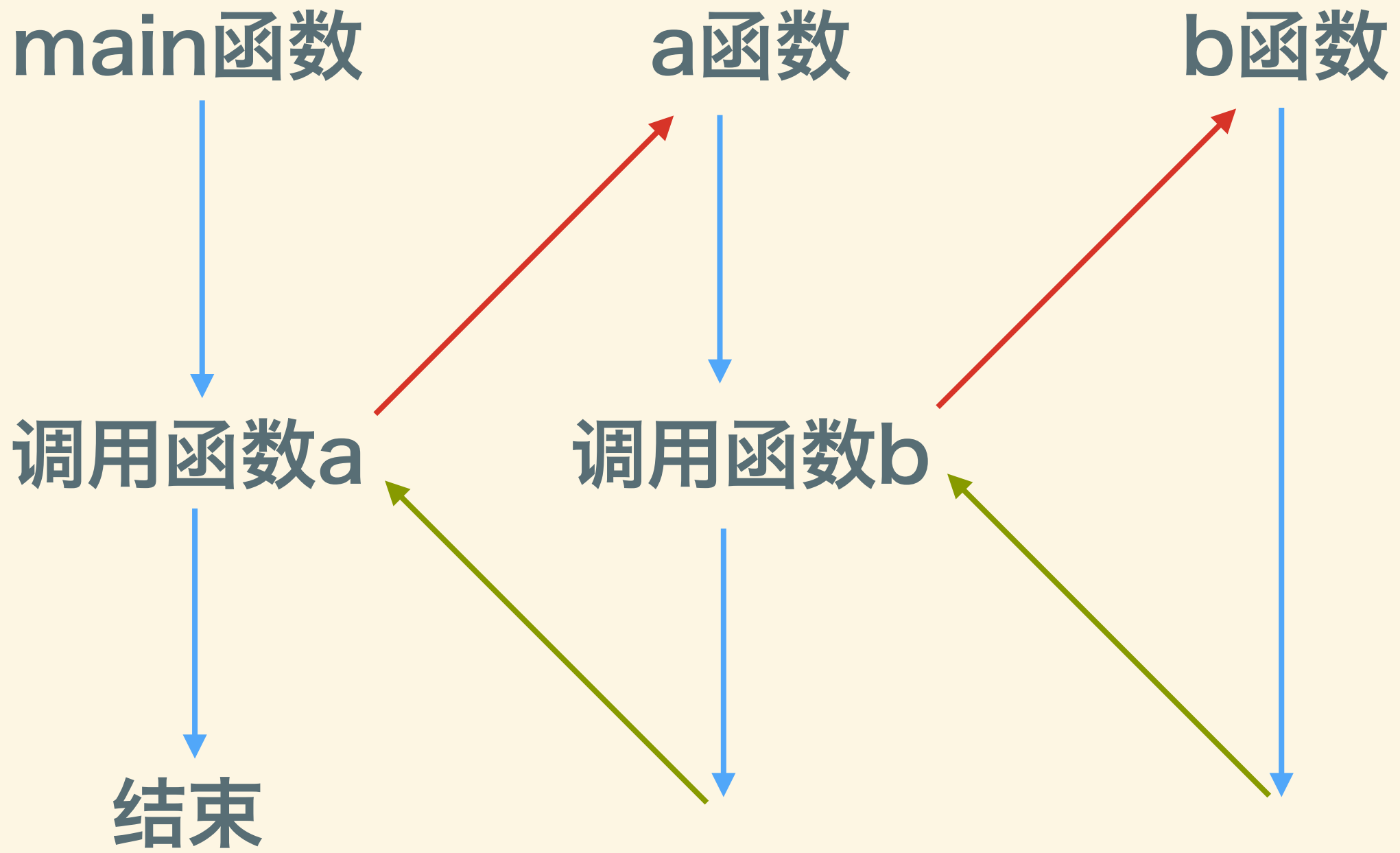
a函数

b函数

调用函数a

调用函数b

结束





# 递归调用

一个函数直接或间接地调用自身……

计算 $n!$

# 循环

```
#include <stdio.h>

int fact(int n)
{
    int i, result;

    result = 1;
    for (i = 1; i <= n; i++)
        result *= i;

    return result;
}

int main()
{
    int n;

    scanf("%d", &n);
    printf("%d\n", fact(n));

    return 0;
}
```

# 递归

```
#include <stdio.h>

int fact(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fact(n - 1);
}

int main()
{
    int n;

    scanf("%d", &n);
    printf("%d\n", fact(n));

    return 0;
}
```

求最大公约数

# 循环

```
int gcd(int u, int v)
{
    int tmp;

    while (v != 0) {
        tmp = u % v;
        u = v;
        v = tmp;
    }

    return u;
}
```

# 递归

```
int gcd(int u, int v)
{
    if (v == 0)
        return u;
    else
        return gcd(v, u % v);
}
```

# 如何写递归函数？



# 递归函数的特点

- 数学归纳法
- 将一个问题转换为规模更小的问题
- 必须有结束条件

# 汉诺塔





## Torres de Hanoi

El juego de las Torres de Hanoi es un juego matemático muy antiguo.

Consiste en mover tres discos de diferentes tamaños de una torre a otra, siguiendo ciertas reglas.

El juego consiste en mover todos los discos a una de las torres, siguiendo ciertas reglas.

Las reglas del juego son las siguientes:

1. Sólo se puede mover un disco cada vez.

2. No se puede poner un disco mayor encima de uno menor.

3. Nunca debes quedar un disco grande vacío en una torre.

El objetivo del juego es mover todos los discos a la torre de destino, siguiendo las reglas.

El juego de las Torres de Hanoi es un juego matemático muy antiguo.

Consiste en mover tres discos de diferentes tamaños de una torre a otra, siguiendo ciertas reglas.

El juego consiste en mover todos los discos a una de las torres, siguiendo ciertas reglas.

Las reglas del juego son las siguientes:

1. Sólo se puede mover un disco cada vez.

2. No se puede poner un disco mayor encima de uno menor.

3. Nunca debes quedar un disco grande vacío en una torre.

El objetivo del juego es mover todos los discos a la torre de destino, siguiendo las reglas.

El juego de las Torres de Hanoi es un juego matemático muy antiguo.

Consiste en mover tres discos de diferentes tamaños de una torre a otra, siguiendo ciertas reglas.

El juego consiste en mover todos los discos a una de las torres, siguiendo ciertas reglas.

Las reglas del juego son las siguientes:

1. Sólo se puede mover un disco cada vez.

2. No se puede poner un disco mayor encima de uno menor.

3. Nunca debes quedar un disco grande vacío en una torre.

El objetivo del juego es mover todos los discos a la torre de destino, siguiendo las reglas.

El juego de las Torres de Hanoi es un juego matemático muy antiguo.

Consiste en mover tres discos de diferentes tamaños de una torre a otra, siguiendo ciertas reglas.

El juego consiste en mover todos los discos a una de las torres, siguiendo ciertas reglas.

Las reglas del juego son las siguientes:

1. Sólo se puede mover un disco cada vez.

2. No se puede poner un disco mayor encima de uno menor.

3. Nunca debes quedar un disco grande vacío en una torre.

El objetivo del juego es mover todos los discos a la torre de destino, siguiendo las reglas.

El juego de las Torres de Hanoi es un juego matemático muy antiguo.

Consiste en mover tres discos de diferentes tamaños de una torre a otra, siguiendo ciertas reglas.

El juego consiste en mover todos los discos a una de las torres, siguiendo ciertas reglas.

Las reglas del juego son las siguientes:

1. Sólo se puede mover un disco cada vez.

2. No se puede poner un disco mayor encima de uno menor.

3. Nunca debes quedar un disco grande vacío en una torre.

El objetivo del juego es mover todos los discos a la torre de destino, siguiendo las reglas.

El juego de las Torres de Hanoi es un juego matemático muy antiguo.

Consiste en mover tres discos de diferentes tamaños de una torre a otra, siguiendo ciertas reglas.

El juego consiste en mover todos los discos a una de las torres, siguiendo ciertas reglas.

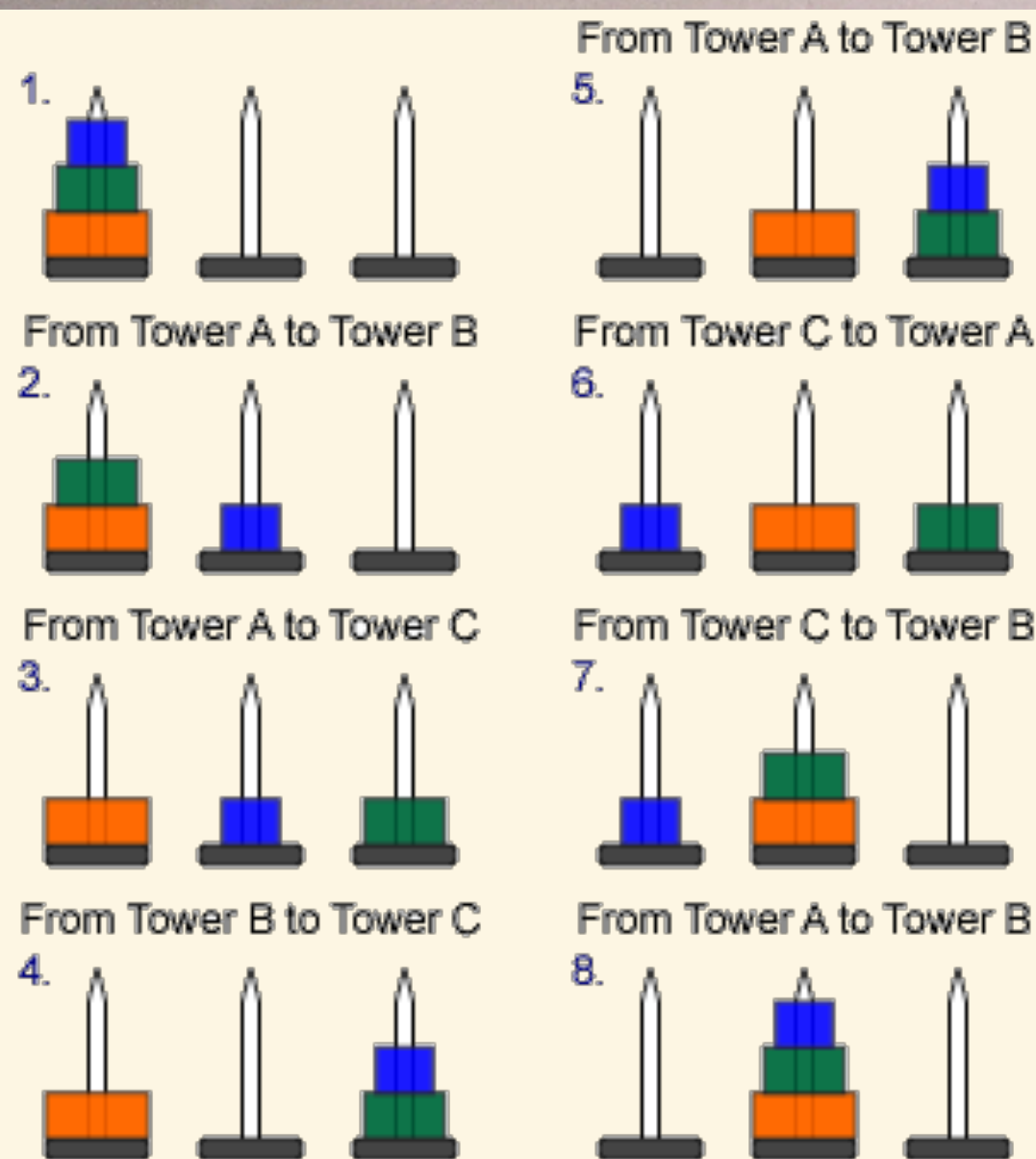
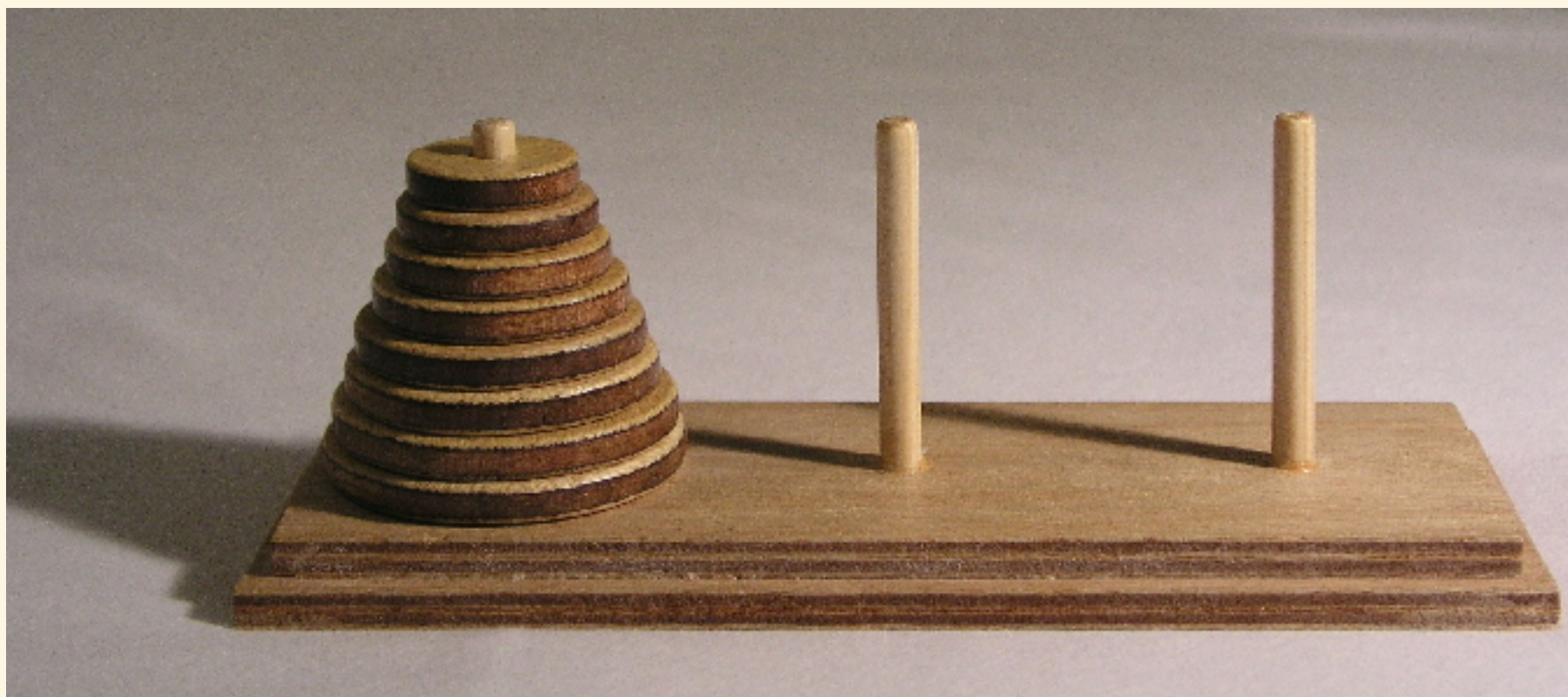
Las reglas del juego son las siguientes:

1. Sólo se puede mover un disco cada vez.

2. No se puede poner un disco mayor encima de uno menor.

3. Nunca debes quedar un disco grande vacío en una torre.





# 数组作为函数参数

求数组最大元素

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

```
#define N 10
```

```
int main()
```

```
{
```

```
    int a[N] = {99, 90, 75, 84, 17, 24, 10, 55, 58, 53};
```

```
    int i, max_val;
```

```
    max_val = a[0];
```

```
    for (i = 1; i < N; i++)
```

```
        if (a[i] > max_val)
```

```
            max_val = a[i];
```

```
    printf("%d\n", max_val);
```

```
    return 0;
```

```
}
```

# 最大值函数



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

```
#define N 10
```

```
int max_element(int a[N])
```

```
{
```

```
    int i, max_val;
```

```
    max_val = a[0];
```

```
    for (i = 1; i < N; i++)
```

```
        if (a[i] > max_val)
```

```
            max_val = a[i];
```

```
    return max_val;
```

```
}
```

```
int main()
```

```
{
```

```
    int a[N] = {99, 90, 75, 84, 17, 24, 10, 55, 58, 53};
```

```
    printf("%d\n", max_element(a));
```

```
    return 0;
```

```
}
```

## 数组参数

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

```
#define N 10
```

```
int max_element(int a[N])
```

```
{
```

```
    int i, max_val;
```

```
    max_val = a[0];
```

```
    for (i = 1; i < N; i++)
```

```
        if (a[i] > max_val)
```

```
            max_val = a[i];
```

```
    return max_val;
```

```
}
```

```
int main()
```

```
{
```

```
    int a[N] = {99, 90, 75, 84, 17, 24, 10, 55, 58, 53};
```

```
    printf("%d\n", max_element(a));
```

```
    return 0;
```

```
}
```

## 数组名

# 数组作为参数

- 数组类型需一致
- 不做数组大小检查
- 传地址

## 数组参数

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

```
#define N 10
```

```
int max_element(int a[N]) ⇔ int max_element(int a[])
```

```
{
```

```
    int i, max_val;
```

```
    max_val = a[0];
```

```
    for (i = 1; i < N; i++)
```

```
        if (a[i] > max_val)
```

```
            max_val = a[i];
```

```
    return max_val;
```

```
}
```

```
int main()
```

```
{
```

```
    int a[N] = {99, 90, 75, 84, 17, 24, 10, 55, 58, 53};
```

```
    printf("%d\n", max_element(a));
```

```
    return 0;
```

```
}
```

## 数组名

如何处理数组长度？

输入n个数， 求平均数

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

```
float average(???)
```

```
{
```

```
    ???
```

```
}
```

```
int main()
```

```
{
```

```
    int a[100], n, i;
```

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

```
    for (i = 0; i < n; i++)
```

```
        scanf("%d", &a[i]);
```

```
    printf("Average = %f\n", ???);
```

```
    return 0;
```

```
}
```

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

```
float average(int a[], int n)
{
    int i;
    float sum = 0.0f;

    for (i = 0; i < n; i++)
        sum += a[i];

    return sum / n;
}
```

```
int main()
{
    int a[100], n, i;

    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Average = %f\n", average(a, n));

    return 0;
}
```



# 数组的参数传递方式

# 多维数组作为参数

3x4 矩阵的平均值

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

```
float average(int a[][4])  
{  
    int i, j;  
    float sum = 0.0f;  
  
    for (i = 0; i < 3; i++)  
        for (j = 0; j < 4; j++)  
            sum += a[i][j];  
  
    return sum / 12;  
}
```

```
int main()  
{  
    int a[3][4] = {{31, 17, 82, 31},  
                   {77, 16, 10, 91},  
                   {31, 57, 16, 25}};  
  
    printf("Average = %f\n", average(a));  
  
    return 0;  
}
```

可见性

存在性

# 作用域与生命周期

# 变量作用域

# 作用域

块内部有效

- 局部变量
- 函数内部变量
- 函数参数
- 全局变量
- 所有函数外部

全局有效

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

```
void f()
```

```
{
```

```
    int i = 3;
```

```
    printf("i = %d\n", i);
```

```
}
```

```
int main(void)
```

```
{
```

```
    int i = 1;
```

```
    printf("i = %d\n", i);
```

```
{
```

```
    printf("i = %d\n", i);
```

```
    int i = 2;
```

```
    printf("i = %d\n", i);
```

```
}
```

```
    printf("i = %d\n", i);
```

```
    f();
```

```
    return 0;
```

```
}
```



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

```
void f()
```

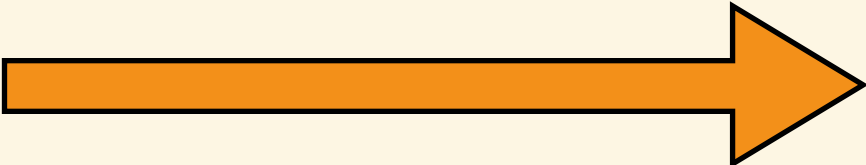
```
{  
    int i = 3;  
    printf("i = %d\n", i);  
}
```

```
int main(void)
```

```
{  
    int i = 1;  
    printf("i = %d\n", i);  
  
    {  
        printf("i = %d\n", i);  
        int i = 2;  
        printf("i = %d\n", i);  
    }  
  
    printf("i = %d\n", i);  
    f();  
  
    return 0;  
}
```

局部变量作用域从定义到块结束

局部优先原则



i	=	1
i	=	1
i	=	2
i	=	1
i	=	3

# 变量解析顺序

- 先在当前块查找
- 如果没有找到，到上一层的块继续查找

# 例子分析

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

```
int a = 1;  
int b = 2;
```

全局变量

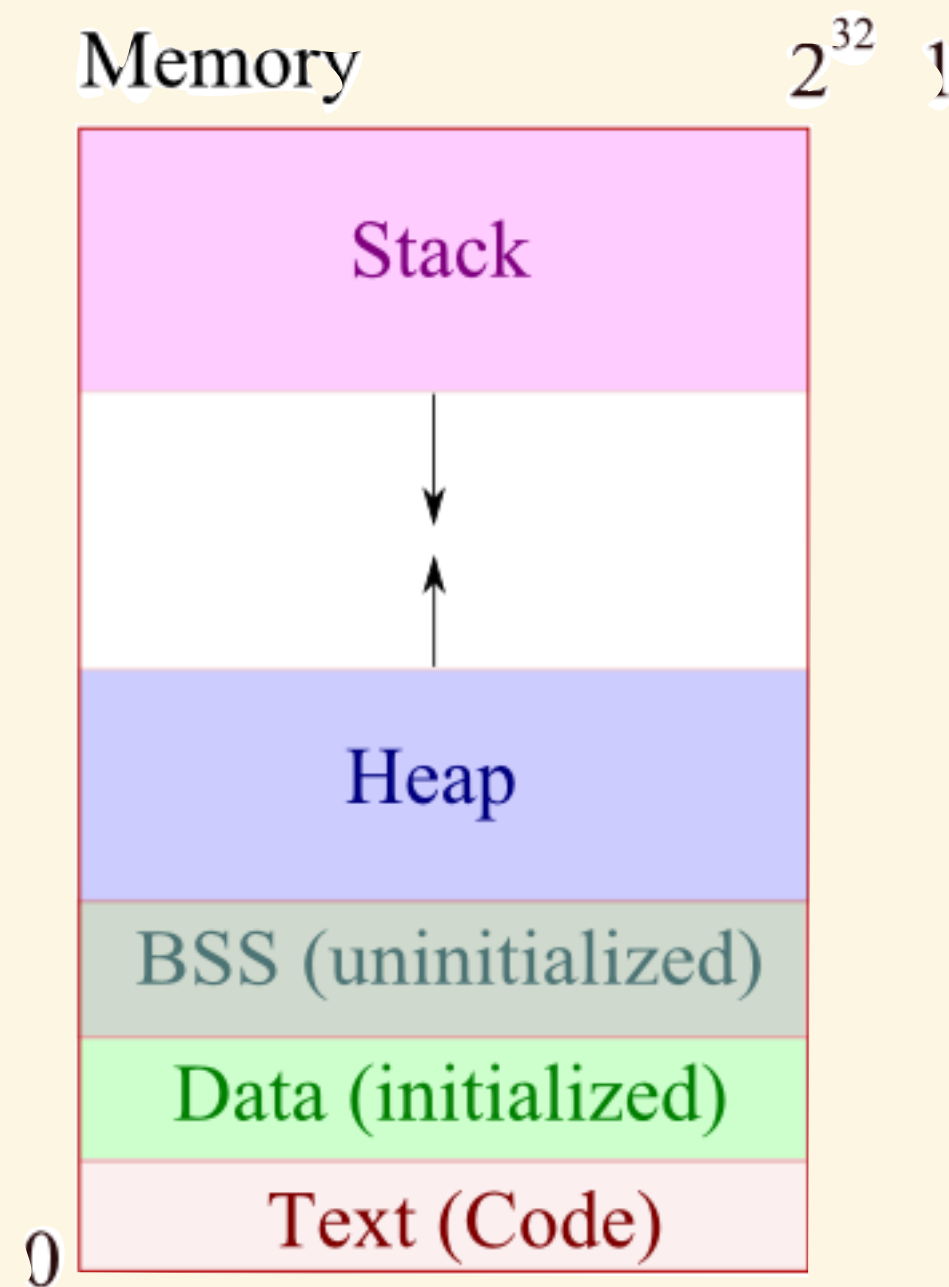
```
int max(int a, int b)  
{  
    return a > b ? a : b;  
}
```

```
int main()  
{  
    int a = 3;  
  
    printf("Max = %d\n", max(a, b));  
  
    return 0;  
}
```

# 生命周期



# 堆与栈





# 堆与栈



栈



堆

# 变量生存周期

- 局部（自动）变量
- 局部静态变量
- 全局变量
- 静态全局变量



# 局部（自动）变量

`auto int a; ⇔ int a;`

- 动态存储（栈）
- 定义时创建
- 块结束时释放
- 未初始化时，值不确定

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

```
int main()
```

```
{
```

```
    int a, b = 0;
```

```
    printf("a = %d, b = %d\n", a, b);
```

```
    if (a == b) {
```

```
        int c = 2;
```

```
        printf("c = %d\n", c);
```

```
    }
```

```
    return 0;
```

```
}
```

# 静态局部变量

```
static int a;
```

- 静态存储（堆）
- 只被初始化一次
- 未初始化时默认为0
- 在程序结束时被释放

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

```
void f()
```

```
{
```

```
    int a = 1;
```

```
    static int b;
```

```
    printf("a = %d, b = %d\n", a, b);
```

```
    a++;
```

```
    b++;
```

```
}
```

```
int main()
```

```
{
```

```
    f();
```

```
    f();
```

```
    f();
```

```
    return 0;
```

```
}
```

# 统计函数调用次数

# 全局变量

- 静态存储（堆）
- 程序结束时释放
- 未初始化时值为0
- 在其它文件中用extern声明

```
/* main.c */
#include <stdio.h>

int global = 3;

void print_global();

void change_global(int value);

int main()
{
    printf("global = %d\n", global);
    print_global();
    change_global(5);
    print_global();
    printf("global = %d\n", global);

    return 0;
}
```

```
/* global.c */
#include <stdio.h>

extern int global;

void print_global()
{
    printf("print global: %d\n", global);
}

void change_global(int value)
{
    global = value;
}
```



# 静态全局变量

- 与全局变量的区别在于，静态全局变量只能用在本文中

```
/* main.c */
#include <stdio.h>

static int global = 3;

void print_global();

void change_global(int value);

int main()
{
    printf("global = %d\n", global);
    print_global();
    change_global(5);
    print_global();
    printf("global = %d\n", global);

    return 0;
}
```



2. ~/sandbox (zsh)



```
> clang main.c global.c
```

```
Undefined symbols for architecture x86_64:
```

```
  "_global", referenced from:
```

```
    _print_global in global-w80Fr4.o
```

```
    _change_global in global-w80Fr4.o
```

```
  (maybe you meant: _change_global, _print_global )
```

```
ld: symbol(s) not found for architecture x86_64
```

```
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

```
> █
```



# 静态函数

```
static void f()  
{  
    printf("Hello\n");  
}
```

- 只能在该文件中使用

# 外部函数

```
extern void f()  
{  
    printf("Hello\n");  
}
```

- 可以在其它文件中使用
- extern可以省略