



中国科学技术大学  
University of Science and Technology of China

计算机程序设计

# 第四章 模块化程序设计

白雪飞

中国科学技术大学微电子学院

- 引言
- 模块化思想
- 函数
- 模块化设计与实践
- 模块化与计算思维实践
- 小结



# 引言

## ■ 模块化程序设计 (Modular Programming)

- 将软件系统按照功能层层分解为若干模块
  - **模块**: 独立、可替换、具有预定功能, 函数(Function)
- 各模块之间通过接口实现调用, 互相协作解决问题
  - **接口**: 对输入与输出的描述

## ■ 模块化的特点

- 复杂程序的设计更加灵活高效
  - 设计良好的模块能够重复使用、易于替换
- 程序设计过程更加复杂
  - 需要考虑如何划分模块、设计模块间的接口

## ■ 模块化与结构化

- **模块化**编程: 将整个程序划分成各部分的**高层分解**
- **结构化**编程: 采用结构化控制语句的**低层代码编写**



# 模块化思想

- 精简程序代码
- 改善程序结构
- 增强程序的通用性

## ■ 例4.1-1：字符画树程序的精简代码设计。

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

```
int main()
{
    printf("  *\n");
    printf("  ***\n");
    printf("  *****\n");
    printf("  *****\n");
    printf("  *\n");
    printf("  ***\n");
    printf("  *****\n");
    printf("  *****\n");
    printf("  #\n");
    printf("  #\n");
    printf("  #\n");
    return 0;
}
```

重复代码段  
定义为函数

运行结果：

```
  *
  ***
  *****
  *****
  *
  ***
  *****
  *****
  #
  #
  #
```

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

```
void treetop() // 函数定义
{
    printf("  *\n");
    printf("  ***\n");
    printf("  *****\n");
    printf("  *****\n");
}

int main()
{
    treetop(); // 函数调用
    treetop(); // 函数调用
    printf("  #\n");
    printf("  #\n");
    printf("  #\n");
    return 0;
}
```

调用与实现分离  
精简main函数

运行结果：

```
  *
  ***
  *****
  *****
  *
  ***
  *****
  *****
  #
  #
  #
```

- 精简程序代码
- 改善程序结构
- 增强程序的通用性



# 改善程序结构举例



## ■ 例4.1-2：字符画树程序的模块化设计。

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

```
void treetop()  
{  
    printf("  *\n");  
    printf(" ***\n");  
    printf(" *****\n");  
    printf("*****\n");  
}
```

```
int main()  
{  
    treetop();  
    treetop();  
    printf("  #\n");  
    printf("  #\n");  
    printf("  #\n");  
    return 0;  
}
```

功能实现语句  
定义为函数

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

```
void treetop()  
{  
    printf("  *\n");  
    printf(" ***\n");  
    printf(" *****\n");  
    printf("*****\n");  
}
```

```
void treetrunk()  
{  
    printf("  #\n");  
    printf("  #\n");  
    printf("  #\n");  
}
```

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

功能实现与应用分离  
程序结构更清楚

运行结果：

```
  *  
 ***  
*****  
*****  
  *  
 ***  
*****  
*****  
  #  
  #  
  #
```

- 精简程序代码
- 改善程序结构
- 增强程序的通用性

# 增强程序的通用性举例

## ■ 例4.1-3：字符画树程序的通用性设计。

```
#include <stdio.h>

void treetop(char ch)           // 树叶字符定义为参数，可以在调用时指定不同字符
{
    printf("  %c\n", ch);
    printf("  %c%c%c\n", ch, ch, ch);
    printf(" %c%c%c%c%c\n", ch, ch, ch, ch, ch);
    printf("%c%c%c%c%c%c%c\n", ch, ch, ch, ch, ch, ch, ch);
}

void treetrunk()
{
    printf("  #\n");
    printf("  #\n");
    printf("  #\n");
}

int main()
{
    treetop('*');               // 画出以 '*' 为叶的树冠
    treetop('@');               // 画出以 '@' 为叶的树冠
    treetrunk();
    return 0;
}
```

### 思考拓展：

考虑进一步  
增强程序的通用  
性，为函数添加  
相应参数：

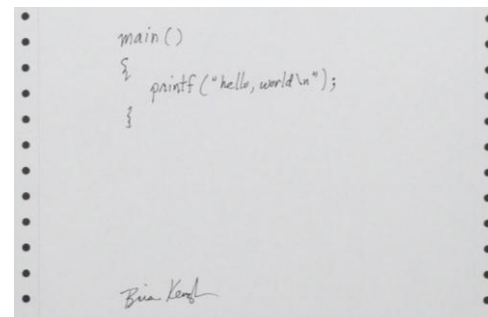
- ① 树冠字符；
- ② 树冠层数；
- ③ 树干字符；
- ④ 树干长度；
- ⑤ ... ..

### 运行结果：

```
  *
 ***
*****
*****
  @
  @@@
 @@@@@
 @@@@@@
  #
  #
  #
```

# 函数

- 函数定义
- 函数调用
- 函数原型声明
- 深入理解函数



```
main()
{
    printf("hello, world\n");
}
```

Brian Kernighan

*The "hello, world" Program  
by Brian Kernighan, 1978*

## ■ 一般形式

函数类型 函数名(形参类型 形参名, 形参类型 形参名, ...) 函数体

## ■ 说明

- **函数类型**即函数**返回值**的类型
  - 若函数定义的函数**类型缺省**, 则函数类型**默认为int**类型
  - 若函数**没有返回值**, 则函数类型应定义为**void**类型
- **函数名**遵循标识符命名规范, 应尽量体现函数的功能
- 函数定义中的参数称为**形式参数**, 简称**形参**
  - 函数可拥有零个、一个或多个形参, 当函数没有形参时, **()**内为空或写作**(void)**
  - 形参被视作函数体中定义的**局部变量**, 并且**在函数调用时使用实参的值初始化**
- **函数体**是一条用**{ }**括起来的复合语句
  - 函数体中可包含一条或多条**return**语句, 执行**return**语句会立刻结束函数调用
  - 有返回值的函数中, **return**语句带有表达式, 表达式的值转换为函数类型并返回
  - 无返回值的函数中, **return**语句不带表达式, 也可以没有**return**语句

# 函数定义举例：判断素数



## ■ 例4.2-a：定义判断素数的函数。

```
int IsPrime(int n)           // 函数头，有一个形式参数（形参）
{                             // 函数体复合语句开始
    int i=2;                 // 函数体中可以定义变量，且只能用于本函数内部

    while (i <= n-1) {       // 函数体中，形参用作变量，其初值来自函数调用时的实参
        if (n%i == 0)        // 若能整除，则n为合数
            return 0;        // 结束函数调用并返回0（“假”）
        else                 // 这个else可以没有
            i++;              // 若不能整除，准备测试[2,n-1]范围内的下一个整数
    }

    return 1;                // 能执行到此处说明n是素数，结束函数调用并返回1（“真”）
}
```

- 函数定义
- 函数调用
- 函数原型声明
- 深入理解函数



## ■ 一般形式

函数名(实参, 实参, ...)

## ■ 说明

- 函数调用中的参数称为**实际参数**，简称**实参**
  - 实参与形参**数量相同**，并按照顺序**一一对应**
  - 实参与形参的**类型应相同**或**赋值兼容**
  - 实参可以是常量、变量、表达式，其值在函数调用时传递给相应的形参
  - 若实参是变量，实参与形参可以同名或不同名，即使它们同名，也是**不同**的变量
- 函数调用中的函数名和实参前都**不需要**书写数据类型
- 即使函数没有参数，函数名后的**()**也不能省略
- **有**返回值的函数，其函数调用可以用作**表达式**，函数返回值即表达式的值
- **无**返回值的函数，其函数调用不能用作表达式，**只能用于函数调用语句**

# 函数定义和调用举例：判断素数



## ■ 例4.2-1：输入一个整数并判断其是否为素数。

```
#include <stdio.h>
#include <math.h>           // 数学函数头文件

int is_prime(int n)         // 函数定义和形参n
{
    int i, m;

    if (n <= 1)              // 处理异常输入
        return 0;           // 异常输入，返回0

    m = sqrt(n);              // 取 $\sqrt{n}$ 的整数部分

    for (i=2; i<=m; i++)     // 只需循环到 $\lfloor\sqrt{n}\rfloor$ 
        if (n%i == 0)        // 若发现n的因数
            return 0;         // n是合数，返回0

    return 1;                // n是素数，返回1
}
```

```
int main()                  // 程序从主函数开始执行
{
    int n;                   // 与形参n是不同的变量

    printf("输入一个整数: ");
    scanf("%d", &n);

    if (is_prime(n))         // 函数调用和实参变量n
        printf("%d是素数\n", n);
    else
        printf("%d不是素数\n", n);

    return 0;                // 程序在主函数结束执行
}
```

### 运行结果一：

输入一个整数：97↵  
97是素数

### 运行结果二：

输入一个整数：100↵  
100不是素数

## ■ 值传递机制

- 实参到形参的**单向值传递**
  - 实参与形参是**不同**的数据对象
  - 实参的值**复制**给形参，由形参参与函数内部的运算
  - 形参的值在函数内部的任何变化**无法反向传递**回实参
- 通过函数调用**无法修改**实参的值

## ■ 传递地址值

- 实参和形参是**地址类型**（**指针类型**）
- 地址值传递仍然遵循**单向值传递**规则
  - 形参从实参获得其地址类型的值，与实参指向了**相同的存储空间**
  - 对**形参指向的数据**进行的操作，就是对**实参指向的数据**所作的操作
  - 借助地址值传递可以改变**多个**数据的值
- 通过函数调用**不会修改**地址类型实参的值

# 值传递机制举例



## ■ 例4.2-2：编写函数求三个数的最大值。

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

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

```
int main()
{
    int a;
    float b;

    scanf("%d%f", &a, &b);
    printf("最大值: %d\n", max(a, b+2, 9*9));
    return 0;
}
```

类型转换  
float→int

### 程序解析：

- 1 形参的初值来自实参，不需要在函数内部对形参再次进行初始化或赋值，即可直接使用；
- 2 实参可以是常量、变量、表达式；
- 3 实参数据类型应和形参相同，或能够转换为形参数据类型，即赋值兼容；
- 4 若实参数据类型和形参不同，则先将实参的值隐式转换为形参数据类型，再进行参数传递。

### 运行结果一：

1 2  
最大值：81

### 运行结果二：

100 234.56  
最大值：236

### 运行结果三：

10 -3.14  
最大值：81

### 运行结果四：

1000 234.56  
最大值：1000

# 单向值传递举例



## ■ 例4.2-3：编写函数交换两个参数的值。

```
#include <stdio.h>

void swap(int a, int b)
{
    int t;
    t = a, a = b, b = t;    // 交换形参的值
    printf("In swap()\t : a=%d b=%d\n", a, b);
}

int main()
{
    int a=3, b=5;

    printf("Before call swap(): a=%d b=%d\n", a, b);
    swap(a, b);              // 能否交换实参的值?
    printf("After call swap() : a=%d b=%d\n", a, b);

    return 0;
}
```

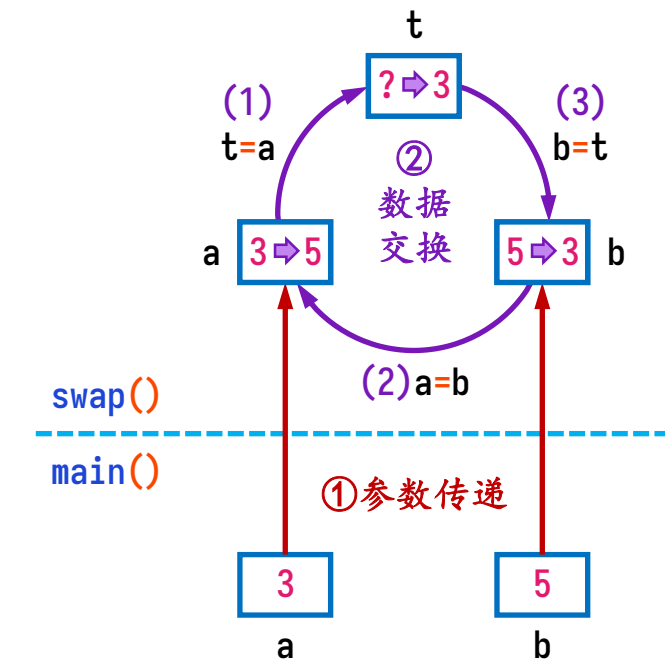
### 运行结果：

Before call swap(): a=3 b=5

In swap() : a=5 b=3

After call swap() : a=3 b=5

### 变量监测：



# 传递地址值举例

## ■ 例4.2-4：编写函数交换数组中元素的值。

```
#include <stdio.h>

void swap(int b[])           // 形参b用作数组地址
{
    int t;
    t = b[0], b[0] = b[1], b[1] = t;
    printf("b的内容是: %d, %d\n", b[0], b[1]);
}

int main()
{
    int a[2]={3,5};

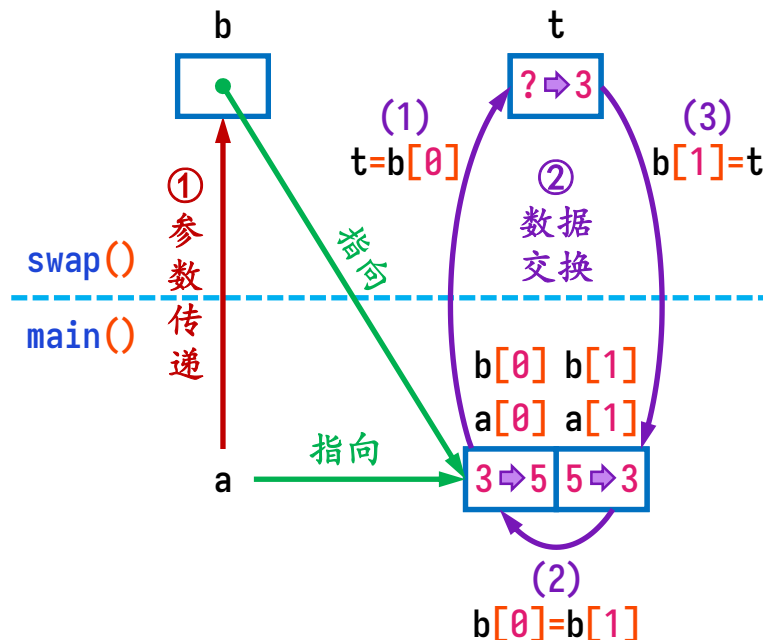
    printf("a的内容是: %d, %d\n", a[0], a[1]);
    swap(a);                 // 数组a的地址作为实参
    printf("a的内容是: %d, %d\n", a[0], a[1]);

    return 0;
}
```

### 运行结果：

a的内容是：3, 5  
b的内容是：5, 3  
a的内容是：5, 3

### 变量监测：



## ■ 返回值

- 若被调函数有返回值，则可以通过`return`语句向主调函数返回一个值
- `main`函数的返回值由操作系统接收

## ■ 无返回值

- 若函数类型为`void`类型，则函数没有返回值
- 函数可以没有`return`语句，执行完最后一条语句后，返回主调函数
- 若函数有`return`语句，则不带有返回值

## ■ 有返回值

- 若函数类型不是`void`类型，则函数有返回值
- 若函数定义时没有指定函数类型，则函数类型默认为`int`类型
- `return`语句带有表达式（包括常量、变量），其值用作返回值
  - `return`语句带有的表达式类型，应与函数类型相同或赋值兼容
  - 若二者类型不同，则将表达式的值转换为函数类型后再返回

## ■ 一般形式

```
return;  
return 表达式;
```

## ■ 说明

- 结束函数调用并返回主调函数
- 若函数有返回值，则将表达式的值返回给主调函数
- 若表达式类型与函数类型不同，则先将表达式的值转换为函数类型后再返回
- 若函数中有多个return语句，则执行其中任何一个时，函数即结束调用

```
void test(float data)  
{  
    if (data < 0)  
        return;    // 函数无返回值  
    ...  
}
```

```
float max(float a[])  
{  
    ...  
    return a[k]*2.0; // 表达式值转换为  
                    // float类型返回  
}
```



- 函数定义
- 函数调用
- 函数原型声明
- 深入理解函数

## ■ 一般形式

函数类型 函数名(形参类型 形参名, 形参类型 形参名, ...);  
函数类型 函数名(形参类型, 形参类型, ...);

## ■ 说明

- 函数原型声明, 简称**函数声明**, 函数应**“先声明, 后调用”**
- 函数原型声明包含函数调用和语法检查所需的所有信息
  - 函数名、函数类型、形参数量、形参类型、形参顺序
  - 函数调用不需要检查形参名, 函数原型声明可以省略形参名
- 若函数原型声明在**函数外部**, 则作用范围**从声明处至源程序文件结束**
- 若函数原型声明在**函数内部**, 则作用范围仅限于**从声明处至当前函数结束**
- 函数定义包含函数原型声明的内容, 可以起到函数原型声明的作用
- 库函数的原型声明在相应的头文件中, 正确**#include**头文件即可
- 函数原型声明习惯上放在预处理命令及结构体类型定义之后、函数定义之前

# 函数原型声明举例



## ■ 例4.2-7：编写函数判断素数。

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

int is_prime(int n);           // 函数原型声明

int main()
{
    int n;

    printf("输入一个整数: ");
    scanf("%d", &n);

    if (is_prime(n))           // 函数调用
        printf("%d是素数\n", n);
    else
        printf("%d不是素数\n", n);

    return 0;
}
```

```
int is_prime(int n)           // 函数定义
{
    int i, m;

    if (n <= 1)
        return 0;

    m = sqrt(n);

    for (i=2; i<=m; i++)
        if (n%i == 0)
            return 0;

    return 1;
}
```

### 运行结果一：

输入一个整数：97↵  
97是素数

### 运行结果二：

输入一个整数：100↵  
100不是素数

# 函数嵌套调用举例



## ■ 例4.2-8：编写函数求四个数的最大值。

```
#include <stdio.h>

int max4(int a, int b, int c, int d);
int max2(int a, int b);

int main()
{
    int a, b, c, d;

    printf("请输入4个整数: ");
    scanf("%d%d%d%d", &a, &b, &c, &d);
    printf("最大值: %d\n", max4(a, b, c, d));

    return 0;
}
```

```
int max4(int a, int b, int c, int d)
{
    int m;

    m = max2(a, b);           // 嵌套调用
    m = max2(m, c);
    m = max2(m, d);

    return m;
}

int max2(int a, int b)
{
    return (a > b ? a : b);   // 括号可以没有
}
```

### 运行结果一：

请输入4个整数：1 2 3 4↵  
最大值：4

### 运行结果二：

请输入4个整数：-33 29 81 67↵  
最大值：81



- 函数定义
- 函数调用
- 函数原型声明
- 深入理解函数

## ■ 多参数和多函数

- 程序可以包含多个函数，每个函数可以包含多个参数
- 函数之间存在相互调用关系
  - 一个函数可以调用多个函数
  - 一个函数可能被多个函数调用
- 自定义函数相对独立地实现模块功能
- 主函数关注总体流程调度

## ■ 数组用作函数参数

- 数组名表示数组存储空间的**起始地址**
- 实参数组将**地址值**传递给形参数组
- 形参数组名也表示**实参数组起始地址**，可以通过下标运算访问**实参数组元素**
- 形参数组实际上是**指针类型**，而不是真正的数组，所以可以接收实参传值

## ■ 例4.2-9：编写函数求三个电阻的串联阻值和并联阻值。

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

```
float series(float a, float b, float c);  
float parallel(float, float, float);
```

// 函数原型声明中的形参名可与定义不同  
// 函数原型声明也可以没有参数名

```
int main()  
{  
    float r1, r2, r3, rs, rp;  
    printf("请输入3个电阻值: ");  
    scanf("%f%f%f", &r1, &r2, &r3);  
    rs = series(r1, r2, r3);  
    rp = parallel(r1, r2, r3);  
    printf("串联阻值: %.2f, 并联阻值: %.2f\n", rs, rp);  
    return 0;  
}
```

```
float series(float r1, float r2, float r3)  
{  
    return (r1 + r2 + r3);  
}
```

```
float parallel(float r1, float r2, float r3)  
{  
    return (1.0 / (1.0 / r1 + 1.0 / r2 + 1.0 / r3));  
}
```

$$R_S = R_1 + R_2 + R_3$$

$$R_P = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$$

### 运行结果一：

请输入3个电阻值: 1 2 3↵  
串联阻值: 6.00, 并联阻值: 0.55

### 运行结果二：

请输入3个电阻值: 2.3 3.4 4.5↵  
串联阻值: 10.20, 并联阻值: 1.05

# 一维数组用作函数参数举例



## ■ 例4.2-11: 编写函数求学生平均成绩。

```
#include <stdio.h>
float average(float array[], int n);

int main()
{
    float score_1[10]={67,87,90,93,74,65,78,88,99,78};
    float score_2[ 5]={98,76,87,83,95};
    printf("aver1=%5.1f\n", average(score_1, 10));
    printf("aver2=%5.1f\n", average(score_2,  5));
    return 0;
}

float average(float array[], int n)
{
    int i;
    float aver, sum=array[0];
    for (i=1; i<n; i++)
        sum = sum + array[i];
    aver = sum / n;
    return aver;
}
```

### 程序解析:

- 1** 函数原型和定义中, 形参数组实际上是指针类型, 数组长度将被编译器忽略, 可以省略不写, 但是[]表示形参类型, 不可省略;
- 2** 实参数组仅使用数组名表示, 不需要带有[]和数组长度, 其值为实参数组的起始地址;
- 3** 形参数组是对应的指针类型, 参数传递中不携带数组长度信息, 通常需要通过另一参数显式传递数组长度;
- 4** 形参分别通过参数传递获得实参数组的起始地址和长度, 即可以访问实参数组的所有元素。

### 运行结果:

```
aver1= 81.9
aver2= 87.8
```



# 多维数组用作函数参数举例



## ■ 例4.2-12: 求二维数组的最大元素。

```
#include <stdio.h>
int max_value(int array[][4], int m);

int main()
{
    int a[3][4]={{1,3,5,7},{2,4,6,8},{15,17,34,12}};
    printf("max is %d\n", max_value(a, 3));
    return 0;
}

int max_value(int array[][4], int m)
{
    int i, j, max;
    max = array[0][0];
    for (i=0; i<m; i++)
        for (j=0; j<4; j++)
            if (array[i][j] > max)
                max = array[i][j];

    return max;
}
```

### 程序解析:

- 1** 形参二维数组行数（第一维长度）可以省略，并以另一参数传递，列数（第二维长度）必须予以指定，不需要以参数传递；
- 2** 实参二维数组的列数必须与形参二维数组列数相同，在函数调用中，仅使用二维数组名表示，不需要带有[]和行列数。

### 运行结果:

max is 34

### 思考拓展:

当函数的形参是二维数组时，对应实参必须是与形参数组列数相同的二维数组，缺少灵活性。

请思考如何把形参和实参都改为一维数组来解决这个问题。

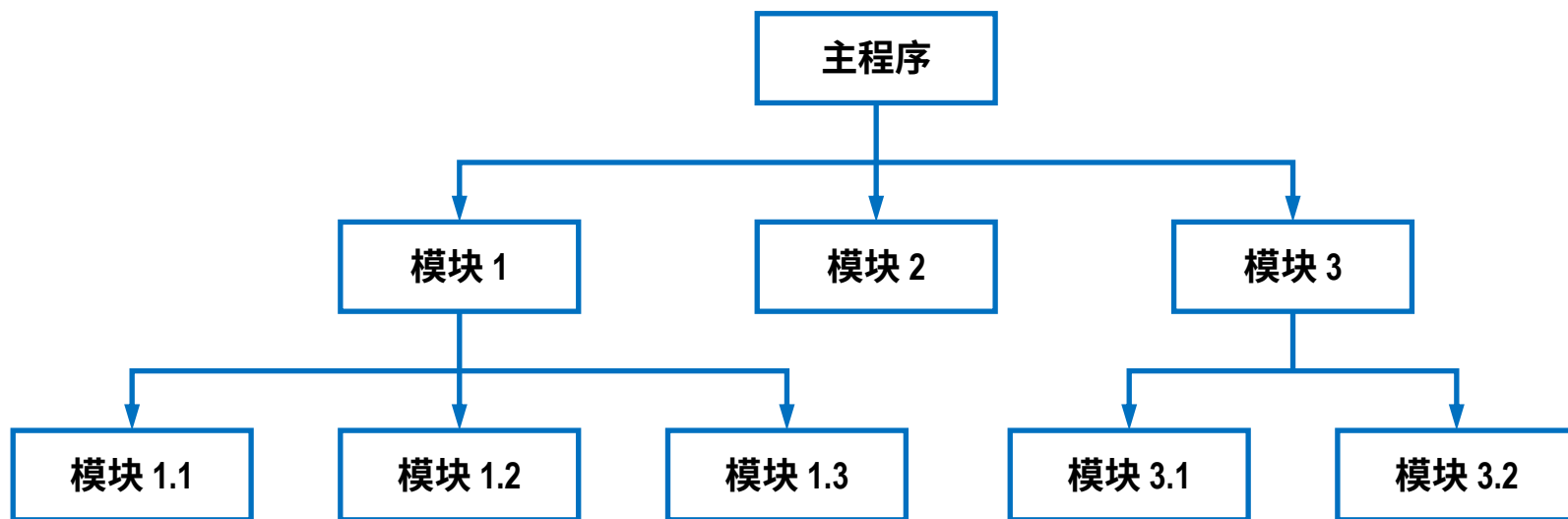


# 模块化设计与实践

- 自顶向下设计
- 变量的作用域与生存期
- 文件包含
- 库函数
- 递归

## ■ 自顶向下设计过程

- 按照软件需求进行功能分解，得到多个**模块**
- 设计模块之间的**接口**，即模块输入与输出
- 实现各模块的**功能**
- 将模块**组装**在一起构成可运行的软件系统

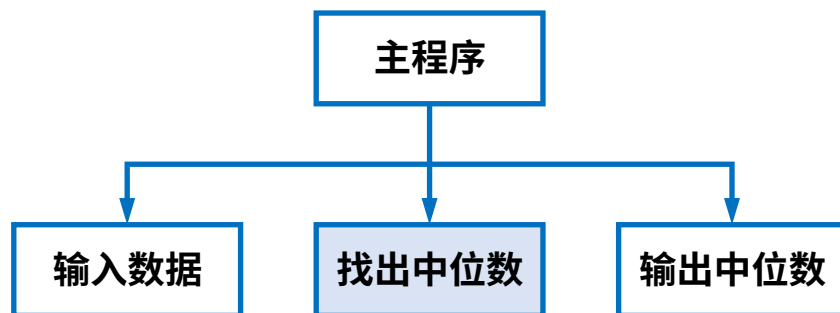


# 设计案例：求中位数——自顶向下分解系统



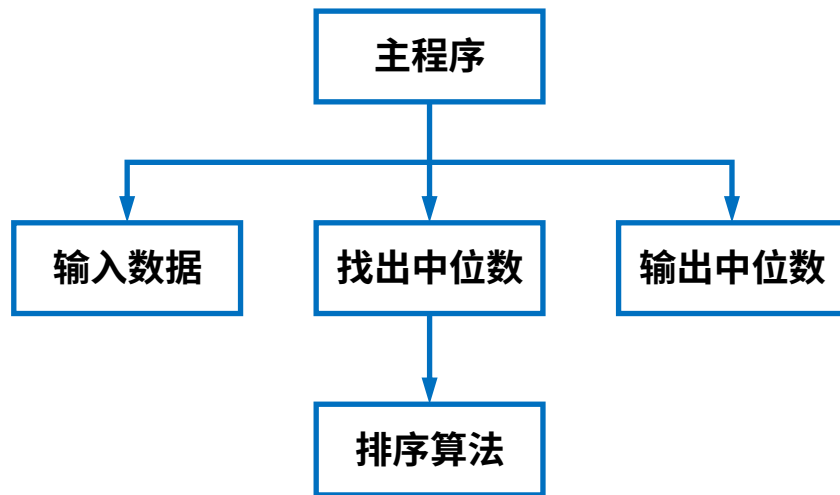
## ■ 顶层设计：描述总体流程

- 输入数据
  - 输入方式可能有键盘、文件等
- 找出中位数
  - 根据总体流程划分功能
- 输出中位数
  - 输出方式可能有屏幕、文件等



## ■ 第二层设计：继续分解功能

- 数据排序算法
  - 常用基本算法，可作为底层模块
- 定位中位数
  - 实现方法简单，不作为单独模块



# 设计案例：求中位数——设计模块接口



接口设计：设计函数头

输入接口：形参；输出接口：返回值

## ■ 输入数据函数

```
int input_data(int a[], int n);
```

- 输入：数组地址、数组长度
- 输出：实际输入数据数量

## ■ 输出中位数函数

```
void output_median(double m);
```

- 输入：中位数
- 输出：无

## ■ 找出中位数函数

```
double find_median(int a[], int n);
```

- 输入：数组地址、实际数据数量
- 输出：求得的中位数

## ■ 排序算法函数

```
void sort(int a[], int n);
```

- 输入：数组地址、实际数据数量
- 输出：无

# 设计案例：求中位数——自底向上实现系统



## ■ 例4.3-a：求给定的一组整数的中位数。

```
#include <stdio.h>

// 输入数据函数
int input_data(int a[], int n)
{
    int i;

    for (i=0; i<n; i++)    // 直接赋值简化功能
        a[i] = i + 2;    // 预留接口以便修改

    return n;
}

// 排序算法函数
void sort(int a[], int n)    // 空壳函数
                             // 在实现之前占位
{}

// 输出中位数函数
void output_median(double m)
{
    printf("中位数是%.1f", m);
}
```

```
// 找出中位数函数
double find_median(int a[], int n)
{
    sort(a, n);    // 调用排序算法

    if (n%2 == 0)
        return (a[n/2-1] + a[n/2]) / 2.0;
    else
        return a[n/2];
}

// 主函数
int main()
{
    int    a[10], n;
    double med;

    n    = input_data(a, 10);    // 输入数据
    med = find_median(a, n);    // 找出中位数
    output_median(med);    // 输出中位数
    return 0;
}
```

**运行结果：**

中位数是6.5

- 自顶向下设计
- 变量的作用域与生存期
- 文件包含
- 库函数
- 递归



## ■ 作用域 (Scope)

- 标识符可使用的**代码范围**
- 变量在作用域内才能被引用

## ■ 变量按照作用域分类

- 局部变量（内部变量）
  - 自动变量
  - 形式参数
  - 静态局部变量
- 全局变量（外部变量）

## ■ 生存期 (Lifetime)

- 数据在内存中存在的**运行时间**
- 变量在生存期内才存在并保有其值

## ■ 变量按照生存期分类

- 动态变量
  - 自动变量
  - 形式参数
- 静态变量
  - 静态局部变量
  - 全局变量

## ■ 局部变量

- 又称内部变量
- 在函数内部定义的变量
  - 复合语句（包括函数体）、分支和循环控制语句、函数定义中的形参列表

## ■ 说明

- 局部变量的作用域从变量定义处开始，至其定义所在的语句块结束为止
- 不同函数内的局部变量可以重名
  - 其作用域互不重叠，不会相互冲突
  - 函数定义时不需要考虑其他函数中的同名局部变量
- 同一函数内不同语句块中的局部变量可以重名
  - 若作用域发生重叠，内层局部变量屏蔽外层局部变量
- 局部变量不能为函数之间提供共享数据的渠道

## ■ 例4.3-1：复合语句中的局部变量。

```
#include<stdio.h>
int max_value(int array[][4], int);    // array不是局部变量
```

**运行结果：**  
max is 34

```
int main()
{
    int a[3][4]={{1,3,5,7},{2,4,6,8},{15,17,34,12}};
    printf("max is %d\n", max_value(a, 3));
    return 0;
}
```

局部数组a的作用域

```
int max_value(int array[][4], int m)
{
```

形参array, m的作用域

```
    int i, max;
```

局部变量i, max的作用域

```
    max = array[0][0];
```

```
    for (i=0; i<m; i++) {
```

// 复合语句开始

```
        int j;
```

// 复合语句中定义变量 局部变量j的作用域

```
        for (j=0; j<4; j++)
```

```
            if (array[i][j] > max)
```

```
                max = array[i][j];
```

```
    }
```

// 复合语句结束

```
    return max;
```

```
}
```

# 重名局部变量举例



## ■ 例4.3-2：重名局部变量的作用域。

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

```
int a1(int a)           // 函数形参也是局部变量    形参a的作用域
{
    a = 7;
    printf("函数a1中的a=%d\n", a);
    return a;
}
```

### 运行结果：

main头部的a=3  
在if{}里的a=9  
函数a1中的a=7  
在if{}后的a=7  
独立{}中的a=2

```
int main()
{
    // 函数中定义的3个a是3个不同的局部变量
    int a = 3;           // 函数体中定义局部变量    局部变量a的作用域
    printf("main头部的a=%d\n", a);
    if (a == 3) {        // 复合语句用作if语句的子句
        int a = 9;       // 复合语句中定义局部变量    局部变量a的作用域
        printf("在if{}里的a=%d\n", a);
    }
    a = a1(a);
    printf("在if{}后的a=%d\n", a);
    {                   // 独立的复合语句
        int a = 2;       // 复合语句中定义局部变量    局部变量a的作用域
        printf("独立{}中的a=%d\n", a);
    }

    return 0;
}
```

## ■ 全局变量

- 又称**外部变量**
- 在函数外部定义的变量

## ■ 说明

- 全局变量的作用域从变量定义处开始，至源程序文件结束为止
- 全局变量可以与局部变量重名
  - 在作用域重叠范围内，**局部变量屏蔽全局变量**
- 全局变量提供了函数之间共享数据的渠道
  - 全局变量作用域范围内的所有函数都可以访问该全局变量
  - 避免了函数之间通过参数和返回值传递数据的复杂操作
  - 各函数通过全局变量产生数据相关性，将降低函数通用性、可读性、可维护性
- 所有全局变量都是**静态变量**

## ■ 例4.3-3：使用全局变量和函数，求数组元素最大值、最小值和平均值。

```
#include <stdio.h>
#define N 10
float Max=0, Min=0;           // 全局变量，为其后所有函数共享           全局变量Max, Min的作用域

float average(float array[], int n)           形参array, n的作用域
{
    float sum, ave;                       局部变量sum, ave的作用域
    sum = Max = Min = array[0];
    for (int i=1; i<n; i++) {               局部变量i的作用域
        if (array[i] > Max)
            Max = array[i];
        if (array[i] < Min)
            Min = array[i];
        sum = sum + array[i];
    }
    ave = sum / n;
    return ave;
}

int main()
{
    float score[N]={67,78,54,98,45,88,83,76,90,92}, ave;           局部变量score, ave的作用域
    ave = average(score, N);
    printf("Max=%.2f Min=%.2f Ave=%.2f", Max, Min, ave);
    return 0;
}
```

**运行结果：**  
Max=98.00 Min=45.00 Ave=77.10

# 全局变量与局部变量举例



## ■ 例4.3-4：全局变量与局部变量的作用域。

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

```
int m=10, n=5, x=1, y=3;
```

全局变量m, n, x, y的作用域

```
int max(int x, int y)    // 形参x, y与全局变量重名
{
    return x > y ? x : y; // 形参x, y屏蔽全局变量x, y
}
```

形参x, y的作用域

```
int z;    // 函数max()不能访问全局变量z
```

全局变量z的作用域

```
int main()
{
```

```
    int m=7;    // 局部变量m与全局变量重名
```

局部变量m的作用域

```
    z = max(m, n);    // 全局变量z, n以及局部变量m
    printf("max=%d\n", z); // 全局变量z
```

```
    return 0;
```

```
}
```

**运行结果：**

max=7

# 全局变量运算逻辑举例



## ■ 例4.3-5：全局变量的运算逻辑。

```
#include <stdio.h>

int a = 3; // 全局变量定义及初始化

void test()
{
    printf("test开始: a=%d\n", a);
    a++;
}

int main()
{
    printf("main开始: a=%d\n", a);
    a *= 4;
    test();
    printf("test结束: a=%d\n", a);

    return 0;
}
```

### 程序解析：

- 1 程序中各个函数引用的变量a都是全局变量a；
- 2 需要注意所有函数都可以引用全局变量并修改其值。

### 变量监测：

a	
3	int a=3;
↓	
12	main(): a*=4;
↓	
13	test(): a++;

### 运行结果：

main开始: a=3  
test开始: a=12  
test结束: a=13



## ■ 静态变量

- 程序加载到内存时分配存储空间，并保留到程序结束
- 对静态变量的修改将会保持到下次修改之前
- 静态变量**仅进行一次初始化**
  - 静态变量若未指定初值，则**默认初值为零**
- 静态局部变量：使用**static**关键字修饰的局部变量
  - 离开作用域后，仍保持存储空间不被释放，其值也不会丢失
  - 当再次调用函数时，静态局部变量**不重新初始化**，仍保持上次修改后的值
- 全局变量：所有全局变量都是静态变量

## ■ 动态变量

- 在每次函数调用时**重新分配存储空间和初始化**，函数返回即释放
  - 动态变量若未指定初值，则**初值不确定**
- 自动变量：使用**auto**关键字修饰的局部变量，是局部变量的缺省类型
- 形式参数：所有形式参数都是动态局部变量，不能使用**static**关键字修饰

# 静态局部变量举例



## ■ 例4.3-6：编写函数求阶乘，并使用静态局部变量记录函数调用次数。

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

```
void fact(int m)
{
    long    product=1;
    int     i; // 未初始化自动变量初值不定
    static int j; // 静态局部变量默认初值为0

    for (i=1; i<=m; i++)
        product *= i;
    printf("第%d次: %ld\n", j+++1, product);
}
```

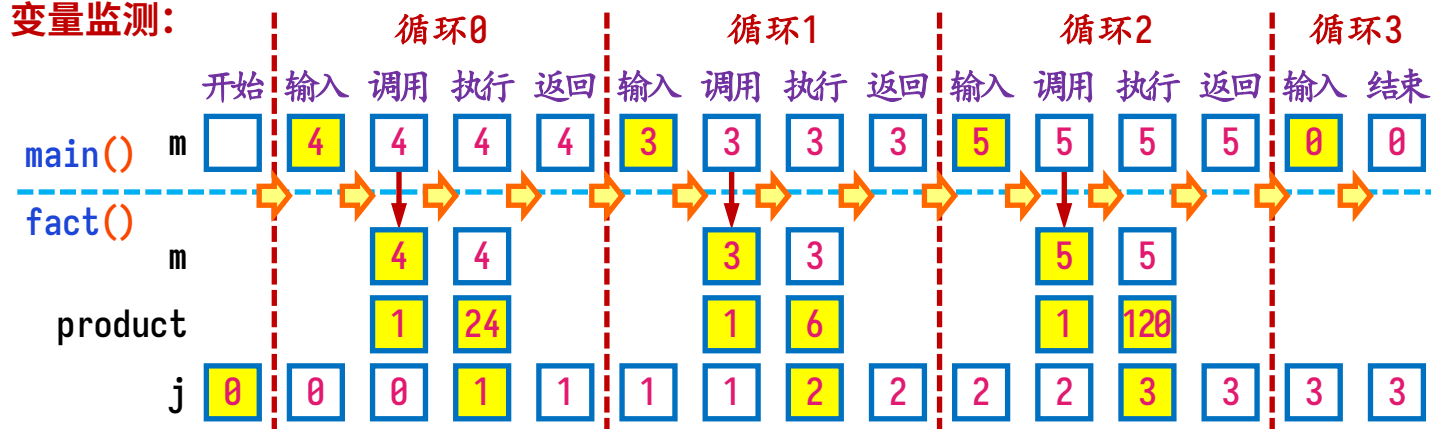
```
int main()
{
    int m;

    while (1) { // 循环条件为真
        scanf("%d", &m);
        if (m > 0)
            fact(m);
        else
            return 0; // 循环在此处结束
    }
}
```

### 运行结果：

```
4↵
第1次： 24
3↵
第2次： 6
5↵
第3次： 120
0↵
```

### 变量监测：



# 静态变量与动态变量举例



## ■ 例4.3-7：静态局部变量与自动变量的比较。

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

```
int f(int a)
```

```
{
```

```
    int      b=0;
```

```
    static int c=3; // 静态局部变量定义及初始化
```

```
    b = b + 1;
```

```
    c = c + 1;      // 静态局部变量保留上次调用所做的修改
```

```
    return (a + b + c);
```

```
}
```

```
int main()
```

```
{
```

```
    int a=2;
```

```
    for (int i=0; i<3; i++)  
        printf("%4d", f(a));
```

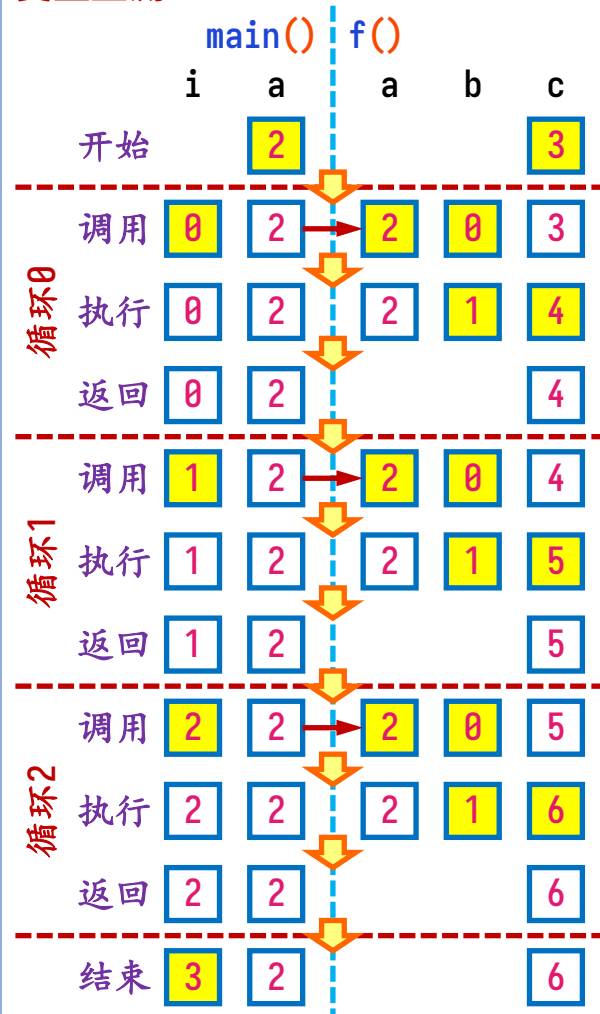
```
    return 0;
```

```
}
```

运行结果：

7    8    9

变量监测：





# 静态局部变量与全局变量举例

## ■ 例4.3-8：静态局部变量与全局变量的比较。

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

```
int var_static=8;
```

全局变量var\_static的作用域

```
void auto_static()  
{
```

```
    int var_auto=0;
```

自动变量var\_auto的作用域

```
    static int var_static;
```

静态局部变量var\_static的作用域

```
    printf("var_auto=%d, ", ++var_auto);
```

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

```
}
```

```
int main()  
{
```

```
    for (int i=0; i<3; ++i) {
```

自动变量i的作用域

```
        auto_static();
```

```
        printf("main中的var_static=%d\n", ++var_static);
```

```
    }
```

```
    return 0;
```

```
}
```

### 运行结果：

var\_auto=1, var\_static=1

main中的var\_static=9

var\_auto=1, var\_static=2

main中的var\_static=10

var\_auto=1, var\_static=3

main中的var\_static=11

# 程序存储空间举例



## ■ 例4.3-9：程序存储空间举例。

```
#include <stdio.h>
double glb;

void addr(int x, int y)
{
    static int st;
    printf("st      : %p\n", &st);
    printf("glb     : %p\n", &glb);
    printf("x       : %p\n", &x);
    printf("y       : %p\n", &y);
}

int main()
{
    int i;
    char str[8]="USTC";
    printf("addr   : %p\n", addr);
    printf("main   : %p\n", main);
    printf("\"USTC\": %p\n", "USTC");
    addr(i, i*i);
    printf("str    : %p\n", str);
    printf("i      : %p\n", &i);
    return 0;
}
```

运行结果：(32位编译环境)

函数	addr	: 00401500
函数	main	: 00401556
字符串常量	"USTC"	: 00404048
静态局部变量	st	: 00405020
全局变量	glb	: 00405430
形式参数	x	: 0061FE80
形式参数	y	: 0061FE84
自动数组	str	: 0061FE94
自动变量	i	: 0061FE9C

0x00401000	Text: 代码区
0x00403000	Data: 常量、已初始化的全局和静态变量
0x00405000	BSS: 未初始化的全局和静态变量
	Heap: 动态存储分配
	堆 ↓
	栈 ↑
	Stack: 形参、自动变量
0x0061FFFF	



- 自顶向下设计
- 变量的作用域与生存期
- 文件包含
- 库函数
- 递归

## ■ 命令格式

```
#include <文件名>  
#include "文件名"
```

## ■ 说明

- 预处理器使用被包含文件的全部内容替换文件包含命令
  - 被包含文件通常是以.h为扩展名的头文件，也可以是其他文本文件
  - 文件名可以带有绝对路径或相对路径，也可以不带路径
  - 文件名的扩展名可以是任意类型，也可以没有扩展名
- 系统头文件一般使用<>
  - 预处理器在系统配置的目录中查找头文件
  - 系统头文件内容包括：库函数声明、宏定义、结构体类型定义、数据类型定义等
- 自定义头文件应使用""
  - 预处理器先在源程序目录下查找头文件，若找不到，再到系统配置的目录中查找
  - 若程序包含多个源程序文件，全局变量与函数较多，通常需要自定义头文件

## ■ 例4.3-b：多文件程序和文件包含举例。

```
/* my.h */  
int sum(int m, int n);           // 函数原型声明
```

```
/* my.c */  
int sum(int m, int n)           // 计算从m到n范围内整数的累加和  
{  
    int i, sum=0;  
    for (i=m; i<=n; i++)  
        sum += i;  
    return sum;  
}
```

```
/* main.c */  
#include <stdio.h>               // 包含系统头文件，内容包括库函数原型声明、宏定义等  
#include "my.h"                 // 包含自定义头文件，内容包括函数原型声明  
  
int main()  
{  
    printf("%d\n", sum(1,100));  // 函数已声明且在其他源程序文件中定义  
    return 0;  
}
```

**运行结果：**

5050



- 自顶向下设计
- 变量的作用域与生存期
- 文件包含
- 库函数
- 递归

头文件	描 述
<code>assert.h</code>	定义断言宏，用于在程序的调试版本中辅助检测逻辑错误以及其他类型的问题
<code>ctype.h</code>	定义字符分类函数、字符大小写转换函数
<code>errno.h</code>	定义通过错误代码回报错误信息的宏
<code>float.h</code>	定义用于浮点库特定实现属性的宏常量
<code>limits.h</code>	定义用于整数库特定实现属性的宏常量
<code>locale.h</code>	定义本地化函数
<code>math.h</code>	定义数学函数
<code>setjmp.h</code>	定义非局部跳转函数
<code>signal.h</code>	定义信号处理函数
<code>stdarg.h</code>	定义可变参数函数所需的数据类型与宏
<code>stddef.h</code>	定义若干常见的类型与宏
<code>stdio.h</code>	定义输入输出函数
<code>stdlib.h</code>	定义数值转换函数、伪随机数生成函数、动态内存分配函数、过程控制函数
<code>string.h</code>	定义字符串处理函数
<code>time.h</code>	定义日期与时间处理函数



# 格式化输入函数

## ■ 函数原型

```
int scanf(const char *format, ...);  
int fscanf(FILE *stream, const char *format, ...);
```

## ■ 参数

- `FILE *stream` – 输入文件流指针
- `const char *format` – 指向格式字符串的指针
- `...` – 输入参数地址列表

## ■ 返回值

- 成功赋值的输入参数数量
- 若在首个输入参数赋值前匹配失败，则返回0
- 若在首个输入参数赋值前输入失败，则返回EOF
- 符号常量EOF在头文件stdio.h中定义为(-1)

```
#define EOF (-1)
```



# 格式化输出函数

## ■ 函数原型

```
int printf(const char *format, ...);  
int fprintf(FILE *stream, const char *format, ...);
```

## ■ 参数

- `FILE *stream` – 输出文件流指针
- `const char *format` – 指向格式字符串的指针
- `...` – 输出参数列表

## ■ 返回值

- 输出的字符数
- 若出现输出错误，则返回负值



## 格式化输入输出函数的格式字符串

函数	格式字符串				
	普通字符		格式转换指示字符		
	非空白字符	空白字符	引导字符	附加格式字符 (可选)	格式字符
格式化输入 scanf() fscanf()	准确匹配 同一字符	匹配连续的 零个、一个 或多个 任意空白字符	%	最大宽度 数据类型长度	数据类型
格式化输出 printf() fprintf()	无更改输出			格式调整 最小宽度 精度 数据类型长度	

# 格式化输入函数格式字符



格式字符	数据类型	缺省格式说明（无附加格式字符）
c	字符	一个字符
s	字符指针	非空白字符序列，并在字符序列结尾添加一个空字符
d	有符号整数	可带有符号的十进制整数
i		可带有符号的八进制、十进制或十六进制整数，以前缀判定进制
u	无符号整数	无符号十进制整数，若输入负数以补码存储
o		无符号八进制整数，可带或不带前缀0，若输入负数以补码存储
x, X		无符号十六进制整数，可带或不带前缀0x或0X，若输入负数以补码存储
f, F e, E g, G	浮点数	十进制小数形式或指数形式单精度浮点数
p	指针	与具体实现有关，一般为十六进制整数形式
%		字符%

# 格式化输入函数附加格式字符



附加格式字符	类型	常用附加格式说明
*		赋值抑制，数据不赋值给任何接收参数
m	宽度	正整数，输入最大宽度
hh	数据长度	有符号或无符号字符型（作为整数输入） (C99)
h		有符号或无符号短整型
l		有符号或无符号长整型、双精度浮点型
ll		有符号或无符号双长整型 (C99)
L		长双精度浮点型

# 格式化输出函数格式字符



格式字符	数据类型	缺省格式说明（无附加格式字符）
c	整数类型	转换为无符号字符型，并以此为ASCII值输出相应字符
s	字符指针	从字符指针指向的字符开始，输出字符串直至但不包括首个空字符
d, i	有符号整数	有符号十进制整数，正数不输出正号
u	无符号整数	无符号十进制整数
o		无符号八进制整数，不输出前缀0
x, X		无符号十六进制整数，不输出前缀0x或0X
f, F	浮点数	小数形式单精度或双精度浮点数，小数点后6位
e, E		规范化指数形式单精度或双精度浮点数，小数点后6位
g, G		小数形式或规范化指数形式单精度或双精度浮点数，依赖于值和精度
p	指针	与具体实现有关，一般为十六进制整数形式
%		字符%



# 格式化输出函数附加格式字符



附加格式字符	类型	常用附加格式说明
-	格式	左对齐（缺省：右对齐）
+		有符号数始终输出前置符号（缺省：正数不输出正号）
空格		有符号数若不输出前置符号，则前置空格
#		八进制和十六进制整数输出前缀，浮点数不输出小数部分时也输出小数点
0		整数和浮点数右对齐时，以前导0填充至宽度要求（缺省：空格填充）
m	宽度	正整数，输出最小宽度，不足部分以空格填充
.n	精度	正整数，字符串最大输出字符数、整数最小位数、浮点数小数点后位数
hh	数据长度	有符号或无符号字符型（作为整数输出）(C99)
h		有符号或无符号短整型
l		有符号或无符号长整型
ll		有符号或无符号双长整型 (C99)
L		长双精度浮点型

## ■ 行缓冲机制

- 输入输出的字符先存放于内存中的缓冲区(Buffer)
- 当遇到换行符、缓冲区满、程序结束等时，再进行实际输入输出操作
- 标准输入函数：输入换行符后，才开始接收数据，残余字符影响后续输入
- 标准输出函数：输出换行符时，才开始输出数据
  - 部分编译器的标准输出采用无缓冲机制

## ■ 清除标准输入缓冲区残余字符

```
fflush(stdin);    // 清空标准输入缓冲区，非标准定义行为，部分编译器不支持
scanf(" %c", &c); // 输入数据时，清除连续空白字符
getchar();        // 清除单个字符，可多次或循环使用
```

## ■ 从键盘输入EOF

- Windows系统：空行行首输入Ctrl-Z
- Linux系统：空行行首输入Ctrl-D

# 格式化输入输出函数举例



## ■ 例4.3-10/11：格式化输入输出函数举例。

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

```
int main()  
{
```

```
    int    a, f;  
    char   b;  
    float  c;  
    double d;  
    char   e[10];
```

```
    f = scanf("%d %c %f %lf %s", &a, &b, &c, &d, e);  
    printf("%d %d %c %f %f %s", f, a, b, c, d, e);
```

```
    printf("\n\n");  
    printf("%d", printf("%d %c %f %s\n", 1, '2', 3.0, "4")); // 输出字符数包括回车符
```

```
    return 0;
```

```
}
```

运行结果:

```
1 2 3 4 567  
5 1 2 3.000000 4.000000 567  
  
1 2 3.000000 4  
15
```

// 返回值为成功输入的数据个数

// 输出字符数包括回车符

## ■ 例4.3-c：格式字符串举例。

```
#include <stdio.h>

int main(void)
{
    const char s[] = "Hello";

    printf("字符串: \n");
    printf("\t.%.10s.\n\t.%.10s.\n", s, s);

    printf("\n字符: \t%c %%\n", 65);

    printf("\n整数: \n");
    printf("十进制: %i %d %+i %u\n", 1, 2, 3, -1);
    printf("八进制: %o %#o %#o\n", 10, 10, 4);
    printf("16进制: %x %x %X %#x\n", 5, 10, 10, 6);

    printf("\n浮点数: \n");
    printf("舍入: \t%f %.0f\n", 1.5, 1.5);
    printf("填充: \t%05.2f %.2f %5.2f\n", 1.5, 1.5, 1.5);
    printf("科学: \t%.2E %.2e\n", 1.5, 1.5);

    return 0;
}
```

## 运行结果:

字符串:

```
.      Hello.
.Hello      .
```

字符: A %

整数:

十进制: 1 2 +3 4294967295

八进制: 12 012 04

16进制: 5 a A 0x6

浮点数:

舍入: 1.500000 2

填充: 01.50 1.50 1.50

科学: 1.50E+000 1.50e+000



# 字符输入输出函数

## ■ 字符输入函数

```
int getchar(void);
```

- 从标准输入设备输入一个字符
- 输入成功时，返回获得的字符
- 输入失败时，返回EOF

## ■ 字符输出函数

```
int putchar(int ch);
```

- 将ch的值转换为unsigned char类型，并输出相应字符至标准输出设备
- 输出成功时，返回输出的字符
- 输出失败时，返回EOF



# 字符串输入输出函数

## ■ 字符串输入函数

```
char *gets(char *str);
```

- 从标准输入设备输入可带有空格和制表符的字符串直至遇到回车
- 将回车符替换为空字符后，将输入字符串保存到str指向的字符数组中
- 若输入字符串长度超过str指向的字符数组长度，则发生**数组越界**
  - 由于安全性问题，C11标准已将此函数移除，**不建议使用**
- 输入成功时，返回str的值；输入失败时，返回空指针**NULL**
- 符号常量**NULL**在头文件**stdio.h**中定义为空指针，其值通常为**0**

```
#define NULL ((void *)0)
```

## ■ 字符串输出函数

```
int puts(const char *str);
```

- 将str指向的字符串末尾空字符替换为回车符，并输出字符串至标准输出设备
- 输出成功时，返回非负值；输出失败时，返回**EOF**

# 字符串输入输出举例：单词统计



## ■ 例4.3-12：统计一行字符中用空格或制表符分隔的单词个数。

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

```
#define IN 1
```

```
#define OUT 0
```

```
int main()
```

```
{
```

```
    char c, str[255];
```

```
    int i, num=0;
```

```
    int state=OUT;
```

```
    gets(str);
```

```
    for (i=0; c=str[i]; i++)
```

```
        if (c==' ' || c=='\t')
```

```
            state = OUT;
```

```
        else if (state == OUT) {
```

```
            num++;
```

```
            state = IN;
```

```
        }
```

```
    printf("字符串中有%d个单词", num);
```

```
    return 0;
```

```
}
```

```
// 状态标识，表示在单词内
```

```
// 状态标识，表示在单词外
```

```
// 存放输入字符串
```

```
// 单词个数计数
```

```
// 初始状态为在单词外
```

```
// 输入一行字符串
```

```
// 遇到空字符时，循环结束
```

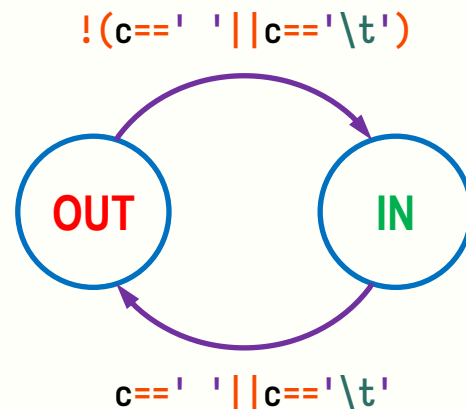
```
// 若读到单词分隔符
```

```
// 则将状态置为在单词外
```

```
// 若当前状态在单词外，且读到非分隔字符
```

```
// 则进入新单词，单词计数自增
```

```
// 改变状态为在单词内，以免重复计数
```



**运行结果：**

C Programming Language↵

字符串中有3个单词

## ■ 字符串复制函数

```
char *strcpy(char *dest, const char *src);
```

- 将src指向的字符串复制到dest指向的字符数组
- dest指向的数组长度应能够容纳复制的字符串
- 返回值为dest的值

## ■ 字符串连接函数

```
char *strcat(char *dest, const char *src);
```

- 将src指向的字符串复制到dest指向的字符串末尾
- src[0]覆盖dest指向的字符串末尾的空字符
- dest指向的数组长度应能够容纳连接后的字符串
- 返回值为dest的值





# 字符串处理函数

## ■ 字符串比较函数

```
int strcmp(const char *lhs, const char *rhs);
```

- 比较lhs和rhs指向的字符串
- 由前向后依次比较两个字符串中的字符，直至遇到不同字符或空字符
- 返回值的符号是两个字符串中首对不同字符的ASCII值之差的符号

## ■ 求字符串长度函数

```
size_t strlen(const char *str);
```

- 求字符串长度，不包括字符串末尾的空字符
- 返回值为字符串长度
- 返回值类型size\_t通常在头文件中定义为unsigned int类型的别名

```
typedef unsigned int size_t;
```

## ■ 例4.3-13: 字符串处理函数举例。

```
#include <stdio.h>
#include <string.h>                                     // 字符串处理函数声明所在头文件

int main()
{
    char src[20];                                       // 源字符串
    char dest[50];                                     // 目标字符串, 大小不小于源字符串
    int ret;

    strcpy(src, "www.ustc.edu.cn");                    // 将字符串常量复制到src
    strcpy(dest, src);                                 // 将src中的字符串复制到dest
    printf("%s\n", strcat(dest, src));                 // 将src中的字符串连接到dest尾部

    ret = strcmp(dest, src);                            // 比较两个字符串
    printf("%s\n%d\t%d", src, strlen(dest), ret);

    return 0;
}
```

### 运行结果:

```
www.ustc.edu.cnwww.ustc.edu.cn
www.ustc.edu.cn
30      1
```

## ■ 随机数函数

```
int rand(void);  
void srand(unsigned int seed);
```

- 函数`rand()`返回`0~RAND_MAX`之间的伪随机整数值
- 函数`srand()`以参数`seed`设置`rand()`中伪随机数发生器的种子
- 符号常量`RAND_MAX`为`rand()`的最大返回值，其值与实现有关，至少为`32767`

```
#define RAND_MAX 0x7FFF
```

## ■ 时间函数

```
time_t time(time_t *arg);
```

- 返回自协调世界时1970年1月1日0时0分0秒起至当前时间的秒数
- 若`arg`不是空指针，返回值也将保存于`arg`指向的对象中
- 数据类型`time_t`在头文件`time.h`中定义为足以表示时间的类型的别名

```
typedef long long int time_t;
```

# 随机数函数与时间函数举例



## ■ 例4.3-14：以当前时间为种子产生伪随机数。

```
#include <stdio.h>
#include <stdlib.h>           // 随机数函数声明所在头文件
#include <time.h>             // 时间函数声明所在头文件

int main()
{
    int i;

    srand((unsigned)time(NULL)); // 以当前时间作为伪随机数发生器种子
    for (i=5; i>0; i--)
        printf("%5d\t", rand()); // 连续产生伪随机数并输出

    return 0;
}
```

### 运行结果一：

4975    12336    12047    23806    5569

### 运行结果二：

5929    5143    18273    5173    2769

### 运行结果三：

6223    22227    20416    8183    13576

### 运行结果四：

6265    30885    23275    26094    16594

- 自顶向下设计
- 变量的作用域与生存期
- 文件包含
- 库函数
- 递归



*Recursive Dolls: the Original Set of  
Matryoshka Dolls, 1892*

## ■ 递归调用

- 函数直接或间接地调用自身
- 直接递归:  $A()$ 调用 $A()$
- 间接递归:  $A()$ 调用 $B()$ ,  $B()$ 调用 $C()$ ,  $C()$ 调用 $A()$

## ■ 有效递归的基本条件

- 有一个或多个能够终止递归的情形
- 在递归过程中, 问题规模将不断缩小, 直至达到某个终止递归的条件

## ■ 递归的特点

- 适用于有递归属性的问题, 逻辑清晰, 代码简洁
- 计算效率较低, 内存消耗较大

# 递归举例：阶乘



## ■ 例4.3-15：递归求阶乘。

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

```
int fac(int n)
{
    if (n < 0) {                // 数据合法性检查
        printf("n<0, data error!");
        return -1;
    }

    if (n==0 || n==1)           // 递归终止条件
        return 1;
    else
        return fac(n-1)*n;      // n!=(n-1)!*n
}
```

```
int main()
{
    int n;

    printf("输入一个正整数: ");
    scanf("%d", &n);
    printf("%d!=%d", n, fac(n));

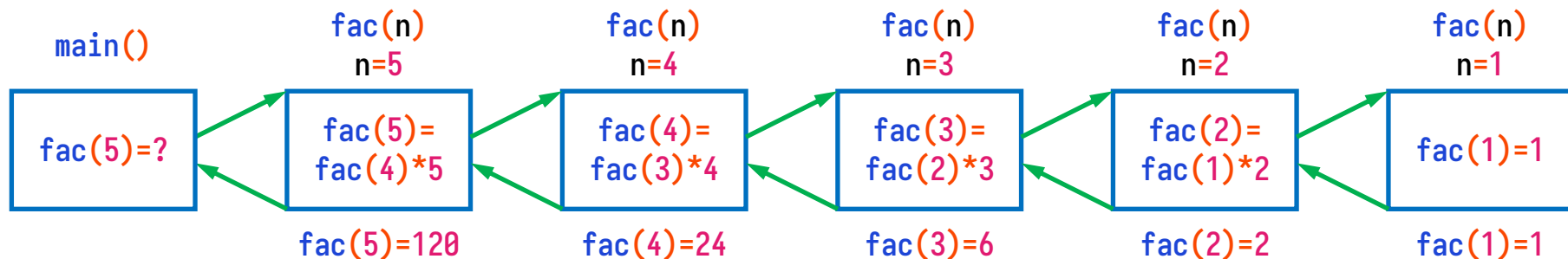
    return 0;
}
```

### 运行结果一：

输入一个正整数：5↵  
5!=120

### 运行结果二：

输入一个正整数：12↵  
12!=479001600





# 模块化与计算思维实践





- 数据与操作
- 排序与查找

## ■ 计算机解决问题的方法

- 将问题中的数据表示出来
- 设计处理数据的算法过程
- 确定问题涉及的数据及对数据的操作

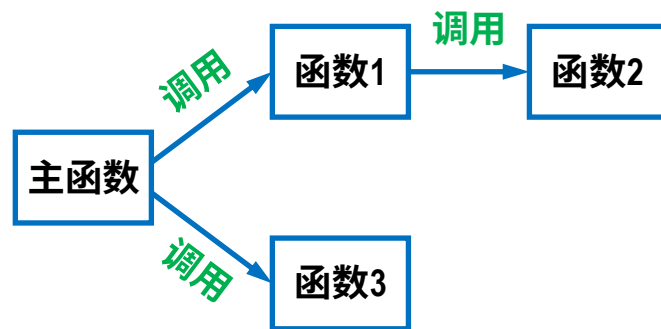
## ■ 程序设计思想与方法

### ■ 面向过程

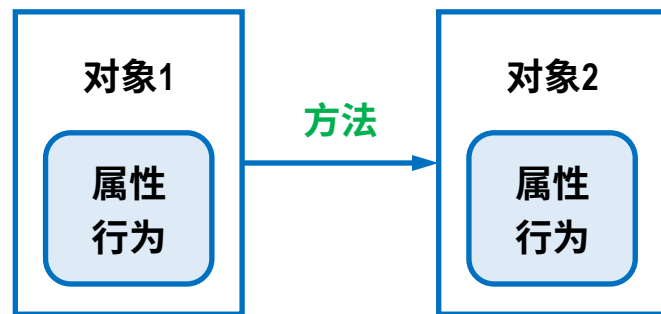
- 以操作为中心，数据与操作分离
- 数据仅表示信息，不表达操作
- 操作驱动程序实现特定功能

### ■ 面向对象

- 以数据为中心，数据与操作不可分离
- 数据与操作结合起来形成对象(Object)
- 对象掌控对自身数据的处理方法



面向过程的观点



面向对象的观点

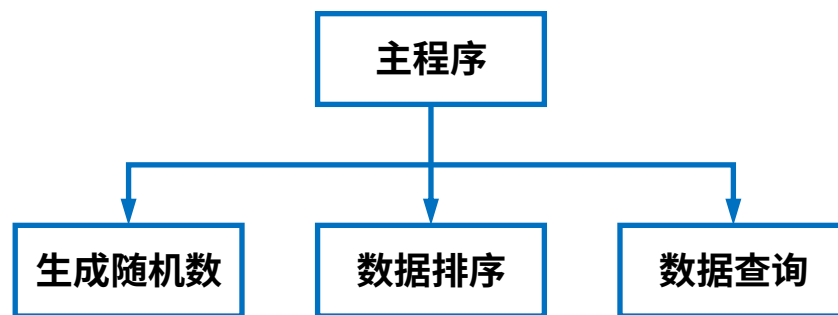


- 数据与操作
- 排序与查找

## ■ 例4.4-1：生成随机整数数组并进行排序和查找。

### ■ 模块分解：

- ① **生成随机数**：利用随机数库函数产生随机数据并存储在数组中
- ② **数据排序**：按照常用排序算法对数组进行升序或降序排序
- ③ **数据查询**：按照常用查找算法在数组中查找给定数据



系统模块分解图

## ■ 生成随机数函数

```
void RandomIntArray(int num, int a[]);
```

- 输入：数组大小、数组地址
- 扩展：可增加数据上下界参数

## ■ 数据排序函数

```
void SortArray(int num, int a[]);
```

- 输入：数组大小、数组地址
- 实现：使用不同的函数名分别实现不同的排序算法

## ■ 数据查询函数

```
int SearchInArray(int num, int a[], int snum);
```

- 输入：数组大小、数组地址、查询数据
- 输出：若查找到，返回元素下标；若未查找到，返回-1
- 实现：使用不同的函数名分别实现不同的查找算法

# 主函数和生成随机数模块的设计实现



## ■ 例4.4-1：生成随机整数数组并进行排序和查找——主函数和生成随机数。

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

#define N 10                // 数组大小

int main()
{
    int a[N], i, snum, pos;

    RandomIntArray(N, a);    // 生成随机整数
    printf("生成的随机数组元素如下: \n");
    PrintArray(N, a);

    SortSelect(N, a);        // 选择排序
    printf("\n排序后的数组元素如下: \n");
    PrintArray(N, a);        // 输出数组

    printf("\n请输入要查询的数: \n");
```

```
scanf("%d", &snum);
if ((pos=SearchBinary(N, a, snum)) >= 0)
    printf("%d是数组的第%d个元素!",
           snum, pos);    // 二分查找
else
    printf("%d不是数组的元素!", snum);

    return 0;
}
```

```
void RandomIntArray(int num, int a[])
{
    int i;

    srand(time(NULL)+rand()); // 随机数种子

    for (i=0; i<num; i++)
        a[i] = rand() % 100; // 范围: 0~99
}
```

# 数据排序模块的设计实现——交换排序



## ■ 例4.4-1：生成随机整数数组并进行排序和查找——交换排序。

### ■ 算法描述： ( $n$ 个数据升序排列)

- ① **第0轮**：将第0个数据与后续数据逐个比较，若顺序不符，则交换这两个数据；
- ② **第1轮**：将第1个数据与后续数据重复以上比较和交换操作；
- ③ 重复以上操作 $n-1$ 轮，排序完毕。

```
void SortExchange(int num, int a[])
{
    int i, j, t;

    for (i=0; i<num-1; i++)
        for (j=i+1; j<num; j++)
            if (a[i] > a[j])
                t=a[i], a[i]=a[j], a[j]=t;
}
```

变量监测：SortExchange(6, a)

i		0	1	2	3	4
a[0]	32	14	14	14	14	14
a[1]	25	32	25	25	25	25
a[2]	87	87	87	32	32	32
a[3]	55	55	55	87	33	33
a[4]	33	33	33	55	87	55
a[5]	14	25	32	33	55	87

j		1	2	3	4	5
a[0]	32	25	25	25	25	14
a[1]	25	32	32	32	32	32
a[2]	87	87	87	87	87	87
a[3]	55	55	55	55	55	55
a[4]	33	33	33	33	33	33
a[5]	14	14	14	14	14	25

# 数据排序模块的设计实现——选择排序



## ■ 例4.4-1：生成随机整数数组并进行排序和查找——选择排序。

### ■ 算法描述： ( $n$ 个数据升序排列)

- ① **第0轮**：从第0个数据开始与后续数据逐个比较，若顺序不符，则记录该数据位置，并使用该数据与后续数据比较；本轮结束后，将记录的数据与第0个数据交换；
- ② 重复以上操作 $n-1$ 轮，排序完毕。

```
void SortSelect(int num, int a[])
{
    int i, j, t, k;
    for (i=0; i<num-1; i++) {
        k = i;
        for (j=i+1; j<num; j++)
            if (a[k] > a[j]) k = j;
        t=a[i], a[i]=a[k], a[k]=t;
    }
}
```

变量监测：SortSelect(6, a)

i		0	1	2	3	4
a[0]	32	14	14	14	14	14
a[1]	25	25	25	25	25	25
a[2]	87	87	87	32	32	32
a[3]	55	55	55	55	33	33
a[4]	33	33	33	33	55	55
a[5]	14	32	32	87	87	87

j		1	2	3	4	5
k	0	1	1	1	1	5
a[0]	32	32	32	32	32	32
a[1]	25	25	25	25	25	25
a[2]	87	87	87	87	87	87
a[3]	55	55	55	55	55	55
a[4]	33	33	33	33	33	33
a[5]	14	14	14	14	14	14



# 数据排序模块的设计实现——冒泡排序



## ■ 例4.4-1：生成随机整数数组并进行排序和查找——冒泡排序。

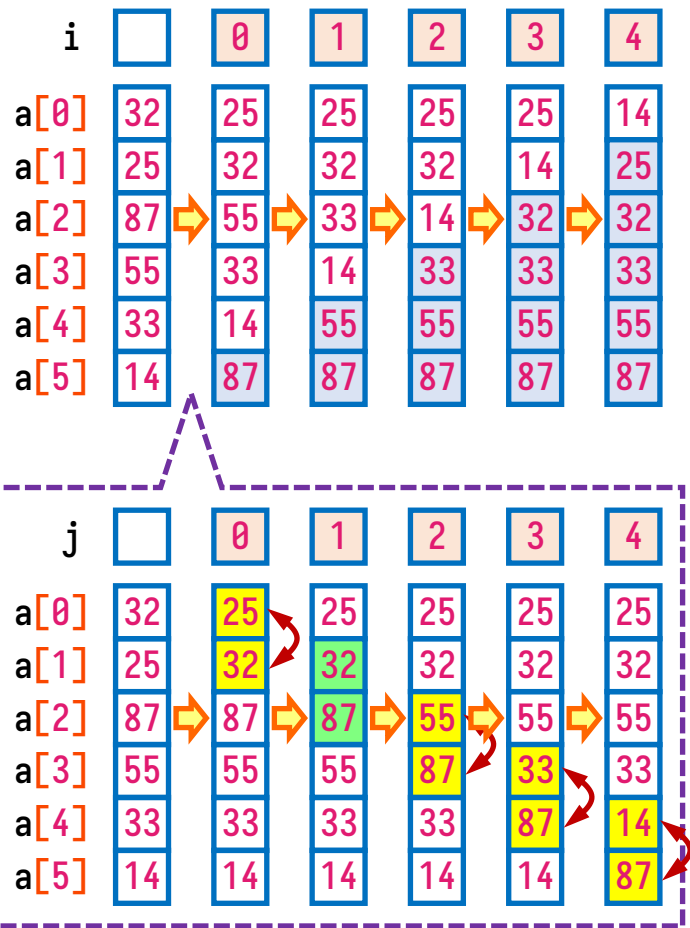
### ■ 算法描述： ( $n$ 个数据升序排列)

- ① 第0轮：比较第0个和第1个数据，若顺序不符，则交换；继续比较第1个和第2个数据，直至末尾；
- ② 第1轮：从第1个和第2个数据起，重复以上比较和交换操作；
- ③ 重复以上操作 $n-1$ 轮，排序完毕。

```
void SortBubble(int num, int a[])
{
    int i, j, t;

    for (i=0; i<num-1; i++)
        for (j=0; j<num-1-i; j++)
            if (a[j] > a[j+1])
                t=a[j], a[j]=a[j+1], a[j+1]=t;
}
```

变量监测：SortBubble(6, a)



# 数据查询模块的设计实现——二分查找



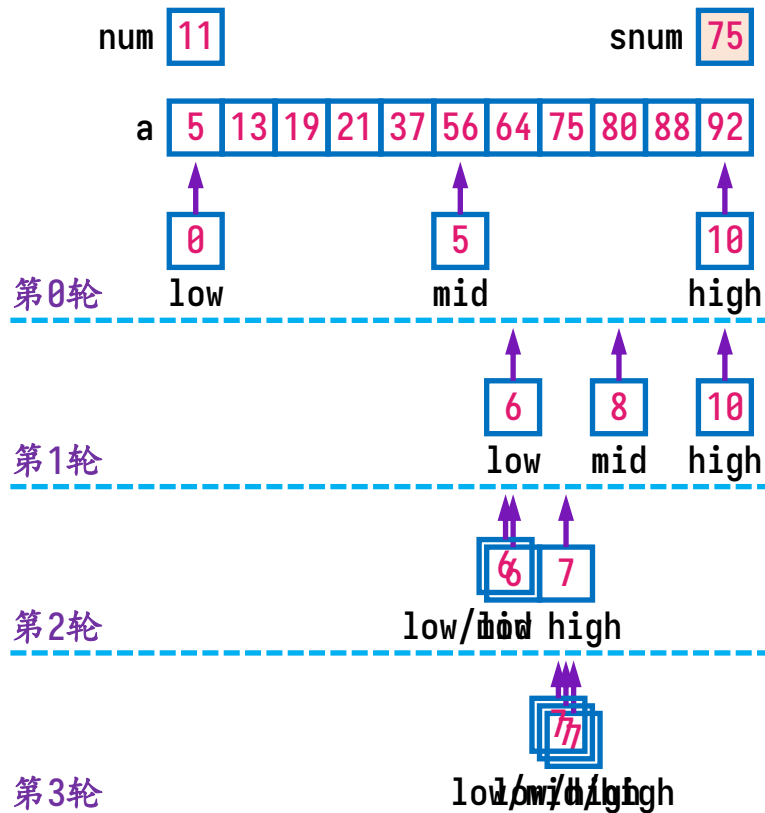
## ■ 例4.4-1：生成随机整数数组并进行排序和查找——二分查找。

```
int SearchBinary(int num, int a[], int snum)
{
    int low=0;           // 查询区间左边界
    int high=num-1;      // 查询区间右边界
    int mid;             // 查询区间中点

    while(low <= high) { // 查询区间不为空
        mid = (low+high) / 2; // 计算区间中点
        if (a[mid] == snum) // 中点等于待查数
            return mid;     // 找到待查数
        else if (a[mid] > snum) // 中点比待查数大
            high = mid - 1; // 查找左半区间
        else // 中点比待查数小
            low = mid + 1;  // 查找右半区间
    }

    return -1;           // 未找到待查数
}
```

变量监测：SearchBinary(11, a, 75)



# 小 结

## ■ 模块化程序设计的步骤

- 分析问题，明确程序的总体任务
- 对任务进行逐层分解，直至大小合适的模块
- 设计模块间的接口与模块算法流程
- 编码实现模块并通过调用组成程序
- 测试模块与程序

## ■ 模块化程序设计的目标

- 使程序结构更清晰，提高程序的可读性，便于交流
- 降低代码间的耦合，便于独立开发
- 提高代码的重用性，减少程序中的重复代码，使程序更简洁
- 控制局部代码规模，便于编码、调试与维护

## ■ 主要知识点

- 函数的定义、调用、原型声明
- 函数的参数传递、返回值
- 变量的作用域、生存期、存储类型
- 模块化程序设计的思想与实现方法

## ■ 能力要求

- 较为全面地掌握模块化程序设计的思想与方法，初步了解软件工程过程
- 养成先设计再实现的良好编程习惯，注重程序规范写、完善性
- 熟练定义与调用函数，有一定的算法优化能力
- 熟练编写基本的排序算法与字符串处理函数
- 理解递归的逻辑，能编写简单的递归程序
- 能编写包含多个函数的单文件程序，解决常见的数据处理与计算问题

# 本章结束