

# 《计算机语言与程序设计》

## 第5周 函数

清华大学 自动化系  
范 静 涛

# 本节课主要内容

为什么有函数?

怎么写函数?

怎么用函数?

函数调用过程发生了什么?

作用域与生命周期

# 概述：模块化程序设计

## 基本思想

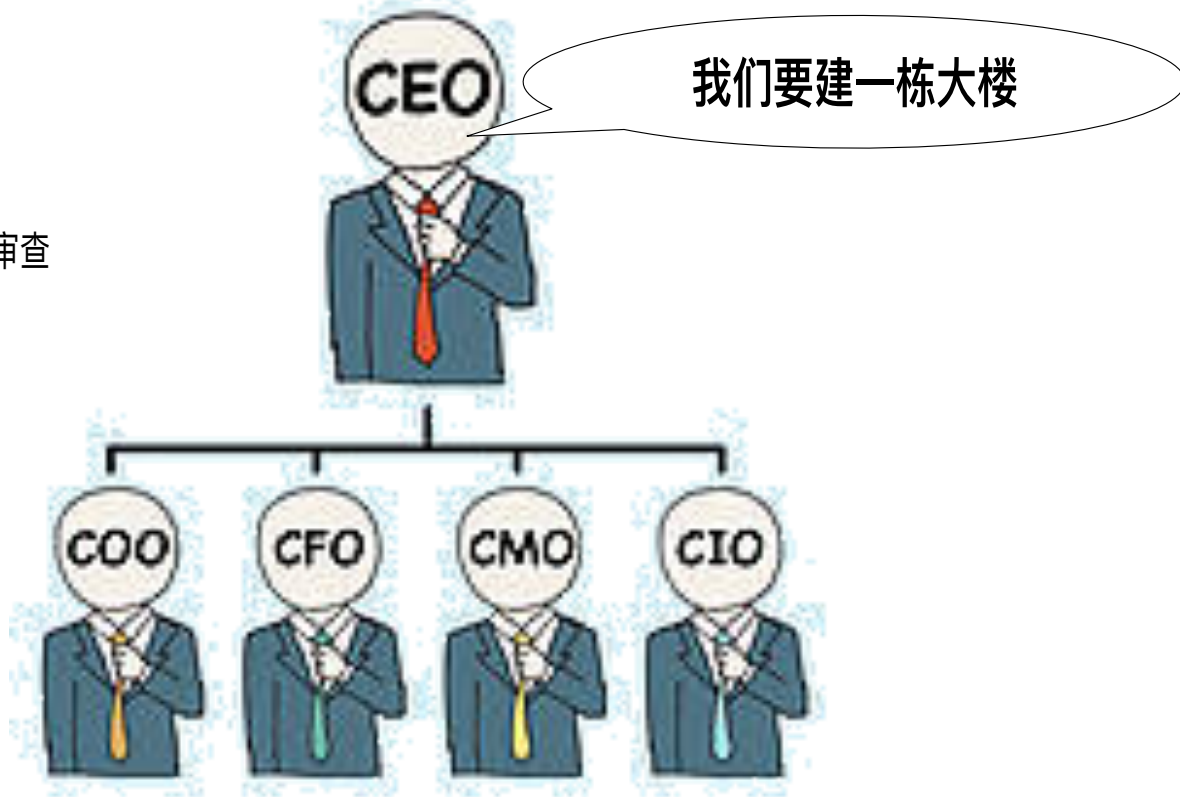
- ❑ 把较大的任务**分解**成若干个较小的任务，并**提炼**出公用任务

## 特点

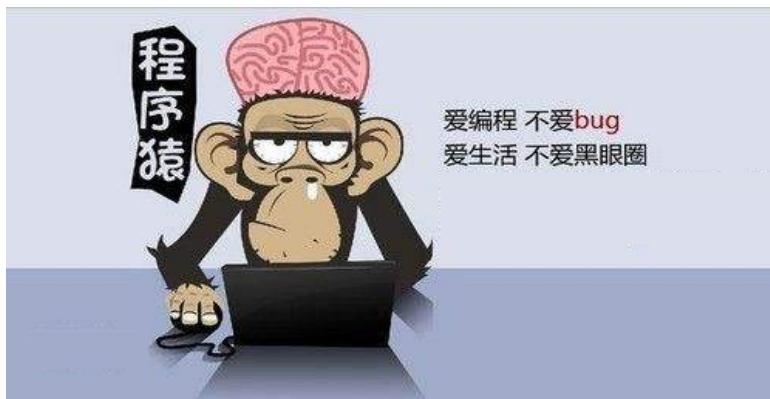
- ❑ 使整个程序结构清楚
- ❑ 各模块相对独立、功能单一、结构清晰、接口简单
- ❑ 控制了程序设计的复杂性
- ❑ 提高元件的可靠性
- ❑ 避免程序开发的重复劳动、缩短开发周期
- ❑ 易于维护和功能扩充

# 概述：模块化程序设计

1. 选址
2. 规划总图
3. 初步设计
4. 施工图审查
5. 规划报建图审查
6. 施工报建
7. 综合验收



# 概述：模块化程序设计



解决一个实际问题需要多少行程序？

`main()`当中能放多少行程序？

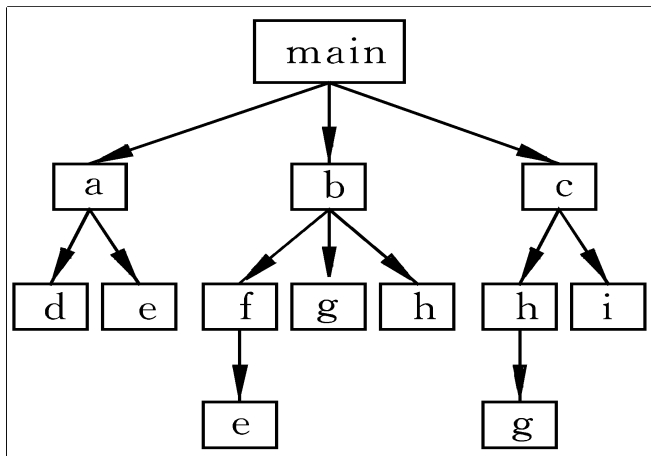
读/写多少行的程序能保持头脑清晰？

如果所有代码都在`main()`当中，怎么团队合作？

如果代码都在一个文件中，怎么团队合作？

# 概述：函数与文件

## 功能上, C语言通过函数来实现模块化



自上向下  
逐步分解  
分而治之

- C程序的执行总是从**main**函数开始，在**main**中结束
- 所有函数定义是平行的、独立的，不能嵌套定义
- 函数间可以互相调用，但不能调用**main**函数 (main函数是系统调用的)

# 函数的声明和定义: 要素

函数定义包括**四大要素**

- ?** 确定返回值类型
- ?** 确定函数名
- ?** 确定形式参数列表
- ?** 编写函数体代码

**函数声明** 也叫**函数头**  
仅表示存在性  
不管具体实现

**函数定义**  
存在性且要有具体实现

**说明:**

- 函数名、形式参数列表、返回值类型组成的**函数头**，也称为**函数接口** (interface)
- 一组适合应用程序开发的函数接口统称为应用程序接口API。

```
DataType FunctionName(ParameterList) {  
  
  
}
```

# 本节课主要内容

为什么有函数？

**怎么写函数？**

怎么用函数？

函数调用过程发生了什么？

作用域与生命周期



# 函数的声明和定义: 要素

返回值类型

函数名

(形式参数列表)

```
int main(int argc, char* argv[]) {  
    return 0;  
}
```

函数体

返回值类型

函数名

(形式参数列表)

```
int max(int iNum1, int iNum2) {  
    return iNum1 > iNum2 ? iNum1 : iNum2;  
}
```

函数体

# 函数的声明和定义：函数名与形参

## 函数名

- ❓ 函数需要确定函数名，以便使用函数时能够按名引调用。
- ❓ 函数名遵守C语言标识符规则，通常要“见其名知其意”。
- ❓ 一个程序中（包括多个文件的情况），不允许存在多个同名函数

## 形式参数列表

- ❓ 实现函数需要确定参数的个数、类型
- ❓ 形式参数列表是函数与调用者进行数据交换的途径，多个参数用逗号（，）分隔，且每个参数都要有自己的类型说明
- ❓ 参数类型可以是任意数据类型

```
int fun(int x, int y, double m)
{
    return m > 12.5 ? x : y;
}
```

# 函数的声明和定义：返回值类型

## 返回值类型

- ❓ 实现函数需要确定**有无返回数据**、返回**什么类型的数据**。
- ❓ 返回值是函数**向调用者返回数据**的途径之一，本质上函数返回值也起到与调用者进行数据交换的作用，只不过它是**单向**的，即从函数向调用者传递，故称返回。

说明：

- 返回值类型，可以是**除数组之外的任何类型**。
- 函数可以不返回数据，此时返回值类型应写成**void**，表示没有返回值。

```
void fun(int x, int y, double m)
```

# 函数的声明和定义：返回值类型

碰到**return**，当前函数立即结束执行

函数定义中的 函数类型	return形式	是否正确
非void	无	实际返回了随机值
	return;	错误
	return 表达式;	实际返回了表达式的值
void	无	正确
	return;	
	return 表达式;	错误

# 函数的声明和定义：函数体

## 函数体

- ❓ 实现函数最重要的是编写函数体。
- ❓ 函数体 (function body)  
    ) 包含：函数内的声明部分和执行语句，是一组能实现特定功能的语句序列的集合。

### 说明：

- 函数体本身可以理解为一个完整的程序
- 在函数体内部可以**声明**数据类型、**定义**变量.....;
- 可以使用任意结构的程序**流程**: while, for, do-while, exit, ...
- 可以使用简单**语句**、复合语句、控制语句及语句嵌套
- 可以**调用**别的**函数**

可以嵌套调用  
但不可以嵌套定义

# 函数的声明和定义：定义不可嵌套

C语言不允许在函数体内嵌套定义函数

```
返回值类型 函数名1(类型 参数1,类型 参数2,... )
{
    ...
    返回值类型 函数名2(类型 参数1,类型 参数2,... )
    {
        函数体
        return 表达式;
    }
    ...
    return 表达式;
}
```



# 函数的声明和定义：平时编程的要求

## 无参函数

- ❓ 执行过程与参数无关（不需要受参数控制）。
- ❓ 形式参数列表必须写为**void**

```
float GetPI(void) {  
    return 3.1415926f;  
}
```

## 无返回值类型函数

- ❓ 一般用来执行一组特定操作，无需执行结果。
- ❓ 返回值类型必须写为**void**

```
void IncArray(int Array[], int Size) {  
    for (int i = 0; i < Size; i++) {  
        Array[i]++;  
    }  
    return;  
}
```

# 本节课主要内容

为什么有函数?

怎么写函数?

**怎么用函数?**

函数调用过程发生了什么?

作用域与生命周期



# 函数的调用：形式

## 函数调用的一般形式

函数名 (实际参数1, 实际参数2, ...);

```
#include <stdio.h>
int max(int x, int y);
int main(int argc, char* argv[]) {
    int iNum1 = 12;
    int iNum2 = 24;
    int result;
    result = max(iNum1, iNum2);
    printf("Max is %d\n", result);
    return 0;
}
```

实际参数

发起调用的，主调函数

```
// 此处省略N多字
int max(int x, int y) {
    //函数体中的声明部分
    int z;
    z = x > y ? x : y;
    return z;
}
```

形式参数

被调用的函数，被调函数

先声明，后调用，定义在哪里无所谓

# 函数的调用：实际参数

```
/* 函数定义*/      返回值类型 函数名(类型 形式参数1,类型 形式参数2,... )
{
    函数体
    return 表达式;
}
```

```
/* 函数调用*/      函数名 (实际参数1, 实际参数2, ...);
```

主调函数提供给被调函数的参数称为实际参数，简称实参。

- ❓ 实参必须有确定的值，因为调用函数会将这些值传递给形参。
- ❓ 实参可以是常量、变量或表达式，还可以是函数的返回值。

例如：

```
x = max(a, b); //max函数调用，实参为a,b
y = max(a + 3, 128); //max函数调用，实参为a+3,128
z = max(max(a, b), c); //max函数调用，实参为max(a,b), c
```

# 函数的调用：数据传输

形式参数是实际参数的**值拷贝**

函数值是被调函数return表达式的**值拷贝**

```
#include <stdio.h>
int max(int x, int y);
int main(int argc, char* argv[]) {
    int iNum1 = 12;
    int iNum2 = 24;
    int result;
    result = max(iNum1, iNum2);
    printf("Max is %d\n", result);
    return 0;
}
```

实际参数

```
// 此处省略N多字
int max(int x, int y) {
    //函数体中的声明部分
    int z;
    z = x > y ? x : y;
    return z;
}
```

形式参数

**x不是iNum1**

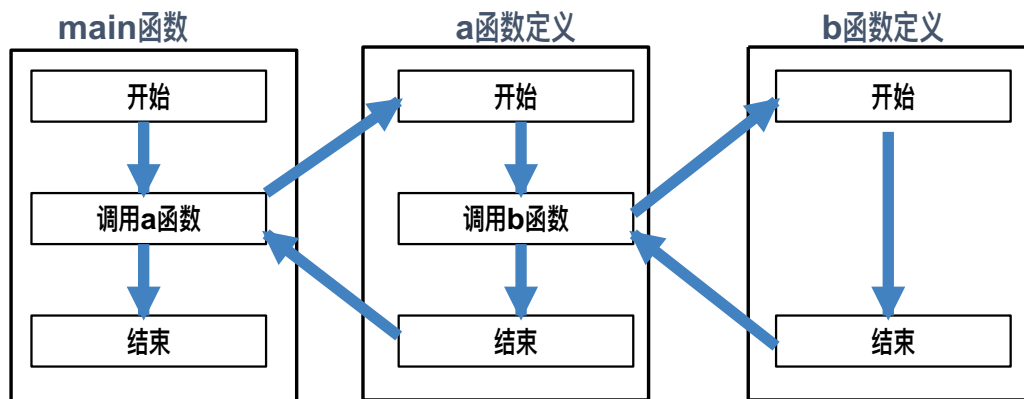
**y不是iNum2,** 它们都有各自的存储空间, 仅仅是值相同

**result不是z**

# 函数的调用：调用方式

## 嵌套调用

- ❓ C语言的函数定义互相平行、独立
- ❓ 不能嵌套定义函数，但可以嵌套调用函数



# 本节课主要内容

为什么有函数?

怎么写函数?

怎么用函数?

**函数调用过程发生了什么?**

作用域与生命周期

# 栈 (stack)

所有函数的内部变量、形参、返回值都开辟在一种叫做栈 (stack) 的内存中。

栈是内存管理中的一种数据结构，是一种先进后出的数据表。栈最常见操作：进栈 (push) 和出栈 (pop) 。

系统为每次函数调用在“栈”中建立独立的栈框架，称为函数调用栈帧 (stack frame) ，其建立和撤销是自动维护的。

# 栈 (stack)

```
→ int main(int argc, char* argv[]) {  
    float fNum1;  
    float fNum2;  
    int result;  
    .....  
    return 0;  
}
```

栈顶和栈底是同一个单元时, 叫做**空栈**

内存地址最大的单元, 叫做**栈底**

→ 指向将要执行的指令, 叫做**IP**

→ 第一个为存放数据的单元, 叫做**栈顶**

内存地址	对应变量/常量名	存放数值
.....		
0x4004		
0x4008		
0x400C		
0x4010		
0x4014		
0x4018		
0x401C		
0x4020		
0x4024		



# 栈 (stack)

```
int main(int argc, char* argv[]) {  
    float fNum1;  
    float fNum2;  
    int result;  
    .....  
    return 0;  
}
```

开始调用main，返回值、形参push进栈

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

内存地址	对应变量/常量名	存放数值
.....		
0x4004		
0x4008		
0x400C		
0x4010		
0x4014		
0x4018		
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??



# 栈 (stack)

```
int main(int argc, char* argv[]) {  
    float fNum1;  
    float fNum2;  
    int result;  
    .....  
    return 0;  
}
```

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**



接着，内部变量/常量**PUSH**进展

内存地址	对应变/常量名	存放数值
.....		
0x4004		
0x4008		
0x400C		
0x4010	result	??
0x4014	fNum2	??
0x4018	fNum1	??
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??

# 栈 (stack)

```
int main(int argc, char* argv[]) {  
    float fNum1;  
    float fNum2;  
    int result;  
    .....  
    return 0;  
}
```

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**



执行完**return**后，对返回值空间赋值

内存地址	对应变/常量名	存放数值
.....		
0x4004		
0x4008		
0x400C		
0x4010	result	??
0x4014	fNum2	??
0x4018	fNum1	??
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	0

内存地址最大的单元，叫做**栈底**

# 栈 (stack)

```
int main(int argc, char* argv[]) {  
    float fNum1;  
    float fNum2;  
    int result;  
    .....  
    return 0;  
}
```



执行完整个函数后，全部main  
函数的返回值、形参、内部变量/常量POP  
。再次变为空栈  
但是，仅仅调整了栈顶，并未清除数据

内存地址最大的单元，叫做栈底



➡ 指向将要执行的指令，叫做IP

➡ 第一个为存放数据的单元，叫做栈顶

内存地址	对应变/常量名	存放数值
.....		
0x4004		
0x4008		
0x400C		
0x4010		??
0x4014		??
0x4018		??
0x401C		指向可执行文件名字符串
0x4020		1
0x4024		0

# 栈 (stack)

```
int main(int argc, char* argv[]) {  
    float fNum1;  
    float fNum2;  
    int result;  
    .....  
    return 0;  
}
```

在某一函数执行过程中，在栈中占据的最大范围叫做**栈帧 (stack frame)**

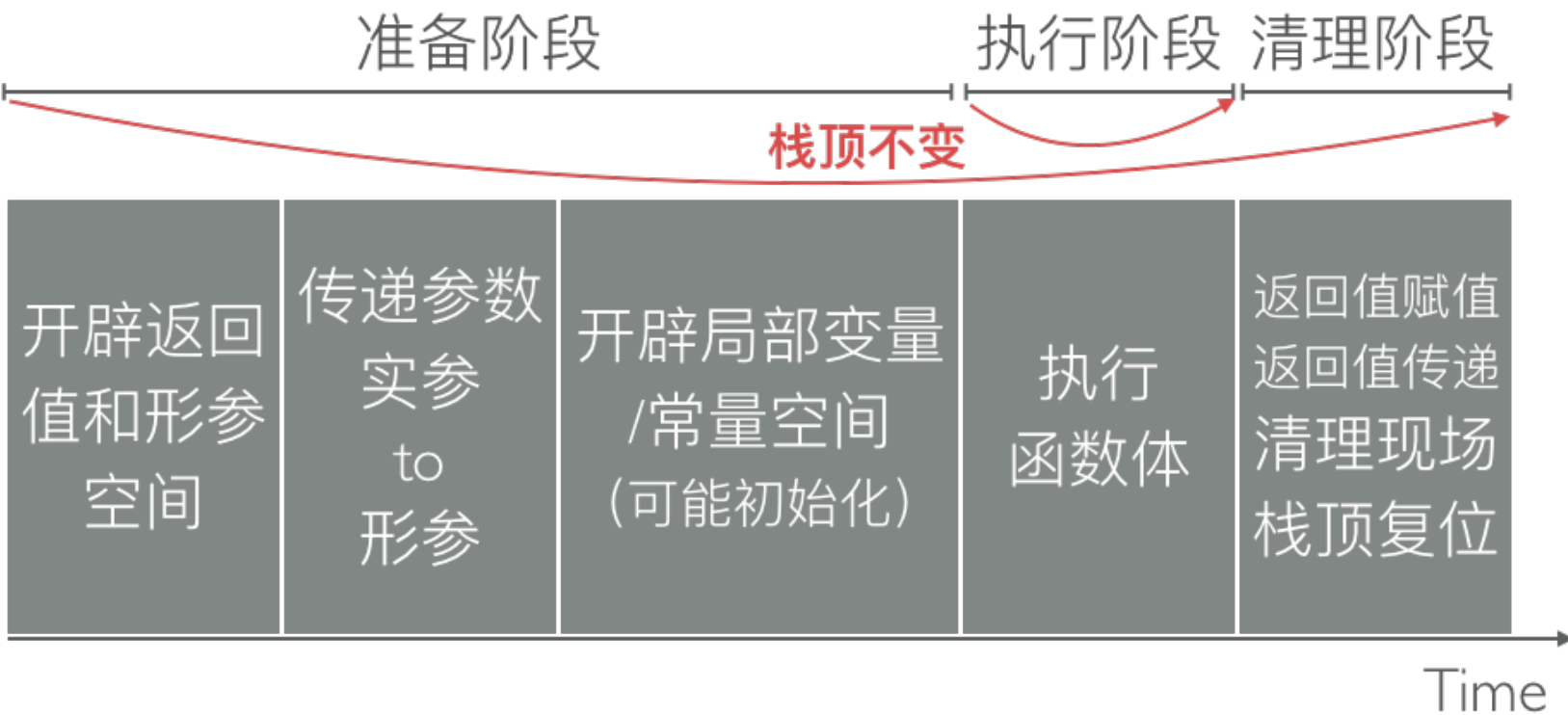
main函数的栈帧

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

内存地址	对应变/常量名	存放数值
.....		
0x4004		
0x4008		
0x400C		
0x4010	result	??
0x4014	fNum2	??
0x4018	fNum1	??
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	0

# 调用时序: 三个阶段



# 栈帧：实例解析

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

→

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

系统调用main前：空栈

→ 指向将要执行的指令，叫做**IP**

→ 第一个为存放数据的单元，叫做**栈顶**

内存地址	对应变量/常量名	存放数值
0x4000		??
0x4004		??
0x4008		??
0x400C		??
0x4010		??
0x4014		??
0x4018		??
0x401C		??
0x4020		??
0x4024		??



# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main函数，准备阶段，STEP 1：**  
**返回值、形参PUSH进栈**

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

内存地址	对应变量/常量名	存放数值
0x4000		??
0x4004		??
0x4008		??
0x400C		??
0x4010		??
0x4014		??
0x4018		??
0x401C	argv	??
0x4020	argc	??
0x4024	main函数返回值	??



# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main函数，准备阶段，STEP 2:**  
**实参向形参传值**

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

内存地址	对应变量/常量名	存放数值
0x4000		??
0x4004		??
0x4008		??
0x400C		??
0x4010		??
0x4014		??
0x4018		??
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??





# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main函数，准备阶段，STEP 3:**  
**内部变量PUSH进栈并初始化**

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

内存地址	对应变名/常量名	存放数值
0x4000		??
0x4004		??
0x4008		??
0x400C		??
0x4010	result	??
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??

# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main函数，执行阶段：**  
**根据控制流程，依次执行每条指令**

**max函数，准备阶段，STEP 1：**  
**返回值、形参PUSH进栈**

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**



内存地址	对应变量/常量名	存放数值
0x4000		??
0x4004	y	??
0x4008	x	??
0x400C	max函数返回值	??
0x4010	result	??
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??

# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main函数，执行阶段：**  
**根据控制流程，依次执行每条指令**

**max函数，准备阶段，STEP 2：**  
**实参向形参传值**

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**



内存地址	对应变量/常量名	存放数值
0x4000		??
0x4004	y	2.5f
0x4008	x	1.5f
0x400C	max函数返回值	??
0x4010	result	??
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??



# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main函数，执行阶段：**  
**根据控制流程，依次执行每条指令**

**max函数，准备阶段，STEP 3：**  
**内部变量PUSH进栈并初始化**

→ 指向将要执行的指令，叫做**IP**

→ 第一个为存放数据的单元，叫做**栈顶**



0x3FFC	对应变量/常量名	存放数值
0x4000	z	??
0x4004	y	2.5f
0x4008	x	1.5f
0x400C	max函数返回值	??
0x4010	result	??
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??

# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main**函数，执行阶段：  
根据控制流程，依次执行每条指令

**max**函数，执行阶段：  
内部变量**PUSH**进栈并初始化

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

	对应变量/常量名	存放数值
0x3FFC		
0x4000	z	2.5f
0x4004	y	2.5f
0x4008	x	1.5f
0x400C	max函数返回值	??
0x4010	result	??
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??

# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main函数，执行阶段：**  
根据控制流程，依次执行每条指令

**max函数，返回阶段，STEP 1：**  
从return表达式向返回值空间赋值

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

	对应变量/常量名	存放数值
0x3FFC		
0x4000	z	2.5f
0x4004	y	2.5f
0x4008	x	1.5f
0x400C	max函数返回值	2.5f
0x4010	result	??
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??

# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main函数，执行阶段：**  
根据控制流程，依次执行每条指令

**max函数，返回阶段，STEP 2：**  
从max返回值向main函数中传值

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**



0x3FFC	对应变量/常量名	存放数值
0x4000	z	2.5f
0x4004	y	2.5f
0x4008	x	1.5f
0x400C	max函数返回值	2.5f
0x4010	result	2.5f
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??



# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

**main**函数，执行阶段：  
根据控制流程，依次执行每条指令

**max**函数，返回阶段，**STEP 3**：  
**栈帧POP**（栈顶复位）

➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

0x3FFC	对应变量/常量名	存放数值
0x4000		2.5f
0x4004		2.5f
0x4008		1.5f
0x400C		2.5f
0x4010	result	2.5f
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	??



# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

main函数，返回阶段，STEP 1：  
从return表达式向返回值空间赋值

➡ 指向将要执行的指令，叫做IP

➡ 第一个为存放数据的单元，叫做栈顶

0x3FFC	对应变量/常量名	存放数值
0x4000		2.5f
0x4004		2.5f
0x4008		1.5f
0x400C		2.5f
0x4010	result	2.5f
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	0

# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

main函数，返回阶段，STEP 2:  
从main返回值向操作系统传值

➡ 指向将要执行的指令，叫做IP

➡ 第一个为存放数据的单元，叫做栈顶

0x3FFC	对应变量/常量名	存放数值
0x4000		2.5f
0x4004		2.5f
0x4008		1.5f
0x400C		2.5f
0x4010	result	2.5f
0x4014	fNum2	2.5f
0x4018	fNum1	1.5f
0x401C	argv	指向可执行文件名字符串
0x4020	argc	1
0x4024	main函数返回值	0

# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

main函数，返回阶段，STEP 3:  
栈帧POP (栈顶复位)

➡ 指向将要执行的指令，叫做IP

➡ 第一个为存放数据的单元，叫做栈顶

0x3FFC	对应变量/常量名	存放数值
0x4000		2.5f
0x4004		2.5f
0x4008		1.5f
0x400C		2.5f
0x4010		2.5f
0x4014		2.5f
0x4018		1.5f
0x401C		指向可执行文件名字符串
0x4020		1
0x4024		0



# 栈帧：实例解析

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

```
int main(int argc, char* argv[]) {  
    float fNum1 = 1.5f;  
    float fNum2 = 2.5f;  
    int result;  
    result = max(fNum1, fNum2);  
    return 0;  
}
```

随着main函数结束，整个程序运行结束

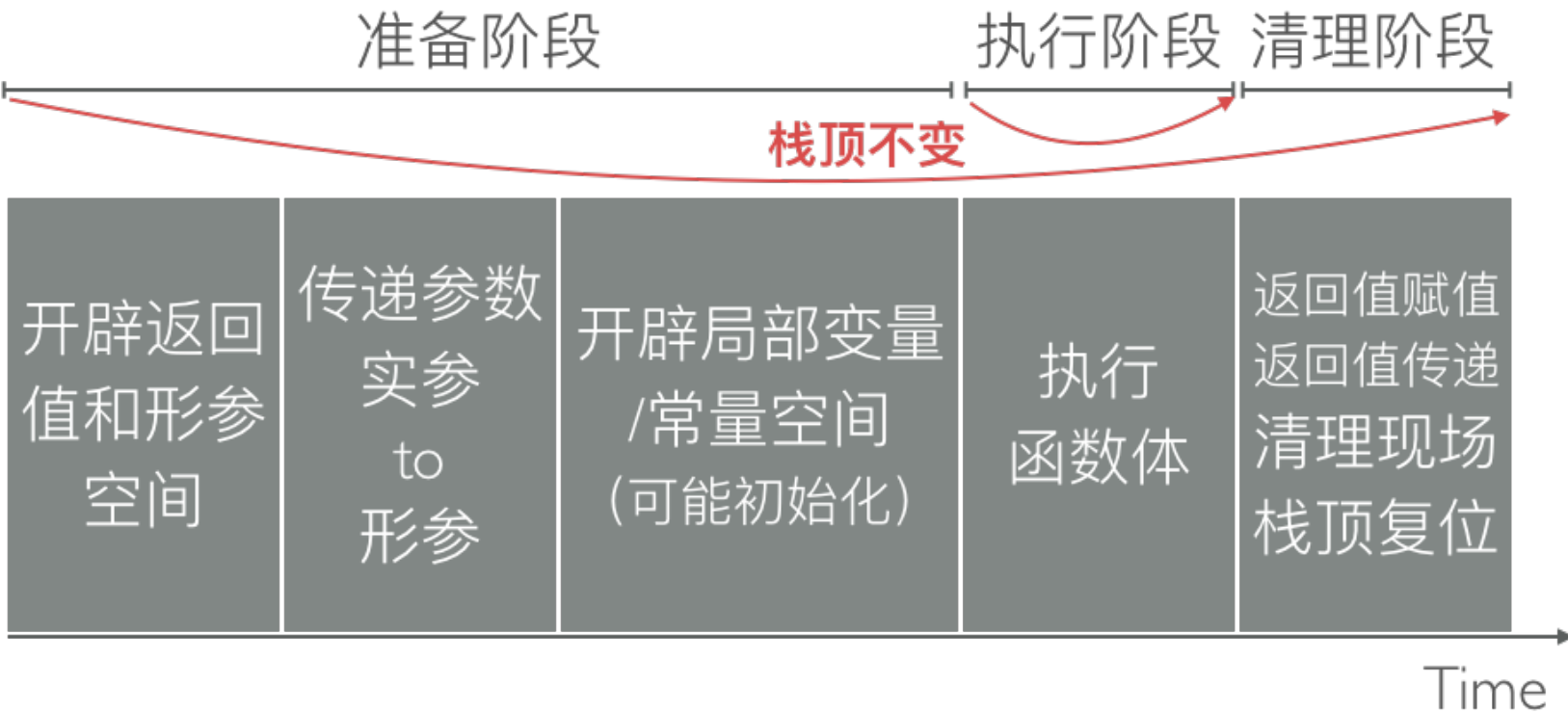
➡ 指向将要执行的指令，叫做**IP**

➡ 第一个为存放数据的单元，叫做**栈顶**

0x3FFC	对应变量/常量名	存放数值
0x4000		2.5f
0x4004		2.5f
0x4008		1.5f
0x400C		2.5f
0x4010		2.5f
0x4014		2.5f
0x4018		1.5f
0x401C		指向可执行文件名字符串
0x4020		1
0x4024		0



# 思考：写什么样的函数最划算？



执行阶段相对与准备阶段和清理阶段，**越长越好**

# 思考：写什么样的函数最划算？

准备阶段(1/3)，执行阶段(1/3)，清理阶段(1/3)

每次调用：1/3的时间用于计算，2/3的时间用于准备和清理

调用次数越多，相对损失越大！！

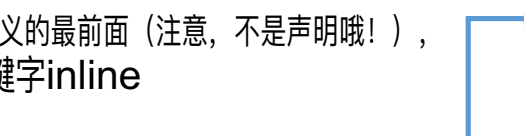
怎么办呢？

# 思考：写什么样的函数最划算？

## C99中，使用内联函数(inline function)

内联表明编译器将函数的每一次调用都用函数的机器指令来代替，减低函数调用消耗的时间和空间资源

在函数定义的最前面（注意，不是声明哦！），  
加上关键字inline



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

**内联函数的定义，必须！必须！必须！写在头文件里**

内联函数里不能有循环和switch

# 本节课主要内容

为什么有函数?

怎么写函数?

怎么用函数?

函数调用过程发生了什么?

**作用域与生命周期**



# 函数的声明和定义：平时编程的要求

如何定义自己的函数？**任何一个自己定义的函数，都要拆分成两个文件：**

- ❓ 头文件（.h文件），放入函数声明
- ❓ 源文件（.c文件），放入函数定义
- ❓ 如何在VS2019、DEV C++等环境下添加.c和.h文件，请自行学习

```
#ifndef _MATHCOMPARE_H_
#define _MATHCOMPARE_H_
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);
#endif
```

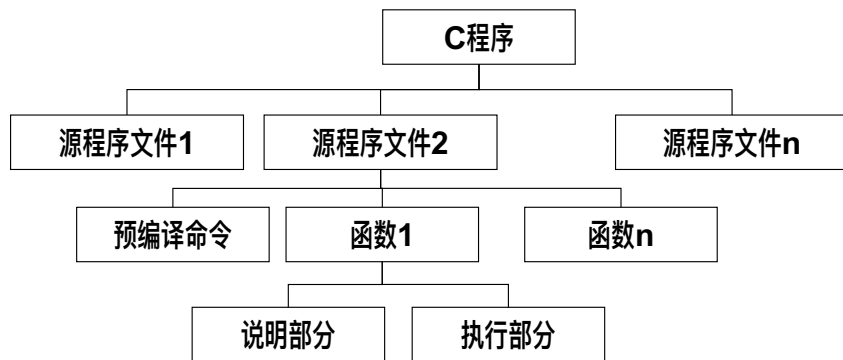
MathCompare.h

```
#include "MathCompare.h"
int Max(int iNum1, int iNum2) {
    if (iNum1 > iNum2){
        return iNum1;
    }
    return iNum2;
}
int Min(int iNum1, int iNum2) {
    if (iNum1 < iNum2){
        return iNum1;
    }
    return iNum2;
}
```

MathCompare.c

# 函数的声明和定义：函数与文件

形式上，C语言通过文件实现模块化



说明：

- 一个C程序由一个或多个程序模块组成，每个程序模块作为一个源程序文件，较大的程序通常放在若干源文件中。
- 各文件分别编写、分别编译，提高调试效率。
- 一个源程序文件（.c文件）是一个编译单位，程序编译是以源程序文件为单位，而不是以函数为单位。
- 一个源程序文件包含一个或多个函数及其他有关内容（命令行、数据定义等）。
- 一个源程序文件可以为多个C程序公用。

# 函数的声明和定义：自定义函数要求

头文件（.h文件） 构成

- ❓ 第1行、第2行、最后一行，称之为“哨兵”，防止重复编译
- ❓ 其余行只能是函数声明，不能有函数定义

## MathCompare.h



```
#ifndef _MATHCOMPARE_H_
#define _MATHCOMPARE_H_
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);
#endif
```

头文件名的所有字母都变为大写  
'\_'改为'\_'  
前后各加一个'\_'

# 函数的声明和定义：自定义函数要求

为什么要有哨兵？

- ◆ 从源代码到生成可执行文件的过程中，第一个步骤称为预处理（preprocessing）。负责把#include指定的头文件插入到c文件中。
- ◆ 但如果，在c文件中有多次 #include "MathCompare.h"，又没有使用哨兵时

```
#include "MathCompare.h"
#include "MathCompare.h"
int main(int argc, char* argv[]) {
    return 0;
}
```



```
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);
int main(int argc, char* argv[]) {
    return 0;
}
```

声明了多个同名函数  
不符合编译要求

# 函数的声明和定义：自定义函数要求

哨兵如何工作？

- ◆ `#ifndef _MATHCOMPARE_H_`  
询问编译器，之前是否没有定义过一个叫做 `_MATHCOMPARE_H_` 的宏？
- ◆ 如果没有定义过，则一直到 `#endif` 之间的代码参与编译
- ◆ 如果有定义过，则一直到 `#endif` 之间的代码不参与编译
- ◆ `#define _MATHCOMPARE_H_` 定义一个叫做 `_MATHCOMPARE_H_` 的宏

```
#include "MathCompare.h"  
#include "MathCompare.h"  
int main(int argc, char* argv[]) {  
    return 0;  
}
```



只有第一个 `include` 起作用

```
int Max(int iNum1, int iNum2);  
int Min(int iNum1, int iNum2);  
int main(int argc, char* argv[]) {  
    return 0;  
}
```

# 函数的声明和定义：自定义函数要求

源文件（.c文件） 构成

- ❓ 第1行，对同名头文件的包含
- ❓ 添加必要的其他头文件包含
- ❓ 全部自定义函数的定义

```
#include "MathCompare.h"
int Max(int iNum1, int iNum2) {
    if (iNum1 > iNum2){
        return iNum1;
    }
    return iNum2;
}
int Min(int iNum1, int iNum2) {
    if (iNum1 < iNum2){
        return iNum1;
    }
    return iNum2;
}
```

MathCompare.c

# 函数的声明和定义：自定义函数要求

第1行，对同名头文件的包含。为什么？

```
#include "MathCompare.h"
int Max(int iNum1, int iNum2) {
    if (iNum1 < Min(iNum1, iNum2)){
        return iNum2;
    }
    return iNum1;
}
int Min(int iNum1, int iNum2) {
    if (iNum1 < iNum2){
        return iNum1;
    }
    return iNum2;
}
```

MathCompare.c

如果没有对自身头文件的包含，**Max**函数的定义中使用了**Min**函数。那么不满足“**先声明后使用**”的编译要求

包含自身头文件的目的：

**使此原文件中的各个自定义函数可以相互调用**

# 函数的声明和定义：自定义函数要求

添加必要的其他头文件包含。为什么？

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

int Max(int iNum1, int iNum2) {
    if (iNum1 < Min(iNum1, iNum2)){
        return iNum2;
    }
    printf("\n");
    return iNum1;
}

int Min(int iNum1, int iNum2) {
    if (iNum1 < iNum2){
        return iNum1;
    }
    return iNum2;
}
```

MathCompare.c

为了满足“**先声明后使用**”的编译要求，使用函数就要包含其头文件。

怎么知道某个函数使用什么头文件：  
**助教习题课讲**



# 函数的声明和定义：编译过程扩展知识

扩展名为.c的源文件，是编译的最小单元

对于每个.c源文件：

**[?] STEP 1, 预编译/预处理(preprocessing)**, 生成.I中间文件

- 将所有#include <...>和#include "...", 预编译指令, 都用对应的头文件内容替换
- 将所有宏进行替换和展开

```
#include <stdio.h>
#define PI 3.1415629
#define max(a, b) (a > b ? a : b)
int main(int argc, char* argv[]) {
    float fNum;
    scanf("%f", &fNum);
    fNum *= PI;
    return max(60, fNum);
}
```

source file, before preprocessing

用头文件内容替换

宏替换和展开

```
extern FILE * __stdin;
extern FILE * __stdout;
extern FILE * __stderr;
...
int scanf(const char * __restrict, ...);
...
↑引入了scanf的声明 (函数原型)

int main(int argc, char* argv[]) {
    float fNum;
    scanf("%f", &fNum);
    fNum *= 3.1415629;
    return (60 > fNum ? 60 : fNum);
}
```

Intermediate file, after preprocessing

# 函数的声明和定义：编译过程扩展知识

对于每个.i中间文件：

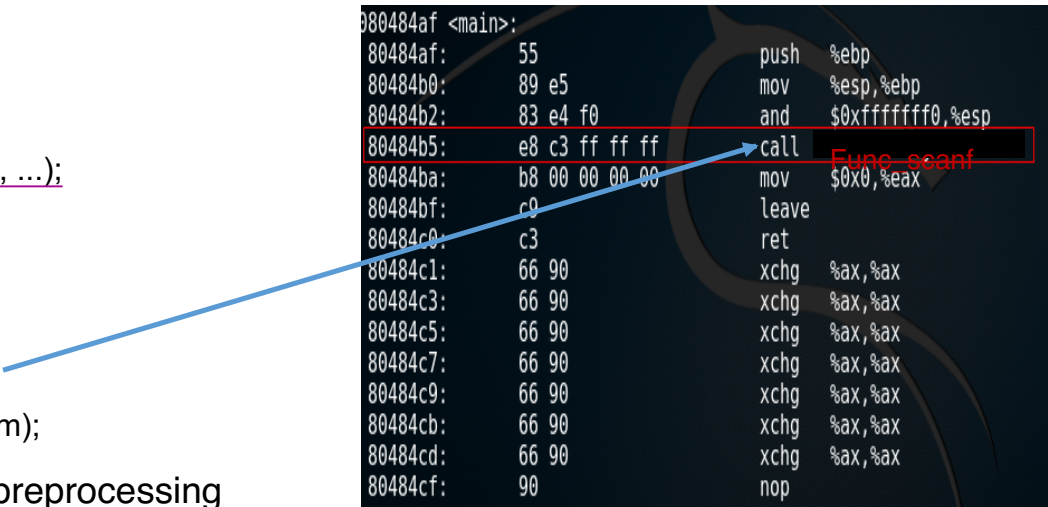
## STEP 2, 编译 (compile), 生成.obj目标文件

- 将所有C语言语句，转换为可执行的机器指令
- 中间文件中的自定义函数，都保存在一个全局的函数查找表中（所有.i共用一个查找表，每一项是函数名和函数机器指令地址）
- 所有调用函数的语句，还是用函数名，而非机器指令地址表示

```
extern FILE * __stdinp;
extern FILE * __stdoutp;
extern FILE * __stderrp;
...
int scanf(const char * restrict, ...);
...
```

```
int main(int argc, char* argv[]) {
    float fNum;
    scanf("%f", &fNum);
    fNum *= 3.1415629;
    return (60 > fNum ? 60 : fNum);
}
```

Intermediate file, after preprocessing



```
080484af <main>:
80484af: 55          push    %ebp
80484b0: 89 e5      mov     %esp,%ebp
80484b2: 83 e4 f0   and     $0xffffffff0,%esp
80484b5: e8 c3 ff ff ff call    Func:scanf
80484ba: b8 00 00 00 mov     $0x0,%eax
80484bf: c9        leave  %eax
80484c0: c3        ret
80484c1: 66 90     xchg    %ax,%ax
80484c3: 66 90     xchg    %ax,%ax
80484c5: 66 90     xchg    %ax,%ax
80484c7: 66 90     xchg    %ax,%ax
80484c9: 66 90     xchg    %ax,%ax
80484cb: 66 90     xchg    %ax,%ax
80484cd: 66 90     xchg    %ax,%ax
80484cf: 90        nop
```

# 函数的声明和定义：编译过程扩展知识

对于所有.Obj目标文件：

**[?] STEP 3, 链接(Link), 生成.exe可执行文件**

- 将所有.Obj文件和必要.lib库文件的顺序链接在一起
- 所有调用函数的语句的函数名，替换为对应的机器指令地址

```
080484af <main>:
80484af: 55          push    %ebp
80484b0: 89 e5      mov     %esp,%ebp
80484b2: 83 e4 f0   and     $0xffffffff0,%esp
80484b5: e8 c3 ff ff call    804847d
80484ba: b8 00 00 00 mov     $0x0,%eax
80484bf: c9        leave
80484c0: c3        ret
80484c1: 66 90     xchg    %ax,%ax
80484c3: 66 90     xchg    %ax,%ax
80484c5: 66 90     xchg    %ax,%ax
80484c7: 66 90     xchg    %ax,%ax
80484c9: 66 90     xchg    %ax,%ax
80484cb: 66 90     xchg    %ax,%ax
80484cd: 66 90     xchg    %ax,%ax
80484cf: 90        nop
```

**main.obj + stdio.lib = main.exe**

# 编译过程实例：源文件和头文件

## MathCompare.h

```
#ifndef _MATHCOMPARE_H_
#define _MATHCOMPARE_H_
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);
#endif
```

## MathCompare.c

```
#include "MathCompare.h"
int Max(int iNum1, int iNum2) {
    if (iNum1 > iNum2){
        return iNum1;
    }
    return iNum2;
}
int Min(int iNum1, int iNum2) {
    if (iNum1 < iNum2){
        return iNum1;
    }
    return iNum2;
}
```

## 2016990079.c

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

int main(int argc, char* argv[]) {
    printf("%d\n", max(1.5f, 2.5f));
    return 0;
}
```

# 编译过程实例：预编译

## MathCompare.h 不直接参与

```
#ifndef _MATHCOMPARE_H_
#define _MATHCOMPARE_H_
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);
#endif
```

## MathCompare.c

```
#include "MathCompare.h"
int Max(int iNum1, int iNum2) {
    ...
}
int Min(int iNum1, int iNum2) {
    ...
}
```

## 2016990079.c

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

int main(int argc, char* argv[]) {
    printf("%d\n", max(1.5f, 2.5f));
    return 0;
}
```

## MathCompare.l

```
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);

int Max(int iNum1, int iNum2) {
    ...
}
int Min(int iNum1, int iNum2) {
    ...
}
```

## 2016990079.l

```
int printf(const char* Format, ...);
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);

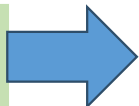
int main(int argc, char* argv[]) {
    printf("%d\n", max(1.5f, 2.5f));
    return 0;
}
```

# 编译过程实例：编译

## MathCompare.i

```
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);

int Max(int iNum1, int iNum2) {
    ...
}
int Min(int iNum1, int iNum2) {
    ...
}
```



## MathCompare.obj

```
...
call Min
...
...
...
...

```

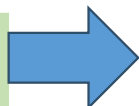
Max函数的机器码

Min函数的机器码

## 2016990079.i

```
int printf(const char* Format, ...);
int Max(int iNum1, int iNum2);
int Min(int iNum1, int iNum2);

int main(int argc, char* argv[]) {
    printf("%d\n", max(1.5f, 2.5f));
    return 0;
}
```



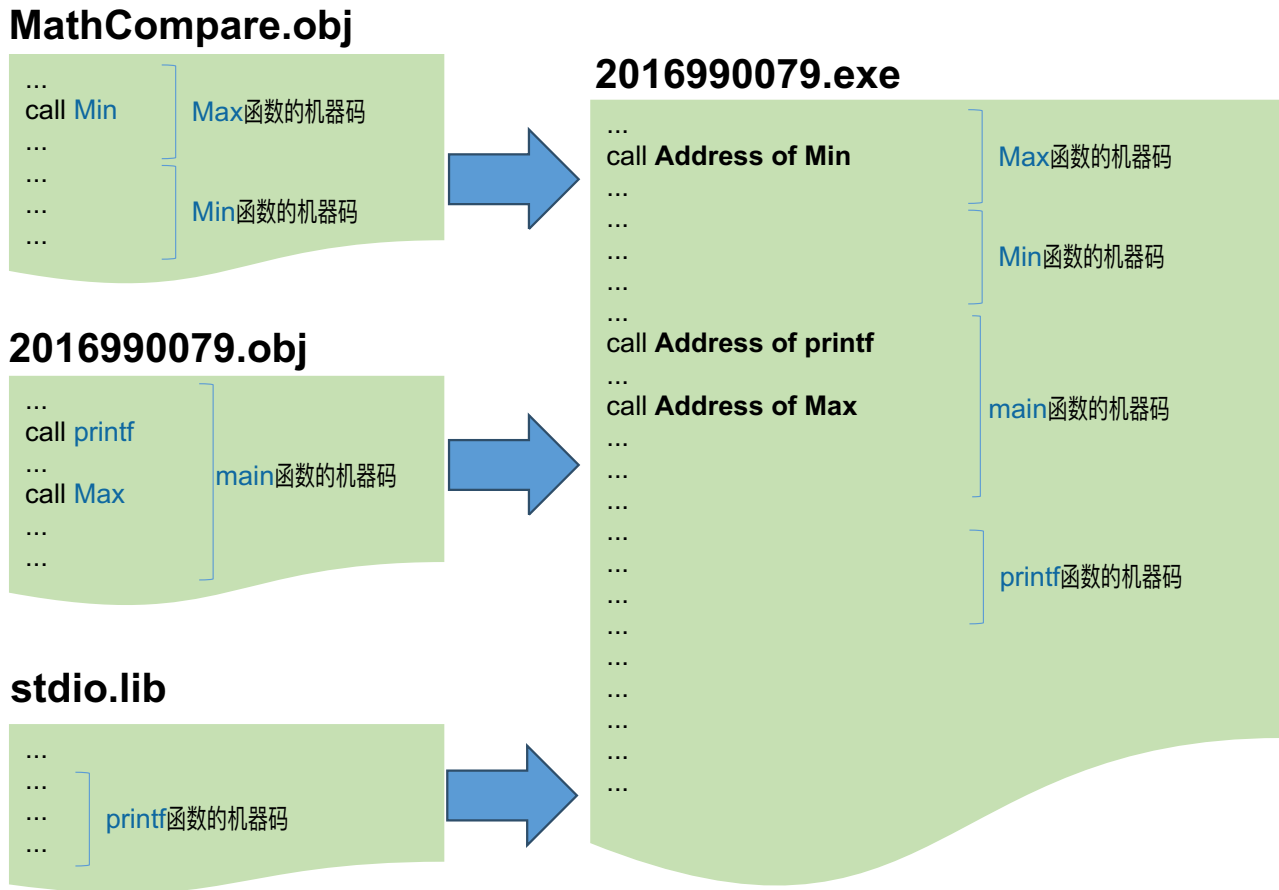
## 2016990079.obj

```
...
call printf
...
call Max
...
...

```

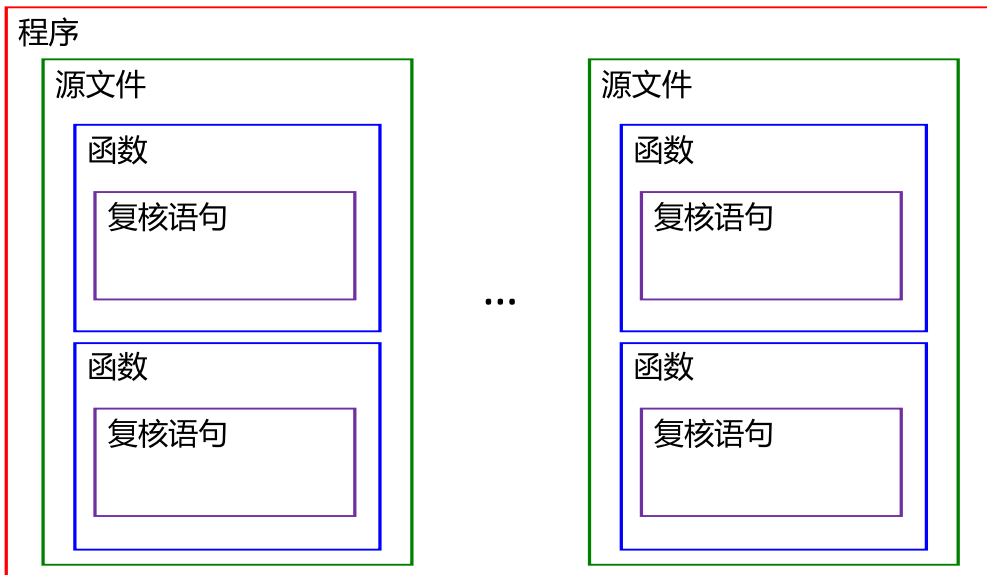
main函数的机器码

## 编译过程实例：链接



# 常量/变量的作用域

**作用域：**常量/变量有效性的代码区域，在哪里可以使用常量/变量



在函数外部定义的常量/变量，称为**全局常量/变量**

在函数内部或复合语句中定义的常量/变量，称为**局部常量/变量**

**作用域：**从**定义位置**到**区间结束**（文件/函数/复合语句的结束）



# 常量/变量的作用域

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

```
int iArray1[100];
```

iArray1作用域

```
int main(int argc, char* argv[])
```

```
{
```

main函数形参作用域

```
    for (int i = 0; i < 100; i++) {
```

```
        int j = i * 2;
```

i作用域

```
        iArray1[i] = j;
```

j作用域

```
    }
```

```
    return 0;
```

```
}
```

```
int iArray2[50];
```

```
.....
```

iArray2作用域

# 常量/变量的作用域

**重名与屏蔽：**相同作用域内标识符不能重名。不同作用域内可以重名，遵循“**小作用域屏蔽大作用域**”原则

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

```
int iMax = 3;
```

全局iMax作用域

```
int main(int argc, char* argv[])
```

```
{
```

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

```
    int iMax = 2;
```

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

```
    return 0;
```

```
}
```

局部iMax作用域

```
int GetMax() {
```

```
    return iMax;
```

```
}
```

# 作用域：补充

作用域规则不仅仅应用于变量，还包括其他实体

- ❑ 变量，包括基本类型变量、数组和指针变量以及结构体变量等
- ❑ 常量
- ❑ 函数
- ❑ 自定义数据类型，比如结构体类型、共用体类型。

作用域可以理解为程序中的一段区域

- ❑ 同一作用域，名字与实体一一对应
- ❑ 不同作用域，可以使用相同名字

作用域分类

- ❑ 文件作用域、函数作用域、块作用域、类型声明/定义作用域、函数原型作用域

# 作用域：补充

说明：

- 不同函数中可以使用相同名字的变量，它们代表不同的对象(不同栈帧)，互不干扰。
- **主函数中定义**的变量也**只在主函数中有效**，而不因为在主函数中定义而在整个文件或程序中有效。
- **主函数不能使用**其他函数中定义的变量。
- 形式参数也是该函数的局部变量。
- **C99中，可以在复合语句中定义变量，其作用域限于所在复合语句。**

# 常量/变量的生命期：静态与动态存储

从值**存在的时间**角度(生命期)来分，又分为静态存储方式和动态存储方式。

- ❓ 静态存储：程序运行就开辟（main函数执行之前），程序运行结束才销毁
- ❓ 动态存储：暂时认为函数运行才开辟，函数运行结束就销毁

# 存储类别

变量和函数可以指定**存储类别**，标识数据在内存中存储的方式(静态与动态)。

存储类别包含（考虑进度和基础，只讲了一部分）：

- ❑ 自动的(auto)
- ❑ 静态的(static)
- ❑ 外部的(extern)

# 存储类别: **auto**变量

默认情况下, 函数或复合语句中的变量(包含局部变量和形参)称为**自动对象**, 其存储方式是动态存储。

自动对象进入**函数时分配空间, 结束函数时释放空间**。

自动变量用关键字**auto**作存储类别的声明, **可以省略**。

例如:

```
int f(int a) /*定义f函数, a为形参*/  
{  
    auto int b;  
    auto int c = 3; /*定义b、c为自动变量*/  
}
```

**定义位置 优先于 存储类别关键字, 决定生存周期**

# 储类别：局部static变量

当函数中的局部变量的值在函数调用结束后不消失而保留原值时，在**局部对象**的前面加上**static**存储类别修饰，用来指明对象是**静态局部对象**

静态局部对象与全局对象一样是按**静态存储处理**的，即它的生命期与程序运行期相同，所以静态局部对象可以将其值一致**保持到程序结束**，或者下次修改时。

## 说明：

- 当一个函数会多次调用又希望将它的某些对象的值保持住时，就应该使用局部static对象
- 局部static对象属于持久占有存储空间，所以谨慎和适度使用。



# 存储类别：局部static变量

例如

```
#include <stdio.h>

int CounterInc() {
    static int Count = 0; //初始化只执行一次
    Count++;
    return Count;
}

int main(int argc, char* argv[]) {
    for (int i = 0; i < 10; i++) {
        printf("%d\n", CounterInc());
    }
    return 0;
}
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

# 存储类别：局部static变量 vs. auto变量

生命期不同：

- ❓ 局部static变量，有程序生存期。
- ❓ auto变量，栈帧，函数调用开辟，调用结束后即释放。

初始化：

- ❓ 局部static变量，在编译时赋初值的，即只初始化一次。尽管初始化代码写在函数内，但每次函数调用时都不会初始化，而是保留上次函数调用结束时的值。
- ❓ auto变量，每次函数调用时才初始化

# 变量存储类别: 局部static变量 vs. 全局变量

## 作用域不同

- ❓ **局部static变量**，不可在其他函数中使用。作用域是函数作用域或者块作用域，即它只能在局部区域内使用。
- ❓ **全局变量**，作用域是定义点至文件结果

## static修饰词意义不同

- ❓ **局部static变量**来说，**static**改变的是生命期，不改变作用域。它使变量获得与程序相同的生命期，但作用域依然是函数内。
- ❓ **对全局对象来说** **static**改变作用域，不改变生命期。**static**是私有的意思，将全局对象的作用域限定在所处的源文件中（其他文件不可见）。

**定义位置 优先于 存储类别关键字，决定生存周期**

# 变量存储类别:extern变量修饰

有时要用**扩展变量的作用域**（文件内或文件外），可以采用extern来**声明（声明、声明、声明，重要的事情说三遍）**外部变量

外部变量的“外部”是相对的，是别的文件（或作用域）的全局变量，它的**作用域是从变量的extern声明处开始，到本程序文件的末尾。**

在此作用域内，全局变量可以为程序中各个函数所引用。

编译时将外部变量分配在**静态存储区**。

# 存储类别: extern扩展作用域

## 情形1：同一文件内的作用域扩展

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    //声明（不是定义哦）使用一个外部的（非当前语句所处的作用域）整型变量A
    //将外部的变量A的作用域扩展到当前作用域
    extern int A;
    //声明（不是定义哦）使用一个外部的（非当前语句所处的作用域）整型变量A
    //将外部的变量A的作用域扩展到当前作用域
    extern int B; /*外部变量声明*/
    //声明（不是定义哦）使用一个外部的（非当前语句所处的作用域）函数int max(int)
    //将外部的函数int max(int)的作用域扩展到当前作用域
    extern int max(int x, int y);
    //之后，外部变量/函数已满足：先声明后调用的要求
    printf("%d\n", max(A, B));
}
int A = 13;
int B = -8;
int max(int x, int y) {
    return x > y ? x : y;
}
```

# 存储类别: extern扩展作用域

## 情形2：跨文件作用于扩展

file1.c

```
#include <stdio.h>
int A = 3; //这里是定义
int main(int argc, char* argv[]) {
    printf("%d\n", A);
    return 0;
}
```

file2.c

```
//仅仅是声明将使用一个外部的int A, 将A的作用域扩展到当前文件
extern int A;
void PrintA() {
    printf("%d\n", A);
}
```

# 存储类别: extern 扩展作用域

特别地, **static**全局对象, 不可在当前文件的其他位置, 或其他文件中被声明为 **extern**。因为**static**意味着“私有”(作用域不可被扩展)

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4     extern int A;
5     printf("%d\n", A);
6     return 0;
7 }
```

```
8 static int A = 3;
```



Static declaration of 'A' follows non-static declaration

# 存储类别: **extern**扩展作用域

## 关于变量的声明和定义

**[?] 定义性声明**: 需要建立存储空间的声明。

如: `int a;`

**[?] 引用性声明**: 不需建立存储空间的声明。

如: `extern int a;`

注意:

**[?] 声明包括定义**, 但并非所有的声明都是定义。

**[?] 对“`int a;`”而言**, 它既是声明, 又是定义; 而对“`extern int a;`”而言, 它是声明而不是定义。



# 存储类别：总结

## 从作用域角度分



### 局部变量

- **auto**变量 （离开函数，值就消失）
- 局部**static**变量（离开函数，值仍保持）
- （形式参数可以定义为自动变量或寄存器变量）



### 全局变量

- **static**全局变量 （作用域不可扩展）
- 非**static**全局变量（作用域可扩展）
- **extern**全局变量 （从其他文件扩展到当前作用域的非**static**全局变量）

# 函数与变量规则

变量的作用域和生存期，描述了程序运行期间数据的运作和管理

## 作用域

❓ 空间概念，刻画变量的“可见”性

## 生存期

❓ 时间概念，刻画变量的“存在”性

# 函数与变量规则：单文件单函数(main)

## 作用域规则

- ❑ 变量先定义后使用
- ❑ 变量定义不能同名
- ❑ 变量可以在复合语句及嵌套中定义
- ❑ 变量在复合语句及嵌套中定义允许同名

最后两条限于C99  
及后续标准

## 初始化规则

- ❑ auto变量使用前应该初始化为指定值，否则为随机值

# 函数与变量规则：单文件多函数

## 包含main函数与自定义函数的程序

### 作用域规则：变量分局部变量和全局变量

#### [?] 局部变量

- auto变量，即动态局部变量(多数情况下使用)
- 局部static变量(函数多次调用仍保持数据值情况下使用)
- 形式参数(函数间数据传递时使用)
- 寄存器(register)变量(已有编译器优化工具，极少使用)

#### [?] 全局变量

- 函数间数据传递，不用或少用

# 函数与变量规则：单文件多函数

生命期规则：变量分动态存储和静态存储



动态存储

- auto变量(auto进入函数分配，函数退出释放)
- 形式参数(进入函数分配，函数退出释放)



静态存储

- 局部static变量
- 全局变量

# 函数与变量规则: 单文件多函数

初始化规则:变量初始化与存储方式有关



静态存储

- 设定值(已初始化的全局变量、局部static变量, 运行前一次设置)
- 0(未初始化的全局变量、局部static变量, 运行前一次设置)



动态存储

- 设定值(已初始化的局部非static变量, 函数调用每次重新设置)
- 随机值(未初始化的局部非static变量)

# 函数与变量规则：多文件多函数

编译器是按文件为单位编译的，现今的编译器都有增量编译的功能，即当编译器发现某个源文件未曾改动，那么就不重新编译它，以节省编译时间

程序较大时，为了提高编译效率要使用多文件的工程模式

变量的应用情况越来越复杂

# 函数与变量规则：多文件多函数

## 作用域规则

- ❓ 变量和函数公有使用(允许多个文件中使用)
  - 全局变量(在需要使用的文件中extern声明)
  - 函数(在需要使用的文件中extern声明)
  
- ❓ 变量和函数私有使用(作用域规则，只限一个文件中使用)
  - 全局变量(在需要限定的变量定义中static声明)
  - 函数(在需要限定的函数定义中static声明)

### 注意：

const限定声明对象是只读的，从而保护对象不会意外修改。  
这一方法也适用于单文件多函数的情况



# 函数与变量规则：多文件多函数

## 实体可见(可见规则)

❓ 文件、函数、复合语句、嵌套复合语句区域逐级包含

❓ 局部实体

- 包含关系中子区域在父区域不可见
- 包含关系中父区域在子区域同名不可见、不同名可见
- 同一个父区域的平行区域互不可见

❓ 全局实体

- 使用extern声明在别的文件可见
- 使用static声明仅限本文件可见

# 函数与变量规则：多文件多函数

## unit1.c

#include <必要的系统头文件>

#include "unit1.h"

#include 必要的其他头文件

可被其他文件访问的全局变量定义（对别的文件来说，是外部的）

static 不可被其他文件访问的全局变量

static 不可被其他文件调用的函数声明

所有函数定义

## unit1.h

#ifndef \_UNIT1\_H\_

#define \_UNIT1\_H\_

#include 必要的其文件

extern 可被其他文件访问的全局变量声明

可被其他文件调用的函数声明

#endif

# 数组作为函数参数：数组元素参数

## 数组元素

可以当作变量或者组成表达式，作为函数实参。与基本类型的变量作实参一样，是**单向传递，即“值传送”方式**。

例，假设a、b各有10个元素，按如下规则比较两个数组a和b：

将它们对应地逐个相比(即 $a[i]$ 跟与 $b[i]$ )。

如果a数组中的元素大于b数组中的相应元素的数目多于b数组中元素大于a数组中相应元素的数目(例如， $a[i] > b[i]$  6次， $a[i] < b[i]$  3次)，则认为a数组大于b数组；

如果a数组中的元素大于b数组中的相应元素的数目少于b数组中元素大于a数组中相应元素的数目，则认为a数组小于b数组；

否则，a数组等于b数组。

```

#include <stdio.h>
int cmp(int x, int y); /* 函数声明 */
int main(void) {
    int a[10];
    int b[10];
    int i;
    int n_Larger = 0;
    int nEqual = 0;
    int n_Smaller = 0;

    for(i = 0; i < 10; i++) {
        scanf("%d", &a[i]);
    }
    for(i = 0; i < 10; i++) {
        scanf("%d", &b[i]);
    }

    for(i = 0; i < 10; i++)
    {
        if(cmp(a[i], b[i]) == 1) {
            n_Larger++;
        }

        else if(cmp(a[i], b[i]) == 0) {
            nEqual++;
        }
        else {
            n_Smaller++;
        }
    }

    if(n_Larger > n_Smaller){
        printf("array a is larger b\n");
    }
    else if (n_Larger < n_Smaller) {

```

```

int cmp(int x, int y) {
    int flag;
    if (x > y) {
        flag = 1;
    }
    else if( x < y) {
        flag = -1;
    }
    else {
        flag = 0;
    }
    return flag;
}

```

# 数组作为函数参数：数组元素参数

```
#include <stdio.h>
void swap(int a,int b) /*函数声明*/
int main(void)
{

    int i, j, a[10];
    for (i = 0; i < 10; i++){
        scanf("%d,", &a[i]);
    }
    for (j = 0; j < 9; j++) {
        for (i = 0; i < 9 - j; i++) {
            if (a[i] > a[i + 1]) {
                swap(a[i],a[i+1]);
            }
        }
    }
    for (i = 0; i < 10; i++)
        printf("%d,", a[i]);
    printf("\n");
    return 0;
}
```

```
/* 函数定义*/
void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

**Failed !!!**

# 数组作为函数参数：数组名参数

用数组名作函数实参时，形参应当用数组名或指针变量。

例，有一个一维数组score，内放10门课的学生成绩，求平均成绩。

```
#include <stdio.h>
float average(float array[10]); /*函数声明*/
int main(void)
{
    float score[10];
    float aver;
    int i;
    printf("input 10 scores:\n");
    for (i = 0; i < 10; i++) {
        scanf("%f", &score[i]);
    }
    printf("\n");
    aver = average(score);
    printf("average score is %5.2f\n", aver);
}
```

```
/* 函数定义*/
float average(float array[10])
{
    float aver;
    float sum = 0;
    int i;
    for (i = 0; i < 10; i++) {
        sum += array[i];
    }
    aver = sum / 10.0;
    return aver;
}
```

# 数组作为函数参数：数组名参数



用数组名作函数实参时，地址发生了值传递，实际上形参和实参存储了相同的值（地址）。因此通过形参改变数组元素时，实参对应的数组元素也随之变化。

```
#include <stdio.h>

void sort(int arr[10]); /*函数声明*/

int main(void)
{
    //...
    int i;
    int a[10];
    sort(a);
    for (i = 0; i < 10; i++) {
        printf("%d,", a[i]);
    }
    printf("\n");
    return 0;
}
```

```
/* 函数定义*/
void sort(int arr[10])
{
    int i;
    int j;
    int temp;
    for (j = 0; j < 9; j++) {
        for (i = 0; i < 9 - j; i++) {
            if (arr[i] > arr[i + 1]) {
                temp = arr[i + 1];
                arr[i + 1] = arr[i];
                arr[i] = temp;
            }
        }
    }
}
```

**Succeed !!!**

# 数组作为函数参数：数组长度传递

数组名作函数实参时，形参数组不定义和检查长度

```
#include <stdio.h>
float average(float array[10]); /*函数声明*/
int main(void)
{
    float score[5];
    float aver;
    int i;
    printf("input 10 scores:\n");
    for (i = 0; i < 10; i++) {
        scanf("%f", &score[i]);
    }
    printf("\n");
    aver = average(score);
    printf("average score is %5.2f\n", aver);
}
```

```
/* 函数定义*/
float average(float array[10])
{
    float aver;
    float sum = 0;
    int i;
    for (i = 0; i < 10; i++) {
        sum += array[10];
    }
    aver = sum / 10.0;
    return aver;
}
```

编译时正确；  
调用时数组访问越界



# 数组作为函数参数：数组长度传递

数组长度作为参数传递，有效避免数组访问越界

数组长度作为参数传递，增加了程序的通用性

```
/*函数定义*/  
void sort(int arr[], int len)  
{  
    int i;  
    int j;  
    int temp;  
    for (j = 0; j < len - 1; j++) {  
        for (i = 0; i < len - j - 1; i++) {  
            if (arr[i] > arr[i + 1]) {  
                temp = arr[i + 1];  
                arr[i + 1] = arr[i];  
                arr[i] = temp;  
            }  
        }  
    }  
}
```

⇒ 更具通用性的代码！

# 数组作为函数参数：数组名参数

## 注意：

- 实参数组与形参**数组类型应一致**，如不一致，结果出错；
- 在**函数定义中声明形参数组的大小是不起任何作用的**；
- **形参数组可以不指定大小**，有时为了在被调用函数中处理数组元素个数的需要，**可以另外设一个形参**；
- 用数组名作为函数实参时，不是把数组元素的值传递给形参，**而是（数组首地址）作为实参进行数值传递，两个数组就共占同一段内存空间，在被调函数中访问（读或写）形参数组元素，是通过地址访问数值。**

# 数组作为函数参数：多维数组参数

用多维数组名作为函数实参和形参，参数声明中必须指明数组的列数

- ❓ 在被调函数中对形参数组定义时可以指定每一维的大小
- ❓ 也可以省略最高维的大小，但不能省略最高维的[]
- ❓ 在维度相同的前提下，最高维大小可以与实参数组不同

例，求3x4矩阵所有元素中的最大值。

```
int max_value(int array[][4]);
int main(void) {
    int a[3][4] = {
        {1,3,5,7},
        {2,4,6,8},
        {1, 4, 9, 16}
    };
    printf("max value is %d\n", max_value(a));
}
```

```
int max_value(int array[][4]) {
    int max = array[0][0];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            if(array[i][j] > max) {
                max= array[i][j];
            }
        }
    }
    return max;
}
```