07 数组

统一数据表示，明确访问范围，组织成组处理

本讲学习目标

- 解释数组、元素、下标与元素个数的关系。
- 定义和初始化数组，用循环读写有效范围内的数据。
- 完成简单统计与查找，用函数组织数组处理。
- 使用二维数组表示行列数据，检查访问范围与处理结果。

数组：从数据组合到批量处理

程序 = 数据对象 + 操作 + 控制 + 组合

- 数组将多个同类数据对象组合成整体，使程序能够统一组织和访问数据。
- 数组实现**对象规模突破**，大量同类元素不再需要逐个独立命名。
- 数组与循环相结合，使程序按照规则批量处理大量同类数据。

1 数组与元素

用同类型元素统一表示一组数据，通过下标定位并建立有效初值

一组数据需要统一表示

```
int value_1 = 12;
int value_2 = 7;
int value_3 = 15;
int value_4 = 7;
int value_5 = 9;
int sum = value_1 + value_2 + value_3
        + value_4 + value_5;
```

- 某次实验取得五个整数：12、7、15、7、9。
- 五个独立变量能够保存这些值，求和结果为 50。
- 记录增至几十项时，逐个定义与命名会使代码随对象数量增长。
- 改变一个整数 i，不能自动把变量名 value_1 换成 value_2；需要可以按位置计算的访问方式。

数组统一组织同类型元素

```
int values[5] = {12, 7, 15, 7, 9};
```

- **数组**是由同一类型的若干元素连续组成的对象。
  - values 是整个数组的名称；本例含五个 int 类型元素。
- **数组元素**是构成数组的一个对象。
  - 五个元素分别保存 12、7、15、7、9；每个元素都有自己的当前值。
- 数组统一组织数据，元素仍可分别读取和修改。

下标从零开始定位元素

数据次序	下标	元素表达式	当前值
第 1 项	0	values[0]	12
第 2 项	1	values[1]	7
第 3 项	2	values[2]	15
第 4 项	3	values[3]	7
第 5 项	4	values[4]	9

- **下标**是用于指定数组元素位置的整数；本例下标为 0～4，元素个数为 5。

元素访问沿用变量的读写规则

```
1 int values[5] = {12, 7, 15, 7, 9};
2 int first = values[0];
3 values[1] = 10;
4 int pair_sum = values[0] + values[2];
```

读取元素

first 取得 values[0] 的值 12；读取不改变原元素

写入元素

values[1] 改为 10，其他四个元素不变

在表达式中使用元素

pair_sum 取得两个元素值之和 27

下标表达式在运行时选择元素

```
int values[5] = {12, 7, 15, 7, 9};
int i = 2;
printf("%d %d\n", values[i], values[i - 1]);
```

- 同一写法 values[i] 会随 i 的值选择不同元素
  - i 为 2 时，values[i] 是第三个元素，其值为 15
  - values[i - 1] 选择第二个元素，其值为 7
- 独立变量名要逐项写出；values[i] 用同一写法访问由 i 决定的元素，不必为每项另取变量名。
- 这是对象规模突破的关键：统一声明之外，还能按可计算的下标访问；范围仍须检查。

一维数组定义的基本形式

```
array-definition ::= type-specifier identifier"[" element-count "]" ";"
```

- type-specifier：元素类型；identifier：数组名称。
- element-count：固定的正整数元素个数；本讲使用整数字面量或固定常量。
- [ ] 是定义中的实际方括号；本形式省略可选初始化部分。
- 例如 int values[5]; 定义五个 int 元素，但没有为普通局部元素提供可读取的初值。

元素个数与占用字节数

```
int values[5] = {12, 7, 15, 7, 9};
printf("%zu\n", sizeof values / sizeof values[0]);
```

- 数组的字节数
  - sizeof values 查询整个数组对象的字节数。
- 一个元素的字节数
  - sizeof values[0] 查询一个 int 元素的字节数。
- 元素个数
  - 两者相除得到 5；int 的字节数由实现决定。
  - 此写法用于数组对象所在处，不能在数组形参上推断原数组长度。

初始化列表依次给元素设置初值

```
int values[5] = {12, 7, 15, 7, 9};
values[2] = 18;
```

- 花括号中的五个值按下标 0～4 依次初始化元素。
- 初始化在定义时完成；此后可用元素赋值改变某一项。
- int values[5]; 是无初始化定义；读取前须先给要使用的元素写入有效值。
- 本讲函数内普通局部数组的无初始化元素不能直接当作零。

部分初始化补零，列表可确定个数

```
int first[5] = {12, 7};
int second[5] = {0};
int third[] = {12, 7, 15};
```

- 只给部分初值
  - first 含五个元素：12、7、0、0、0。
- 全部置零
  - second 的五个整数元素都初始化为 0。
- 由初值确定个数
  - third 含三个元素，值为 12、7、15。

初始化与无初始化的区别

```
int a[4] = {3, 5};
int b[] = {2, 4, 6};
int c[4];
```

- a 的四个元素为 3、5、0、0，最后一个有效下标为 3。
- b 含三个元素，b[2] 为 6。
- c 是普通局部数组，没有初始化；c[0] 不能直接读作零。
- 给 c[0] 写入有效值后，才可读取它。

数组复制需要逐元素完成

```
1 int source[3] = {2, 4, 6};
2 int destination[3] = {0};
3 destination[0] = source[0];
4 for (int i = 0; i < 3; i++) {
5     destination[i] = source[i];
6 }
```

合法处理

初始化 destination 后，可把 source[0] 赋给 destination[0]

逐个有效下标赋值，完成全部元素复制

整体限制

destination = source; 不能作为 C 数组整体赋值

初始化列表用于定义时，不是定义后的整数组赋值表达式

2 遍历与有效范围

用循环访问有效元素，区分容量与当前数据个数

数组的顺序遍历

```
int values[5] = {12, 7, 15, 7, 9};
for (int i = 0; i < 5; i++) {
    printf("%d\n", values[i]);
}
```

- 数组与循环协作
  - **遍历**是按规定次序访问给定范围内的每个元素
  - i 从 0 增至 4，values[i] 定位元素，循环重复输出
- 范围与结果
  - i 为 5 时条件 i < 5 为假，停止；依次输出 12、7、15、7、9
- 两种职责
  - 数组组织和定位数据，循环安排重复，配合形成批量处理
  - 数组不会自动执行；循环不会自动建立数组或检查下标

数组越界与访问边界

```c
int values[5] = {12, 7, 15, 7, 9};
for (int i = 0; i <= 5; i++) {
    printf("%d\n", values[i]);
}
```

越界与修正

越界访问是访问数组有效下标范围之外的位置

本例下标为 0 ~ 4；i <= 5 会尝试读取 values[5]，应改为 i < 5

范围责任

C 传统设计强调信任程序员和高效执行，普通数组访问不强制检查下标

省去逐次检查可避免相应开销；程序须用循环条件或访问前检查保证范围

越界结果

读取 values[5] 是**未定义行为**，C 标准不保证程序行为

不保证自动提示；没有报错也不能证明安全或依赖该次输出

数组容量与有效数据个数

```
1 int values[8] = {12, 7, 15};
2 int count = 3;
```

记录情境

前例五项都要处理；本例最多记录八项，本次只有 12、7、15 三项

两个数量

容量是数组可容纳的元素数，本例为 8

有效数据个数是本次已存入、需要处理的数据数，本例为 3

程序记录个数

count 由程序记录，数组不会自动记录；要求 0 ≤ count ≤ 容量

遍历范围

只处理下标 0～2；循环用 i < count，本例为 i < 3

i < 8 虽不越界，却会把未使用的五个位置也纳入处理

数组中的有效数据与未使用位置

项目	当前含义	本例
数组容量	能容纳的元素数	values 有 8 个元素
有效数据个数	本次已存入、需要处理的数据数	count 为 3
本次有效数据的位置	三项数据所在的下标	0～2，依次保存 12、7、15
本次未使用的位置	不作为本次记录数据处理	下标 3～7，本例初始化为 0

- 沿用前页：values 容量为 8，count 为 3；下标 0～2 保存本次三项数据。
- 下标 3～7 的 0 来自部分初始化，不代表本次记录了五项值为 0 的数据；遍历应使用 i < count。

数组数据的输入与检查

```c
#define CAPACITY 8
int values[CAPACITY];
int count;
if (scanf("%d", &count) != 1 ||
    count < 1 || count > CAPACITY) {
    return 1;
}
for (int i = 0; i < count; i++) {
    if (scanf("%d", &values[i]) != 1) {
        return 1;
    }
}
```

输入顺序

容量为 8；先读个数 count，再读 count 个 int 范围内的整数

检查个数

scanf 返回 1 且 count 在 1 ~ 8 时，才进入循环

失败时执行 return 1，结束当前函数

逐项读取

i 从 0 开始，每次读入 values[i]；读取失败立即结束

有效范围

全部读取成功后，前 count 项有效，下标为 0 ~ count - 1

数组的逆序遍历

```
1 int values[5] = {12, 7, 15, 7, 9};
2 int count = 5;
3 for (int i = count - 1; i >= 0; i--) {
4     printf("%d\n", values[i]);
5 }
```

起点

前 count 项的末下标为 count - 1，本例从 4 开始

更新与结束

每次访问后，i 减 1；访问下标 0 后，i 变为 -1，i >= 0 不成立，循环结束

本例的 i 使用有符号 int，可表示停止时的 -1

输出顺序

按下标递减访问，逐行输出 9、7、15、7、12

3 数组统计与查找

用累计、比较和首匹配规则得到处理结果

数组求和的初值与累加

```c
int values[5] = {12, 7, 15, 7, 9};
int sum = 0;
for (int i = 0; i < 5; i++) {
    sum += values[i];
}
printf("%d\n", sum);
```

累计初值

尚未处理元素时，sum 为 0

更新规则

每访问一个元素，sum += values[i] 把当前值加入总和

结果与边界

输出 50，原数组元素不变；本例总和在 int 范围内

数组求和的累计过程

处理下标	当前元素值	更新前 sum	更新后 sum
0	12	0	12
1	7	12	19
2	15	19	34
3	7	34	41
4	9	41	50

- 沿用前页，sum 从 0 开始；每行执行 sum += values[i]，原数据不变。例如 12 + 7 = 19；本行结果传给下一行，sum 始终保存已处理项之和，最终为 50。

数组平均值的计算条件

```
double average = (double)sum / count;
printf("%.1f\n", average);
```

- 使用已有的 sum 和 count
  - sum 为 50、count 为 5，平均值为 10.0。
- 浮点除法
  - 转为 double 后相除，保留小数结果。
  - 如果 sum 为 7、count 为 2，整数除法为 3，浮点结果为 3.5。
  - 也 可以乘以 1.0
- 前提
  - count 必须大于 0；空数据不能按此表达式计算平均值。

数组的最大值

```
1 int values[5] = {12, 7, 15, 7, 9};
2 int maximum = values[0];
3 for (int i = 1; i < 5; i++) {
4     if (values[i] > maximum) {
5         maximum = values[i];
6     }
7 }
8 printf("%d\n", maximum);
```

初值

maximum 从 values[0] 开始，而不是默认从 0 开始

更新

从下标 1 起比较，当前值更大时替换 maximum

前提与结果

count 至少为 1；本例最大值为 15

全部值为负数时，首元素初值仍适用

全负数数组的最大值初值

```
1 int values[3] = {-8, -3, -11};
2 int maximum = 0;
3 for (int i = 0; i < 3; i++) {
4     if (values[i] > maximum) {
5         maximum = values[i];
6     }
7 }
```

错误初值

三个元素 -8、-3、-11 都不大于 0，maximum 从未更新，错误结果为 0

正确做法

maximum 取 values[0]，再从下标 1 起比较；结果为 -3

适用条件

首元素法适用于非空数组；另选初值须有已知的合适下界

满足条件的元素计数

```c
int values[5] = {12, 7, 15, 7, 9};
int matched = 0;
for (int i = 0; i < 5; i++) {
    if (values[i] >= 10) {
        matched++;
    }
}
printf("%d\n", matched);
```

任务

统计五项数据中大于或等于 10 的项数

计数

matched 从 0 开始；满足条件才加 1

结果

12 和 15 符合条件，matched 为 2

matched 是项数，不是这些元素值之和

一次遍历统计正数的个数与总和

```
int values[5] = {3, -2, 0, 5, -1};
int positive_count = 0;
int positive_sum = 0;
for (int i = 0; i < 5; i++) {
    if (values[i] > 0) {
        positive_count++;
        positive_sum += values[i];
    }
}
```

共同条件

只处理大于 0 的元素；本例为 3 和 5，0 不计入

两种更新

两个统计量从 0 开始；项数加 1，总和加当前元素值

结果

一次遍历得到 positive_count 为 2、positive_sum 为 8；原元素不变

顺序查找的首个匹配位置

```
1 int values[5] = {12, 7, 15, 7, 9};
2 int target = 7;
3 int position = -1;
4 for (int i = 0; i < 5; i++) {
5     if (values[i] == target) {
6         position = i;
7         break;
8     }
9 }
```

逐项比较

顺序查找按下标依次比较元素，直到找到目标或查完规定范围

本例比较下标 0～4 的 values[i] 与 target

保存下标

position = i 保存匹配下标；查找 7 得到 1，保存的是位置，不是数值 7

找到即停止

break 退出本例的 for 循环，继续执行后续代码，不退出函数

下标 1 匹配后停止，不再检查下标 3 的另一个 7，保留首个位置

未找到

position 初值 -1；查完仍无匹配则保持 -1，不能用它访问数组

不同目标值的查找结果

目标值	结果	原因
12	0	首元素匹配
7	1	第一个 7 匹配后停止
9	4	末元素匹配
8	-1	五次比较均未匹配

- -1 是查找状态，不是可以访问的数组下标。
- 查找结果只有在确认找到后，才能用于元素访问。

查找结果的检查与使用

```
1 int values[5] = {12, 7, 15, 7, 9};
2 int target = 7;
3 int position = -1;
4 for (int i = 0; i < 5; i++) {
5     if (values[i] == target) {
6         position = i;
7         break;
8     }
9 }
10
11 if (position >= 0) {
12     printf("%d %d\n",
13            position, values[position]);
14 } else {
15     printf("not found\n");
16 }
```

找到时

position >= 0 时，用 values[position] 读取找到的元素，并输出下标与元素值

本例查找 7，position 为 1，输出 1 7

未找到时

查完仍未匹配时，position 保持 -1，输出 not found

先检查再访问，避免读取 values[-1]

检查成立的前提

本页查找只访问下标 0～4，position 只可能为 -1 或该范围内的下标

因此本例确认 position >= 0 后即可访问；任意来源的非负下标仍须核对上界

4 用函数处理数组

说明调用数据、处理个数、返回结果和原元素修改

数组求和函数的定义

```
1 int sum_array(const int values[], int count)
2 {
3     int sum = 0;
4     for (int i = 0; i < count; i++) {
5         sum += values[i];
6     }
7     return sum;
8 }
```

数组参数

values[] 用于访问调用方的数组元素

const 与只读职责

const 是类型限定符；本例的 const int values[] 允许通过 values 读取元素，但不允许通过它给元素赋值或修改元素值

求和只读取数据，因此采用只读参数；这不表示调用方的原数组永远不能修改

范围与返回值

count 指定处理范围，返回 sum；count 为 0 时返回 0

调用条件

前 count 个元素须有有效值且可访问，总和须在 int 范围内

数组求和函数的调用

```c
int sum_array(const int values[], int count)
{
    int sum = 0;
    for (int i = 0; i < count; i++) {
        sum += values[i];
    }
    return sum;
}

int values[5] = {12, 7, 15, 7, 9};
int total = sum_array(values, 5);
int first_three = sum_array(values, 3);
printf("%d %d\n", total, first_three);
```

函数定义与调用

上方保留 sum_array 的完整定义，下方是调用方片段；完整程序先定义 sum_array，再将调用方片段放入 main

两次调用

处理五项得到 50，只处理前三项得到 34

调用方责任

传入的个数须对应可访问的有效数据，函数不能从 values[] 自动取得容量

数组参数的传值与元素访问

```c
#include <stdio.h>
void change_first(int values[], int count)
{
    values[0] += 2;
    count = 0;
}
int main(void)
{
    int values[1] = {12};
    int count = 1;
    change_first(values, count);
    printf("%d %d\n", values[0], count);
    return 0;
}
```

两种实参都传值

沿用第 6 讲：用实参求得的值建立形参的值

地址是定位对象的值；数组名 values 转换为首元素地址，函数接收地址副本

count 传入整数 1，整数形参也得到副本

地址副本访问原元素

values[0] 用接收的地址访问原元素；只复制地址，不复制数组元素

values[0] += 2 将原元素从 12 改为 14

整数副本独立修改

count = 0 只修改形参；调用方 count 仍为 1

输出 14 1：原元素改变，调用方个数不变

数组元素修改与普通形参修改

```
1 void add_two(int values[], int count)
2 {
3     for (int i = 0; i < count; i++) {
4         values[i] += 2;
5     }
6     count = 0;
7 }
```

调用条件

原数组为 12、7、15，调用方 count 为 3；调用 add_two(values, count)

原元素变化

三个元素分别变为 14、9、17；函数访问并修改调用方的原元素

普通形参变化

count = 0 只改变函数中的局部形参，调用方 count 仍为 3

数组形参与容量信息

- 两种参数写法
  - 在本例形参位置，int values[] 与 int values[5] 都接收首元素地址值。
  - [5] 不会复制五个元素，也不自动检查实参数组是否含五个元素。
- sizeof 的边界
  - 对数组对象使用 sizeof，可查询整体字节数。
  - 对数组形式形参使用 sizeof，得到参数类型的大小，不能推断原数组长度。
- 明确接口
  - 元素个数单独传入，并与实际可访问范围一致。

数组接口的只读与修改职责

处理职责	参数写法	结果方式
只读求和	const int values[]	return 返回总和
增加元素值	int values[]	修改原元素

- const 限制通过该参数写元素，有助于表达只读职责。
- 返回整数与修改原元素分别约定；可访问范围、整数范围和修改权限须明确。

阈值计数函数的参数与实现

```
1  int count_at_least(const int values[],
2                     int count, int threshold)
3  {
4      int matched = 0;
5      for (int i = 0; i < count; i++) {
6          if (values[i] >= threshold) {
7              matched++;
8          }
9      }
10     return matched;
11 }
```

参数职责

count 指定项数，threshold 指定下限；const 限制写入

计数规则

matched 从 0 开始；大于或等于阈值时加 1，最后返回项数

调用结果

values 为 12、7、15、7、9 时，count_at_least(values, 5, 10) 返回 2

count 为 0 时不访问元素，返回 0

阈值计数函数的结果与边界

数组数据	count	返回值	依据
12、7、15、7、9	5	2	12 和 15 满足条件
单项 10	1	1	包含等于阈值的情形
单项 9	1	0	没有满足条件的元素
12、7、15、7、9	0	0	不访问任何元素

- 本页 threshold 为 10；约定 0 ≤ count ≤ 有效元素数，原元素不变。
- 函数不自动检查容量；超出可访问范围的调用违反接口约定。

5 二维数组

用行列下标定位数据，以嵌套循环逐行处理

二维数组的定义与元素访问

```c
int grid[2][3] = {
    {2, 4, 6},
    {1, 3, 5}
};
printf("%d\n", grid[1][2]);
```

数组结构

二维数组是元素本身为一维数组的数组，可以看作由若干个一维数组组成的数组

grid 含两行，每一行都是含三个 int 元素的数组

定义与初值

[2][3] 分别规定行数和每行列数。

内层花括号按行给出初值。

元素访问

行、列下标从 0 开始；grid[1][2] 的行下标是 1，列下标是 2

它访问第二行第三列的元素，值为 5

二维数组的按行存放顺序

存放次序	元素	当前值
0	grid[0][0]	2
1	grid[0][1]	4
2	grid[0][2]	6
3	grid[1][0]	1
4	grid[1][1]	3
5	grid[1][2]	5

- 存放次序为同行列下标先变化，再接下一行；示意不指定实际地址或每项字节数。

嵌套循环的逐行遍历与输出

```c
int grid[2][3] = {
    {2, 4, 6},
    {1, 3, 5}
};
for (int row = 0; row < 2; row++) {
    for (int col = 0; col < 3; col++) {
        printf("%d ", grid[row][col]);
    }
    printf("\n");
}
```

外层循环

row 依次选择两行，范围为 0 ~ 1

内层循环

每行 col 都从 0 重新开始，访问三列

输出分行

内层结束后换行，显示 2 4 6 和 1 3 5

二维数组行列下标的有效范围

- grid 定义为 int grid[2][3]，六个元素均已初始化。
- grid[1][2] 有效：行下标 1 和列下标 2 都在范围内，值为 5。
- grid[2][0] 无效：行下标 2 超过末行下标 1。
- grid[0][3] 无效：列下标 3 超过末列下标 2。
- 必须同时满足 0 ≤ row < 2 和 0 ≤ col < 3，不能只根据六个元素的总数判断。

逐行求和的初值与累加

```c
int grid[2][3] = {
    {2, 4, 6},
    {1, 3, 5}
};
for (int row = 0; row < 2; row++) {
    int row_sum = 0;
    for (int col = 0; col < 3; col++) {
        row_sum += grid[row][col];
    }
    printf("%d\n", row_sum);
}
```

每行独立求和

row_sum 在外层循环体中设为 0，每行重新累计

内层处理

三列元素依次加入当前行的 row_sum

结果

第 0 行输出 12，第 1 行输出 9

若在外层循环前只初始化一次，会形成跨行累计

各行最大值的初值与更新

```
int grid[2][3] = {
    {-8, -3, -11},
    {5, 5, 2}
};
for (int row = 0; row < 2; row++) {
    int maximum = grid[row][0];
    for (int col = 1; col < 3; col++) {
        if (grid[row][col] > maximum) {
            maximum = grid[row][col];
        }
    }
    printf("%d\n", maximum);
}
```

每行初值

maximum 在每行开始时取 grid[row][0]，不沿用上一行结果

比较范围

每行非空，col 从 1 开始比较其余元素；当前值更大时更新

输出与边界

两行依次输出 -3 和 5；首元素法适用于全负数，重复最大值不影响结果

二维数组形参的列数要求

```
1 int sum_row(int grid[][3], int row)
2 {
3     int sum = 0;
4     for (int col = 0; col < 3; col++) {
5         sum += grid[row][col];
6     }
7     return sum;
8 }
```

数组结构与列数

二维数组是数组的数组，每一行也是一个数组

grid[][3] 可省略行数，但须保留列数 3

列数决定一行大小，用于定位指定行

访问条件

row 是调用方提供的有效行下标；本函数不自动检查行数

调用结果

sum_row(grid, 0) 为 12，sum_row(grid, 1) 为 9

函数只读元素，返回一行的总和

6 综合应用

示范输入、统计和查找的组合及运行结果

实验记录处理的任务与结果

处理一组整数记录：求总和、平均值和最大值，并查找 7 首次出现的下标

- 输入：先读入个数 count（1～8），再读入 count 个整数（每项为 -100～100），存入 values
- 输出：总和、平均值（保留一位小数）、最大值、7 的首个下标；未找到时下标为 -1
- 例如：输入 5 12 7 15 7 9 → 输出 50 10.0 15 1；下标 1 对应第二条记录
- 读取失败或输入超出范围时结束，不输出结果；有效个数至少为 1，可求最大值并计算平均值

实验记录处理的职责分工

处理步骤	承担位置	完成的操作
输入	main	检查个数与数据，存入数组
求和	sum_array(values, count)	返回总和 sum
求最大值	max_array(values, count)	返回最大值；要求数组非空
查找 7	find_first(values, count, 7)	返回首个匹配下标或 -1
平均与输出	main	(double)sum / count；输出四项结果

- main 组织输入、调用和输出，三个函数分别求和、求最大值和查找。

首匹配查找函数的返回约定

```
1  int find_first(const int values[],
2                 int count, int target)
3  {
4      for (int i = 0; i < count; i++) {
5          if (values[i] == target) {
6              return i;
7          }
8      }
9      return -1;
10 }
```

查找范围

count 指定有效项数，target 指定目标；原元素不变

找到即返回

返回当前下标并结束函数；查找 12、7、15、7、9 中的 7，返回 1

未找到

查完整个范围后返回 -1；count 为 0 时也返回 -1

调用方先检查返回值，再将找到的位置用于元素访问

实验记录处理的运行结果

输入数据	总和	平均值	最大值	7 的首下标
12、7、15、7、9	50	10.0	15	1
单项 7	7	7.0	7	0
-8、-3、-11	-22	-7.3	-3	-1
八个 0	0	0.0	0	-1

- count 为 0 或 9、读取失败、数值超范围：退出，不输出统计结果。
- 此表是当前程序用例结果，不替代函数的有效范围约定。

冒泡排序的相邻比较与交换

```
1  void bubble_sort(int values[], int count)
2  {
3      for (int pass = 0;
4           pass < count - 1; pass++) {
5          int swapped = 0;
6          for (int i = 0;
7               i < count - 1 - pass; i++) {
8              if (values[i] > values[i + 1]) {
9                  int temp = values[i];
10                 values[i] = values[i + 1];
11                 values[i + 1] = temp;
12                 swapped = 1;
13             }
14         }
15         if (!swapped) { break; }
16     }
17 }
```

比较与交换

将记录升序排列；从左到右比较相邻两项，左大于右时交换

temp 暂存左项，避免覆盖原值

每轮“冒泡”

未排好部分的最大值移到末端；下轮少比较一项

n 项最多 n - 1 轮，剩余一项自然就位

提前结束

swapped 每轮先置 0，有交换则置 1；整轮为 0，用 break 结束

某轮排好后，可能还需下一轮确认无交换

调用条件

count 非负且不超过有效可修改项数；直接修改原数组，0 或 1 项无需排序

冒泡排序的每轮结果

完成的轮次	数组当前内容	本轮结果
初始状态（尚未排序）	12、7、15、7、9	—
第 1 轮	7、12、7、9、15	末项 15
第 2 轮	7、7、9、12、15	倒数第 2 项 12
第 3 轮	7、7、9、12、15	无交换，确认有序并提前结束

- 调用 bubble_sort(values, 5)；表中为每轮结束状态。第 2 轮排好，第 3 轮无交换，省去第 4 轮。
- 输出 7、7、9、12、15；只改变排列顺序，保留重复值与元素个数。

折半查找的范围缩小规则

- 排序后，左侧元素不大于右侧元素，才能根据中间值排除半边
- 在 7、7、9、12、15 中查找 12：中间值 9 及其左侧都小于 12，只需继续查右侧的 12、15
- **折半查找**（也称二分查找）：比较中间元素，找到即停止，否则排除半边，在剩余范围重复查找
- left、right 为候选范围首、末下标，初始为 0、count - 1；比较 values[mid]
  - 等于目标：返回 mid
  - 小于目标：排除中间项及左半边，令 left = mid + 1
  - 大于目标：排除中间项及右半边，令 right = mid - 1
- left > right 时范围为空，返回 -1；顺序查找无需排序

折半查找的比较次数

- 每次排除半边，候选位置逐步减少
  - 8 个位置最多剩下：**8 → 4 → 2 → 1 → 0**
  - 最多比较 4 个中间元素；顺序查找最坏需比较 8 个元素
- n 个元素的比较次数约随 $\log_2 n$ 增长
  - n ≥ 1 时，最多比较 $\lfloor \log_2 n \rfloor + 1$ 个中间元素
- 以上只计算查找，不含排序成本；先排序再查一次未必更快
- 保留原始记录顺序时，应排序副本

折半查找函数的实现

```
1  int binary_search(const int values[],
2                    int count, int target)
3  {
4      int left = 0, right = count - 1;
5      while (left <= right) {
6          int mid = left + (right - left) / 2;
7          if (values[mid] == target) {
8              return mid;
9          }
10         if (values[mid] < target) {
11             left = mid + 1;
12         } else {
13             right = mid - 1;
14         }
15     }
16     return -1;
17 }
```

候选范围

left <= right 时仍有候选元素；count 为 0 时不访问数组，直接返回 -1

中间位置

left + (right - left) / 2 取当前范围的中间下标

比较与更新

找到即返回；否则排除中间项及不可能匹配的半边，每次都缩小范围

调用约定

前 count 项须升序排列且有效可访问；本函数只读元素，返回一个匹配下标或 -1

有重复值时不保证返回首个位置；返回下标属于排序后的数组，不能当作原始记录位置

折半查找的过程与结果

比较次数	left	right	mid	中间元素	比较后的操作
第 1 次	0	4	2	9	9 小于 12，令 left = 3
第 2 次	3	4	3	12	找到，返回下标 3

- 数组为 7、7、9、12、15，下标 0～4；表中查找 12，返回 3。查找 7 依次比较下标 2、0，返回 0，不保证首匹配
- 查找 10 比较 9、12 后，left = 3 > right = 2，返回 -1；count 为 0 也返回 -1。返回值非负后才能访问元素

数组程序的检查要点

- 对象与初值：访问的是哪个元素？读取前是否已有有效值？
- 下标与范围：起点、末次访问、停止条件是否和 count 对应？
- 累计与结果：初值是否符合求和、最值或查找职责？
- 函数接口：是否传入正确个数，是否允许修改，是否按返回约定使用结果？
- 二维处理：两维分别检查，行累计量在正确层级重新初始化。
- 折半查找：数组是否升序排列？候选范围是否缩小？是否把一个匹配位置误当作首匹配或原始记录位置？

本讲总结：统一数据，明确范围，组织处理

- 数组把同类元素统一组织，用可计算下标访问，突破逐个命名的限制；个数与末下标不同。
- 初始化或成功赋值提供有效数据；容量限定可存范围，count 限定当前处理范围。
- 数组提供对象组织与访问规则，循环提供迭代自动化；二者配合完成批量处理。
- 数组函数访问原元素，并显式传入个数；返回值与元素修改承担不同结果职责。
- 二维数组保留行列关系；两级下标和循环分别受各维范围约束，不会自动检查越界。

教材阅读与复习

本讲内容	《C Primer Plus》第六版对应
一维定义、初始化、元素访问与边界	§10.1、§10.1.1、§10.1.3～10.1.4
处理数组的函数与只读参数	§10.4、§10.6.1
二维数组与固定列数接口	§10.2、§10.2.1、§10.7.3

- 对照课件代码，说明元素、下标、有效范围和调用后的数据变化。
- 指针表示法、指针运算与多维指针将在后续单元学习。