XCPCer 从零开始的 C/C++ 入门指南——第 5 章 函数、结构体与 sort 自定义排序

SYUCT-ACM 2026 新生训练 · 第五次讲义 · 2026-11-16

第 5 章 函数、结构体与 sort 自定义排序

> 本文是《XCPCer 从零开始的 C/C++ 入门指南（算法竞赛向）》系列第 5 章。本章配套第五次训练的题单，15 道自编练习已全部导入该场比赛（题号 **V–AJ**），逐题链接见 5.10。

5.0 导语：把逻辑打包，把数据打包

前四章里，我们处理的数据大多是”散装”的：一个数、一排数、一个字符串。但数据往往**成组出现**——一个学生有姓名、语文、数学三样信息，一个点有横纵两个坐标；而代码里的逻辑也往往**要重复使用**比如“判断质数”这段代码，可能在一道题里要写三遍。

本章解决的就是这两件事：

- **函数**把一段逻辑打包成一个可以反复调用的工具；
- **结构体**把一组相关的数据打包成一个整体；
- 最后把两者接上 `sort` **的自定义比较规则**，让排序不再只能按”数值大小”，而是按”总分降序、总分相同按语文降序、再相同按姓名字典序”这样的规则来。

本次训练结束后，你应该能够：

- 定义并调用函数，理解返回值、形参与实参，分清**传值**与**传引用**；
- 把数组作为函数参数传进去，并知道为什么必须同时传长度；
- 定义结构体、访问成员、使用结构体数组，并用别名简化类型名；
- 给 `sort` 传一个 `cmp` **函数**或一个 **lambda**，完成降序、多关键字与”按另一个数组排序”；
- 说清楚比较器必须满足的**严格弱序**，知道写成 `<=` 会出什么事；
- 通过三道综合示例，把函数、结构体、排序串起来解决完整问题。

内容分层：

- **主线**：5.1 函数、5.2 结构体、5.3 自定义排序，全部是基础必做内容；
- **【拓展】**：5.7 递归。解决汉诺塔、斐波那契这类问题。

> **函数打包”怎么做”，结构体打包”是什么”，比较器告诉 `sort` “谁在前”。**

5.1 函数：把一段逻辑打包

5.1.1 为什么需要函数

先看一段”没有函数”的代码。题目要求读入三个数，输出其中最大的那个：

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a, b, c;
    cin >> a >> b >> c;

    int maximum = a;
    if (b > maximum)
    {
        maximum = b;
    }
    if (c > maximum)
    {
        maximum = c;
    }

    cout << maximum << '\n';
    return 0;
}
```

输入：

```
3 7 5
```

输出：

```
7
```

这段代码没问题。但如果题目变成”读入两组数据，每组三个数，分别输出最大值”，就要把中间那五行**再抄一遍**。抄的时候很容易抄错一个变量名，而且以后想改”最大值怎么算”，得改两处。

函数就是给这段逻辑起一个名字，以后用名字调用它：

```cpp
int maxOf(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    return b;
}
```

现在”取两个数的较大值”这件事有了名字 `maxOf`，想用几次就调用几次。这就是函数最直接的价值：**写一次，用很多次；改一处，全都改。**

5.1.2 定义、调用与返回

函数的定义由四部分组成：

```
返回类型   函数名(参数列表)
{
    函数体
    return 返回值;
}
```

对照 `maxOf` 逐项看：

部分	内容	说明
返回类型	`int`	这个函数算出来的结果是什么类型
函数名	`maxOf`	调用时用的名字
参数列表	`(int a, int b)`	调用时要传进来的数据，每个都写类型
函数体	`{ ... }`	真正干活的代码
返回	`return a;`	把结果交回调用处，同时结束函数

调用就是写”函数名(实参)”。下面是一个完整程序：

```
#include <iostream>
using namespace std;

int maxOf(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    return b;
}

int main()
{
    int x, y;
    cin >> x >> y;

    int answer = maxOf(x, y);
    cout << answer << '\n';
    return 0;
}
```

输入：

```
3 7
```

输出：

```
7
```

执行过程：`main` 里执行到 `maxOf(x, y)` 时，程序会做三件事：

```
1. 把实参 x = 3、y = 7 按顺序拷贝给形参 a、b
2. 跳进 maxOf 的函数体，从第一行开始执行
3. 遇到 return 7，把 7 带回调用处，answer 得到 7，继续执行 main 的下一行
```

所以 `maxOf(x, y)` 这个整体，在表达式里的地位和一个变量完全一样——它**产生一个值**。

三个容易混的点：

- **函数名后面一定要有括号。**`maxOf` 是函数本身，`maxOf(x, y)` 才是”调用它并拿到结果”；
- `return` **之后的代码不会执行。**`maxOf` 里 `a > b` 成立时直接 `return a`，后面的 `return b` 被跳过；
- `main` **也是函数**，只不过它由系统调用，返回值 `0` 表示”程序正常结束”。

5.1.3 没有返回值的函数：`void`

有些函数只是”做一件事”，不需要交回结果，比如打印一行分隔线。这时返回类型写 `void`（空）：

```cpp
#include <iostream>
using namespace std;

void printLine(int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << '-';
    }
    cout << '\n';
}

int main()
{
    int n;
    cin >> n;

    printLine(n);
    cout << "done\n";
    return 0;
}
```

输入：

```
5
```

输出：

```
-----
done
```

`void` 函数里可以写 `return;`（不带值）提前结束，但**不能写** `return 某个值;`，**否则编译错误。反过来，返回类型写了** `int` **却一条** `return` **都没有，也是编译警告甚至错误——每个非** `void` **函数都必须保证所有路径都有返回值。**

5.1.4 传值：形参是实参的副本

这是本章第一个必须掰清楚的概念。看下面的程序：

```cpp
#include <iostream>
using namespace std;

void addOne(int x)
{
    x = x + 1;
    cout << "inside: " << x << '\n';
}

int main()
{
    int a = 10;
    addOne(a);
    cout << "outside: " << a << '\n';
    return 0;
}
```

输出：

```
inside: 11
outside: 10
```

函数里明明把 `x` 加了一，`main` 里的 `a` 却还是 10。原因：

```
调用 addOne(a)：
  实参 a = 10  ──拷贝一份──▶  形参 x = 10
                                x = x + 1  →  x 变成 11
  实参 a 从头到尾都是 10，没有被碰过
```

形参是实参的副本，函数里改形参，改的是副本，实参不受影响。这叫**传值**（pass by value），是 C++ 的默认方式。

传值的好处是安全：函数怎么折腾自己的副本，都伤不到调用方。代价是每次调用都要拷贝一份，如果参数是一个很大的结构体，拷贝本身就要花时间。

5.1.5 传引用：形参是实参的别名

如果确实想让函数**改动**外面的变量，就在参数类型后面加一个 `&`：

```cpp
#include <iostream>
using namespace std;

void addOne(int &x)
{
    x = x + 1;
}

int main()
{
    int a = 10;
    addOne(a);
    cout << a << '\n';
    return 0;
}
```

输出：

```
11
```

`int &x` 读作”`x` 是一个 `int` 的引用”。引用不是副本，而是**实参的另一个名字**：

```
调用 addOne(a)：
  形参 x 就是 a 的别名
  x = x + 1  等价于  a = a + 1
```

两个必须记住的细节：

1. **调用时不要写 `&` 符号。**定义时写 `int &x`，调用时写 `addOne(a)`，不是 `addOne(&a)`；
2. 引用必须**绑定到一个已经存在的变量**，不能凭空创建。

传引用最经典的用途是交换两个变量。第 3 章我们一直用标准库的 `swap`，现在可以自己写一个：

```cpp
#include <iostream>
using namespace std;

void swapTwo(int &x, int &y)
{
    int t = x;
    x = y;
    y = t;
}

int main()
{
    int a = 3;
    int b = 8;
    swapTwo(a, b);
    cout << a << ' ' << b << '\n';
    return 0;
}
```

输出：

```
8 3
```

如果把 `swapTwo` 的参数写成 `int x, int y`（去掉两个 `&`），函数里交换的只是两个副本，`main` 里的 `a`、`b` 一点变化都没有——这是新手最常踩的坑之一。

5.1.6 传址：用指针达到同样的效果

C 语言没有引用，它用**指针**做同一件事。C++ 里两种写法都能用，但竞赛代码更常用引用，因为它不用到处写 `*`。

```cpp
#include <iostream>
using namespace std;

void addOne(int *x)
{
    *x = *x + 1;
}

int main()
{
    int a = 10;
    addOne(&a);
    cout << a << '\n';
    return 0;
}
```

输出：

```
11
```

对照表：

写法	定义	调用	函数内访问
传值	`void f(int x)`	`f(a)`	直接用 `x`，改不动外面
传引用	`void f(int &x)`	`f(a)`	直接用 `x`，能改外面
传址	`void f(int *x)`	`f(&a)`	用 `*x`，能改外面

指针的完整用法（指针与数组、指针运算）不属于ACM学习内容，本章只需要**看懂**传址写法，自己写代码时优先用引用。

5.1.7 数组作为函数参数

把数组交给函数用，常见的写法有两种：**传参**和**开全局**。

写法一：传参

数组开在 `main` 里，函数用”数组名 + 长度”接收：

```cpp
#include <iostream>
using namespace std;

#define MAXN 1000

long long sumOf(int a[], int n)
{
    long long sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum += a[i];
    }
    return sum;
}

int main()
{
    int n;
    cin >> n;

    int a[MAXN]; // 数组开在 main 里（局部数组）
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    cout << sumOf(a, n) << '\n';
    return 0;
}
```

输入：

```
5
7 -2 7 -5 3
```

输出：

```
10
```

写法二：全局数组（推荐）

数组开在所有函数外面，函数照样用 `int a[], int n` 接收：

```cpp
#include <iostream>
using namespace std;

#define MAXN 1000

int a[MAXN]; // 数组开在全局，不在 main 里

long long sumOf(int a[], int n)
{
    long long sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum += a[i];
    }
    return sum;
}

int main()
{
    int n;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    cout << sumOf(a, n) << '\n';
    return 0;
}
```

输入输出与写法一完全相同（答案还是 `10`）——两种写法的差别只在**数组放在哪**，调用的那一行 `sumOf(a, n)` 一模一样。

两种写法怎么选

对比项	写法一：传参	写法二：全局数组（推荐）
数组放在哪	`main` 里的栈上	静态存储区
函数怎么拿到数组	只能靠参数传	参数传，或直接读全局
`n` 很大时	可能栈溢出、直接 RE	不受影响
是否自动清零	不会，要手动初始化	会，默认全是 0
多组数据	每组重新声明即可	旧值会留下，要手动 `fill`
一眼看出函数依赖什么	看参数表就知道	要往上翻代码

主线统一用全局数组。 这是本章、也是后面所有章节的默认写法，原因有三条：

1. **不用纠结作用域。** `a` 定义在所有函数外面，`main` 里读入、函数里使用，看到的是同一个数组，不必再想“这个 `a` 是哪个函数里的、要不要加 `&`”；
2. **不占 `main` 的栈空间。**局部数组放在栈上，`n` 一大就可能栈溢出、直接 RE；同样的规模开在全局（静态存储区）就没事（第 3 章 3.2.2）；

3. **自动清零。**全局数组默认全是 0，计数数组、标记数组能省掉一步初始化。

两个注意点：

- **开全局不等于不用传参。**本系列的写法仍然是”显式传 `a` 和 `n` “（5.1.8 会讲为什么）：函数签名里写清楚依赖什么，换一个数组还能复用同一个函数；
- **多组数据要重置。**全局数组的旧值会留到下一组，该清的时候得 `fill/memset`（第 3 章 3.4.2）。

> 想再省一步的话，`sumOf` 也可以不写数组参数、直接在函数里用全局的 `a`。但那样这个函数就只对这一个数组有用了，换一个数组就得再写一个函数，所以本系列统一”数组开全局 + 显式传 `a` 和 `n` “。

三个要点（两种写法都适用）

1. **长度必须一起传。**`int a[]` 在参数里会退化成”首地址”，函数内部**拿不到数组长度**，`sizeof(a)` 得到的不是整个数组的字节数。第 3 章 3.2.1 说过”始终显式维护 `n` “，原因就在这里；
2. **函数里改数组元素，外面会变。**数组传的是首地址，不是整份拷贝，所以 `a[i] = 0` 改的就是原数组；
3. 形参写成 `int a[]` 或 `int *a` 是等价的，前者更好读。

第 2 点用一个例子确认：

```
void fillZero(int a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        a[i] = 0;
    }
}
```

在 `main` 里调用 `fillZero(a, n)` 之后，全局数组 `a` 的前 `n` 个元素真的都变成 0 了。

【拓展】如果坚持用**写法一**，二维数组作参数时**第二维的长度必须写出来**，因为编译器要靠它计算 `a[i][j]` 的地址；而按推荐的写法二开全局，函数只要 `int n, int m`，这个坑也就不存在了。二维数组还**更该开全局**：`1000 × 1000` 的 `int` 就是 4 MB，开在函数里几乎必然爆栈。

```
#define MAXN 1000
#define MAXM 1000

int grid[MAXN][MAXM]; // 二维数组开全局

void printGrid(int n, int m)
{
    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            cout << grid[row][column] << (column + 1 == m ? '\n' : ' ');
        }
    }
}
```

5.1.8 作用域与嵌套调用

第 1 章 1.8 讲过作用域，函数让这件事更明显：**函数内部声明的变量是局部变量**，函数一结束就消失，别的函数看不见它。

```cpp
#include <iostream>
using namespace std;

int square(int x)
{
    return x * x;
}

int sumOfSquares(int a, int b)
{
    return square(a) + square(b);
}

int main()
{
    int x, y;
    cin >> x >> y;
    cout << sumOfSquares(x, y) << '\n';
    return 0;
}
```

输入：

```
3 4
```

输出：

```
25
```

`main` 调用 `sumOfSquares`，`sumOfSquares` 又调用 `square`，这叫**嵌套调用**。执行顺序是”进去、再进去、回来、再回来”：

```
main
 └─ sumOfSquares(3, 4)
      ├─ square(3) → 9
      ├─ square(4) → 16
      └─ return 9 + 16 = 25
main 拿到 25
```

两个作用域规则要记牢：

- 函数里声明的变量，**出了函数就不存在**，别的函数同名也不冲突；
- 函数里可以读全局变量（定义在所有函数外面的），但**能通过参数传的，就不要靠全局变量传**——参数写清楚了”这个函数依赖什么”，读代码的人一眼能看懂。

5.1.9 声明与定义：先有名字，再有身体

C++ 要求**调用之前编译器至少见过这个函数的名字。**如果函数定义写在 `main` 前面（本章所有例子都是这样），直接调用即可；如果定义写在后面，就要先在前面写一行**函数声明**（也叫函数原型）：

```cpp
#include <iostream>
using namespace std;

int maxOf(int a, int b); // 声明：只写返回类型、名字、参数类型

int main()
{
    int x, y;
    cin >> x >> y;
    cout << maxOf(x, y) << '\n';
    return 0;
}

int maxOf(int a, int b) // 定义：真正实现
{
    if (a > b)
    {
        return a;
    }
    return b;
}
```

声明以分号结尾、没有函数体；定义有函数体。**声明和定义的参数类型必须完全一致**，否则编译器会认为是两个不同的函数，链接时报”未定义”。

初学阶段最简单的做法是：**把函数定义都写在 `main` 前面**，按”被调用的在前、调用的在后”排好顺序，就不需要单独写声明了。

5.1.10 写函数的三条约定

1. **一个函数只做一件事。**`sumOf` 只求和，不顺便输出；输出留给 `main`。这样同一个函数在别的题里也能直接用；
2. **参数不要太多。**超过三四个参数时，通常说明该用结构体把数据打包了——这正是下一节的内容；
3. **函数名说清楚做什么。**`maxOf`、`sumOf`、`isPrime`、`fillZero` 都比 `f1`、`work`、`solve` 好读。

5.2 结构体：把一组数据打包

5.2.1 为什么需要结构体

假设题目要处理 `n` 个学生，每个学生有学号、语文成绩、数学成绩。用数组存的话，得开三个数组，而且**下标必须对齐**：

```cpp
int id[MAXN];
int chinese[MAXN];
int math[MAXN];
```

第 `i` 个学生的信息散落在三个数组的同一个下标上。这带来两个麻烦：

1. **排序时非常难受。**想按总分排序，一交换就得同时交换三个数组的同一个位置，少写一个就全乱；
2. **传参时非常难受。**想把”一个学生”传给函数，得传三个参数。

结构体（`struct`）把这几样数据打包成一个整体，就像给它们装进一个盒子：

```
struct Student
{
    int id;
    int chinese;
    int math;
};
```

现在 `Student` 是一个**新的类型**，用法和 `int`、`double` 一样：

```
Student s;      // 声明一个学生
s.id = 1;       // 用 . 访问成员
s.chinese = 90;
s.math = 85;
```

5.2.2 定义与成员访问

结构体的定义写在所有函数外面（和全局变量同一个位置），语法是：

```
struct 类型名
{
    成员类型 成员名;
    成员类型 成员名;
    ...
};              ← 注意这个分号，漏了会报一堆莫名其妙的错
```

完整示例：

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Student
{
    string name;
    int chinese;
    int math;
};

int main()
{
    Student s;
    s.name = "Alice";
    s.chinese = 90;
    s.math = 85;

    cout << s.name << ' ' << s.chinese << ' ' << s.math << '\n';
    cout << s.chinese + s.math << '\n';
    return 0;
}
```

输出：

```
Alice 90 85
175
```

三个要点：

- **成员的类型可以不同。** `name` 是 `string`，两个成绩是 `int`，这正是结构体比数组强的地方——数组只能装同类型；
- **`.` 读作“的”的意思。** `s.name` 读作“ `s` 的 `name` ”，和 `a[i]` 一样是访问数据的方式；
- **成员可以继续用下标。**如果成员本身是数组（`int score[3];`），就写 `s.score[0]`。

结构体定义**只是描述“长什么样”，并不占内存**；声明变量 `Student s;` 时才真正分配空间。

5.2.3 结构体数组

结构体最常用的形态是**结构体数组**：一批同类型的结构体。

```cpp
#include <iostream>
#include <string>
using namespace std;

#define MAXN 1000

struct Student
{
    string name;
    int chinese;
    int math;
};

Student students[MAXN];

int main()
{
    int n;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cin >> students[i].name >> students[i].chinese >> students[i].math;
    }

    for (int i = 0; i < n; i++)
    {
        int total = students[i].chinese + students[i].math;
        cout << students[i].name << ' ' << total << '\n';
    }
    return 0;
}
```

输入：

```
3
Alice 90 85
Bob 78 92
Cindy 88 88
```

输出：

```
Alice 175
Bob 170
Cindy 176
```

`students[i]` 是第 `i` 个学生（一个完整的结构体），`students[i].name` 是他的姓名。**下标和成员是两层访问**，先定位”第几个”，再定位”哪一项”。

把结构体数组定义在函数外面（全局）有两个好处：自动清零，且不占函数内的栈空间——和第 3 章的大数组是同一个道理。

5.2.4 结构体的初始化

三种写法：

```
Student a = {"Alice", 90, 85};   // 按成员顺序依次给值
Student b = {};                  // 全部清零：name 为空串，成绩为 0
Student c;                       // 全局变量自动清零；局部变量则是不确定值
```

第一种叫**列表初始化**，值的顺序必须和成员声明顺序一致。第二种 `= {}` 是 最常用的写法，尤其在多组数据需要重置时。

如果只想给一部分成员赋值，剩下的会自动补零：

```
Student d = {"Bob"};   // name = "Bob"，两个成绩都是 0
```

5.2.5 结构体作为函数参数

结构体可以像 `int` 一样传进函数。默认是**传值**，也就是整份拷贝：

```
int totalOf(Student s)
{
    return s.chinese + s.math;
}
```

这样写能跑，但每次调用都要拷贝整个结构体。如果结构体里有大数组，拷贝开销会很明显。**只读不改时，用常引用**：

```
int totalOf(const Student &s)
{
    return s.chinese + s.math;
}
```

`const Student &s` 的意思是" `s` 是调用方那个结构体的引用，但我在函数里保证不改它" 。这样既不拷贝，又不会误改，是竞赛代码里传结构体的标准写法。

需要修改时去掉 `const`：

```
void addBonus(Student &s, int bonus)
{
    s.chinese += bonus;
    s.math += bonus;
}
```

调用 `addBonus(students[0], 5)` 之后， `students[0]` 的两个成绩真的都加了 5。

对照表：

写法	是否拷贝	能否修改调用方	什么时候用
`Student s`	拷贝	不能	结构体很小、且不改
`const Student &s`	不拷贝	不能	**只读时的默认选择**
`Student &s`	不拷贝	能	需要改调用方

5.2.6 别名：`typedef` 与 `using`

每次写 `struct Student` 有点长。C 语言用 `typedef` 起别名，C++ 更推荐 `using`：

```
struct Student
{
    string name;
    int chinese;
    int math;
};

typedef struct Student Student;   // C 风格
using Stu = Student;              // C++ 风格，等价
```

之后 `Stu s;` 和 `Student s;` 完全一样。竞赛代码里常见的是把结构体名起短一点，比如 `using P = Point;`。

【拓展】C 语言里必须写 `typedef struct Student Student;` 才能在声明变量时省略 `struct` 关键字；C++ 不需要，`struct Student { ... };` 之后直接写 `Student s;` 就行。看到旧代码里的 `typedef` 不必照抄。

5.2.7 `pair`：只有两个成员的结构体

第 3 章 3.6.4 提过 `pair`，现在可以正式理解它了：`pair<A, B>` **就是只有两个成员的结构体类型**，成员名固定叫 `first` 和 `second`。

```
#include <iostream>
#include <utility>
using namespace std;

int main()
{
    pair<int, int> p = {3, 7};
    cout << p.first << ' ' << p.second << '\n';

    p.first = 10;
    cout << p.first << ' ' << p.second << '\n';
    return 0;
}
```

输出：

```
3 7
10 7
```

`pair` 的最大好处是**自带比较规则**：先比 `first`，相同再比 `second`。所以 `sort` 一个 `pair` 数组不需要写比较器，默认就是”第一关键字升序、第二关键字升序”。

什么时候用 `pair`、什么时候用 `struct`？

场景	选择
就是两个数据，且”第一、第二”的含义清楚	`pair`
三个及以上成员	`struct`
成员有明确业务含义（姓名、成绩）	`struct`，可读性远好于 `p.first`
需要自定义比较规则	两者都行，`struct` 更清楚

5.2.8 结构体里放数组与字符串

成员可以是任何类型，包括数组和 `string`：

```
struct Player
{
    string name;
    int score[3];   // 三局成绩
    int total;
};
```

访问时继续用下标：`players[i].score[1]` 是第 `i` 个玩家的第二局成绩。

注意一个坑：**结构体里的数组不能直接用 `=` 整体赋值**（`p.score = {1, 2, 3};` 是错的），要逐个元素赋值，或者在初始化时一次给全：

```
Player p = {"Alice", {10, 20, 30}, 60};   // 正确
```

5.3 自定义排序：给 `sort` 一个比较规则

5.3.1 回顾：`sort` 默认按什么排

第 3 章 3.6.3 学过：

```
sort(a, a + n);                 // 升序
sort(a, a + n, greater<int>()); // 降序
```

默认的升序规则是”小的在前”，也就是 `a < b` 时 `a` 排在 `b` 前面。`greater<int>()` 则是”大的在前”。

这两个规则只对**单个数值**有意义。一旦要排的是结构体，问题就来了：两个学生，谁该排在前面？编译器不知道”总分”是什么，更不知道”总分相同怎么办”。**必须由我们告诉它。**

5.3.2 `cmp` 函数：返回 `true` 表示”排前面”

给 `sort` 传第三个参数——一个比较函数。约定只有一条：

> **`cmp(x, y)` 返回 `true`，表示 `x` 应该排在 `y` 前面。**

先看最简单的降序：

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

#define MAXN 1000

bool cmp(int x, int y)
{
    return x > y;   // x 比 y 大 → x 排前面 → 降序
}

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    sort(a, a + n, cmp);
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << (i + 1 == n ? '\n' : ' ');
    }
    return 0;
}
```

输入：

```
6
3 1 4 1 5 9
```

输出：

```
9 5 4 3 1 1
```

和 `sort(a, a + n, greater<int>())` 的结果完全一样。`cmp` **就是** `greater<int>()` **的手写版本**——理解了这一层，后面所有自定义排序都是同一个套路。

5.3.3 多关键字排序

现在排学生：**总分降序；总分相同按语文降序；语文也相同按姓名字典序升序。**

比较函数要一层一层判断，**每一层不相等就立刻返回**：

```cpp
struct Student
{
    string name;
    int chinese;
    int math;
    int total;
};

bool cmp(const Student &x, const Student &y)
{
    if (x.total != y.total)
    {
        return x.total > y.total;   // 第一关键字：总分降序
    }
    if (x.chinese != y.chinese)
    {
        return x.chinese > y.chinese; // 第二关键字：语文降序
    }
    return x.name < y.name;           // 第三关键字：姓名升序
}
```

三个细节：

1. **参数用 `const Student &` 类型。**排序过程中要比较很多次，传引用避免反复拷贝整个结构体（5.2.5）；
2. **每一层都用 `!=` 判断，而不是 `<=` 判断。**写成 `x.total <= y.total` 就违反了严格弱序，见 5.3.6；
3. **最后一层直接 `return` 即可**，不需要再判断相等——前面所有层都相等时，就由最后一层决定。

完整程序：

```cpp
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

#define MAXN 1000

struct Student
{
    string name;
    int chinese;
    int math;
    int total;
};

Student students[MAXN];

bool cmp(const Student &x, const Student &y)
{
    if (x.total != y.total)
    {
        return x.total > y.total;
    }
    if (x.chinese != y.chinese)
    {
        return x.chinese > y.chinese;
    }
    return x.name < y.name;
}

int main()
{
    int n;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cin >> students[i].name >> students[i].chinese >> students[i].math;
        students[i].total = students[i].chinese + students[i].math;
    }

    sort(students, students + n, cmp);

    for (int i = 0; i < n; i++)
    {
        cout << students[i].name << ' ' << students[i].total << '\n';
    }
    return 0;
}
```

输入：

```
5
bob 80 90
alice 95 85
cindy 70 70
dave 90 80
eve 95 85
```

输出：

```
alice 180
eve 180
dave 170
bob 170
cindy 140
```

手算验证：`alice` 和 `eve` 总分都是 180、语文都是 95，最后按姓名升序，`alice` 在前；`dave` 和 `bob` 总分都是 170，`dave` 语文 90 高于 `bob` 的 80，所以 `dave` 在前。

总分要在读入时算好存进结构体，不要在比较函数里现算 `x.chinese + x.math`——排序会调用比较函数约 `n log n` 次，每次都重算一遍是白费力气。

5.3.4 lambda：把比较规则写在调用处

`cmp` 函数有个小麻烦：它必须定义在 `sort` 外面，离调用处很远，读代码时要来回翻。C++11 起可以用 **lambda**（匿名函数）把规则直接写在 `sort` 那一行：

```
sort(students, students + n, [](const Student &x, const Student &y)
{
    if (x.total != y.total)
    {
        return x.total > y.total;
    }
    return x.name < y.name;
});
```

lambda 的语法是：

```
[捕获列表](参数列表)
{
    函数体
}
```

对照 `cmp` 看：

部分	`cmp` 函数	lambda
名字	`cmp`	没有名字（匿名）
返回类型	写在最前面 `bool`	由 `return` 自动推导
参数列表	`(const Student &x, const Student &y)`	一样
函数体	`{ ... }`	一样
位置	必须定义在外面	**可以直接写在调用处**

lambda 和 `cmp` 函数在 `sort` 眼里完全等价——都是”接受两个元素、返回 `bool` “的东西。选哪个只看可读性：

- 规则简单、只用一次 → lambda，就近写清楚；
- 规则复杂、多处复用 → 写成 `cmp` 函数，起个名字。

5.3.5 捕获列表：让比较规则用上外部数据

lambda 开头那个 `[]` 叫**捕获列表**，它决定 lambda 能用哪些外部变量。这是 lambda 比普通函数强的地方。

场景：给下标数组排序，但比较的是另一个数组里的值。

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

#define MAXN 1000

int a[MAXN];
int idx[MAXN];

int main()
{
    int n;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
        idx[i] = i;
    }

    sort(idx, idx + n, [&](int x, int y)
    {
        return a[x] < a[y];
    });

    for (int i = 0; i < n; i++)
    {
        cout << a[idx[i]] << (i + 1 == n ? '\n' : ' ');
    }
    return 0;
}
```

输入：

```
5
30 10 20 50 40
```

输出：

```
10 20 30 40 50
```

这里排序的是 `idx`（下标），但比较的是 `a[x]` 和 `a[y]`。`[&]` 表示“按引用捕获所有外部变量”，所以 lambda 里能直接读 `a`。

捕获列表的四种写法：

写法	含义	什么时候用
`[]`	不捕获任何外部变量	比较只依赖参数（5.3.2、5.3.3）
`[&]`	按引用捕获全部	**竞赛里最常用**，能读也能改
`[=]`	按值捕获全部（拷贝一份）	需要快照、不希望被外部改动影响
`[&a]` / `[a]`	只捕获指定的变量	想明确写出依赖了什么

按引用捕获不产生拷贝，但**被捕获的变量必须在排序期间一直活着。**上面 `a` 是全局数组，天然满足；如果 `a` 是某个函数里的局部数组，就必须保证 `sort` 在同一个函数里调用。

5.3.6 严格弱序：比较器不能乱写

这是本章**最容易埋雷**的一节。`sort` 内部依赖比较器满足**严格弱序**（strict weak ordering），三条要求：

1. `cmp(x, x)` **必须返回** `false`，即不能有元素排在自己前面；
2. `cmp(x, y)` **为** `true` **时，** `cmp(y, x)` **必须为** `false`，即不能互相排前面；
3. **传递性**：`cmp(x, y)` 和 `cmp(y, z)` 都为 `true` 时，`cmp(x, z)` 也必须为 `true`。

最常见的违例是把 `<` 写成 `<=`：

```
bool cmp(int x, int y)
{
    return x <= y;   // 错误！
}
```

代入第一条：`cmp(3, 3)` 返回 `true`，意味着”3 应该排在 3 前面”。这直接违反要求 1。`sort` 拿到这样的比较器，行为是**未定义的**——可能死循环、可能越界、可能输出乱序，而且**换一台机器结果可能不同。**

正确写法永远是 `<` 或 `>`，**不带等号。**

多关键字排序时，等号的处理方式是”**相等就交给下一层**“，而不是”相等就返回 `true`“：

```
// 错误：总分相等时返回 true，违反严格弱序
bool cmp(const Student &x, const Student &y)
{
    if (x.total == y.total)
    {
        return true;
    }
    return x.total > y.total;
}

// 正确：总分相等时继续比下一关键字
bool cmp(const Student &x, const Student &y)
{
    if (x.total != y.total)
    {
        return x.total > y.total;
    }
    return x.name < y.name;
}
```

还有一个隐蔽的坑：**比较依据不能在排序过程中改变**。比如比较函数里写 `return rand() % 2;` 或者修改全局变量，都会让 `sort` 的行为彻底失控。需要额外信息时，**在排序前算好存进结构体**（5.3.3 的 `total` 就是这么做的）。

5.3.7 排序的稳定性与并列名次

`std::sort` **不保证稳定**：两个”相等”的元素，排序后的相对顺序是不确定的。如果题目要求”总分相同时保持输入顺序”，就要用 `std::stable_sort`，用法和 `sort` 完全一样：

```
stable_sort(students, students + n, cmp);
```

另一种常见需求是**并列名次**：总分相同的人名次相同，下一个名次跳号（1、1、3 而不是 1、1、2）。做法是排序后扫一遍，比较当前元素和上一个元素：

```
int rank = 1;
for (int i = 0; i < n; i++)
{
    if (i > 0 && students[i].total != students[i - 1].total)
    {
        rank = i + 1;   // 名次直接等于"前面有几个人 + 1"
    }
    cout << rank << ' ' << students[i].name << '\n';
}
```

这里 `rank = i + 1` 而不是 `rank++`，因为并列之后名次要跳号。5.6 的综合示例三会完整走一遍。

5.3.8 用 `sort` 排其他东西

`sort` 的适用范围比”数组”广得多，只要给的是”一对位置”：

要排的东西	写法
原生数组前 `n` 个	`sort(a, a + n);`
原生数组降序	`sort(a, a + n, greater<int>());`
结构体数组	`sort(s, s + n, cmp);`
`pair` 数组	`sort(p, p + n);`（自带规则）
`string` 数组	`sort(words, words + n);`（字典序）
`vector`（后续学）	`sort(v.begin(), v.end());`

一个常被问到的组合：**按字符串长度排序，长度相同按字典序**：

```cpp
bool cmp(const string &x, const string &y)
{
    if (x.size() != y.size())
    {
        return x.size() < y.size();
    }
    return x < y;
}
```

注意 `x.size()` 返回的是无符号类型，两个无符号数相减再比较符号会出错，所以这里写成“先判断不等、再比较大小”，而不是 `return x.size() - y.size() < 0`。

5.4 综合示例一：成绩排名（函数 + 结构体 + 排序）

题目：读入 `n` 个学生的姓名、语文成绩、数学成绩，按总分降序输出；总分相同按语文降序；语文也相同按姓名字典序升序。

这道题把本章三块内容全用上了：

1. **结构体**存一个学生的全部信息；
2. **函数**负责“算总分”和“判断谁在前”；
3. `sort` + `cmp` 完成排序。

先看数据怎么组织。一个学生有四项信息，其中总分是算出来的：

```cpp
struct Student
{
    string name;
    int chinese;
    int math;
    int total;
};
```

总分在读入时就算好，不要留到比较函数里现算（5.3.3 解释过原因）。

再看函数怎么拆。比较规则写成独立函数，起名 `cmpStudent`，一眼能看出它比较的是什么：

```cpp
bool cmpStudent(const Student &x, const Student &y)
{
    if (x.total != y.total)
    {
        return x.total > y.total;
    }
    if (x.chinese != y.chinese)
    {
        return x.chinese > y.chinese;
    }
    return x.name < y.name;
}
```

完整程序：

```cpp
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

#define MAXN 1000

struct Student
{
    string name;
    int chinese;
    int math;
    int total;
};

Student students[MAXN];

bool cmpStudent(const Student &x, const Student &y)
{
    if (x.total != y.total)
    {
        return x.total > y.total;
    }
    if (x.chinese != y.chinese)
    {
        return x.chinese > y.chinese;
    }
    return x.name < y.name;
}

int main()
{
    int n;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cin >> students[i].name >> students[i].chinese >> students[i].math;
        students[i].total = students[i].chinese + students[i].math;
    }

    sort(students, students + n, cmpStudent);

    for (int i = 0; i < n; i++)
    {
        cout << students[i].name << ' ' << students[i].total << '\n';
    }
    return 0;
}
```

输入：

```
5
bob 80 90
alice 95 85
cindy 70 70
dave 90 80
eve 95 85
```

输出：

```
alice 180
eve 180
dave 170
bob 170
cindy 140
```

手算验证排序结果：

```
原始数据（姓名 语文 数学 → 总分）：
  bob    80 90 → 170
  alice  95 85 → 180
  cindy  70 70 → 140
  dave   90 80 → 170
  eve    95 85 → 180

第一关键字 总分降序：180 组（alice, eve）→ 170 组（bob, dave）→ 140（cindy）
第二关键字 语文降序：alice 95 = eve 95，打平；dave 90 > bob 80，dave 在前
第三关键字 姓名升序：alice < eve，alice 在前

最终：alice 180 / eve 180 / dave 170 / bob 170 / cindy 140
```

三个可以复用的写法：

- **结构体数组开在全局**，自动清零且不占栈空间；
- **比较函数参数用 `const Student &` 引用**，避免排序过程中反复拷贝；
- **比较函数名带业务含义**（`cmpStudent` 而不是 `cmp`），一个程序里有多套排序规则时不会混。

5.5 综合示例二：区间合并（结构体 + 排序）

题目：给出 `n` 个闭区间 `[l[i], r[i]]`，合并所有有重叠的区间，按左端点升序输出合并结果。

例如 `[1,3]`、`[2,6]`、`[8,10]`、`[15,18]` 合并后是 `[1,6]`、`[8,10]`、`[15,18]`。

思路：先按左端点排序，然后从左往右扫一遍。维护”当前正在合并的区间” `[curL, curR]`：

- 下一个区间的左端点 `l <= curR`：有重叠，把 `curR` 更新成 `max(curR, r)`；
- 否则：当前区间结束，输出它，然后把 `[curL, curR]` 换成新区间。

用 `[1,3] [2,6] [8,10] [15,18]` 手算：

```
排序后： [1,3] [2,6] [8,10] [15,18]

初始 curL = 1, curR = 3
[2,6]：2 <= 3，重叠 → curR = max(3, 6) = 6
[8,10]：8 > 6，不重叠 → 输出 [1,6]，curL = 8, curR = 10
[15,18]：15 > 10，不重叠 → 输出 [8,10]，curL = 15, curR = 18
扫描结束 → 输出 [15,18]

结果：[1,6] [8,10] [15,18]
```

结构体存区间，比较规则只按左端点：

```
struct Range
{
    int left;
    int right;
};

bool cmpRange(const Range &x, const Range &y)
{
    return x.left < y.left;
}
```

完整程序：

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

#define MAXN 1000

struct Range
{
    int left;
    int right;
};

Range ranges[MAXN];

bool cmpRange(const Range &x, const Range &y)
{
    return x.left < y.left;
}

int main()
{
    int n;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cin >> ranges[i].left >> ranges[i].right;
    }

    sort(ranges, ranges + n, cmpRange);

    int curLeft = ranges[0].left;
    int curRight = ranges[0].right;

    for (int i = 1; i < n; i++)
    {
        if (ranges[i].left <= curRight)
        {
            if (ranges[i].right > curRight)
            {
                curRight = ranges[i].right;
            }
        }
        else
        {
            cout << curLeft << ' ' << curRight << '\n';
            curLeft = ranges[i].left;
            curRight = ranges[i].right;
        }
    }
    cout << curLeft << ' ' << curRight << '\n';
    return 0;
}
```

输入：

```
4
1 3
2 6
8 10
15 18
```

输出：

```
1 6
8 10
15 18
```

三个容易写错的地方：

1. `curR` **要取** `max` **值。**如果写成 `curRight = ranges[i].right`，遇到 `[1,10] [2,3]` 这种”被包含”的区间就会把右端点缩回去；
2. **最后一个区间要在循环外输出。**循环里只在”发现不重叠”时输出，扫完最后一个区间后它还没被输出；
3. `n = 0` **要特判。**上面程序直接访问 `ranges[0]`，题目若允许空输入需要先判断。

这道题是**排序 + 扫描**的典型结构：排序把”乱序”变成”有序”，有序之后一次线性扫描就能解决。后面很多题（去重、找中位数、贪心选点）都是这个套路。

5.6 综合示例三：并列名次（排序 + 扫描）

题目：读入 `n` 个学生的姓名与总分，按总分降序输出名次。总分相同的人名次相同，下一个名次跳号（例如 1、1、3）。

思路：先排序，再扫一遍。名次等于”前面有几个人 + 1”，所以并列之后名次会自然跳号。

用 `180 180 170 170 140` 手算：

```
下标 i   总分    判断                                rank
0        180     第一个，rank 保持 1                  1
1        180     180 == 180，并列，rank 不变          1
2        170     170 != 180，rank = i + 1 = 3         3
3        170     170 == 170，并列，rank 不变          3
4        140     140 != 170，rank = i + 1 = 5         5
```

关键就一行：`rank = i + 1` **这一行**，不是 `rank++`。因为 `i` 恰好是”排在我前面的人数”，名次就是它加一。

完整程序：

```cpp
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

#define MAXN 1000

struct Student
{
    string name;
    int total;
};

Student students[MAXN];

bool cmpStudent(const Student &x, const Student &y)
{
    if (x.total != y.total)
    {
        return x.total > y.total;
    }
    return x.name < y.name;
}

int main()
{
    int n;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cin >> students[i].name >> students[i].total;
    }

    sort(students, students + n, cmpStudent);

    int rank = 1;
    for (int i = 0; i < n; i++)
    {
        if (i > 0 && students[i].total != students[i - 1].total)
        {
            rank = i + 1;
        }
        cout << rank << ' ' << students[i].name << ' ' << students[i].total << '\n';
    }
    return 0;
}
```

输入：

```
5
alice 180
eve 180
dave 170
bob 170
cindy 140
```

输出：

```
1 alice 180
1 eve 180
3 bob 170
3 dave 170
5 cindy 140
```

注意比较函数里**总分相同时按姓名升序**——这不是题目要求，而是为了让输出确定。如果总分相同又没写后备规则，`sort` 给出的顺序是不确定的，本地跑对了交上去可能 WA。

如果题目要求的是”1、1、2”（不跳号），把 `rank = i + 1` 改成 `rank++` 即可。**读题时看清是哪种。**

5.7 【拓展】递归：函数调用自己

> 本节是选读内容。解决汉诺塔、斐波那契、全排列这类问题。

5.7.1 递归的两个要素

递归就是函数直接或间接地调用自己。写递归必须同时想清楚两件事：

1. **基线条件**（base case）：什么时候不再往下调用，直接返回答案；
2. **递归关系**：规模为 `n` 的问题，怎么用规模更小的问题的答案算出来。

缺了第一条会无限调用下去，直到栈溢出（RE）；缺了第二条就写不出代码。

5.7.2 阶乘

`n! = 1 * 2 * ... * n`，也可以写成 `n! = n * (n-1)!`，而 `0! = 1`。

```cpp
long long factorial(int n)
{
    if (n <= 1)
    {
        return 1;              // 基线条件
    }
    return n * factorial(n - 1); // 递归关系
}
```

以 `factorial(4)` 看”递下去、归回来”：

```
递下去：
  factorial(4) → 4 * factorial(3)
  factorial(3) → 3 * factorial(2)
  factorial(2) → 2 * factorial(1)
  factorial(1) → 1            ← 基线条件，不再往下

归回来：
  factorial(1) = 1
  factorial(2) = 2 * 1  = 2
  factorial(3) = 3 * 2  = 6
  factorial(4) = 4 * 6  = 24
```

5.7.3 斐波那契：为什么朴素递归很慢

`F[1] = F[2] = 1`，`F[n] = F[n-1] + F[n-2]`。

```
long long fib(int n)
{
    if (n <= 2)
    {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}
```

这个写法**正确但极慢**。算 `fib(5)` 时，`fib(3)` 被算了两遍，`fib(2)` 被算了三遍：

```
              fib(5)
            /        \
        fib(4)       fib(3)
       /     \      /     \
   fib(3)  fib(2) fib(2) fib(1)
   /    \
fib(2) fib(1)
```

调用次数随 `n` 指数增长，`n = 40` 就要跑好几秒。**递归写法直观，但不代表可以直接用**——这是本章要建立的第一个”复杂度直觉”：看到递归，先数一数同一个子问题被算了几次。

改进方向有两个（都超出本章范围，知道存在即可）：把算过的答案存起来（记忆化），或者干脆用循环从 `F[1]` 递推到 `F[n]`。

5.7.4 汉诺塔：递归的经典形态

三根柱子 A、B、C，A 上有 `n` 个从大到小叠放的盘子，要求全部移到 C，每次只能移动一个，且大盘不能压在小盘上。

把”把 `n` 个盘子从 `from` 移到 `to`、借助 `via`“写成一个函数，递归关系是：

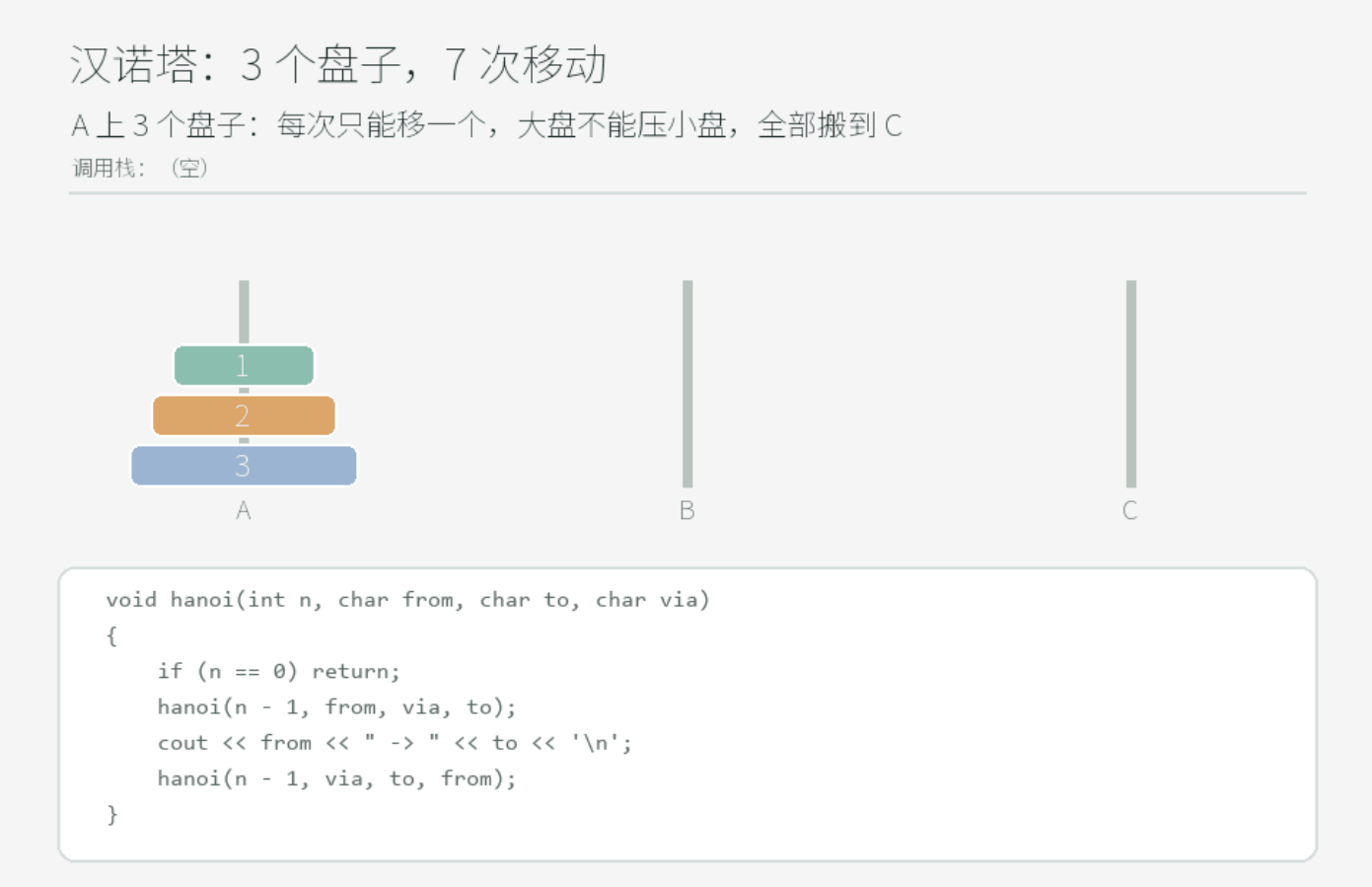
1. 先把上面 `n - 1` 个从 `from` 移到 `via`（借助 `to`）；
2. 把最大的那个从 `from` 移到 `to`；
3. 再把 `n - 1` 个从 `via` 移到 `to`（借助 `from`）。

```
void hanoi(int n, char from, char to, char via)
{
    if (n == 0)
    {
        return;                    // 基线条件：没有盘子要移
    }
    hanoi(n - 1, from, via, to);   // 第 1 步
    cout << from << " -> " << to << '\n'; // 第 2 步
    hanoi(n - 1, via, to, from);   // 第 3 步
}
```

`n = 3` 时的输出（共 `2^3 - 1 = 7` 行）：

```
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

把代码跑起来看一遍：下面的动画把上面这段函数逐行执行的过程画了出来——三根柱子上的盘子实时变化，下方代码面板里**橙色高亮的就是当前正在执行的那一行**，顶部还显示着当前的递归调用栈。注意看”递下去”（连续进入 `hanoi(n - 1, ...)`）和”归回来”（`n == 0` 返回后逐层执行第 2、3 步）是怎么交替出现的：



对照动画看三件事：

1. **每次真正搬盘子，都发生在第 2 步那一行**（`cout << from << " -> " << to`）——递归本身不搬盘子，它只负责把”什么时候该搬哪个”安排好；
2. **调用栈先一路加深到 `hanoi(0, ...)` 这一层**，然后才开始一层层返回，返回的过程中才出现第一次真正的移动；
3. **7 次移动的顺序**与上面 `text` 块里的输出逐行对应，动画里每一次移动的编号（第 1 次、第 2 次……）就是输出的第几行。

移动次数满足 `T(n) = 2 * T(n-1) + 1`，解得 `T(n) = 2^n - 1`。`n = 64` 时约 `1.8e19` 次，这就是”梵塔传说”里世界末日的由来。

5.7.5 什么时候该用递归

适合递归	不适合递归
问题天然分层（树、分治、汉诺塔）	同一个子问题被反复计算（朴素斐波那契）
递归深度小（`n <= 1e4` 量级）	深度可能达到 `1e5` 以上（栈溢出）
递推关系一眼能写出来	递推关系需要额外状态才能表达

竞赛里的默认选择是循环：能写成 `for` 的绝不写递归。递归留给”循环写不出来”的场合，比如树形结构、分治、搜索回溯。

5.8 常见错误

错误一（CE）：结构体定义末尾漏分号

```
struct Student
{
    string name;
    int total;
}          // ← 少了分号
```

现象是编译器在**下一行**报一堆莫名其妙的错（”expected unqualified-id”之类），因为编译器以为你在声明一个变量。看到结构体后面的报错，先回头看分号。

错误二（WA）：以为传值能改外面

```
void addOne(int x) { x = x + 1; }                       // 改的是副本
void swapTwo(int x, int y) { int t = x; x = y; y = t; }  // 交换的是副本
```

现象是”函数里明明改了，外面没变”。需要改调用方时，参数加 `&`（5.1.5）。

错误三（CE / 链接错误）：函数定义写在调用之后却没声明

```
int main()
{
    cout << maxOf(3, 7);   // 编译器还没见过 maxOf
}
int maxOf(int a, int b) { ... }
```

现象是 `'maxOf' was not declared in this scope`。要么把定义挪到前面，要么在 `main` 前补一行声明（5.1.9）。

错误四（RE）：数组作参数忘了传长度

```
long long sumOf(int a[])   // 函数里不知道有几个元素
{
    long long sum = 0;
    for (int i = 0; i < ???; i++) { ... }
}
```

`sizeof(a)` 在函数里得到的是指针大小（8 字节），不是数组大小。**长度必须作为参数一起传**（5.1.7）。

错误五（WA / 未定义行为）：比较器写成 `<=`

```
bool cmp(int x, int y) { return x <= y; }   // 违反严格弱序
```

现象是本地能跑、交上去 WA，或者干脆 RE。`sort` 拿到违反严格弱序的比较器后行为未定义（5.3.6）。**永远写 `<` 或 `>` 之一。**

错误六（WA）：多关键字排序漏了后备规则

```
bool cmp(const Student &x, const Student &y)
{
    return x.total > y.total;   // 总分相同时返回 false，顺序不确定
}
```

总分相同的两个学生，谁在前由 `sort` 内部实现决定，**换台机器可能不一样。**题目要求”总分相同按语文”时必须把第二关键字写出来；题目没要求时也建议补一条（比如按姓名），让输出确定。

错误七（WA）：在比较函数里现算

```
bool cmp(const Student &x, const Student &y)
{
    return x.chinese + x.math > y.chinese + y.math;   // 每次比较都重算
}
```

排序会调用比较函数约 `n log n` 次，每次重算两遍加法。数据量大时这是实打实的常数开销。**排序前算好存进结构体**（5.3.3）。

错误八（CE）：lambda 捕获列表写错

```
sort(idx, idx + n, [](int x, int y) { return a[x] < a[y]; });   // a 没被捕获
```

现象是 `'a' is not captured`。比较用到了外部变量，捕获列表里必须写 `[&]` 或 `[&a]`（5.3.5）。

错误九（WA）：结构体数组没清零就累加

```
Student students[MAXN];   // 全局，自动清零
// 但如果是局部数组，成员是不确定值
```

多组数据时，上一组的 `total` 会带进下一组。每组开始时用 `= {}` 重置，或者干脆在读入时**直接赋值**而不是累加。

错误十（WA）：并列名次用 `rank++`

```
if (students[i].total != students[i - 1].total) { rank++; }   // 得到 1 1 2
```

题目要求”1、1、3”（跳号）时应该写 `rank = i + 1`；要求”1、1、2”（不跳号）时才用 `rank++`。**读题确认**（5.6）。

5.9 本章小结

本章内容可以压缩成下面这条路线：

```
一段逻辑要反复用
    ↓ 起个名字、定好参数和返回值
函数：传值（副本，改不动外面）/ 传引用（别名，能改外面）
    ↓ 数组作参数要连长度一起传
一组数据要绑在一起
    ↓ 定义 struct，用 . 访问成员
结构体：结构体数组、const 引用传参、typedef / using 别名、pair
    ↓ 排序规则不再是"数值大小"
sort 的第三个参数：cmp 函数 或 lambda
    ↓ 多关键字逐层判断，每层不相等就返回
严格弱序：cmp(x, x) 必须为 false，永远写 < 或 >，不写 <=
    ↓ 三个套路
排序 + 扫描（区间合并）/ 排序 + 名次（并列名次）/ 排序 + 多关键字（成绩排名）
```

需要真正记住的关键点：

- **传值是副本，传引用是别名**；想改调用方的变量就加 `&`，调用时不写 `&`；
- **数组作参数必须连长度一起传**，函数里 `sizeof` 拿不到数组长度；函数里改数组元素会改到原数组；
- **结构体定义末尾有分号**；成员用 `.` 访问；结构体数组是”先下标、再成员”两层访问；
- **传结构体优先用 `const Student &` 引用**，既不拷贝又不会误改；
- **`cmp(x, y)` 返回 `true` 表示 `x` 排前面**；降序就是 `x > y`；
- **多关键字排序逐层判断，每层写 `!=`，最后一层直接返回**；
- **比较器必须满足严格弱序**，写成 `<=` 会让 `sort` 行为未定义；
- **lambda 的 `[&]` 按引用捕获外部变量**，按另一个数组排序时靠它；
- **排序前把要比较的值算好存进结构体**，不要在比较函数里现算；
- **并列名次用 `rank = i + 1` 计算**（跳号），不是 `rank++`。

5.10 配套练习

第五次训练题单已发布在 SYUCTOJ：2026级SYUCT-ACM新生第五次训练

第一部分：函数（对应 5.1）

1. 区间求和（第 V 题）
2. 质数判断（第 W 题）
3. 传值与传引用（第 X 题）
4. 数组的最值与平均值（第 Y 题）
5. 最大公约数与最小公倍数（第 Z 题）

第二部分：结构体（对应 5.2）

6. 学生成绩单（第 AA 题）
7. 两点距离（第 AB 题）
8. `pair` 排序（第 AC 题）
9. 平均分最高的学生（第 AD 题）

第三部分：自定义排序（对应 5.3）

10. 降序排序（第 AE 题）
11. 绝对值排序（第 AF 题）
12. 按长度排序字符串（第 AG 题）
13. 成绩排名（第 AH 题）
14. 按另一个数组排序（第 AI 题）
15. 区间合并（第 AJ 题）