step by step

进阶

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与联系（模块设计方法、函数）

表达与转换（基本操作、数据类型）

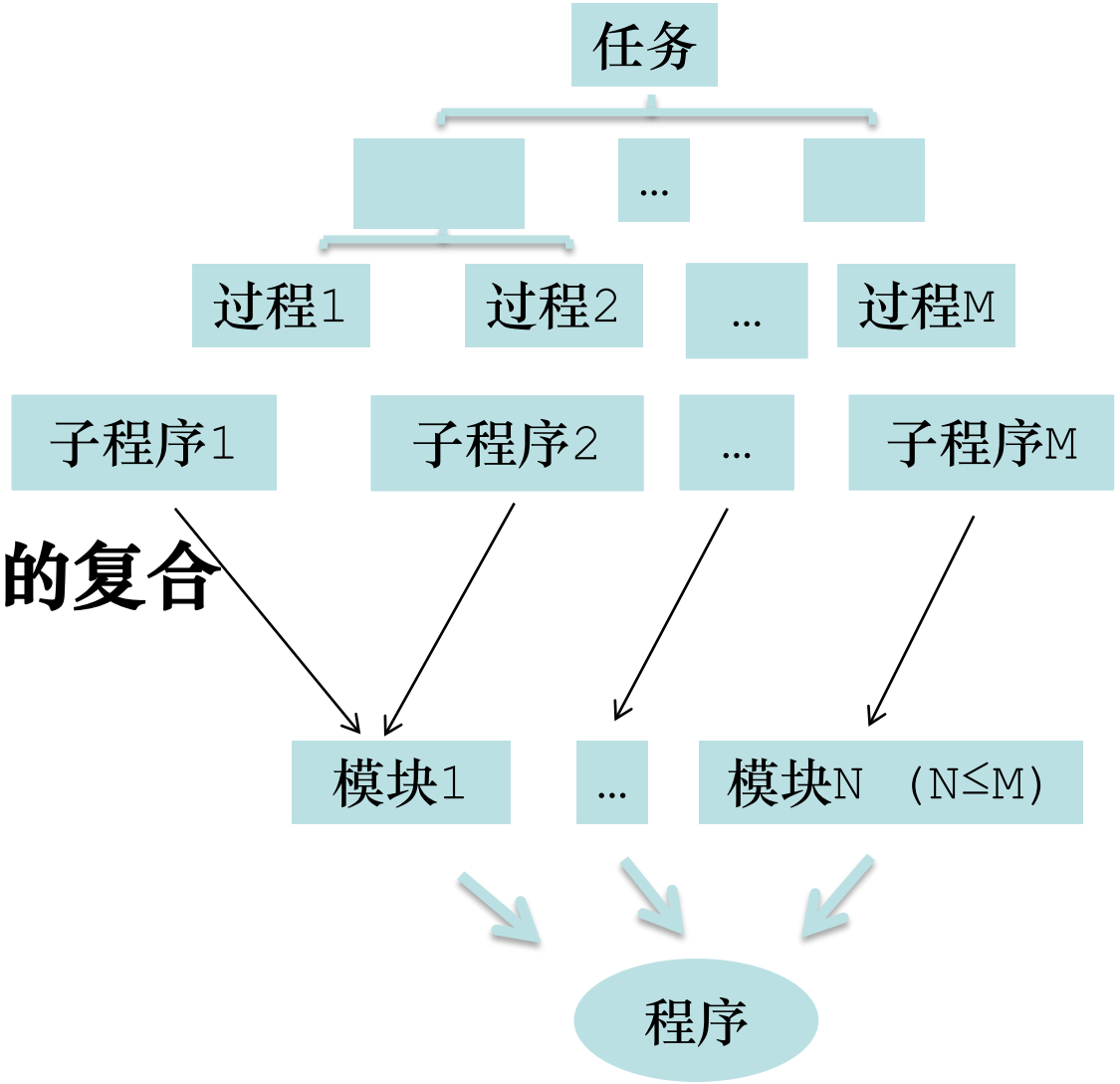
提高：

构造与访问（数组、指针、结构）

归纳与推广（程序设计的本质）

模块设计

- **过程（procedure）的分解**



- **程序的复合**

从特定的任务实例（3^2+4^2、34^2+65^2）中抽出一般化的功能特征（两个整数的平方和）

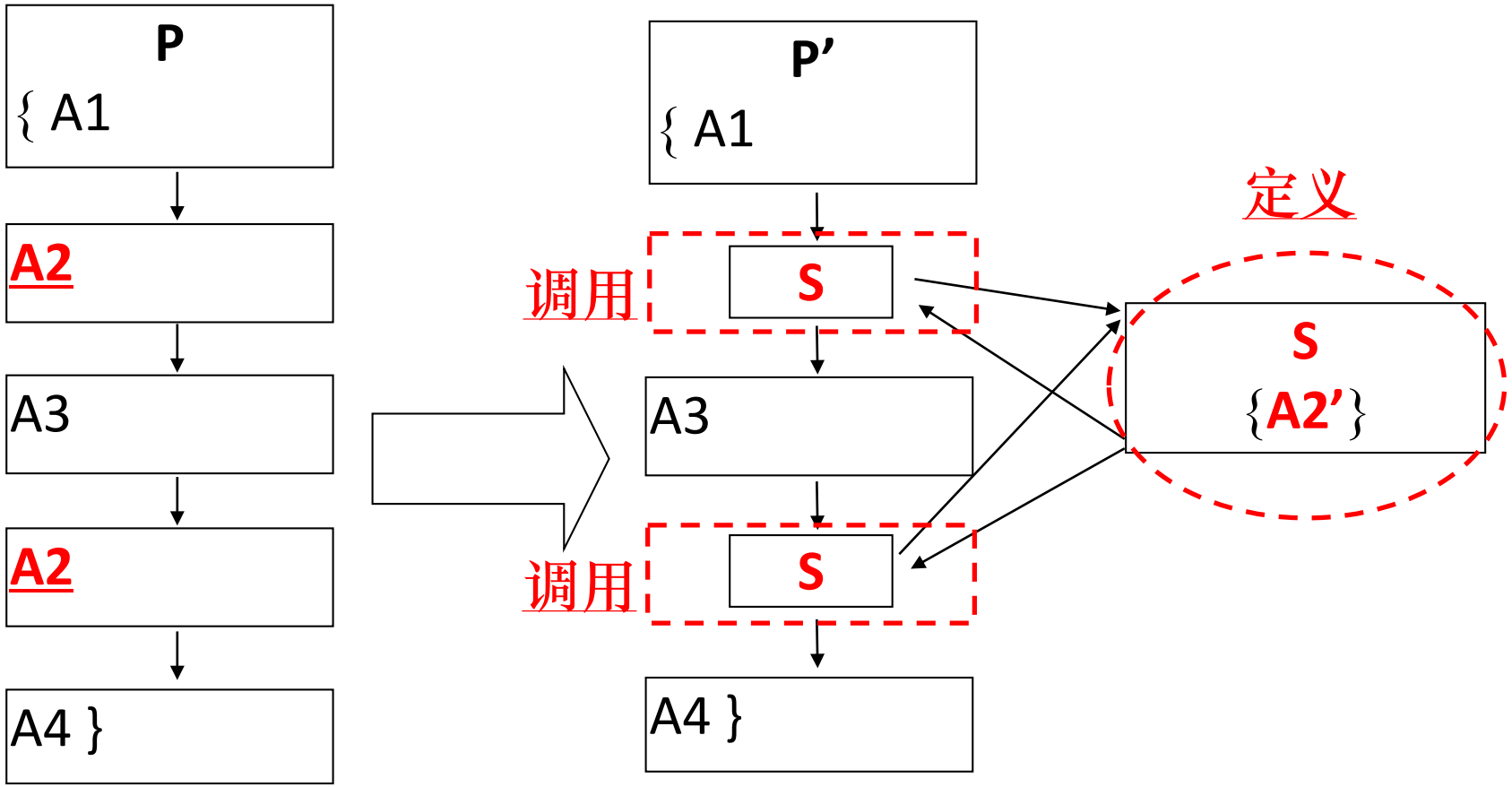
- 过程抽象
- 子程序（subprogram）：封装了一系列操作

发挥头文件的作用，做好每个模块的接口

- 合理安排
- 调用（call）机制：将分布在一个或多个模块（module）中的子程序关联成一个整体

子程序

- 是取了名字的一段程序代码，可以实现一个相对独立的功能



例如：

```
int main( )
{
    int a, b, i, j, c, k;
    scanf("%d%d%d%d", &a, &b, &i, &j);
    c = a*a + b*b;
    k = i*i + j*j;
    printf(…;
    return 0;
}
```

定义

```
int SumSq(int x, int y)
{
    int z;
    z = x*x + y*y;
    return z;
}
```

```
int main( )
{
    int a, b, i, j, c, k;
    scanf("%d%d%d%d", &a, &b, &i, &j);
    c = SumSq(a, b);    调用
    k = SumSq(i, j);    调用
    printf(…;
    return 0;
}
```

C语言子程序——函数（function）

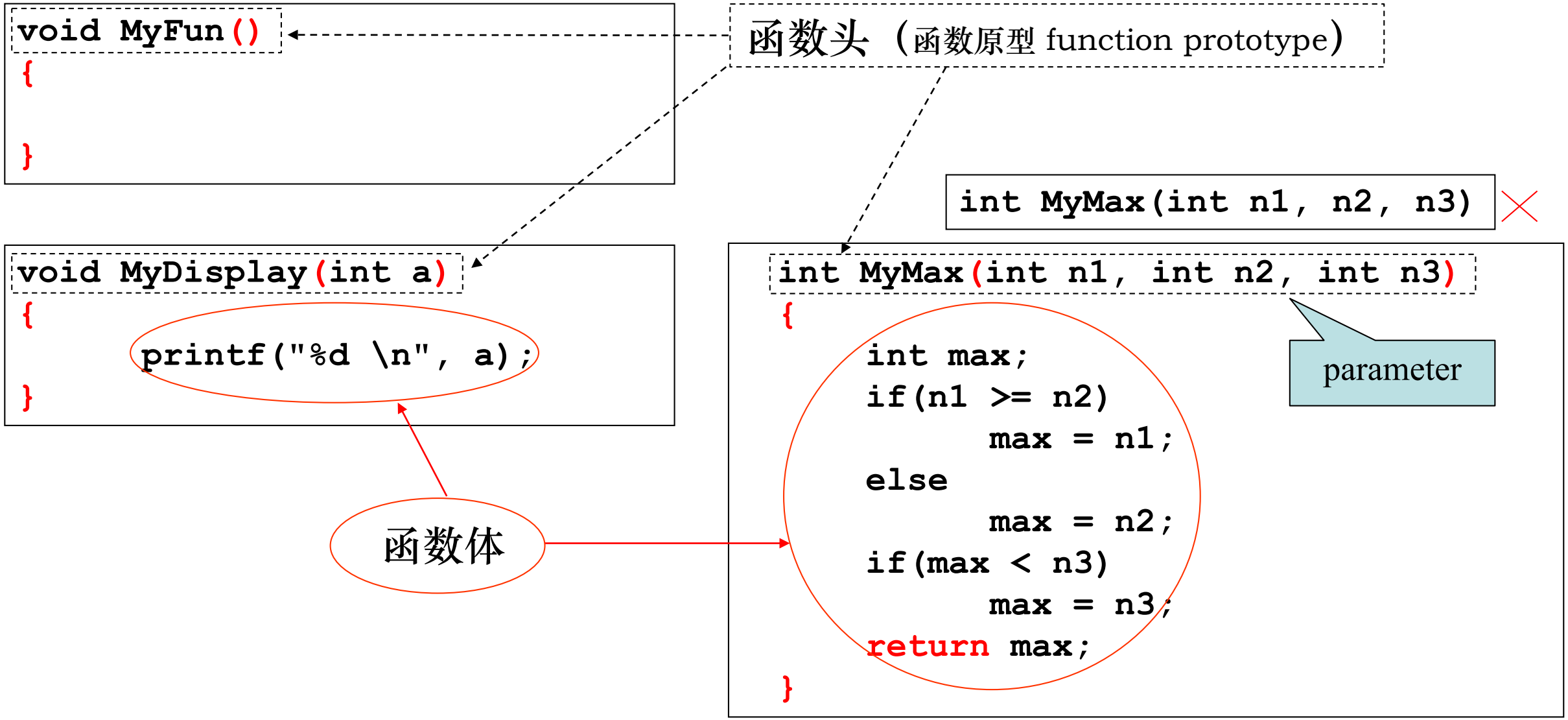
- 17世纪末，在德国的数学家*莱布尼兹*的文章中，首先使用了“function”一词，不过，它表示“幂”、“坐标”、“切线长”等概念。

- 19世纪中期，法国数学家*黎曼*吸收了莱布尼茨及后来*达朗贝尔*和*欧拉*的成果，第一次准确地提出了function的定义：某一个量依赖于另一个量，使后一个量变化时，前一个量也随着变化。

- 清.李善兰：“凡此变数中函（包含）彼变数者，则此为彼之函数”。
  - 注：函（象形。今隶误作函。本义:舌[tongue]。后指盛物的匣子、套子 [case]。）

```
f(x, y, z)

average = (n1+n2+n3)/3
average = f(n1, n2, n3)
          y(n1, n2, n3)

MyFunction(n1, n2, n3)
{average = (n1+n2+n3)/3}
```

函数的定义(definition)



return语句

- C语言中的**return**语句用来**结束**函数的执行，将流程转回主调函数，还可以顺便返回一个返回值。
  - 一个return语句最多只能返回一个值。
  - 如果没有return语句，则函数体的右花括号作为函数执行结束的标志。

```
void MyDisplay(int a)
{
      printf("%d \n", a);
}
```

```
void MyDisplay(int a)
{
      printf("%d \n", a);
      return;
}
```

```
int MyMax(int n1, int n2, int n3)
{
      int max;
      if(n1 >= n2)
            max = n1;
      ……
      return max;
}
```

- **main函数中的return语句用来结束整个程序的执行。**

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

- 一个函数中有多个return语句时，执行到哪一个return语句就从哪儿返回

```
int Min(int a, int b)
{
    int temp ;
    if(a < b)
        temp = a;
    else
        temp = b;
    return temp ;
}
```

```
int Min(int a, int b)
{
    if(a < b)
        return a;
    else
        return b;
}
```

函数定义时的注意事项

```
int main( )
{       int n1, n2, n3;
        printf("Please input three integers: \n");
        scanf("%d%d%d", &n1, &n2, &n3);

        int MyMax(int n1, int n2, int n3)
        {       int max;
                if(n1 >= n2)
                        max = n1;
                else
                        max = n2;
                if(max < n3)
                        max = n3;
                return max;
        } //应把函数 MyMax 的定义写在 main 函数的外面

        printf("The max. is: %d \n", MyMax);
        return 0;
}
```

C程序中的函数体里不能再定义函数

```
int MyMax(int n1, int n2, int n3)
{
	int max;
	if(n1 >= n2)
		max = n1;
	else
		max = n2;
	if(max < n3)
		max = n3;
	return max;
}
int main( )
{
	int n1, n2, n3;
	printf("Please input three integers: \n");
	scanf("%d%d%d", &n1, &n2, &n3);
	int max = MyMax(n1, n2, n3);
	printf("The max. is: %d \n", max);
	return 0;
}
```

定义应该各自独立

函数的调用(call)

```
void MyFun()
{

}
```

```
void MyDisplay(int a)
{
    printf("%d \n", a);
}
```

```
int main( )
{
    MyFun();
    MyDisplay( 7 );
    return 0;
}
```

argument

```
int x = MyFun();
printf("%d", MyFun());
```

```
int MyMax(int n1, int n2, int n3)
{
    int max;
    if(n1 >= n2)
        max = n1;
    else
        max = n2;
    if(max < n3)
        max = n3;
    return max;
}
```

```
int main( )
{
    int i = 3, j = 4, k = 5;
    int m = MyMax(i, j, k);
    printf("%d ", m);
    return 0;
}
```

```
int m = int MyMax(int i, int j, int k);
```

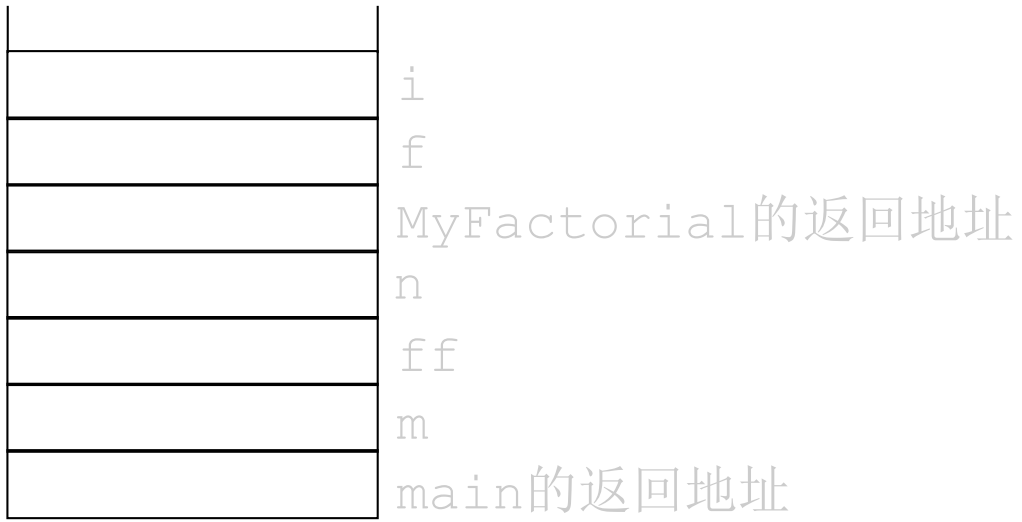
```
int main( )
{
    int i = 3, j = 4, k = 5;
    printf("%d", MyMax(i, j, k));
    return 0;
}
```

例2.1 设计 C 程序，用函数实现 求阶乘问题。

- 分析：求阶乘作为一个独立的功能，用函数实现时，函数的参数为一个整数，函数的返回值为该整数的阶乘。

```
int main( )
{
        int m;
        printf("Input an integer: \n");
        scanf("%d", &m);
        if(m < 0) return -1; //结束整个程序
        int ff = MyFactorial(m);
        printf("Factorial is: %d \n", ff);
        return 0;
}
```

```
int MyFactorial(int n)
{
        int f = 1;
        for(int i = 2; i <= n; ++i)
                f *= i;
        return f;
}
```



例2.2 求输入三个整数中最大值的阶乘，并输出。

```
int main( )
{
      int n1, n2, n3, max, f;
      printf("Input three integers: \n");
      scanf("%d%d%d", &n1, &n2, &n3);
      max = MyMax(n1, n2, n3);
      f = MyFactorial(max);
      printf("Factorial of max. is
      return 0;
}
```

```
int MyFactorial(int n)
{       int f = 1;
        for(int i = 2; i <= n; ++i)
                f *= i;
        return f;
}
```

```
int MyMax(int n1, int n2, int n3)
{       int max;
        if(n1 >= n2)
                max = n1;
        else
                max = n2;
        if(max < n3)
                max = n3;
        return max;
}
```

函数调用结果的不同“身份”（作为赋值操作右操作数或实参）

```
f = MyFactorial(MyMax(n1, n2, n3));
printf("Factorial of max. is: %d \n", f);
```

或

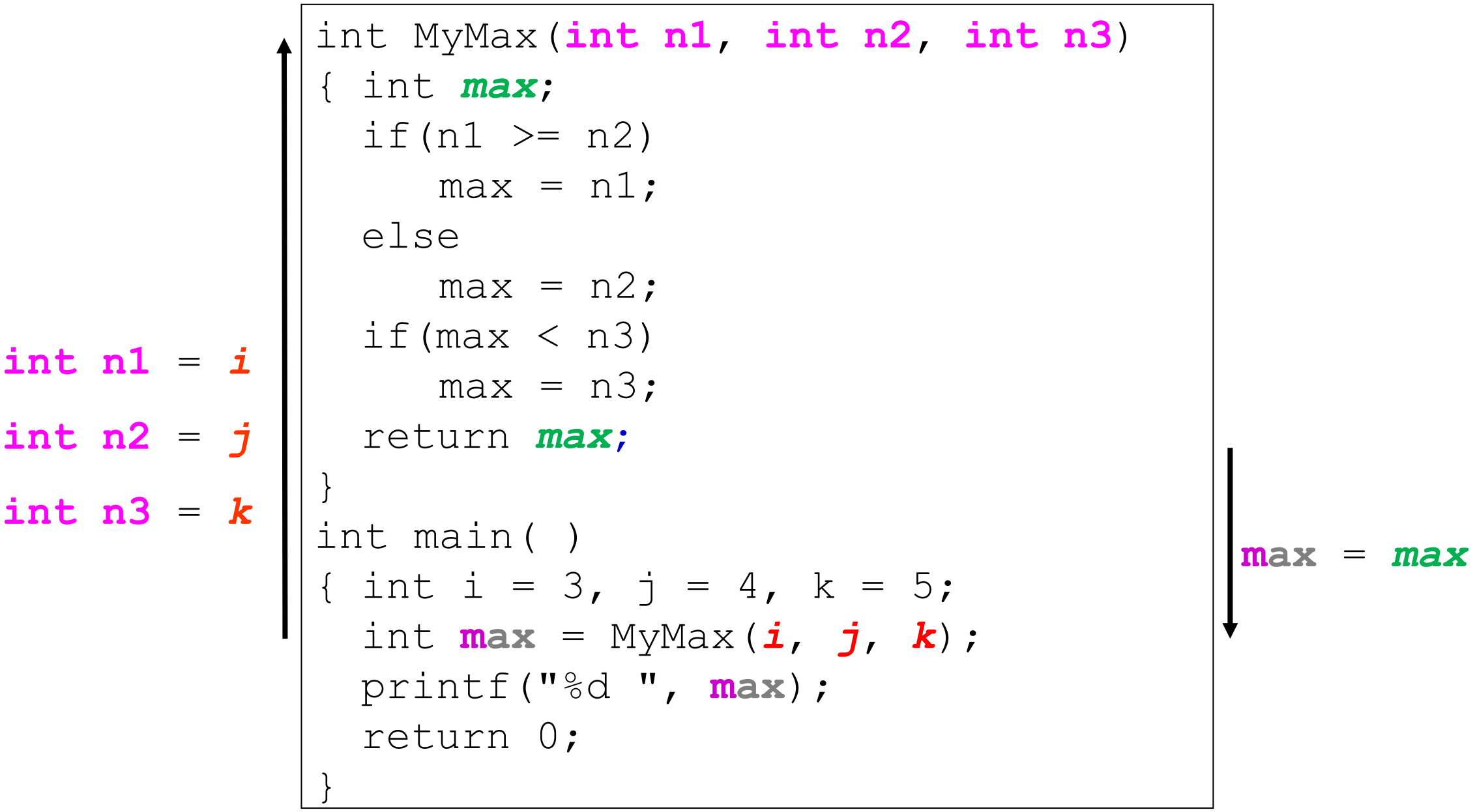
```
max = MyMax(n1, n2, n3);
printf("Factorial of max. is: %d \n", MyFactorial(max));
```

或

```
printf("…is: %d \n", MyFactorial(MyMax(n1, n2, n3)));
```

在上述不同的函数调用形式中，main函数都是先调用 MyMax 函数，
在 MyMax 函数执行结束后，再调用 MyFactorial 函数的。

函数间的通讯方式 I


int n1 = i
int n2 = j
int n3 = k
int MyMax(int n1, int n2, int n3)
{ int max;
if(n1 >= n2)
max = n1;
else
max = n2;
if(max < n3)
max = n3;
return max;
}
int main( )
{ int i = 3, j = 4, k = 5;
int max = MyMax(i, j, k);
printf("%d ", max);
return 0;
}
max = max

函数的声明(declaration)及其好处

```
int MyMax(int n1, int n2, int n3); //函数的声明

int main( )
{       int n1, n2, n3;
        printf("Please input three integers: \n");
        scanf("%d%d%d", &n1, &n2, &n3);
        int max = MyMax(n1, n2, n3);  //函数的调用
        printf("The max. is: %d \n", max);
        return 0;
}
int MyMax(int n1, int n2, int n3) //函数的定义
{       int max;
        if(n1 >= n2)
                max = n1;
        else
                max = n2;
        if(max < n3)
                max = n3;
        return max;
}
```

全局变量

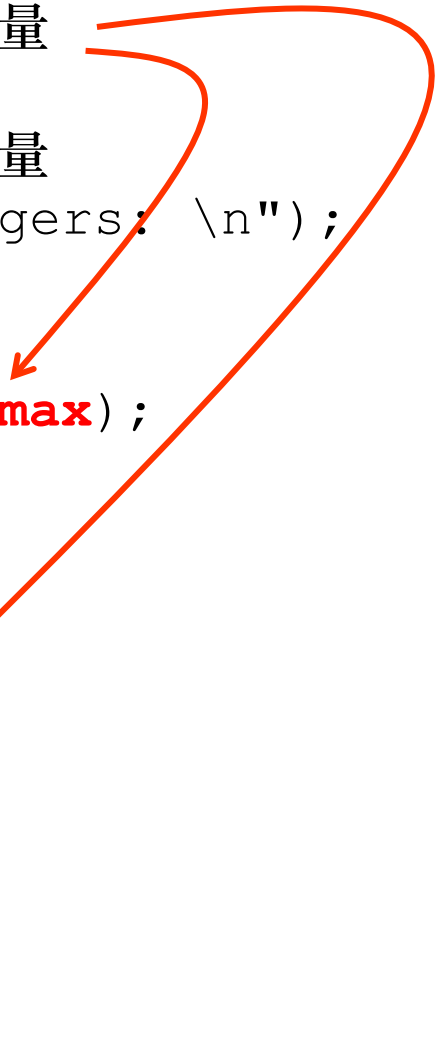
```
void MyMax(int n1, int n2, int n3);
int g_max = 0;                    //全局变量
int main( )
{     int n1, n2, n3;             //局部变量
      printf("Please input three integers: \n");
      scanf("%d%d%d", &n1, &n2, &n3);
      MyMax(n1, n2, n3);
      printf("The max. is: %d \n", g_max);
      return 0;
}
void MyMax(int n1, int n2, int n3)
{
      if(n1 >= n2)
            g_max = n1;
      else
            g_max = n2;
      if(g_max < n3)
            g_max = n3;
}
```

全局变量的声明

```
void MyMax(int n1, int n2, int n3);
extern int g_max;                          //全局变量的声明
int main( )
{       int n1, n2, n3;
        printf("Please input three integers: \n");
        scanf("%d%d%d", &n1, &n2, &n3);
        MyMax(n1, n2, n3);
        printf("The max. is: %d \n", g_max);
        return 0;
}
int g_max = 0;                             //全局变量的定义
void MyMax(int n1, int n2, int n3)
{
        if(n1 >= n2)
                g_max = n1;
        else
                g_max = n2;
        if(g_max < n3)
                g_max = n3;
}
```

函数间的通讯方式 II

```
void MyMax(int n1, int n2, int n3);
int g_max = 0;                          //全局变量
int main( )
{       int n1, n2, n3;                 //局部变量
        printf("Please input three integers: \n");
        scanf("%d%d%d", &n1, &n2, &n3);
        MyMax(n1, n2, n3);
        printf("The max. is: %d \n", g_max);
        return 0;
}
void MyMax(int n1, int n2, int n3)
{
        if(n1 >= n2)
                g_max = n1;
        else
                g_max = n2;
        if(g_max < n3)
                g_max = n3;
}
```

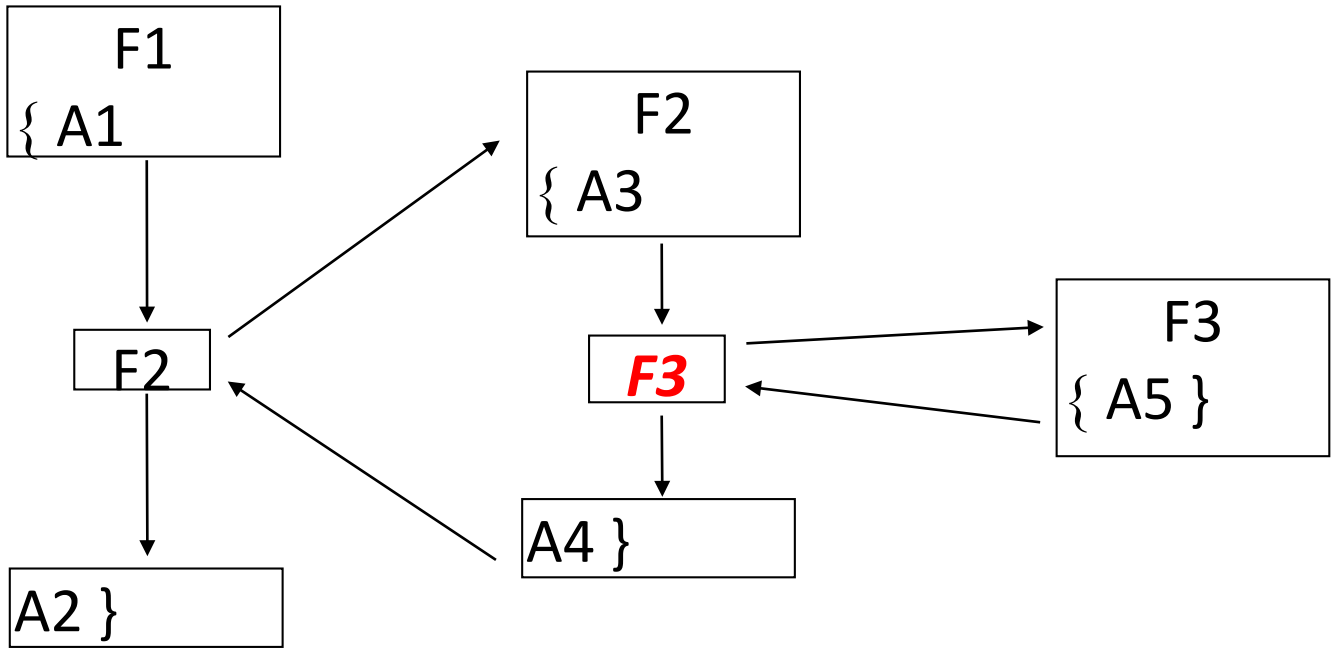


函数的副作用

```
void MyMax(int, int, int);      //形参名在函数的声明中可以省略
int g_max;                      //默认值为0
int main( )
{       int n1, n2, n3;
        printf("Please input three integers: \n");
        scanf("%d%d%d", &n1, &n2, &n3);
        MyMax(n1, n2, n3);
        printf("The max. is: %d \n", g_max);
        return 0;
}
void MyMax(int n1, int n2, int n3)
{
        if(n1 >= n2)
                g_max = n1;
        else
                g_max = n2;
        if(g_max < n3)
                g_max = n3;
}
```

函数的*嵌套*调用

- 即被调函数的定义中含有函数的调用操作



被调函数 F2 中含有函数 F3 的调用操作

不含嵌套调用

- **例2.2 求输入三个整数中最大值的阶乘，并输出。**

```
int main( )
{
      int n1, n2, n3, max, f;
      printf("Input three integers: \n");
      scanf("%d%d%d", &n1, &n2, &n3);
      max = MyMax(n1, n2, n3);
      f = MyFactorial(max);
      printf("Factorial of max. is
      return 0;
}
```

```
int MyFactorial(int n)
{       int f = 1;
        for(int i = 2; i <= n; ++i)
                f *= i;
        return f;
}
```

```
int MyMax(int n1, int n2, int n3)
{       int max;
        if(n1 >= n2)
                max = n1;
        else
                max = n2;
        if(max < n3)
                max = n3;
        return max;
}
```

含嵌套调用

- 例2.2’求输入三个整数中最大值的阶乘，并输出。

```
int main( )
{
      int n1, n2, n3, f;
      printf("Input three integers: \n");
      scanf("%d%d%d", &n1, &n2, &n3);
      f = MyFactorialNew(n1, n2, n3);
      printf("Factorial of max. is
      return 0;
}
```

```
int MyFactorialNew(int n1, int n2, int n3)
{     int n = MyMax(n1, n2, n3);
      int f = 1;
      for(int i = 2; i <= n; ++i)
            f *= i;
      return f;
}
```

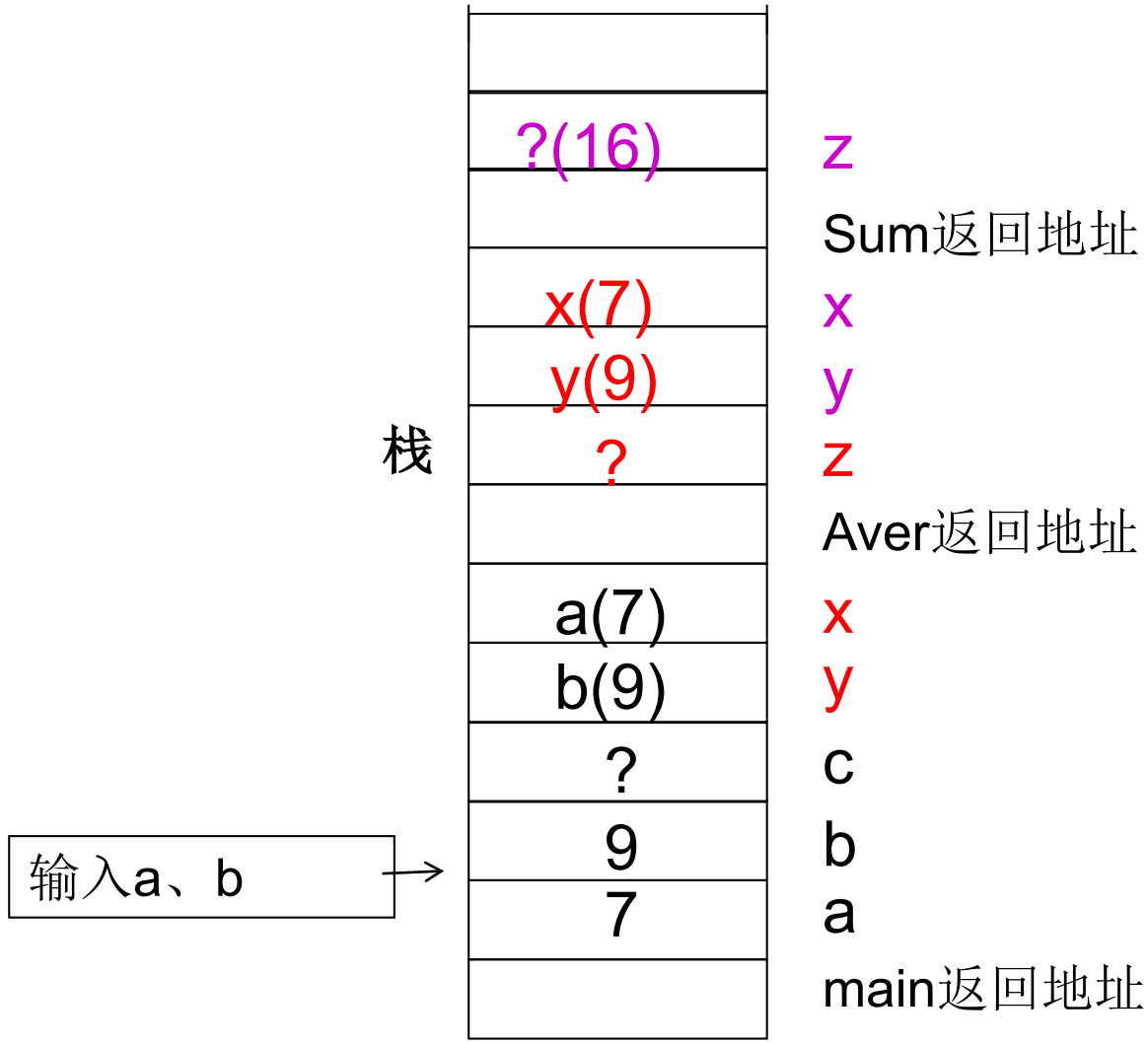
```
int MyMax(int n1, int n2, int n3)
{     int max;
      if(n1 >= n2)
            max = n1;
      else
            max = n2;
      if(max < n3)
            max = n3;
      return max;
}
```

注意对比 不含嵌套调用的情形：

```
f = MyFactorial(MyMax(n1, n2, n3));
```

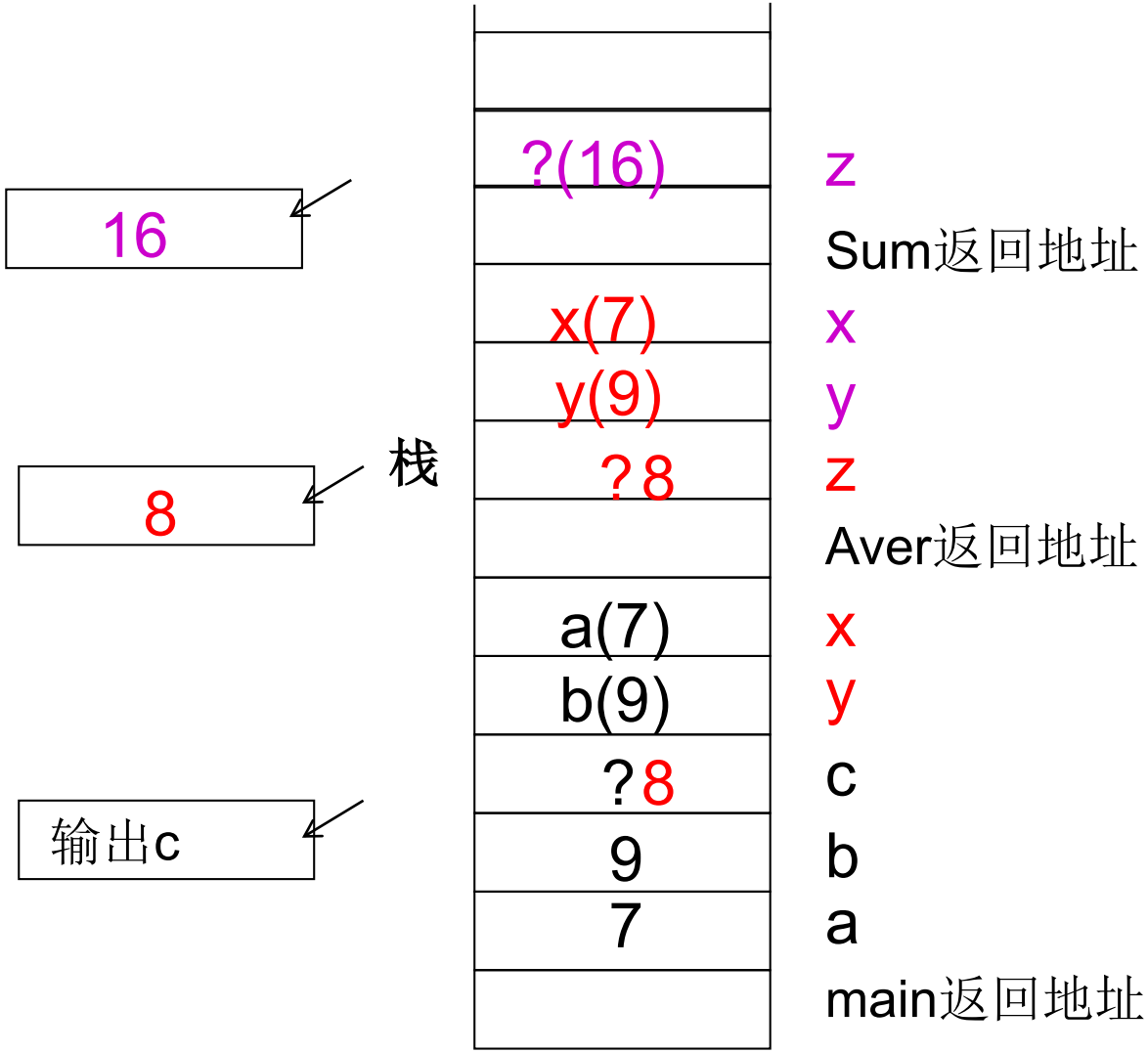
C函数嵌套调用的过程

```
int Sum(int x, int y)
{   int z;
    z = x + y;
    return z;
}

int Aver(int x, int y)
{   int z;
    z = Sum(x, y) / 2;
    return z;
}

int main()
{   int a, b, c;
    scanf("%d%d",&a, &b);
    c = Aver(a, b);
    printf("%d", c);
    return 0;
}
```



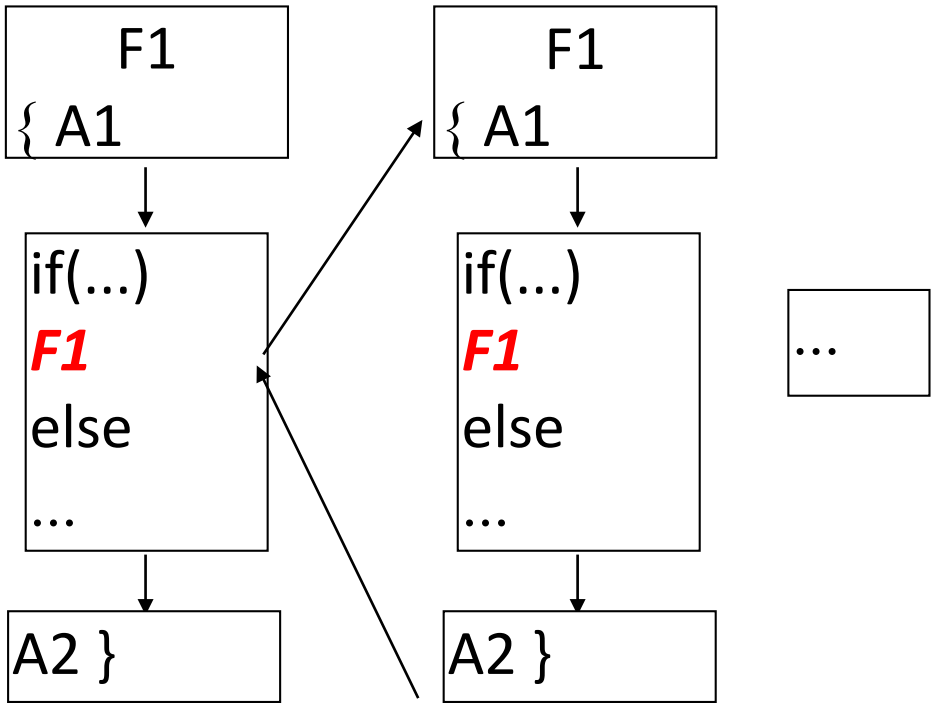
C函数嵌套调用过程（续）

```
int Sum(int x, int y)
{   int z;
    z = x + y;
    return z;
}

int Aver(int x, int y)
{   int z;
    z = Sum(x, y) / 2;
    return z;
}

int main()
{   int a, b, c;
    scanf("%d%d",&a, &b);
    c = Aver(a, b);
    printf("%d", c);
    return 0;
}
```

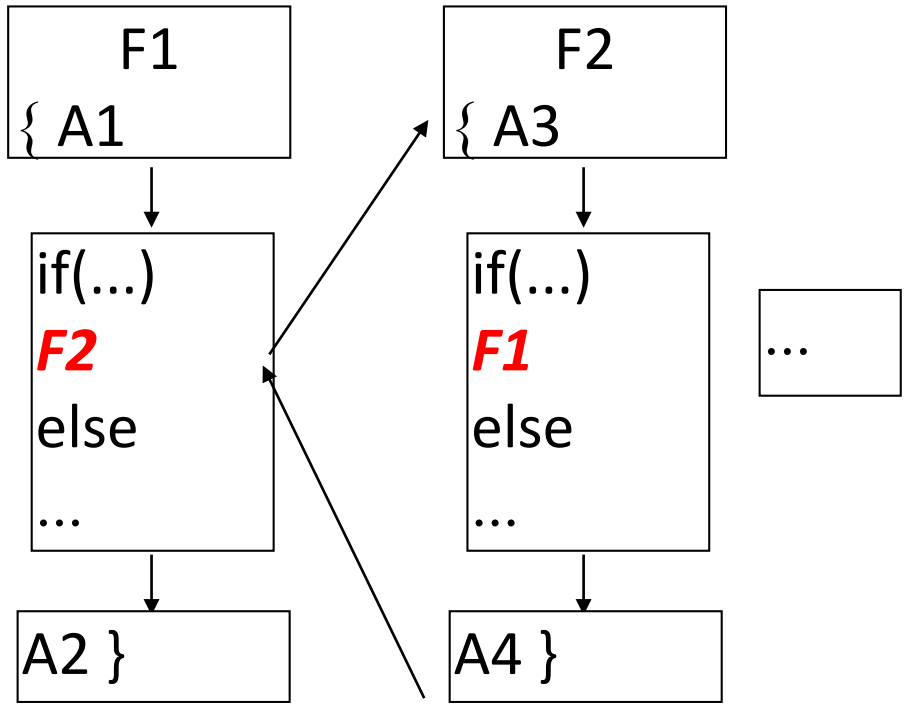


函数的*递归*（recursion）调用

- 即被调函数的定义中直接或间接含有*本*函数的调用操作



直接递归



间接递归

- **例2.3 设计 C 程序，用*递归调用*的函数实现 求阶乘问题。**
  - 分析：

$$n! = \begin{cases} 1 & (n = 0, 1) \\ n\cdot(n-1)! & (n > 1) \end{cases}$$

$1\cdot2\cdot3\cdot\ldots\cdot(n-1)\cdot n$

```
int MyFactorial(int n)
{
        int f = 1;
        for(int i = 2; i <= n; ++i)
                f *= i;
        return f;
}
```

```
int MyFactorialR(int n)
{
        if(n==0 || n==1)
                return 1;
        else
                return n * MyFactorialR(n-1);
}
```

或

递归调用函数的执行过程

```
int MyFactorialR(int n)
{
        if(n==0 || n==1)
                return 1;
        else
                return n * MyFactorialR(n-1);
}
```

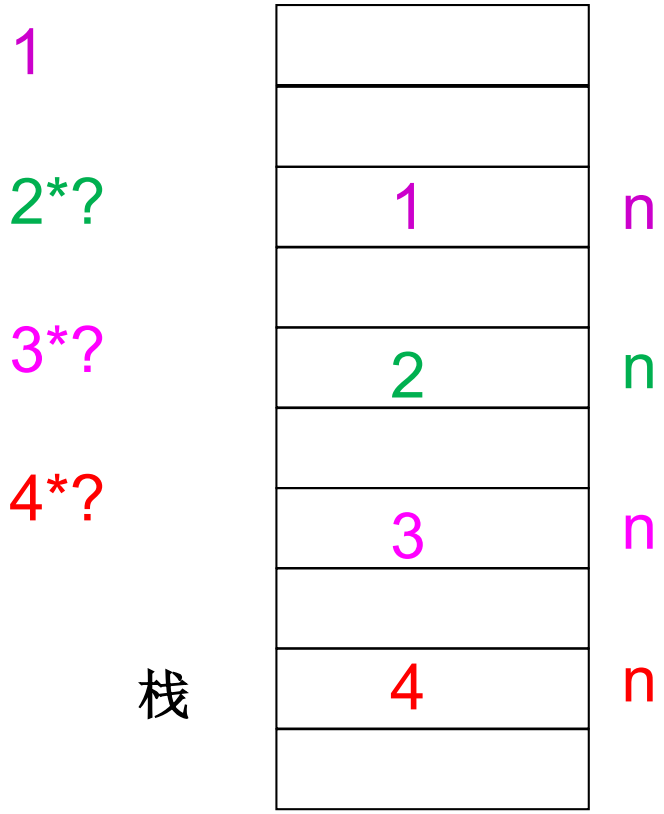
```
int main()
{       printf("%d", MyFactorialR(4));
        return 0;
}
```

实例一 => 4 * MyFactorialR(3)

实例二 => 3 * MyFactorialR(2)

实例三 => 2 * MyFactorialR(1)

实例四 => 1



递归调用函数的执行过程

```
int MyFactorialR(int n)
{
        if(n==0 || n==1)
                return 1;
        else
                return n * MyFactorialR(n-1);
}
```

```
int main()
{       printf("%d", MyFactorialR(4));
        return 0;
}
```

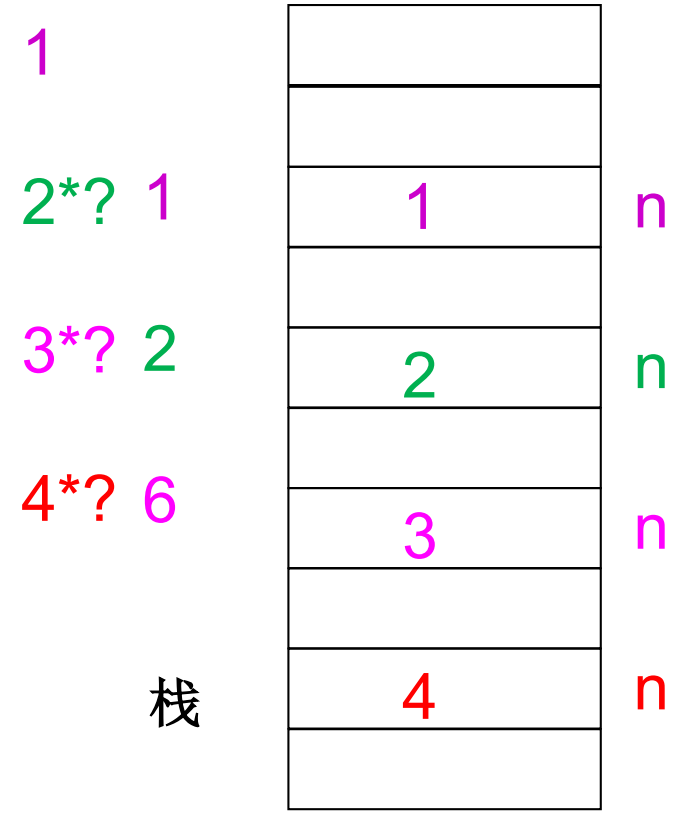
24

实例一 => 4 * 6

实例二 => 3 * 2

实例三 => 2 * 1

实例四 => 1

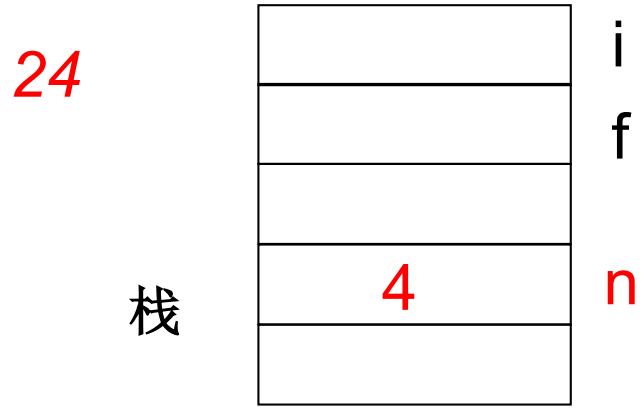


迭代法函数的执行过程

```
int MyFactorial(int n)
{
        int f = 1;
        for(int i = 2; i <= n; ++i)
                f *= i;
        return f;
}
```

```
int main()
{     printf("%d", MyFactorial(4));
      return 0;
}
```

24



斐波那契Fibonacci数列：有一对兔子，从出生后第3个月起每个月生一对兔子，小兔子长到第3个月后每个月又生一对兔子，假设所有兔子都不死，求第n个月的兔子总对数。

1，1，2，3，5，8，13，...

Fibonacci 数的定义：

$$fib(n) = \begin{cases} 1 & (n=1) \\ 1 & (n=2) \\ fib(n-2) + fib(n-1) & (n \geq 3) \end{cases}$$

```
int main
{
    int n;
    scanf("%d", &n);
    printf("第 %d 个月有 %d 对兔子.\n", n, temp);
    return 0;
}
```

```
int MyFib(int n)
{
    int fib1 = 1, fib2 = 1, temp = 1;
    for(int i = 3; i <= n; ++i)
    {       temp = fib1 + fib2;
            fib1 = fib2 ;
            fib2 = temp;
    }
    return temp;
}
```

```
int MyFibR(int n)
{
    if(n == 1 || n == 2)
       return 1;
    else
       return MyFibR(n-2) + MyFibR(n-1);
}
```

推广：

$$fib(n) = \begin{cases} 1 & (n < m) \\ fib(n-m+1) + fib(n-1) & (n \geq m) \end{cases}$$

```
int Fib(int n, int m)
{
    if(n < m)
       return 1;
    else
       return Fib(n-m+1, m) + Fib(n-1, m);
}
```

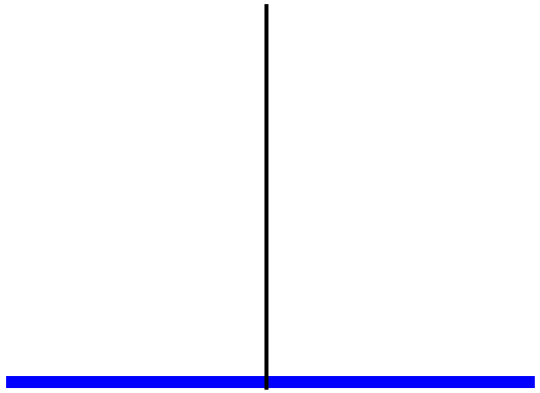
有一头小母牛，从出生后第 4 年起每年生一头母牛，小母牛长到第 4 年后每年又生一头母牛，假设所有母牛都不死，求第 n 年的母牛头数。

1，1，1，2，3，4，6，9, …

```
int FibCow(int n)
{   int fib1=1, fib2=1, fib3=1, temp;
    if(n < 4)
        return 1;
    else
    {
        for(int i = 4; i <= n; ++i)
        {   temp = fib1 + fib3;
            fib1 = fib2;
            fib2 = fib3;
            fib3 = temp;
        }
        return temp;
    }
}
```

- **例2.4 设计 C 程序，用*递归调用*的函数求解 河内塔（Tower of Hannoi，Tower of Brahma，Lucas’ Tower）问题。**

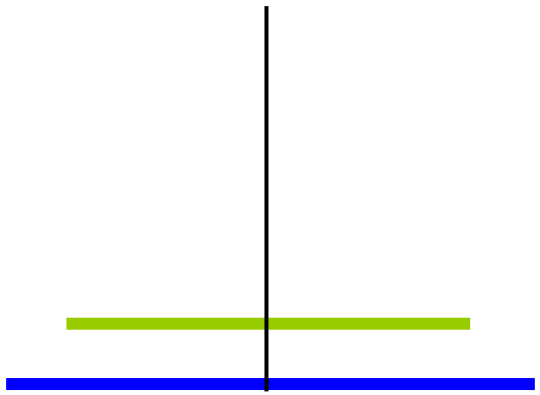
A → C



原来-A

借助-B

目标-C



原来-A

借助-B

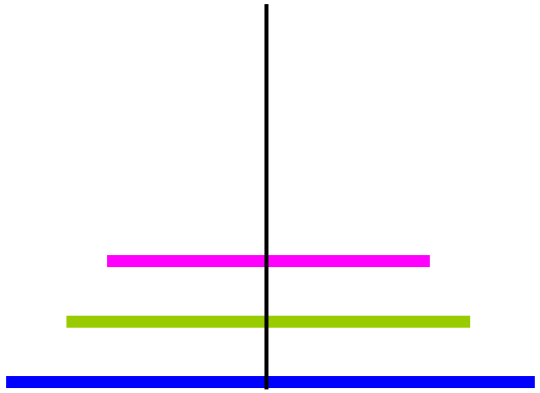
目标-C

A → B
A → C
B → C

A → C
A → B
C → B

A → C

B → A
B → C
A → C



原来-A

借助-B

目标-C

原来-A
借助-A

目标-B
原来-B

借助-C
目标-C

河内塔问题

- 分析:

$$H(n) = \begin{cases} 1 & (n=1) \\ 2 \cdot H(n-1) + 1 & (n>1) \end{cases}$$

$$= 2 \cdot (2 \cdot (2 \cdot (\ldots) + 1) + 1) + 1$$

```
int H(int n)
{
        int cnt = 0;
        for(int i = 1; i <= n; ++i)
                cnt = 2 * cnt + 1;
        return cnt;
}
```

```
int HR(int n)
{
    if(n == 1)
        return 1;
    else
        return 2*HR(n-1) + 1;
}
```

$2^n - 1$

按一秒钟移动一次计算，3 个圆盘需要 7 秒钟，如果有 20 个圆盘，需要一百多万秒，若有 64 个圆盘，则需要五千多亿年！

河内塔问题

- 分析：

（1）当 n 为 1 时：x → z
（2）当 n 大于 1 时：
① n-1 个盘子：**x → y**（借助 **z**）
② 第 n 个盘子：x → z
③ n-1 个盘子：**y → z**（借助 **x**）

```
void Hanoi(char x, char y, char z, int n)
{
    if(n == 1)
        printf("%c → %c \n", x, z);
    else
    {
        Hanoi(x, z, y, n-1);
        printf("%c → %c \n", x, z);
        Hanoi(y, x, z, n-1);
    }
}
```

河内塔问题

```
Hanoi('A', 'B', 'C', 3);
```

```
void Hanoi(char x, char y, char z, int n)
{   if(n == 1)
        printf("%c → %c \n", x, z);
    else
    {
        Hanoi(x, z, y, n-1);
        printf("%c → %c \n", x, z);
        Hanoi(y, x, z, n-1);
    }
}
```

```
printf("A → C\n");
printf("A → B\n");
printf("C → B\n");
```

```
printf("A → C\n");
```

```
printf("B → A\n");
printf("B → C\n");
printf("A → C\n");
```

```
Hanoi('A', 'C', 'B', 2);
printf("%c → %c \n", x, z);
Hanoi('B', 'A', 'C', 2);
```

```
Hanoi('A', 'B', 'C', 1);
printf("A → B\n");
Hanoi('C', 'A', 'B', 1);
```

```
printf("A → C\n");
```

```
Hanoi('B', 'C', 'A', 1);
printf("B → C\n");
Hanoi('A', 'B', 'C', 1
```

递归函数 vs. 循环流程

- 数据操作：
  - 循环流程是在*同一组变量*上进行重复操作；
  - 递归函数则是在*不同的变量组*（属于递归函数的不同实例）上进行重复操作。

- 递归函数的优势：
  - 为某些带有重复性操作的任务提供了一种比采用循环流程更为*自然*、*简洁*的实现方式。

- 递归函数的缺陷：
  - 由于递归函数表达的重复操作是通过函数调用来实现的，所以需要额外开销，栈空间的大小会限制递归的深度，从而降低了递归函数的可行性。
  - 有时会出现重复计算。

- 可用“动态规划”（Dynamic Programming）解决重复计算问题
  - 把计算过的内容保存下来（比如保存在数组中），需要时不再计算，直接用保存的结果。
  - 这是一种以空间换时间的策略。

单模块（Single module）

多模块（Multiple modules）

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与联系（模块设计方法、函数）

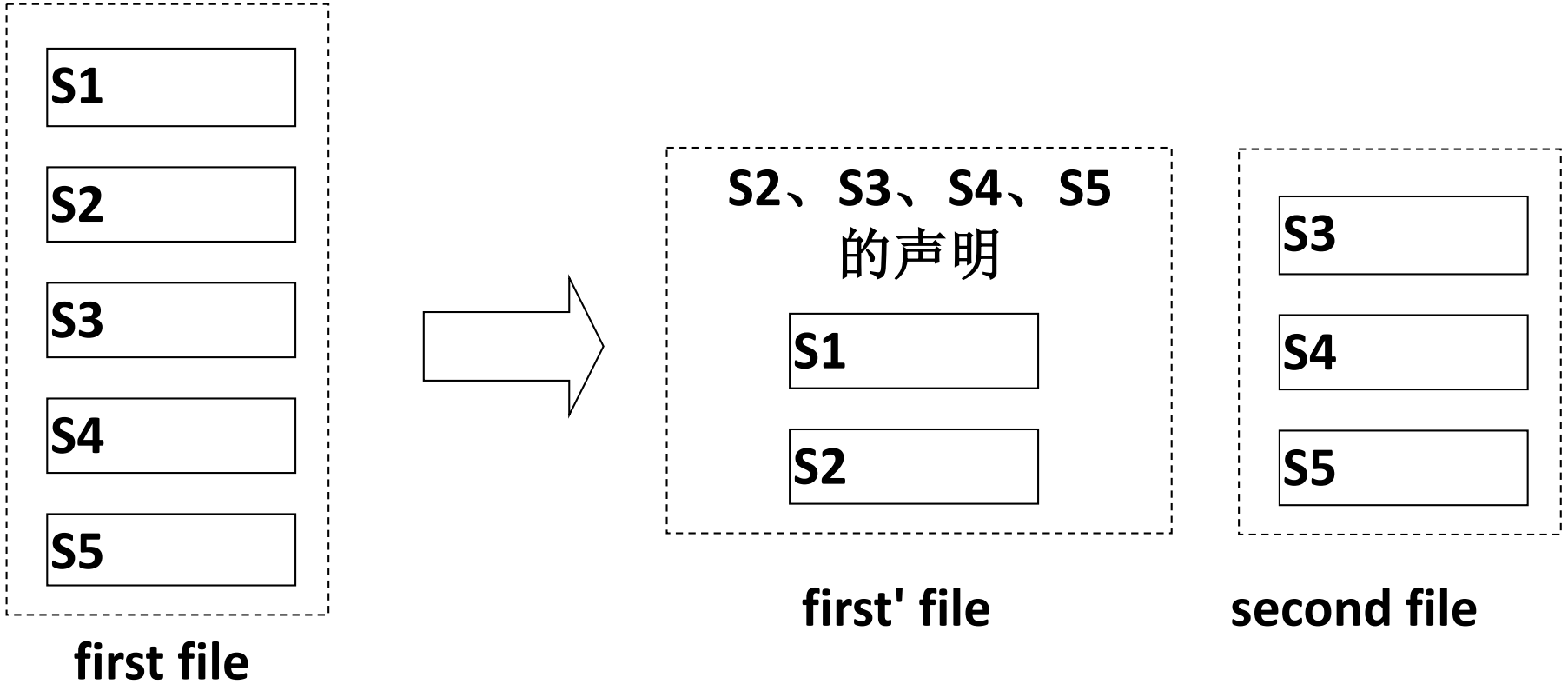
表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、结构体、指针）

归纳与推广（程序设计的本质）

- 一个源文件通常被看成一个模块，是一个独立的编译单元。



S1调用S2、S3，S2调用S4、S5

例2.5 甲乙两人共同开发一个程序，实现求最大数的阶乘功能。

```
//first.c 甲
#include <stdio.h>
int MyMax(int, int, int);
extern int MyFactorial(int);  //声明乙编写的函数
int main()
{    int n1, n2, n3, max, f;
     printf("Please input three integers: \n");
     scanf("%d", &n1, &n2, &n3);
     max = MyMax(n1, n2, n3);
     f = MyFactorial(max);
     printf …
     return 0;
}

int MyMax(int n1, int n2, int n3)
{     …     }
```

```
//second.c 乙
int MyFactorial(int n)
{    int f = 1;
     for(int i=2; i <= n; ++i)
            f *= i;
     return f;
}
```

文件包含预处理（preprocess）命令与头文件（head file）

```
//first.c 甲
#include <stdio.h>
int MyMax(int, int, int);
#include "second.h" //包含乙编写的头文件
int main()
{    int n1, n2, n3, max, f;
     printf("Please input three integers: \n");
     scanf("%d", &n1, &n2, &n3);
     max = MyMax(n1, n2, n3);
     f = MyFactorial(max);
     printf …
     return 0;
}

int MyMax(int n1, int n2, int n3)
{     …     }
```

```
//second.h
extern int MyFactorial(int);
```

```
//second.c 乙
int MyFactorial(int n)
{    int f = 1;
     for(int i=2; i <= n; ++i)
          f *= i;
     return f;
}
```

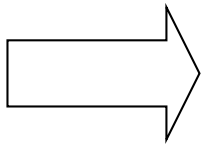
- 头文件里还可以放全局变量的声明等内容

//xlib.c

```
int x, y;
int F()
{     x = 1000;
      y = x*x;
      return y;
}
```

//my.c

```
extern int x, y;
int F();
int G()
{     int z;
      F();
      z = x + y;
      return z;
}
```



//xlib.h

```
extern int x, y;
int F();
```

//xlib.c

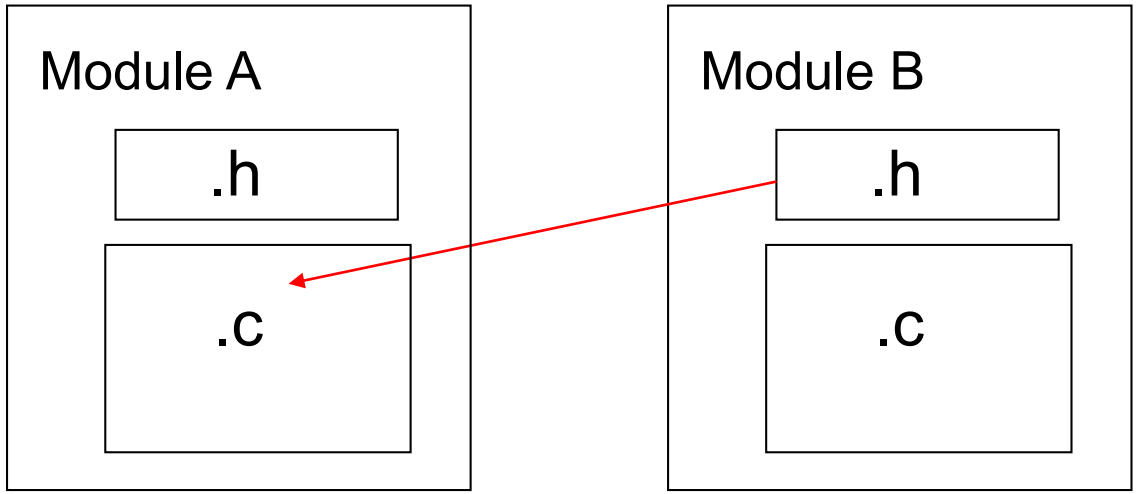
```
int x, y;
int F()
{     x = 1000;
      y = x*x;
      return y;
}
```

//my.c

```
#include "xlib.h"
int G()
{     int z;
      F();
      z = x + y;
      return z;
}
```

模块 = 接口 + 实现

- 接口（interface，.h 文件）
  - 在本模块中定义的、提供给其他模块使用的函数等程序实体的声明
- 实现（implementation，.c 文件）
  - 函数等程序实体的定义



标识符的属性

- **作用域（scope）**

标识符的有效范围

文件作用域

```
// second.c
extern int s; //从声明之后有效
void MySum(int n) //从定义开始有效
{
   int sum = 0;
   for(int i=1; i <= n; ++i)
        sum += i;
   s = sum;
}
```

```
// first.c
#include <stdio.h>
int s = 0; //从定义开始有效
extern void MySum(int); //从声明之后有效
int main()
{     int n;
      printf("Please input an integer: \n");
      scanf("%d", &n);
      if(n <= 0) goto L1;
      MySum(n);
L1:   printf("s = %d \n",  s);
      return 0;
}
```

文件作用域

```
// second.c
static void MySum(int n) //从定义开始有效
{
        int sum = 0;
        for(int i=1; i <= n; ++i)
                sum += i;
        printf("%d", sum);
}
```

```
// first.c
#include <stdio.h>

static int s = 0; //从定义开始有效
int main()
{     int n;
      printf("Please input an integer: \n");
      scanf("%d", &n);
      if(n <= 0) goto L1;
      s = ... ;
L1:   printf("s = %d \n", s);
      return 0;
}
```

文件作用域

```
// first.c
#include <stdio.h>
int s = 0; //从定义开始有效
void MySum(int); //从声明之后有效
int main()
{
    int n;
    printf("Please input an integer: \n");
    scanf("%d", &n);
    if(n <= 0) goto L1;
    MySum(n);
L1: printf("s = %d \n", s);
    return 0;
}
```

```
void MySum(int n)
{
    int sum = 0;
    for(int i=1; i <= n; ++i)
        sum += i;
    s = sum;
} //写在 first.c 文件后部
```

块作用域

```
void MySum(int n)                          //n
{
    int sum = 0;                           //sum
    for(int i=1; i <= n; ++i)              //i
    {     sum += i;          }
    s = sum;
} //写在 first.c 文件后部
```

```
// first.c
#include <stdio.h>
int s = 0;
void MySum(int);
int main()
{
    int n;                  //n
    printf("Please input an integer: \n");
    scanf("%d", &n);
    if(n <= 0) goto L1;
    MySum(n);
L1: printf("s = %d \n", s);
    return 0;
}
```

函数作用域

```
// first.c
#include <stdio.h>
int s = 0;
void MySum(int n);
int main()
{
    int n;
    printf("Please input an integer: \n");
    scanf("%d", &n);
    if(n <= 0) goto L1;
    MySum(n);
L1: printf("s = %d \n", s); //L1
    return 0;
}
```

```
void MySum(int n)
{
    int sum = 0;
    for(int i=1; i <= n; ++i)
        sum += i;
    s = sum;
} //写在 first.c 文件后部
```

函数原型作用域

```
// first.c
#include <stdio.h>
int s = 0;
void MySum(int n);        //n
int main()
{
    int n;
    printf("Please input an integer: \n");
    scanf("%d", &n);
    if(n <= 0) goto L1;
    MySum(n);
L1: printf("s = %d \n", s);
    return 0;
}
```

```
void MySum(int n)
{
    int sum = 0;
    for(int i=1; i <= n; ++i)
        sum += i;
    s = sum;
} //写在 first.c 文件后部
```

- **内层作用域的标识符会“挤走”外层作用域的同名标识符。**

```
int MyFactorial(int n)
{
    int f = 1;
    for(int i=2; i <= n; ++i)
    {
        int f = f*i;
    }
    return f;
}
```

//for语里重新定义了一个 **f**

//这里的 f 仍为1

- 在内层作用域中若要使用与其同名的外层作用域中的标识符，则需要用全局域选择符（::）进行限制。

```
int f = 1;
int MyFactorial(int n, int f)
{
      for(int i=2; i <= n; ++i)
      {
            ::f *= i;
      }
      return f * ::f;
}
```

标识符的属性

- 作用域（scope）
  - 链接（linkage）

标识符的有效范围

不同子程序中的同名标识符之间的关系

链接

```
//second.c-by 乙
```

链接

```
//first.c-by 甲
#include <stdio.h>
int m = 2;  //具有外部链接或内部链接属性，被 second.c 中的使用外部链接
extern int MyFact (int); //要求定义的 MyFact 应具有外部链接属性
static int MyMax(int, int, int); //要求定义的 MyMax 应具有内部链接属性
int main()
{       int n1, n2, n3;
        scanf("%d%d%d", &n1, &n2, &n3);      //n1, n2, n3, max, f无链接
        int max = MyMax(n1, n2, n3);  //链接下面定义的 MyMax
        int f = MyFact (max);   //链接 second.c 中定义的 MyFact
        printf("The factorial of max. is: %d \n", f);
        return 0;
}
static int MyMax(int n1, int n2, int n3)//仅具有内部链接属性，被上面的调用内部链接
{       int max;
        if(n1 >= n2)          // n1, n2无链接
                max = n1;     // max无链接
        else
                max = n2;
        if(max < n3)          // n3无链接
                max = n3;
        return max;
```

```
//second.c-by 乙
int MyFact(int n) //具有外部链接或内部链接属性，被 first.c 中的调用外部链接
{
      extern int m;                    //要求定义的 m 应具有外部链接属性
      int f = 1;
      for(int i=2; i <= n; ++i)        //i, f, n无链接
            f *= i;
      return m*f;                      //链接 first.c 中定义的 m
}
int MyMax(int n1, int n2, int n3)//具有外部链接或内部链接属性，没有被链接过
{
      int max;
      if(n1 >= n2)
            max = n1;
      else
            max = n2;
      if(max < n3)
            max = n3;
      return max;
}
```

标识符的属性

- **作用域（scope）**
  - 链接（linkage）
  - 名空间（namespace）

标识符的有效范围

不同子程序中的同名标识符之间的关系

相同子程序中的不同类标识符的作用域子空间

C：隐含的名空间

C++：

隐含的名空间

程序中指定名空间和使用名空间

隐含的名空间

- **是一种抽象的标识符的容器。**
- **程序中同一个作用域内逻辑上相关的标识符隐藏在同一个名空间里，否则，隐藏于不同的名空间。**

- **隐含的名空间有四种：**
  - **语句标号的名空间；**
  - 标签的名空间（派生类型名/枚举符）；
  - 某个派生类型（结构/联合）的所有成员的名空间；
  - **其他标识符的名空间，这些标识符包括变量名、函数名、形参名。**

一个程序中，相同作用域内的同一个名空间里不允许定义同名标识符

- **否则编译时会出现“重复定义”错误。**
- **比如，**
  - 一个源文件中不可以定义同名函数或同名全局变量（swap、count…?）
  - 函数里复合语句外不可以定义同名或与形参同名的局部变量
  - 一个复合语句中不可以定义同名局部变量
  - 一个函数中不能有相同的语句标号
  - 函数原型中的形参名如果没省略的话，相互不可以重名

- 多模块程序可能会面临的一个问题：
  - **在一个源文件中要用到两个分别在另外两个源文件中定义的不同全局程序实体（如：外部函数或外部变量），而这两个全局程序实体的名字相同。**
    - **C语言：程序组合的时候让其中一个改名字，加 static 禁止其中一个全局程序实体被外部链接**
    - **C++语言：可以指定名空间 放入不同的名空间**

C++程序中指定名空间和使用名空间

- 指定名空间:
  - **namespace X{…}**
  - **在一个名空间中定义的全局标识符，其作用域为该名空间**

- 使用名空间:
  - **using namespace x**
  - **x::**
    - **当在一个名空间外部需要使用该名空间中定义的全局标识符时，可以用该名空间的名字来修饰或受限**

```
//模块1
namespace A
{ int x=1;
  void f()
  { //.....
  }
}
```

```
//接口1(mh1.h)
namespace A
{ extern int x;
  extern void f();
}
```

1、

```
//模块2
namespace B
{ int x=0;
  void f()
  { //.....
  }
}
```

```
//接口2(mh2.h)
namespace B
{ extern int x;
  extern void f();
}
```

2、

3、

要在头文件和源文件中同时定义名空间

```
#include "mh1.h"
#include "mh2.h"
```

//模块3

```
A::x = 3;          //A中的x
A::f();            //A中的f
B::x = 5;          //B中的x
B::f();            //B中的f
```

```
using namespace A;
x = 3;             //A中的x
f();               //A中的f
B::x = 5;          //B中的x
B::f();            //B中的f
```

```
using A::f;
A::x = 3;          //A中的x
f();               //A中的f
B::x = 5;          //B中的x
B::f();            //B中的f
```

作用域：代码中的有效范围

- 文件
- 块

- 函数
- 函数原型

两个标识符的作用域相同意味着：
作用域种类相同，而且有效范围在同一处结束

- 名空间作用域

看到同名标识符，代表的是不同的实体，意味着：
要么在不同的作用域，要么在不同的名空间

C/C++ 语言的隐含名空间

```
struct Student
{
      int number;
      char name[20];
      int age;
};
struct Student s = {201240999, "xiaolanjing", 19};
int Student;
scanf("%d", &Student);

if(Student == s.number)
      printf("1\n");
else
      printf("0\n");
```

虽然在同一个作用域内，
同名的两个标识符并不冲突，
因为位于两个不同的名空间

C++ 语言的指定名空间

```
//模块1
namespace A
{
   void Foo()
   { ...
   }
}
```

```
//接口1(mh1.h)
namespace A
{
   extern void Foo();
}
```

```
//模块2
namespace B
{
   void Foo()
   { ...
   }
}
```

```
//接口2(mh2.h)
namespace B
{
   extern void Foo();
}
```

```
#include "mh1.h"
#include "mh2.h"
```

```
//模块3

A::Foo();          //A中的Foo
B::Foo();          //B中的Foo
```

```
//模块3

using namespace A;
Foo();             //A中的Foo
B::Foo();          //B中的Foo
```

不同模块的 函数相遇的时候 避免冲突
把它们放在不同的名空间中

标识符的属性

- **作用域（scope）**
  - 链接（linkage）
  - 名空间（namespace）

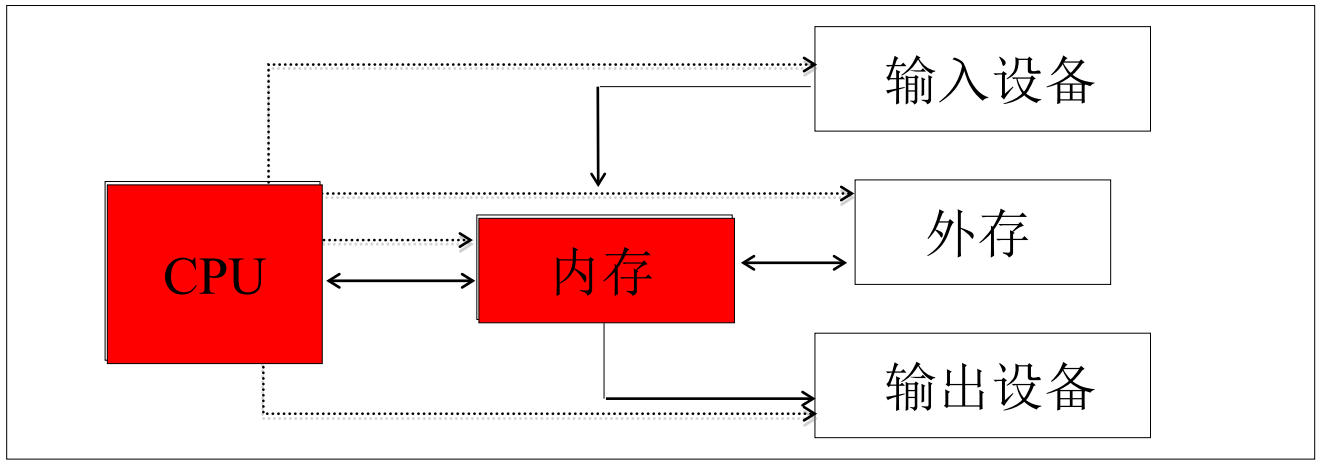
标识符的有效范围

不同子程序中的同名标识符之间的关系

相同子程序中的不同类标识符的作用域子空间

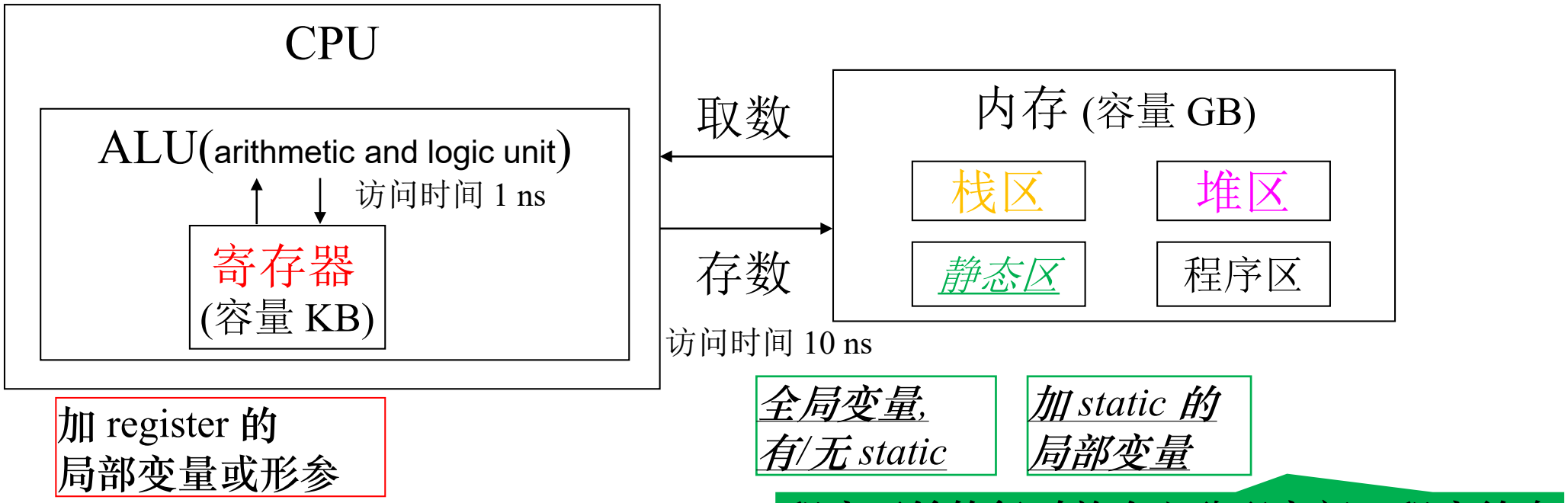
- **存储期（storage duration）**

数据的生存寿命（lifetime）



存储期

- 在程序执行期间，不同存储位置中的数据存储期不同

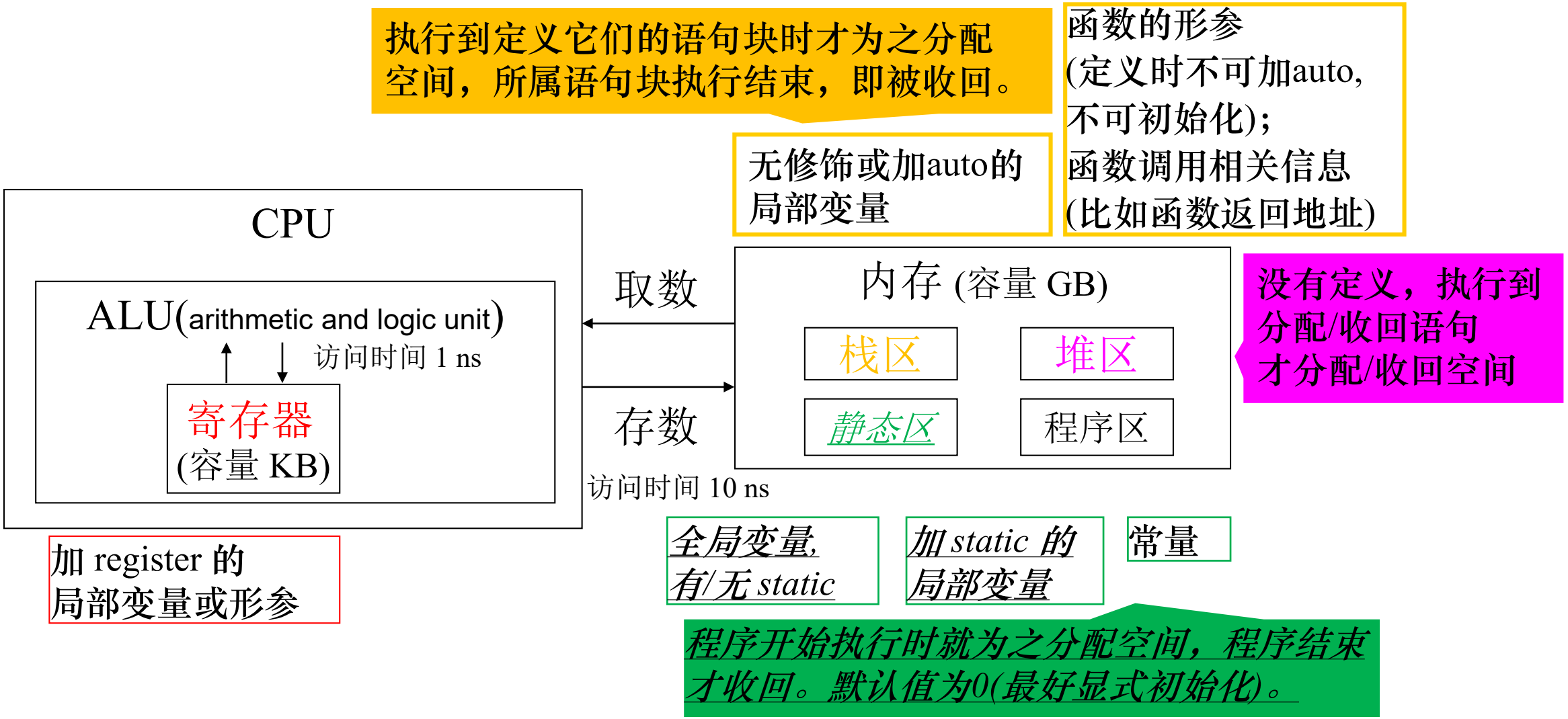


例2.6 输出任意三个不同的整数。

```
#include <stdio.h>
int MyRand(void);
int main()
{
      int m;
      m = MyRand();
      printf("The m is: %d \n", m);
      m = MyRand();
      printf("The m is: %d \n", m);
      m = MyRand();
      printf("The m is: %d \n", m);
      return 0;
}
int MyRand(void)
{
      static int s = 1; //静态变量
      s = (7 * s + 19) % 3;
      return s;
}
```

存储期

- **在程序执行期间，不同存储位置中的数据存储期不同**



static

- **关键字*static*有两种不同的含义。**
  - 指定**局部变量**采用**静态存储分配**（降低了可读性，但具有数据保留作用，如果所在的函数再次被调用，该静态变量不再被初始化，而是保留上次的执行结果）
  - 把**外部链接**改变为**内部链接**（只希望在一个模块内共享的函数和全局变量，定义时可以加static，降低了通用性，但具有数据保护作用）----把一个文件的代码放到一个无名的名空间中，也可以起到相同的效果

```
//模块1
namespace A
{
   void Foo()
   {   ...
   }
}
...//可以使用Foo
```

相当于加了static

```
//模块2
namespace B
{
   void Foo()
   {   ...
   }
}
```

```
//模块3

Foo();       //无法使用A中的Foo

B::Foo();  //B中的Foo
```

跟模块设计有关的优化

宏定义
内联函数
条件编译

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与联系（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、结构体、指针）

归纳与推广（程序设计的本质）

宏（Macro）定义

```
#include <stdio.h>

#define PI 3.142

void BallSize(int r)
{
  printf("Diameter: %f \n", 2*PI);
  printf("Perimeter: %f \n", 2*PI*r);
  printf("Area: %f \n", PI*r*r);
  printf("Volume: %f \n", 4*PI*r*r*r/3);
}
```

符号常量（manifest constant）

```
// BallSize.h
#include <stdio.h>
#define PI 3.142
```

```
#include "BallSize.h"
int main()
{
        ……
        BallSize(m);
        return 0;
}
```

```
#include <stdio.h>

#define QSum(x, y) x*x+y*y
int main()
{
    int m, n;
    scanf("%d%d", &m, &n);
    printf("%d", QSum(m, n)); //printf("%d", m*m+n*n);
}
```

带参数的宏定义

```
#include <stdio.h>

#define QSum(x, y)  ((x)*(x)+(y)*(y))
int main()
{
      int m, n;
      scanf("%d%d", &m, &n);
      printf("%f", 1.0/QSum(m+n, m-n));
}
```

带参数的宏定义

```
printf("%f", 1.0/m+n*m+n+m-n*m-n); //?
```

```
printf("%f", 1.0/((m+n)*(m+n)+(m-n)*(m-n)));
```

内联（inline）函数

```
inline double QSum(double x, double y)
{
  return x*x + y*y;
}
```

```
int main()
{
      int m, n;
      scanf("%d%d", &m, &n);
      printf("%f", 1.0/QSum(m+n, m-n));
}
```

```
printf("%f", 1.0/((m+n)*(m+n)+(m-n)*(m-n)));
```

条件编译

```
#define ABC

#ifdef ABC
      <代码片段1>
#else
      <代码片段2>
#endif

      <其余代码>
#undef ABC
```

```
#if defined(ABC)
```

```
#define ABC

#ifndef ABC
      <代码片段1>
#else
      <代码片段2>
#endif

      <其余代码>
#undef ABC
```

```
#if !defined(ABC)
```

用来避免重复包含头文件

main.cpp

```
#include <stdio.h>

#include "module1.h"

#include "module2.h"

int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double MyFun(double, double);
```

module1.h

```
#define N 100
double MySin(double);
```

用来避免重复包含头文件

main.cpp

```
#include <stdio.h>
#define N 100
double MySin(double);

#include "module2.h"

int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double MyFun(double, double);
```

module1.h

```
#define N 100
double MySin(double);
```

用来避免重复包含头文件

main.cpp

```
#include <stdio.h>
#define N 100
double MySin(double);
#define N 100
double MySin(double);
double MyFun(double, double);
int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double MyFun(double, double);
```

module1.h

```
#define N 100
double MySin(double);
```

用来避免重复包含头文件

main.cpp

```
#include <stdio.h>

#include "module1.h"

#include "module2.h"

int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double MyFun(double, double);
```

module1.h

```
#ifndef MODULE1
#define MODULE1
#define N 100
double MySin(double);
#endif
```

这样，在一个源文件中如果多次包含上面的 **module1.h** 文件，系统只会对第一次包含的内容进行处理。

用来避免重复包含头文件

main.cpp

```
#include <stdio.h>
#define MODULE1
#define N 100
double MySin(double);
#include "module2.h"

int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double MyFun(double, double);
```

module1.h

```
#ifndef MODULE1
#define MODULE1
#define N 100
double MySin(double);
#endif
```

这样，在一个源文件中如果多次包含上面的 **module1.h** 文件，系统只会对第一次包含的内容进行处理。

用来避免重复包含头文件

main.cpp

```
#include <stdio.h>
#define MODULE1
#define N 100
double MySin(double);

double MyFun(double, double);
int main()
{
      //……
}
```

module2.h

```
#include "module1.h"
double MyFun(double, double);
```

module1.h

```
#ifndef MODULE1
#define MODULE1
#define N 100
double MySin(double);
#endif
```

这样，在一个源文件中如果多次包含上面的 **module1.h** 文件，系统只会对第一次包含的内容进行处理。

用于多环境的程序编写

```
#define OS 'W'

#if OS == 'W'
   ……          //适合于 Windows 环境的代码
#elif OS == 'U'
   ……          //适合于 UNIX 环境的代码
#elif OS == 'M'
   ……          //适合于 macOS 环境的代码
#else
   ……          //适合于其他环境的代码
#endif
   ...          //与环境无关的公共代码
```

```
#ifdef OS
#if defined (OS)
#ifndef <OS>
#if !defined (OS)
```

预定义标识符

用于程序的调试

```
#include <stdio.h>
void BallSize(int r);

#define DEBUG

int main()
{
    double r;
    scanf("%lf", &r);
    #ifdef DEBUG
  // printf("%.2f \n", r);   //调试输出
    #endif
    BallSize(r);
    return 0;
}
```

调试结束，注释掉此行

```
#define PI 3.142
void BallSize(int r)
{
    printf("Diameter: %f \n", 2*PI);
    printf("Perimeter: %f \n", 2*PI*r);
    printf("Area: %f \n", PI*r*r);
    printf("Volume: %f \n", 4*PI*r*r*r/3);
}
```

C++

- 函数名重载
- 带默认值的形式参数

函数名重载

- 对于一些功能相同、参数类型或个数不同的函数，有时给它们取相同的名字会带来使用上的方便。例：

  void **Print_int**(int i) { ..... }

  void **Print_double**(double d) { ..... }

  void **Print_char**(char c) { ..... }

  void **Print_A**(A a) { ..... } //A为自定义类型

  定义为：

  void **Print**(int i) { ..... }

  void **Print**(double d) { ..... }

  void **Print**(char c) { ..... }

  void **Print**(A a) { ..... }

- C++规定：在相同的作用域中，可以用同一个名字定义多个不同的函数，这时，要求定义的这些函数应具有不同的参数
- 上述的函数定义形式称为*函数名重载*

对重载函数调用的绑定

- 确定一个对重载函数的调用对应着哪一个重载函数定义的过程称为**绑定（binding，又称定联、联编、捆绑）。**
  - 例：**Print(1.0)将调用void Print(double d) { ..... }**

- 对重载函数调用的绑定在编译时刻由编译程序根据实参与形参的匹配情况来决定。从形参个数与实参个数相同的重载函数中按下面的规则选择一个：
  - **精确匹配**
  - **提升匹配**
  - **标准转换匹配**
  - **自定义转换匹配**
  - **匹配失败**

精确匹配

- **类型相同**
- 对实参进行“微小”的类型转换：
  - 数组变量名->数组首地址
  - 函数名->函数首地址
  - 等等
- **例如，对于下面的重载函数定义：**

void Print(int);

void Print(double);

void Print(char);

下面的函数调用：

Print(1); 绑定到函数：void Print(int);

Print(1.0); 绑定到函数：void Print(double);

Print('a'); 绑定到函数：void Print(char);

提升匹配

- 先对实参进行下面的类型提升，然后进行精确匹配：
  - 按整型提升规则提升实参类型
  - 把float类型实参提升到double
  - 把double类型实参提升到long double
- 例如，对于下述的重载函数：

  void Print(int);

  void Print(double);

  根据提升匹配，下面的函数调用：

  Print('a'); 绑定到函数：void Print(int);

  Print(1.0f); 绑定到函数：void Print(double);

标准转换匹配

- **任何算术类型可以互相转换**
- **枚举类型可以转换成任何算术类型**
- **零可以转换成任何算术类型**或指针类型
- 任何类型的指针可以转换成void *
- 派生类指针可以转换成基类指针
- **每个标准转换都是平等的。**

绑定失败

- 例如，对于下述的重载函数：

  float sqrt(float);

  double sqrt(double);

  根据标准转换匹配，下面的函数调用：

  sqrt(2.56); 绑定到函数：double sqrt(double);

  而sqrt(256); 会绑定失败

- **如果不存在匹配或存在多个匹配，则绑定失败**
  - **根据标准转换，256（属于int型）既可以转成float，又可以转成double**

- 解决办法是：
  - 对实参进行显式类型转换，如，
    - sqrt(*(float)*256)或sqrt(*(double)*256)
  - 增加额外的重载，如，
    - 增加一个重载函数定义int sqrt(int);

带默认值的形式参数

- 在C++中允许在定义或声明函数时，为函数的某些参数指定默认值。如果调用这些函数时没有提供相应的实参，则相应的形参采用指定的默认值，否则相应的形参采用调用者提供的实参值。
- 例如，对于下面的函数声明：

```
void Print(int value, int base=10);
```

下面的调用：

```
Print(28);          //28传给value；base为10
Print(32, 2);       //32传给value；2传给base
```

- 在指定函数参数的默认值时，应注意下面几点：
  - **有默认值的形参应处于形参表的右部。例如：**
    - **void F(int a, int b=1, int c=0);**
    - **void F(int a, int b=1, int c); //×**
  - **对参数默认值的指定*只在函数声明或定义处有意义*。**
  - 在不同的源文件中，对同一个函数的声明可以对它的同一个参数指定不同的默认值；在同一个源文件中，对同一个函数的声明只能对它的每一个参数指定一次默认值。

小结

- 程序的模块设计
  - 分解与复合：过程抽象、子程序；合理安排、调用
- 单模块程序的设计
  - C语言函数（子程序）基础
    - 函数的概念
    - 函数的定义
    - 函数的调用
    - 函数的参数与返回值
    - 函数的声明
    - 全局变量及其声明
    - 函数的副作用
  - C语言函数的嵌套调用
    - 嵌套调用及其过程
    - 递归
- 多模块程序的设计
  - 文件包含命令
  - 头文件及其作用
  - 标识符的作用域、存储期、名空间
- 与程序模块设计有关的优化
  - 带参数的宏定义
  - 内联函数
  - 条件编译
  - 函数名重载（C++）
  - 带默认值的形式参数（C++）

- 要求：
  - 会运用C语言函数实现独立的计算任务，并被main函数或其他函数调用
    - 在函数中运用顺序、分支、循环流程，设计变量
    - 一个程序代码量≈30行，
  - 能够用递归函数实现一个递归算法
  - 能够分析递归函数的功能与结果
  - 能够用模块化思想设计程序：
    - 仔细阅读并分析需求
    - 分解出关键子任务
    - 尝试从大规模问题中分析出小规模同质问题
    - 理解程序多模块结构的相关概念
    - 能够实现多模块结构的程序，并分析其功能与结果
  - 能够调试、判断程序中的逻辑错误
  - 继续保持良好的编程习惯
    - 函数名命名、宏名命名
    - 使用声明、头文件
    - 少用全局变量
    - …

Thanks!