第 4 章 字符串与二维数组

> 本文是《XCPCer 从零开始的 C/C++ 入门指南（算法竞赛向）》系列第 4 章，系列训练大纲见本文。

4.0 导语：文本与表格

前三章里，我们处理的主要是数字。但竞赛题里还有两大类“非数字”数据：

- **字符串**：姓名、单词、文章、DNA 序列、棋盘的一行……本质是一串字符；
- **二维数组**：矩阵、棋盘、成绩表、地图……本质是一张表格，需要**两个下标**才能定位一个格子。

第 3 章末尾已经见过二维数组的声明，但从训练反馈看，不少同学对“行、列、两个下标”还没有形成直觉。所以本章把两件事放在一次训练里：**先讲字符串，再把二维数组从头系统重讲一遍**，每一项都配 3×3 或 3×4 的小矩阵手算。

本次训练结束后，你应该能够：

- 定义、读入、遍历和修改 `string`，并能正确选择 `cin >> s` 与 `getline`；
- 完成拼接、字典序比较、大小写转换、反转、回文判断、`find` / `substr`、数字串求和；
- 使用字符串数组和字符网格；
- 系统掌握二维数组的读写与十个固定套路：加法、行和列和、最值位置、两条对角线、每行逆序、转置、顺时针旋转、边框、四邻格、子矩阵求和；
- 通过三道综合题，初步见识贪心、构造、数学三种解题思路。

内容分层：

- **主线**：`string` + 二维数组，全部是基础必做内容；
- **【拓展】**：`char[]`、`'\0'`、`strlen` / `strcmp` 等 C 风格字符串写法。这部分只需要“能看懂旧代码”，主线一律使用 `string`。//请注意

> **字符串是"一排带下标的字符"，二维数组是"一张带两个下标的格子"。把下标和循环这两件旧工具用在新数据上，就是本章的全部内容。**

4.1 从 char 到 string

第 1、2 章已经学过：`char` 存一个字符，字符本质是 ASCII 编码，其中 `'0'~'9'`、`'A'~'Z'`、`'a'~'z'` 各自连续。

现实中我们很少只处理一个字符，更多时候是一串：一个单词、一句话。C++ 标准库提供了 `string`（字符串）类型，头文件是 `<string>`：

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

int main()
{
    string s = "hello";
```

```
    cout << s[0] << ' ' << s[4] << '\n';
    cout << s << '\n';
    return 0;
}
```

运行结果：

```
h o
hello
```

把 `s` 想象成一排带下标的格子，和第 3 章的数组一模一样：

```
下标:  0   1   2   3   4
字符:  h   e   l   l   o
```

要点：

- `string` 的下标同样从 **0** 开始，长度为 5 时合法下标是 `[0, 5)`；
- `"hello"` 用**双引号**括起来，表示字符串；`'h'` 用**单引号**括起来，表示单个字符，两者不是一回事；
- 声明时不给内容，`string s;` 就是一个**空字符串**，长度为 0，后续可以往里添加内容。

4.2 读入字符串：`cin >> s` 与 `getline`

字符串有**两种常用读法**，选错是本章最高频的 WA 来源，必须从一开始分清楚。

4.2.1 `cin >> s`：读到空白为止

`cin >> s` 会**跳过开头的空格和换行**，从第一个非空白字符开始读，一直读到**下一个空白**（空格、换行、制表符）为止，空白本身不会进入字符串：

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

int main()
{
    string s;
    cin >> s;
    cout << "[" << s << "] size = " << s.size() << '\n';
    return 0;
}
```

输入：

```
hello world
```

输出：

```
[hello] size = 5
```

输入里明明有 "hello world"，但 `s` 只装下了 `hello`——读到中间的空格就停了。竞赛题里的“一个单词”“一个不含空格的串”，都用这种读法。

4.2.2 `getline(cin, s)`：读进整行

`getline` 从当前位置一直读到**换行符**为止，中间的空格全部保留，行尾的换行符会被吃掉、但不进入字符串：

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

int main()
{
    string line;
    getline(cin, line);
    cout << "[" << line << "] size = " << line.size() << '\n';
    return 0;
}
```

输入：

```
hello world
```

输出：

```
[hello world] size = 11
```

`5 个字母 + 1 个空格 + 5 个字母 = 11` 个字符。题目给出“一行文本”“一句话”时，用 `getline`。要数一行里有几个空格，就遍历这个字符串、统计字符 `' '` 出现了多少次（配套练习 2“空格计数”就是这个）。

4.2.3 混用时的经典坑：残留的换行符

把输入想象成一根传送带，上面排着一个个字符。来看一个“看起来没错”的程序：

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

int main()
{
    int n;
    string name;
    cin >> n;
    getline(cin, name);
    cout << n << ' ' << "[" << name << "]" << '\n';
    return 0;
}
```

输入：

```
3
Alice
```

预期读入 `n = 3`、`name = "Alice"`，实际输出：

```
3 []
```

`name` 是空的。用传送带模型看发生了什么：

```
输入内容：    3  '\n'  A  l  i  c  e  '\n'
              ↑
cin >> n 取走 "3"，停在 '\n' 前面
剩下：            '\n'  A  l  i  c  e  '\n'
getline 一上来就遇到 '\n' → 读到空串，换行符被吃掉
```

`cin >> n` 只取走了 `3`，**行尾的 `'\n'` 还留在输入里**；`getline` 遇到它，读到的是空串。这就是“读数后残留的换行符”问题。

修复方法：在两者中间用 `cin.ignore()` 把这一个残留的换行符丢掉。下面把“先读一个数、再连续读 `n` 行文本”这个完整套路写在一起：

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

#define MAXN 100

int main()
```

```
{
    int n;
    cin >> n;
    cin.ignore(); // 丢掉 n 后面那一行的换行符

    string lines[MAXN];
    for (int i = 0; i < n; i++)
    {
        getline(cin, lines[i]);
    }

    for (int i = 0; i < n; i++)
    {
        cout << i << ": [" << lines[i] << "]\n";
    }
    return 0;
}
```

输入：

```
2
hello world
good morning
```

输出：

```
0: [hello world]
1: [good morning]
```

如果删掉 `cin.ignore()`，第 0 行会读成空串，后面整体错一行。记住判断口诀：**用 `cin >>` 读过数字或单词之后再接 `getline`，中间就可能需要一次 `ignore`**；全程只用 `cin >>` 或全程只用 `getline`，则不需要。

4.3 长度、下标、遍历与修改

4.3.1 长度与下标

`s.size()` 返回字符串长度。它的返回值类型是 `size_t`（一种无符号整数），初学阶段直接把它存进 `int` 变量即可：

```
int len = s.size();
```

之后的一切都和一维数组相同：下标 `[0, len)`，最后一个字符是 `s[len - 1]`，访问 `s[len]` 就是越界。

4.3.2 遍历与修改

读入一个字符串，先按“下标 : 字符”的格式打印一遍，再把首字符改成 `'A'`：

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

int main()
{
    string s;
    cin >> s;

    int len = s.size();
    for (int i = 0; i < len; i++)
    {
        cout << i << ": " << s[i] << '\n';
    }

    s[0] = 'A'; // 像数组元素一样修改
    cout << s << '\n';
    return 0;
}
```

输入：

```
code
```

输出：

```
0: c
1: o
2: d
3: e
Aode
```

字符串的首尾字符就是 `s[0]` 和 `s[len - 1]`。配套练习 1“字符串的头和尾”还要同时输出长度，写法是：

```
cout << len << ' ' << s[0] << ' ' << s[len - 1] << '\n';
```

如果只需要读取每个字符、不关心下标，可以用第 3 章见过的范围 `for`：

```
for (char c : s)
{
    cout << c << ' ';
```

```
}
cout << '\n';
```

输入 `code`，输出（每个字符后面都有一个空格）：

```
c o d e
```

要修改字符时用下标循环，只读时用范围 `for`。配套练习 5“字符替换”要求把所有等于 `x` 的字符改成 `y`，不能只改第一个。假设 `s` 和 `len` 已准备好，核心代码是：

```
char x, y;
cin >> x >> y;
for (int i = 0; i < len; i++)
{
    if (s[i] == x)
    {
        s[i] = y;
    }
}
cout << s << '\n';
```

例如 `s = "banana"`、`x = 'a'`、`y = 'o'`，三个 `a` 都会改成 `o`，输出 `bonono`。

4.4 拼接、字典序比较与大小写转换

4.4.1 拼接

`+` 把两个字符串接成新串，`+=` 把内容追加到末尾：

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

int main()
{
    string a = "abc";
    string b = "xyz";

    string c = a + b; // 拼出一个新字符串
    a += b;           // 把 b 追加到 a 的末尾

    cout << c << '\n';
    cout << a << '\n';
    return 0;
}
```

运行结果：

```
abcxyz
abcxyz
```

4.4.2 比较：字典序

字符串之间可以直接用 `==`、`!=`、`<`、`>` 比较。**小于号的含义是字典序**——和查英语字典的规则完全一样：

1. 从头开始逐个字符比较，遇到第一对不同的字符，谁的字符编码小，谁就更小；
2. 如果一个串是另一个串的前缀，短的更小。

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

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

    if (a == b)
    {
        cout << "equal\n";
    }
    else if (a < b)
    {
        cout << a << " < " << b << '\n';
    }
    else
    {
        cout << a << " > " << b << '\n';
    }
    return 0;
}
```

输入：

```
apple apply
```

输出：

```
apple < apply
```

逐字符对照：a=a、p=p、p=p、l=l，第 5 个字符 'e' 和 'y' 第一次不同，'e' 编码更小，所以 apple < apply。另外 app < apple（前缀更短更小）、banana > apple（首字符 b > a）。

> **特别提醒：字符串的 "10" 按字典序小于 "9"**——因为比较的是字符 '1' 和 '9'，不是数值 10 和 9。要比较数值请先用 stoi 转成数字（见 4.7 的【拓展】）。

4.4.3 大小写互换与字符分类

第 2 章学过：大小写字母编码都连续，且 'a' 比 'A' 大 32。小写转大写用 c - 'a' + 'A'，大写转小写用 c - 'A' + 'a'。配套练习 4“大小写互换”可能包含空格，所以这里用 getline 读整行；数字、空格和标点都原样保留：

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

int main()
{
    string s;
    getline(cin, s);

    int len = s.size();
    for (int i = 0; i < len; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
        {
            s[i] = s[i] - 'a' + 'A';
        }
        else if (s[i] >= 'A' && s[i] <= 'Z')
        {
            s[i] = s[i] - 'A' + 'a';
        }
    }

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

输入：

```
SyuctOJ 04: AbZ-z!
```

输出：

```
sYUCToj 04: aBz-Z!
```

两个 if 分别处理两种字母；不在英文字母范围内的字符不会进入任何分支。

配套练习 3“元音计数”要求按 `a e i o u` 的顺序分别计数，并把大写元音合并到对应的小写项。沿用上面的分类方法，核心代码如下：

```
int counts[5] = {};
for (char c : s)
{
    if (c >= 'A' && c <= 'Z')
    {
        c = c - 'A' + 'a';
    }

    if (c == 'a') counts[0]++;
    else if (c == 'e') counts[1]++;
    else if (c == 'i') counts[2]++;
    else if (c == 'o') counts[3]++;
    else if (c == 'u') counts[4]++;
}

for (int i = 0; i < 5; i++)
{
    cout << counts[i] << (i == 4 ? '\n' : ' ');
}
```

例如 `HeLloWorld` 输出 `0 1 0 2 0`。`y` 和 `Y` 都不是元音，不要因为它们有时发音像元音就计入。

【拓展】标准库的 `<cctype>` 提供 `toupper(c)`、`tolower(c)`、`isdigit(c)` 等现成的字符函数。主线先坚持手写转换——能手算出 `'c' - 'a' + 'A'` 的结果，再用工具才有底气。

4.5 反转与回文判断

4.5.1 反转：两个下标向中间靠拢

第 3 章数组逆序的思路原封不动搬过来：左下标从 0 向右、右下标从末尾向左，逐对交换：

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

int main()
{
    string s;
    cin >> s;

    int left = 0;
    int right = (int)s.size() - 1;
    while (left < right)
    {
        swap(s[left], s[right]);
        left++;
```

```
        right--;
    }

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

输入：

```
abcdef
```

输出：

```
fedcba
```

交换过程：`(a,f)` 换、`(b,e)` 换、`(c,d)` 换，两个下标在中间相遇。长度为奇数时中间的字符不需要动；长度为 0 时 `right = -1`，循环一次也不执行，同样安全。

4.5.2 回文判断：对称位置两两相等

回文串是正着读、反着读完全一样的字符串，例如 `level`、`noon`。判断方法：对称位置的字符必须全部相等，一旦发现一对不等就可以立刻下结论：

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

int main()
{
    string s;
    cin >> s;

    bool palindrome = true;
    int left = 0;
    int right = (int)s.size() - 1;
    while (left < right)
    {
        if (s[left] != s[right])
        {
            palindrome = false;
            break;
        }
        left++;
        right--;
    }

    if (palindrome)
```

```
    {
        cout << "YES\n";
    }
    else
    {
        cout << "NO\n";
    }
    return 0;
}
```

输入：

```
level
```

输出：

```
YES
```

逐对检查 `level`：第 0 对 `(l, l)` 相等，第 1 对 `(e, e)` 相等，剩下中间的 `v` 不需要任何比较。换成 `noon` 会检查 `(n, n)`、`(o, o)` 两对，同样输出 `YES`；换成 `code`，第一对 `(c, e)` 就不等，输出 `NO`。配套练习 7 要求的就是这种全大写格式。

这个“左右指针向中间靠拢”的写法后面还会反复出现，请把它当成固定套路记住。

4.6 查找子串与截取：`find` 与 `substr`

4.6.1 查找子串 `find`

`s.find(t)` 在 `s` 中查找子串 `t` **第一次出现的位置**（下标从 0 开始）；找不到时返回一个特殊值 `string::npos`。最稳妥的写法是先保留返回值的 `size_t` 类型，再直接和 `string::npos` 比较，不依赖类型转换后的结果：

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

int main()
{
    string text, word;
    cin >> text >> word;

    size_t position = text.find(word);
    if (position == string::npos)
    {
        cout << -1 << '\n';
    }
```

```
    else
    {
        cout << position << '\n';
    }
    return 0;
}
```

输入：

```
abracadabra cad
```

输出：

```
4
```

对照下标看：`a b r a c a d a b r a` 的下标是 0 到 10，子串 `cad` 从下标 4（字符 `c`）开始第一次出现。若输入 `xyz ab`，输出 `-1`，这正是配套练习 8“子串在哪里”要求的格式。

使用 `find` 必须处理“找不到”的分支：不要在没有判断时把返回值当成合法下标使用。`string::npos` 也不等于字符串的最后一个位置。

4.6.2 截取子串 `substr`

`s.substr(pos, len)` 从下标 `pos` 开始，截取 `len` 个字符，返回一个新字符串：

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

int main()
{
    string s;
    cin >> s;

    int pos, len;
    cin >> pos >> len;

    cout << s.substr(pos, len) << '\n';
    return 0;
}
```

输入：

```
abcdefg
```

```
2 3
```

输出：

```
cde
```

对照下标：`0:a 1:b 2:c 3:d 4:e 5:f 6:g`，从下标 2 开始取 3 个，正好是 `c d e`。

配套练习 9“截取一段”给的是**从 1 开始的闭区间** `[l, r]`，要同时包含第 `l` 个和第 `r` 个字符。先换算成 `substr` 需要的“从 0 开始的起点 + 长度”：

```
cout << s.substr(l - 1, r - l + 1) << '\n';
```

其中 `l - 1` 是起点，`r - l + 1` 是长度。比如 `s = "helloworld"`、`l = 4`、`r = 7`，输出 `lowo`。不要把第二个参数误写成 `r` 或 `r - l`。

两个边界规则：

- 起点合法但 `pos + len` 超过结尾时，自动截到结尾为止，例如 `"abcdefg".substr(5, 10)` 得到 `"fg"`；
- 起点本身超过长度（例如 `"abc".substr(5, 1)`）会让程序异常退出（RE），使用前要保证 `pos` 合法，典型写法是 `pos < (int)s.size()`。

4.7 数字串：字符与数值

输入保证是一串数字字符（如 `"20241024"`）时，每个字符的“数值”是 `c - '0'`（第 2 章的 ASCII 区间知识）。把它们累加起来就是数字串求和：

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

int main()
{
    string s;
    cin >> s;

    long long sum = 0;
    for (char c : s)
    {
        sum += c - '0';
    }

    cout << sum << '\n';
```

```
    return 0;
}
```

输入：

```
20241024
```

输出：

```
15
```

手算验证：`2 + 0 + 2 + 4 + 1 + 0 + 2 + 4 = 15`。

两个注意点：

- 若输入里可能混入非数字字符，要先判断 `c >= '0' && c <= '9'` 再转换；
- `'2'` 是字符（编码 50），`2` 才是数值，混用会得到莫名其妙的答案。

【拓展】`<string>` 还提供 `stoi(s)`（字符串转 `int`）、`stoll(s)`（转 `long long`）、`to_string(x)`（数值转字符串）。数字很长（超过 18 位）或需要逐位处理时，仍用本节的逐字符方法。

4.8 字符串数组与字符网格

4.8.1 字符串数组

把 `string` 放进数组，就得到一批单词。读入 `n` 个单词并输出每个单词的长度：

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

#define MAXN 100

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

    if (n <= 0 || n > MAXN)
    {
        return 0;
    }

    string words[MAXN];
    for (int i = 0; i < n; i++)
    {
```

```
        cin >> words[i];
    }

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

输入：

```
3
apple banana cat
```

输出：

```
apple 5
banana 6
cat 3
```

`words[i]` 是第 `i` 个字符串，`words[i][j]` 是其中第 `j` 个字符——**字符串数组天然就是二维结构**。

4.8.2 字符网格

如果题目给出一个 `n` 行 `m` 列的字符棋盘（每行是一个不含空格的串），用字符串数组读入最方便：`cin >> grid[i]` 一整行读进来。下面统计棋盘中 `*` 的数量：

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

#define MAXN 100

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

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

    int count = 0;
    for (int i = 0; i < n; i++)
```

```
    {
        for (int j = 0; j < m; j++)
        {
            if (grid[i][j] == '*')
            {
                count++;
            }
        }
    }

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

输入：

```
3 4
.*..
..*.
*...
```

输出：

```
3
```

把每个字符串看成网格的一行：`grid[i][j]` 就是第 `i` 行第 `j` 列的格子。对照棋盘数一遍：`(0,1)`、`(1,2)`、`(2,0)` 三个 `*`，输出 3。

如果网格的行内可能包含空格，则不能用 `cin >>`，要改用 `getline(cin, grid[i])` 逐行读入，并记得 4.2.3 的残留换行处理。

4.9 【拓展】`char[]`、`'\0'` 与 `strlen` / `strcmp`

> 本节是**兼容阅读**内容：很多旧讲义、旧模板用 C 风格的字符数组写字符串。你不需要用它写新代码，但见到时要能看懂，也要知道它为什么容易出事故。

C 语言没有 `string` 类型，只能用字符数组加**一个特殊约定**来表示字符串：以编码为 0 的空字符 `'\0'` 作为结尾标记，好比字符串的“句号”。

```
char s[100] = "hello";
```

这行代码在内存里实际存放的是 `h e l l o '\0'`，共占 6 个格子——**字符数 + 1 才是它真正占用的空间**。围绕这个约定，C 提供了一组函数（头文件 `<cstring>`）：

函数	作用	与 string 的对应
strlen(s)	求长度（不统计 '\0'）	s.size()
strcmp(a, b)	字典序比较，返回 0 表示相等、非 0 表示不等	a == b、a < b
strcpy(a, b)	把 b 拷贝到 a	赋值 /=
strcat(a, b)	把 b 拼接到 a 末尾	a += b

旧代码里常见的 strlen(s)、strcmp(a, b) == 0 按上表对照 s.size()、a == b 理解即可。

需要知道的三条安全边界：

1. **容量必须 ≥ 字符数 + 1**。char s[5] = "hello"; 看上去正好，实际还要放 '\0'，已经越界。
2. **没有 '\0' 就没有字符串**。strcat、strlen、cout << s 都会从起点一路读到 '\0' 为止；若数组里忘了写它，就会一路读到越界。
3. **strcpy / strcat 不检查目标容量**，历史上因此产生的安全事故不计其数；用 scanf 读入时要写宽度限制 scanf("%99s", s) 防止溢出。

最后一条必须单独强调：**gets 已经从 C 标准中移除**（C11 起正式删除），原因是它完全无法限制读入长度，必然越界。旧讲义中 gets(s) 的写法请不要再用；读一整行请用 getline(cin, s)，或 C 风格的 fgets(s, 100, stdin)。

主线请继续使用 string：长度、越界、拼接都由它自动管理，本章及之后的代码默认都是 string。

4.10 二维数组：行列坐标

第 3 章出现过二维数组的声明，鉴于大家对着鬼东西都不太熟悉，我们这次从坐标角度理解二维数组。

4.10.1 声明与坐标

```
int a[3][4];
```

表示一张 **3 行 4 列**的表格，合法下标是两个范围的组合：

- 第一个下标（行）范围 [0, 3)；
- 第二个下标（列）范围 [0, 4)。

```
          col 0   col 1   col 2   col 3
row 0    [0][0]  [0][1]  [0][2]  [0][3]
row 1    [1][0]  [1][1]  [1][2]  [1][3]
row 2    [2][0]  [2][1]  [2][2]  [2][3]
```

数组容量要按题面给出的最大数据范围估算。本章配套题的行、列上限不超过 1000，因此示例使用 #define MAXN 1000，实际只访问前 n 行前 m 列。较大的二维数组定义在函数外，既能自动清零，也能避免占用过多局部栈空间。

4.10.2 读入与输出模板

外层循环管行、内层循环管列，这是唯一的基本结构：

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

#define MAXN 1000

int a[MAXN][MAXN];

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

    if (n <= 0 || n > MAXN || m <= 0 || m > MAXN)
    {
        return 0;
    }

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

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

输入：

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```

输出：

```
1 2 3 4
5 6 7 8
9 10 11 12
```

矩阵是**按行优先**读入的：第一个数 1 落到 `a[0][0]`，第二个数 2 落到 `a[0][1]`……第 4 个数 4 填完第 0 行，第 5 个数 5 开始填第 1 行。输出端 `column + 1 == m` 控制“每行末尾换行、元素之间空格”。这是本章二维读入输出的基础模板。4.11～4.15 各自保留完整示例；同类的第二个操作只展示需要新增或替换的核心代码，并写明插入位置，输出除非特别说明都沿用本模板。

4.11 逐项训练一：矩阵加法

4.11.1 对应位置相加

矩阵加法的定义就是“对应位置相加”：`c[i][j] = a[i][j] + b[i][j]`，结果仍是一个 `n` 行 `m` 列的矩阵。

拿两个 3×3 矩阵手算一遍：

```
1 2 3      9 8 7      1+9  2+8  3+7     10 10 10
4 5 6  +   6 5 4  =   4+6  5+5  6+4  =  10 10 10
7 8 9      3 2 1      7+3  8+2  9+1     10 10 10
```

例如右下角：`c[2][2] = a[2][2] + b[2][2] = 9 + 1 = 10`。完整程序：

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

#define MAXN 1000

int a[MAXN][MAXN];
int b[MAXN][MAXN];

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

    if (n <= 0 || n > MAXN || m <= 0 || m > MAXN)
    {
        return 0;
    }

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            cin >> a[row][column];
        }
```

```
    }
    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            cin >> b[row][column];
        }
    }

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            int sum = a[row][column] + b[row][column];
            cout << sum << (column + 1 == m ? '\n' : ' ');
        }
    }
    return 0;
}
```

输入：

```
3 3
1 2 3
4 5 6
7 8 9
9 8 7
6 5 4
3 2 1
```

输出：

```
10 10 10
10 10 10
10 10 10
```

与手算完全一致。读入次序是先读完整个 `a`、再读整个 `b`，输入也按这个顺序给出。

4.12 逐项训练二：行和、列和与最值位置

4.12.1 每一行的和与每一列的和

行和：`rowSum[i]` 累加第 `i` 行的所有 `a[i][j]`；列和：`columnSum[j]` 累加第 `j` 列的所有 `a[i][j]`。用 3×3 矩阵手算：

```
1 2 3      第 0 行：1+2+3     = 6
4 5 6      第 1 行：4+5+6     = 15
7 8 9      第 2 行：7+8+9     = 24

第 0 列：1+4+7 = 12
第 1 列：2+5+8 = 15
第 2 列：3+6+9 = 18
```

注意同一个元素 `a[i][j]` 对行和与列和各贡献一次，两层循环里两个累加器同时更新即可：

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

#define MAXN 1000

int a[MAXN][MAXN];

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

    if (n <= 0 || n > MAXN || m <= 0 || m > MAXN)
    {
        return 0;
    }

    long long rowSum[MAXN] = {};
    long long columnSum[MAXN] = {};

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            cin >> a[row][column];
            rowSum[row] += a[row][column];
            columnSum[column] += a[row][column];
        }
    }

    for (int row = 0; row < n; row++)
    {
        cout << rowSum[row] << '\n';
    }

    for (int column = 0; column < m; column++)
    {
        cout << columnSum[column] << (column + 1 == m ? '\n' : ' ');
    }
```

```
    return 0;
}
```

输入：

```
3 3
1 2 3
4 5 6
7 8 9
```

输出：

```
6
15
24
12 15 18
```

`rowSum` 与 `columnSum` 在定义处用 `= {}` 清零，保证累加从 0 开始。配套练习 13“每行的和”使用前面的 `n` 行输出；配套练习 14“每列的和”使用最后那段空格分隔的列和输出，两道题分别只输出题目要求的部分，不要额外添加 `row sums:` 等说明文字。

4.12.2 最大值的位置

“最大值是多少”保存数值；“最大值在哪里”要额外保存一对下标。沿用第 3 章的原则：从 `a[0][0]` 初始化，用严格大于 `>` 更新，这样得到的是**按行优先顺序第一个**最大值。

用下面的 3×3 矩阵走一遍：

```
3 8 1
5 9 2
7 4 6
```

- 初始：最大值 3，位置 `(0, 0)`；
- `8 > 3`：更新为 8，位置 `(0, 1)`；
- 9 出现：`9 > 8`，更新为 9，位置 `(1, 1)`；
- 此后 `2 7 4 6` 都不超过 9，不再更新。

最终最大值 9 位于下标 `(1, 1)`，即**第 2 行第 2 列**（输出时按下标 +1 换算成题目常说的“第几个”）。在 4.10.2 模板读入完成后，把模板的输出循环替换为以下核心代码：

```
    int maximum = a[0][0];
    int answerRow = 0;
    int answerColumn = 0;
    for (int row = 0; row < n; row++)
```

```
    {
        for (int column = 0; column < m; column++)
        {
            if (a[row][column] > maximum)
            {
                maximum = a[row][column];
                answerRow = row;
                answerColumn = column;
            }
        }
    }

    cout << maximum << ' ' << answerRow + 1 << ' ' << answerColumn + 1 << '\n';
```

输入沿用 4.10.2 的读法：

```
3 3
3 8 1
5 9 2
7 4 6
```

输出：

```
9 2 2
```

配套练习 15 要求的就是这一行三个整数：最大值、行号、列号，不要添加英文标签。

若题目要求“任意一个最大值位置”，上面的写法已经够用；要求“最后一个”时把 `>` 改成 `>=` 即可。

4.13 逐项训练三：两条对角线与每行逆序

4.13.1 方阵的两条对角线

对方阵（`n` 行 `n` 列）：

- **主对角线**（左上到右下）：元素满足 `row == column`；
- **副对角线**（右上到左下）：元素满足 `row + column == n - 1`。

用 3×3 矩阵代入验证：

```
(0,0)=1 ····· (0,2)=3
  ·   (1,1)=5   ·
(2,0)=7 ····· (2,2)=9
```

```
主对角线：a[0][0] + a[1][1] + a[2][2] = 1 + 5 + 9  = 15
副对角线：a[0][2] + a[1][1] + a[2][0] = 3 + 5 + 7  = 15
```

副对角线公式请特别手算记忆：3×3 时 `n - 1 = 2`，`(0,2)`：`0+2=2` ✓；`(1,1)`：`1+1=2` ✓；`(2,0)`：`2+0=2` ✓。

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

#define MAXN 1000

int a[MAXN][MAXN];

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

    if (n <= 0 || n > MAXN)
    {
        return 0;
    }

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

    long long mainSum = 0;
    long long antiSum = 0;
    for (int row = 0; row < n; row++)
    {
        mainSum += a[row][row];
        antiSum += a[row][n - 1 - row];
    }

    cout << mainSum << '\n';
    cout << antiSum << '\n';
    return 0;
}
```

输入：

```
3
1 2 3
4 5 6
7 8 9
```

输出：

```
15
15
```

配套练习 16 要求第一行是主对角线和、第二行是副对角线和，不要添加英文标签。方阵有正中心时，这个中心分别属于两条对角线，应当在两个答案中各计算一次。

4.13.2 每一行各自逆序

“每行逆序”是把每一行内部左右翻转，行与行之间的顺序不变。对 3×4 矩阵：

```
 1  2  3  4        4  3  2  1
 5  6  7  8   →    8  7  6  5
 9 10 11 12       12 11 10  9
```

第 0 行的交换过程：`(1,4)` 换、`(2,3)` 换。每一行都是第 3 章数组逆序的小号复刻，只是交换时带上行下标。在 4.10.2 模板读入完成后、输出循环之前，插入以下核心代码，输出仍用模板的输出循环：

```
    for (int row = 0; row < n; row++)
    {
        for (int left = 0, right = m - 1; left < right; left++, right--)
        {
            swap(a[row][left], a[row][right]);
        }
    }
```

输入：

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```

输出：

```
4 3 2 1
8 7 6 5
12 11 10 9
```

4.14 逐项训练四：转置与顺时针旋转

4.14.1 转置

转置就是把行变列、列变行：`n` 行 `m` 列的矩阵转置后变成 `m` 行 `n` 列，公式：

```
b[column][row] = a[row][column]
```

用 3×4 矩阵手算：`a[0][0] = 1` 放到 `b[0][0]`；`a[2][0] = 9` 放到 `b[0][2]`；`a[1][2] = 7` 放到 `b[2][1]`……逐格搬完：

```
1  2  3  4        1 5 9
5  6  7  8   →    2 6 10
9 10 11 12        3 7 11
                  4 8 12
```

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

#define MAXN 1000

int a[MAXN][MAXN];
int transposed[MAXN][MAXN];

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

    if (n <= 0 || n > MAXN || m <= 0 || m > MAXN)
    {
        return 0;
    }

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            cin >> a[row][column];
        }
```

```
    }

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            transposed[column][row] = a[row][column];
        }
    }

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

输入：

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```

输出：

```
1 5 9
2 6 10
3 7 11
4 8 12
```

注意输出循环的行数、列数已经互换：外层 `row < m`、内层 `column < n`。

4.14.2 顺时针旋转 90°

`n` 行 `m` 列的矩阵顺时针旋转 90° 后变成 `m` 行 `n` 列。下标公式：

```
b[column][n - 1 - row] = a[row][column]
```

先用四个角验证这个公式（最容易记错的地方就在 `-1` 和行列互换）：

```
a[0][0] = 1   →  b[0][2]     （左上角 → 右上角）
a[0][3] = 4   →  b[3][2]     （右上角 → 右下角）
a[2][0] = 9   →  b[0][0]     （左下角 → 左上角）
a[2][3] = 12  →  b[3][0]     （右下角 → 左下角）
```

以 `a[2][0] = 9` 为例：代入公式左边 `row = 2, column = 0`，得到 `b[0][3 - 1 - 2] = b[0][0]` ✓。全部搬完：

```
1  2  3  4        9 5 1
5  6  7  8   →    10 6 2
9 10 11 12        11 7 3
                  12 8 4
```

先在函数外、模板的 `a` 旁边添加 `int rotated[MAXN][MAXN];`，避免大数组占用局部栈空间。再在 4.10.2 模板读入完成后，把模板的输出循环替换为以下核心代码：

```
    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            rotated[column][n - 1 - row] = a[row][column];
        }
    }

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

输入：

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```

输出：

```
9 5 1
10 6 2
```

```
11 7 3
12 8 4
```

与手算一致。记忆建议：先在纸上画一个 2×2 或 3×3 的小矩阵，把四角代入公式，确认转对了再写代码。

4.15 逐项训练五：边框、四邻格与子矩阵求和

4.15.1 边框元素之和

“边框”由四部分组成：第 0 行、第 `n - 1` 行、第 0 列、第 `m - 1` 列。一个格子在边框上当且仅当满足下面任意一条：

```
row == 0 || row == n - 1 || column == 0 || column == m - 1
```

对 3×4 矩阵逐部分手算（一个格子只数一次，角落不会被重复计算，因为循环里每个格子只判断一次）：

```
 1  2  3  4     第 0 行:       1+2+3+4    = 10
 5  6  7  8     第 2 行:       9+10+11+12 = 42
 9 10 11 12     中间行两端:    5+8        = 13
                           合计          65
```

在 4.10.2 模板读入完成后，把模板的输出循环替换为以下核心代码：

```
    long long borderSum = 0;
    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            if (row == 0 || row == n - 1 || column == 0 || column == m - 1)
            {
                borderSum += a[row][column];
            }
        }
    }

    cout << borderSum << '\n';
```

输入：

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```

输出：

```
65
```

4.15.2 四邻格统计

很多网格题要处理某个格子的**上下左右四个邻居**（四邻格）。给定格子 `(row, column)`，四个邻居是 `(row - 1, column)`、`(row + 1, column)`、`(row, column - 1)`、`(row, column + 1)`。但邻居可能落在网格外，所以每个方向都要先检查合法性再访问，斜向格子不算。

配套练习 21“四邻格统计”给出一个 `0/1` 网格，要求对**每一个格子**统计四个邻居中有多少个 `1`，最后输出同样形状的整数矩阵。看下面的 3×3 网格：

```
0 1 0
1 0 1
1 1 0
```

- 左上角 `(0, 0)` 没有上、左邻居，右边和下边都是 `1`，所以答案是 `2`；
- 正中间 `(1, 1)` 的上、下、左、右邻居恰好都是 `1`，所以答案是 `4`。斜向格子即使为 `1` 也不计入。

完整程序如下。`grid` 保存原始字符，`counts` 保存答案；每个 `counts[row][column]` 都会先计算再输出，不需要额外清零。两个数组都放在函数外，避免占用局部栈空间。

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

#define MAXN 500

string grid[MAXN];
int counts[MAXN][MAXN];

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

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

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            int count = 0;
            if (row > 0)
```

```
            {
                count += grid[row - 1][column] - '0';
            }
            if (row + 1 < n)
            {
                count += grid[row + 1][column] - '0';
            }
            if (column > 0)
            {
                count += grid[row][column - 1] - '0';
            }
            if (column + 1 < m)
            {
                count += grid[row][column + 1] - '0';
            }
            counts[row][column] = count;
        }
    }

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

输入：

```
3 3
010
101
110
```

输出：

```
2 0 2
1 4 0
2 1 2
```

四个 `if` 各自独立：某个方向越界就跳过，不越界才把对应字符转成数值并累加。`row > 0` 等价于 `row - 1 >= 0`，两种写法都表示“上一行存在”。

【拓展】方向变多时，可以用**方向数组**压缩写法。把上面内层循环里的四个 `if` 换成下面这段即可，`count` 的初始化保留不变：

```
int dRow[] = {-1, 1, 0, 0};
int dColumn[] = {0, 0, -1, 1};
for (int direction = 0; direction < 4; direction++)
{
    int nextRow = row + dRow[direction];
    int nextColumn = column + dColumn[direction];
    if (nextRow >= 0 && nextRow < n && nextColumn >= 0 && nextColumn < m)
    {
        count += grid[nextRow][nextColumn] - '0';
    }
}
```

主线先写四个 `if`，边界判断一目了然；方向数组等做搜索题时再系统使用。

4.15.3 小规模子矩阵求和

题目给出左上角 `(r1, c1)` 和右下角 `(r2, c2)`，求这个矩形区域内所有元素之和。最直接的办法：两层循环枚举区域内的每个格子累加。配套练习 22 中 `n, m <= 100`、`q <= 1000`，直接枚举最多约 `q × n × m = 10^7` 个格子，不需要二维前缀和。

对 3×4 矩阵取 `(r1, c1) = (0, 1)`、`(r2, c2) = (1, 2)` 手算，圈出的区域是：

```
 1 [ 2  3]  4
 5 [ 6  7]  8
 9  10 11  12

2 + 3 + 6 + 7 = 18
```

配套练习 22 有两点与模板不同：第一行多一个询问次数 `q`，而且题目中的行列编号从 1 开始。把模板开头的 `int n, m;` 改成 `int n, m, q;`，把 `cin >> n >> m` 改成 `cin >> n >> m >> q`，再在读入矩阵后、原输出循环的位置写入：

```
    while (q--)
    {
        int r1, c1, r2, c2;
        cin >> r1 >> c1 >> r2 >> c2;
        r1--;
        c1--;
        r2--;
        c2--;

        long long subSum = 0;
        for (int row = r1; row <= r2; row++)
        {
            for (int column = c1; column <= c2; column++)
            {
                subSum += a[row][column];
            }
```

```
        }

        cout << subSum << '\n';
    }
```

输入：

```
3 4 2
1 2 3 4
5 6 7 8
9 10 11 12
1 2 2 3
1 1 3 4
```

输出：

```
18
78
```

第一次询问对应手算中的 18，第二次询问是整个矩阵，总和为 78。`subSum = 0` 必须放在 `while (q--)` 内，保证每次询问都重新计数。循环起点和终点都包含在矩形里，所以条件是 `<=`；先把题目给出的 1 起始编号各减 1，再访问 C++ 的 0 起始下标。

【拓展】如果矩阵和询问次数继续增大，可以学习**二维前缀和**，把每次询问降到常数时间。本题的数据范围下，直接枚举就是最清楚且足够快的做法。

4.16 综合示例一：回文修补（贪心）

题目：给定一个字符串，每次操作可以把任意一个字符改成任意字符。问最少修改多少次，能把它变成回文串？

观察对称位置的一对字符 `(s[left], s[right])`：

- 若两者相等，这一对不用动；
- 若两者不等，这一对**至少要改 1 次**（不然它们永远不同）；而只要把其中一边改成另一边，**改 1 次就够**。

各对之间相互独立，所以答案就是“不等字符对”的数量。这种“每一步都做当前看来最优的选择”的策略叫**贪心**。用 `abca` 手算：

```
下标: 0 1 2 3
字符: a b c a

第 1 对 (s[0], s[3]) = (a, a) 相等，不改
第 2 对 (s[1], s[2]) = (b, c) 不等，改 1 次
答案 = 1
```

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

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

    int changes = 0;
    int left = 0;
    int right = n - 1;
    while (left < right)
    {
        if (s[left] != s[right])
        {
            changes++;
        }
        left++;
        right--;
    }

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

配套练习 23 第一行先给长度 `n`，第二行才是字符串，所以读入写成 `cin >> n >> s`，并用 `n - 1` 作为最右下标。

输入：

```
4
abca
```

输出：

```
1
```

自己再验证几组：`abcba` 输出 0（已经是回文）；`abcda` 只有 `(b, d)` 这一对不等，输出 1；`ab` 输出 1。为什么这样一定最优？每一对不等字符都至少要改 1 次，所以答案不可能比"不等对数"更小；而每对只改 1 次就能全部配平，正好达到这个最小值——所以贪心是对的。

4.17 综合示例二：棋盘染色（构造）

题目：给一个 `n` 行 `m` 列的棋盘染色，要求相邻格子（上下左右）颜色不同。用字符 `A`、`B` 表示两种颜色，左上角固定为 `A`。这就是配套练习 24 的要求。

构造类题目的关键不是"检查所有方案"，而是**找到一条规律，直接批量生成答案**。把行列下标加起来看看：

```
(0,0): 0 是偶数 → A     (0,1): 1 是奇数 → B
(1,0): 1 是奇数 → B     (1,1): 2 是偶数 → A
```

`(row + column)` 的奇偶性！相邻两个格子要么行号差 1、要么列号差 1，所以它们的 `row + column` 恰好差 1，奇偶性必然不同——按 `(row + column) % 2` 染色，相邻格子一定异色。这正好满足题目要求。

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

#define MAXN 1000

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

    if (n <= 0 || n > MAXN || m <= 0 || m > MAXN)
    {
        return 0;
    }

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            if ((row + column) % 2 == 0)
            {
                cout << 'A';
            }
            else
            {
                cout << 'B';
            }
        }
        cout << '\n';
    }
```

```
    return 0;
}
```

输入：

```
3 4
```

输出：

```
ABAB
BABA
ABAB
```

验证规律性结论：3×4 共 12 格，输出中 `A` 和 `B` 各 6 个。一般地，`n * m` 为偶数时两色各占一半；为奇数时，从 `A` 开始的染色会让 `A` 多一个。这类"先找到一个一定成立的规律，再照着规律直接批量生成答案"的思路，就是构造题的基本功。

4.18 综合示例三：最短循环节（数学）

题目：给出一个长度为 `n` 的字符串 `s`，找最小的 `k`，使得 `s` 能由一个长度为 `k` 的短串重复若干次拼成。例如 `abcabcabc` 由 `abc` 重复 3 次组成，答案是 3；若没有更短的循环，答案就是 `n` 本身。

先做数学分析。若 `s` 由长度为 `k` 的短串重复组成，那么：

1. `k` 必须整除 `n`（`n % k == 0`），否则拼不出长度 `n`；
2. 第 `i` 个字符一定等于短串的第 `i % k` 个字符，即 `s[i] == s[i % k]` 对所有 `i` 成立。

这两个条件合起来刚好够用：都满足就一定能拼出 `s`，有任一不满足就一定拼不出。于是从小到大枚举 `k`，第一个满足条件的就是答案。用 `abcabcabc`（`n = 9`）手算：

```
k = 1: 1 整除 9，但 s[1] = 'b' != s[0] = 'a'，失败
k = 2: 9 % 2 != 0，跳过
k = 3: 9 % 3 == 0，逐位检查 s[i] == s[i % 3]:
       i = 0..8 时 i % 3 依次是 0 1 2 0 1 2 0 1 2，
       对应字符 a b c a b c a b c，全部相等，成功
答案 = 3
```

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

int main()
{
```

```
    string s;
    cin >> s;

    int n = (int)s.size();
    int answer = n;
    for (int k = 1; k <= n; k++)
    {
        if (n % k != 0)
        {
            continue;
        }
        bool repeated = true;
        for (int i = 0; i < n; i++)
        {
            if (s[i] != s[i % k])
            {
                repeated = false;
                break;
            }
        }
        if (repeated)
        {
            answer = k;
            break;
        }
    }

    cout << answer << ' ' << s.substr(0, answer) << '\n';
    return 0;
}
```

配套练习 25 不仅要求最短循环单元的长度，还要求输出这个循环单元本身。它一定是 `s` 的长度为 `answer` 的前缀，所以用 `s.substr(0, answer)` 取出即可。

输入：

```
abcabcabc
```

输出：

```
3 abc
```

两个边界放心验证：`k = n` 时 `n % n == 0` 且 `i % n == i` 恒成立，所以循环一定能找到答案，最坏输出 `n` 和整个 `s`；全相同字符的串（如 `aaaa`）在 `k = 1` 处直接成功，输出 `1 a`。本题数据范围不大，枚举完全够用；若 `n` 达到十万以上，只需枚举 `n` 的因数而不是 `1..n` 的所有整数——那是以后优化的话题。

4.19 常见错误

错误一（WA）：用 `cin >>` 读带空格的串

`cin >> s` 遇到空格就停。读"一行文本""一句话"请用 `getline`，并在与 `cin >>` **混用**时补上 `cin.ignore()`。

错误二（WA）：`getline` 读到的内容整体错一行

`cin >> n` 之后直接 `getline`，第一次读到的是残留的空串。现象是"输出总差一行"，排查时先画 4.2.3 的传送带图。

错误三（RE / WA）：字符串下标越界

长度为 `len` 的字符串，最后合法下标是 `len - 1`。`s[len]`、用 `-1` 当下标、`substr` 起点超界，都是越界。

错误四（WA）：`find` 不判断找不到

`find` 找不到时返回 `string::npos`，它不是合法下标。先用 `position == string::npos` 判断，再使用位置；不要依赖把返回值转成 `int` 后的结果。

错误五（CE / WA）：`'a'` 与 `"a"` 混用

单引号是字符，双引号是字符串，`string s = 'a';`、`s[0] = "a";` 这类写法要么编译错误，要么意思全错。

错误六（WA）：字典序与数值序混淆

`"10" < "9"` 按字典序成立，按数值不成立。比较数值先转换，别直接拿数字串比大小。

错误七（RE / WA）：二维数组行列写反

`n` 行 `m` 列时循环条件是 `row < n`、`column < m`。把 `n`、`m` 写反在越界时是 RE，恰好没越界时是 WA，很难肉眼发现——写完拿 3×4 的小数据跑一遍。

错误八（WA）：下标公式不验证就直接写

副对角线、旋转、转置的公式最容易记错。动笔前先在 3×3 或 3×4 矩阵上把四角代入公式，和手算结果一致再写代码。

错误九（RE）：四邻格不判边界

`(0, 0)` 的"上"和"左"不存在。四个方向各自判断合法性，越界的方向跳过。

错误十（WA）：多组数据忘记重置

矩阵和累加器（`rowSum`、`columnSum`、`borderSum` 等）定义在每组数据内部，或每组开始时清零，否则上一组的结果会带进下一组。

4.20 本章小结

本章内容可以压缩成下面这条路线：

```
char：一个字符
    ↓ 排成一串
string：下标 [0, size)，可遍历、可修改
    ↓ 两种读法
cin >> s（读到空白）与 getline（读整行），混用要 ignore
    ↓ 常用操作
拼接 / 字典序比较 / 大小写 / 反转 / 回文 / find / substr / 数字串
    ↓ string 排成行
字符串数组与字符网格
    ↓ 数据排成表格
二维数组：外层行、内层列，下标公式先手算
    ↓ 三个思路各一例
贪心（回文修补）/ 构造（棋盘染色）/ 数学（最短循环节）
```

需要真正记住的关键点：

- `cin >> s` 遇空白停止，`getline` 保留空格；`cin >>` 读数字或单词后接 `getline`，残留的换行要用 `cin.ignore()` 丢掉；
- 字符串下标是 `[0, size)`，最后一个字符是 `s[size - 1]`；要修改字符用下标循环，只读字符可用范围 `for`；
- 字典序比较的是字符编码，`"10" < "9"`；要比较数值先转换；
- 回文类问题先想“左右两个下标向中间靠拢”；
- `find` 找不到返回 `string::npos`，必须用 `position == string::npos` 判断，不能把返回值当合法下标；
- 二维数组先说清“`n` 行 `m` 列”，外层 `row < n`、内层 `column < m` 全程不混；转置、旋转会改变输出行列数；
- 四邻格、子矩阵、边框都要先想清楚边界与区间，再写循环；
- `char[]` 靠 `'\0'` 结尾、`gets` 已被标准移除——这些是兼容旧代码的知识，主线用 `string`。

4.21 配套练习

新增练习（25 题，按建议顺序完成）

题目将陆续发布到沈阳化工大学在线判题平台（SYUCTOJ），此处先给出题名与建议顺序，标题即训练目标：

1. 字符串的头和尾
2. 空格计数
3. 元音计数
4. 大小写互换
5. 字符替换
6. 反转字符串
7. 回文判断
8. 子串在哪里
9. 截取一段
10. 数字串求和
11. 矩阵原样输出
12. 矩阵加法

13. 每行的和
14. 每列的和
15. 最大值的位置
16. 两条对角线
17. 每行逆序
18. 矩阵转置
19. 顺时针旋转
20. 边框元素之和
21. 四邻格统计
22. 子矩阵求和
23. 回文修补
24. 棋盘染色
25. 最短循环节

已有题目（字符串专题训练，共 15 题）

以下题目已在 SYUCTOJ 字符串专题训练 发布，建议与上方新增题目搭配完成；其中 D / K / N 难度较高，标为【拓展】：

- A Hello World!
- B 字符数数
- C 笨小明
- D 最小字符串【拓展】
- E 鹦鹉学舌
- F 鹦鹉学画
- G 字符串大小比较
- H 小明的环形字符串
- I 找整数
- J Your Ride Is Here
- K 小B的传说【拓展】
- L 流星
- M 龙卷风摧毁停车场
- N 龙卷风摧毁停车场（升级版）【拓展】
- O 最小Move串