第 2 章 运算、判断与循环

> 本文是《XCPCer 从零开始的 C/C++ 入门指南（算法竞赛向）》系列第 2 章，系列训练大纲见本文。

2.0 本章地图：先看这里

按照最初的大纲，这一阶段要分三次训练：先讲 ASCII，再讲运算符，最后讲循环。但大家的推进速度比预想中快，不少同学已经自己摸到了循环。所以我们把一个完整的知识链接起来一次走完：**表达式 → 条件判断 → 循环 → 用循环做题**。

本章很长。为了让长长的章节不那么吓人，正文用两种标签明确标出轻重：

- **【主线】**：第二次训练题单里直接要用到的内容。第一次读只读这些，读完就能开始做题；
- **【拓展】**：同样值得学，但不影响你现在做题。可以整节跳过，等主线刷顺了再回来，也可以在卡住某道难题时翻过来查。

主线与拓展的划分大致是：

标签	内容
【主线】	2.1 表达式与运算符、2.2 字符运算、2.3 条件判断、2.4 循环、2.5 break / continue、2.6 嵌套循环、2.7 多组输入与 EOF、2.8 累加 / 计数 / 累乘 / 最值、2.9 数位拆分与反转、2.10 上取整、2.11 格式化输出、2.12 简单过程模拟、2.13 枚举与复杂度直觉、2.14 常见错误对照
【拓展】	主线各节内标【拓展】的小节（switch、do-while、`cout` 版格式化输出），以及 2.15 全部：复杂图形打印、大数据模拟、很大很大的数、复杂贪心方向、二分猜数字

只安排一次训练的话，不必当场吃透一切。给还在消化基础的同学一条最小主线：**条件、边界、更新**——每写一个循环，先问自己三个问题：从哪里开始？什么时候停？每轮怎么接近结束？把这三个问题答出来，本章一半的内容就已经在用了。

学完本章的主线部分，你应该能：

- 把一句数学条件准确地翻译成 C++ 表达式；
- 使用 `if` / `else if` 处理分支；
- 使用 `for`、`while` 重复执行代码，并说清楚每一轮发生了什么；
- 用嵌套循环打印图形、枚举数对；
- 处理"给组数"和"读到文件结束"两种多组输入；
- 拆数位、判质数、做上取整、用 `printf` 精确控制输出格式；
- 看到数据范围时，对程序会不会超时有一个初步判断。

核心心法只有一句：

> **先算出一个条件，再根据条件决定执行什么；需要做很多次，就把这个决定放进循环。**

2.1 表达式：程序怎样"算出一个值"【主线】

把变量、常量和运算符连接起来，就得到一个**表达式**。表达式一定会产生一个值：

- `a + b` 的值是两数之和；
- `a > b` 的值是 `true` 或 `false`；
- `x = 5` 在完成赋值的同时，整个表达式的值也是 `5`。

分号 `;` 通常不属于表达式，它表示一条语句结束。比如 `sum = a + b;` 中，`sum = a + b` 是表达式，末尾分号把它变成一条表达式语句。

2.1.1 算术运算符

运算符	含义	示例	结果
`+`	加法	`7 + 3`	`10`
`-`	减法 / 取负	`7 - 3` / `-7`	`4` / `-7`
`*`	乘法	`7 * 3`	`21`
`/`	除法	`7 / 3`	整数运算时为 `2`
`%`	取余	`7 % 3`	`1`

整数除法会直接舍去小数部分，结果向 0 截断。因此 `7 / 3` 是 `2`，`-7 / 3` 是 `-2`。如果需要 `2.333...`，至少让一个操作数变成浮点数，例如 `7.0 / 3` 或 `(double)7 / 3`。

`%` 只能用于整数。在 C++ 中，余数与被除数同号或为 0：`7 % 3` 是 `1`，`-7 % 3` 是 `-1`。所以当题目要求数学意义上位于 `[0, m-1]` 的余数、而 `x` 可能为负时，可以写成：

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

int main()
{
    int x = -7;
    int m = 3;
    int remainder = x % m;
    if (remainder < 0)
    {
        remainder += m;
    }

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

运行结果：

```
2
```

这里先算 `-7 % 3 = -1`，发现结果为负后再加上 `3`，得到数学中常用的非负余数 `2`。本技巧要求 `m > 0`。不要机械地写成 `(x % m + m) % m`：当 `x % m` 和 `m` 都很大时，中间的加法也可能让有符号整数溢出。

2.1.2 赋值与复合赋值

赋值运算符 `=` 的作用是把右边的值放进左边的变量：

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

int main()
{
    int score = 60;
    score = score + 10;
    cout << score << '\n';
    return 0;
}
```

运行结果：

```
70
```

当一个变量要在自身基础上修改时，可以使用复合赋值：

完整写法	简写
`x = x + y`	`x += y`
`x = x - y`	`x -= y`
`x = x * y`	`x *= y`
`x = x / y`	`x /= y`
`x = x % y`	`x %= y`

`sum += i;` 是循环求和中最高频的一行，意思是"把本轮的 `i` 加进累计答案 `sum`"。

2.1.3 自增与自减

`++` 让变量增加 1，`--` 让变量减少 1：

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

int main()
{
    int a = 5;
    int b = a++; // 先把旧值 5 交给 b，再让 a 变成 6
```

```
    int c = 5;
    int d = ++c; // 先让 c 变成 6，再把 6 交给 d

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

运行结果：

```
6 5
6 6
```

单独占一行时，`i++` 与 `++i` 对本章的基本整数循环没有区别，都让 `i` 增加 1。初学阶段不要写 `a = a++`、`cout << i++ + ++i` 之类炫技表达式：有些写法会产生未定义行为，即使能运行也非常容易看错。

2.1.4 关系运算符：比较之后得到 bool

运算符	含义
`==`	等于
`!=`	不等于
`<`	小于
`>`	大于
`<=`	小于等于
`>=`	大于等于

关系表达式的结果是 `bool`：成立为 `true`，不成立为 `false`。

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

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

    cout << (a < b) << '\n';
    cout << (a == b) << '\n';
    return 0;
}
```

运行结果：

```
1
0
```

`cout` 默认把 `true` 输出为 `1`，把 `false` 输出为 `0`。最容易写错的是：**判断相等用两个等号 `==`，赋值才用一个等号 `=`。**

2.1.5 逻辑运算符：并且、或者、不是

关系运算符只能描述一个条件，例如 `age >= 18`。当题目同时提出多个条件时，就要用逻辑运算符把它们连接起来：

运算符	中文含义	何时为 `true`	示例
`&&`	并且	左右条件同时成立	`x >= 1 && x <= 100`
			或者
`!`	取反 / 不是	原条件不成立	`!(x == 0)`

设 `A`、`B` 都是条件，三种逻辑运算的结果如下：

A	B	A 并且 B	A 或者 B	非 A
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

记忆方法很简单：`&&` 比较严格，要求两边都对；`||` 比较宽松，只要一边对；`!` 直接把真假翻转。

例如，判断 `x` 是否位于闭区间 `[1, 100]`：

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

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

    bool inRange = (x >= 1 && x <= 100);
    bool outOfRange = (x < 1 || x > 100);
    bool notInRange = !inRange;

    cout << inRange << ' '
         << outOfRange << ' '
         << notInRange << '\n';
```

```
    return 0;
}
```

输入：

```
73
```

输出：

```
1 0 0
```

这里 `inRange` 表示"范围内"，`outOfRange` 和 `notInRange` 都表示"范围外"，所以后二者结果相同。由此也能看到 `!(x >= 1 && x <= 100)` 等价于 `x < 1 || x > 100`。

数学里常写 `1 <= x <= 100`，但 C++ **不能照抄**。C++ 会先计算 `1 <= x`，得到 `0` 或 `1`，再拿这个结果与 `100` 比较；由于 `0` 和 `1` 都小于等于 `100`，所以对整数 `x` 这个错误表达式永远为真，输入 `1000` 也拦不住。必须拆成 `x >= 1 && x <= 100`。

另外，逻辑运算符是 `&&`、`||`、`!`，不要漏写成单个 `&` 或 `|`。后两者是位运算符，含义完全不同，会在第 6 章单独讲。

2.1.6 短路求值：后半句可能根本不算

逻辑运算符有一条非常重要的规则：

- `A && B` 中，如果 `A` 已经为假，整体必假，`B` 不再计算；
- `A || B` 中，如果 `A` 已经为真，整体必真，`B` 不再计算。

这叫**短路求值**。它不只是优化，还能保护危险运算。判断 `b` 能否被 `a` 整除时，必须先排除除数为 0：

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

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

    if (a != 0 && b % a == 0)
    {
        cout << "divisible\n";
    }
    else
    {
        cout << "not divisible\n";
    }
```

```
    return 0;
}
```

输入：

```
0 12
```

输出：

```
not divisible
```

因为 `a != 0` 为假，右边的 `b % a` 不会执行，避免了对 0 取模。顺序如果反过来写成 `b % a == 0 && a != 0`，保护就失效了。

2.1.7 运算符优先级：拿不准就加括号

本章常用运算符的优先级从高到低可以先记为：

1. 括号 `()`；
2. 单目运算 `!`、`++`、`--`、取负；
3. `*`、`/`、`%`；
4. `+`、`-`；
5. `<`、`<=`、`>`、`>=`；
6. `==`、`!=`；
7. `&&`；
8. `||`；
9. 赋值 `=`、`+=` 等。

不需要死背整张 C++ 优先级表。比赛中一旦可能误读，直接加括号。`(year % 4 == 0 && year % 100 != 0) || year % 400 == 0` 就比依赖记忆安全得多。

2.1.8 整数运算仍然要防溢出

第 1 章讲过：右侧表达式先完成运算，再把结果赋给左侧变量。因此即使答案变量是 `long long`，`int * int` 仍然可能先溢出。

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

int main()
{
    int a = 100000;
    int b = 100000;
    long long product = 1LL * a * b;

    cout << product << '\n';
```

```
    return 0;
}
```

运行结果：

```
10000000000
```

这里的 `1LL` 让后续乘法从一开始就在 `long long` 中完成。补充一条技术上必须准确的说明：**C++ 对有符号整数溢出没有规定结果**，不能依赖它"必然绕回"；无符号整数才按模意义回绕。竞赛中最实际的处理方式仍然是提前选够大的类型。

2.1.9 浮点数不要直接判断相等

许多小数不能被二进制浮点数精确表示，因此计算结果可能只是在目标值附近。与其写 `a == b`，通常比较两者差的绝对值是否足够小：

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

int main()
{
    double a = 0.1 + 0.2;
    double b = 0.3;
    const double EPS = 1e-9;

    if (fabs(a - b) < EPS)
    {
        cout << "equal\n";
    }
    else
    {
        cout << "not equal\n";
    }
    return 0;
}
```

运行结果：

```
equal
```

`EPS` 取多大取决于题目允许的误差与数值规模，`1e-9` 只是常见入门值，不是万能常量。至于 `1e-9` 这种写法的含义，见下一节。

2.1.10 数学幂、科学计数法，以及 ^ 的误会【主线必学】

从这一节起，题面和代码里会频繁出现大数的简写，先把记号统一掉，不然后面每一节都要打结。

数学里的幂。 题面写数据范围时经常用幂，例如"`n ≤ 10^8`"，意思是 10 的 8 次方，即 100000000（1 后面 8 个 0）。第 1 章类型表里的 $\pm 2.1\times10^{9}$、$\pm 9.2\times10^{18}$ 也是同一个意思的上标写法。

代码里的科学计数法。 C++ 源码里不能写上标，于是用科学计数法：`1e8` 表示 1×10^8 = 100000000，`2.1e9` 表示 2.1×10^9，`1e-9` 表示 10^-9 = 0.000000001（小数点后 8 个 0 再跟 1）。规则是：`e` 后面的整数表示"10 的多少次方"，正数往右移小数点，负数往左移。所以上一节的 `1e-9` 就是"差值小于十亿分之一就当相等"。

`^` 在 C++ 里不是乘方。 这是初学者几乎必踩的坑：数学里的 `2^10` 表示 1024，但 C++ 的 `^` 是另一个运算符（按位异或，第 6 章会讲），`2 ^ 10` 算出来是 `8`。验证一下：

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

int main()
{
    cout << (2 ^ 10) << '\n';
    return 0;
}
```

运行结果：

```
8
```

所以在 C++ 里：读题看到 `10^8` 要翻译成 `1e8` 或 `100000000`；想算"2 的 10 次方"就写循环累乘（见 2.8.3），不要写 `2^10`。

结合第 1 章的类型范围，把最常用的三个量级背下来：`int` 约到 `2.1e9`，`long long` 约到 `9.2e18`，一秒评测约能跑 `1e8` 次极简单运算（2.13.4 展开）。这三句话会在每一场比赛中用到。

2.2 加餐：ASCII 与字符运算【主线必学】

第 1 章已经说过，`char` 本质上存的是一个整数编码。竞赛中常见英文字母、数字和符号使用 ASCII 编码，其中最值得记住的是三个**连续区间**：

字符范围	十进制编码	关键性质
`'0'`～`'9'`	48～57	连续，相减可得到数值
`'A'`～`'Z'`	65～90	连续
`'a'`～`'z'`	97～122	连续

因此：

- 数字字符转数值：`digit = c - '0'`；
- 小写字母转大写：`c - 'a' + 'A'`；

- 大小写字母的 ASCII 差值为 `'a' - 'A'`。

`char` 参与算术运算前会先提升为 `int`。因此 `'A' + 1` 的值虽然是编码 `66`，但它的类型是 `int`；直接 `cout << 'A' + 1` 会输出 `66`，不会输出 `B`。若要把结果当作字符输出，需要先转换回 `char`：

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

int main()
{
    cout << ('A' + 1) << '\n';
    cout << static_cast<char>('A' + 1) << '\n';
    return 0;
}
```

运行结果：

```
66
B
```

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

int main()
{
    char digitCharacter = '7';
    int digitValue = digitCharacter - '0';

    char lower = 'c';
    char upper = lower - 'a' + 'A';

    cout << digitValue + 5 << '\n';
    cout << upper << '\n';
    return 0;
}
```

运行结果：

```
12
C
```

注意 `'7'` 是字符，数值是其编码；`7` 才是整数。`'7' - '0'` 得到整数 `7`，而不是字符 `'7'`。此外，不要默认所有字符编码都适合这种手算；本章只讨论竞赛中常见的 ASCII 数字与英文字母。

结合条件判断，我们可以自行识别字符类型：

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

int main()
{
    char c;
    cin >> c;

    if (c >= '0' && c <= '9')
    {
        cout << "digit " << c - '0' << '\n';
    }
    else if (c >= 'A' && c <= 'Z')
    {
        cout << "uppercase\n";
    }
    else if (c >= 'a' && c <= 'z')
    {
        cout << "lowercase\n";
    }
    else
    {
        cout << "other\n";
    }
    return 0;
}
```

输入：

```
8
```

输出：

```
digit 8
```

2.3 条件判断：让程序选择道路【主线必学】

程序默认从上到下执行。条件语句让它在岔路口根据数据选择不同的道路。

2.3.1 单分支 if

基本形式为 `if (条件) { ... }`。条件为真时执行花括号里的语句，否则直接跳过。

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

```cpp
int main()
{
    int score;
    cin >> score;

    if (score >= 60)
    {
        cout << "pass\n";
    }
    return 0;
}
```

输入 `75` 时输出 `pass`；输入 `40` 时没有输出。

C++ 的条件位置不只接受 `bool`：数值 `0` 会被视为 `false`，非零值会被视为 `true`。因此 `if (x)` 等价于判断 `x != 0`，`if (!x)` 等价于判断 `x == 0`。初学阶段写完整比较通常更易读。

2.3.2 双分支 if / else

如果两种情况必然二选一，使用 `else`：

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

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

    if (n % 2 == 0)
    {
        cout << "even\n";
    }
    else
    {
        cout << "odd\n";
    }
    return 0;
}
```

输入：

```
13
```

输出：

```
odd
```

2.3.3 多分支 else if

多个区间分类时，从上到下依次检查；命中第一个成立条件后，后面的分支不再执行。

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

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

    if (score < 0 || score > 100)
    {
        cout << "invalid\n";
    }
    else if (score >= 90)
    {
        cout << "A\n";
    }
    else if (score >= 80)
    {
        cout << "B\n";
    }
    else if (score >= 60)
    {
        cout << "C\n";
    }
    else
    {
        cout << "D\n";
    }
    return 0;
}
```

输入：

```
86
```

输出：

```
B
```

这里条件必须从严格到宽松排列。若先写 `score >= 60`，那么 95 也会在这一项就被拦住，永远走不到 A。

2.3.4 嵌套 if 与悬空 else

`if` 内部还能再放 `if`。但省略花括号时，`else` 会与**最近且尚未配对的 `if`** 结合，这就是"悬空 else"问题。统一写花括号能直接消灭这种歧义。

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

int main()
{
    int age;
    bool hasTicket;
    cin >> age >> hasTicket;

    if (age >= 18)
    {
        if (hasTicket)
        {
            cout << "enter\n";
        }
        else
        {
            cout << "buy a ticket\n";
        }
    }
    else
    {
        cout << "too young\n";
    }
    return 0;
}
```

输入 `20 1` 时输出 `enter`；输入 `20 0` 时输出 `buy a ticket`；输入 `15 1` 时输出 `too young`。

本讲义后续即使分支只有一行，也会保留花括号。它带来的麻烦远小于它消灭的歧义。

2.3.5 综合例题：判断闰年

公历闰年的规则是：能被 400 整除，或者能被 4 整除但不能被 100 整除。

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

int main()
{
    int year;
    cin >> year;
```

```
    bool leap = (year % 400 == 0)
             || (year % 4 == 0 && year % 100 != 0);

    if (leap)
    {
        cout << "leap year\n";
    }
    else
    {
        cout << "common year\n";
    }
    return 0;
}
```

输入：

```
2000
```

输出：

```
leap year
```

这道题是翻译条件的好例子。先把中文里的"或者""并且""不能"分别对应到 `||`、`&&`、`!=`，再用括号把逻辑层次写出来。自己跑一遍 `1900`（输出 `common year`）和 `2024`（输出 `leap year`），确认三个条件各自起了作用。

2.3.6 switch：按离散值分支【拓展】

> 本节属拓展。第二次题单没有非用 `switch` 不可的题，用 `else if` 都能写；想学就学，想跳过就跳过。

当分支条件都是"某个变量等于一个确定的整数或字符"时，`switch` 往往比很长的 `else if` 更整齐。

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

int main()
{
    char operation;
    int a, b;
    cin >> a >> operation >> b;

    switch (operation)
    {
        case '+':
            cout << a + b << '\n';
            break;
        case '-':
            cout << a - b << '\n';
```

```
            break;
        case '*':
            cout << a * b << '\n';
            break;
        case '/':
            if (b == 0)
            {
                cout << "division by zero\n";
            }
            else
            {
                cout << a / b << '\n';
            }
            break;
        default:
            cout << "unknown operation\n";
            break;
    }
    return 0;
}
```

输入：

```
20 * 6
```

输出：

```
120
```

`switch` 的注意事项：

1. `case` 后面必须是编译期可确定的整型、字符或枚举常量，不能写范围，也不能直接比较字符串；
2. 通常每个 `case` 末尾写 `break`，否则会继续执行下一个分支，这叫"贯穿"；
3. `default` 类似最后的 `else`，处理所有未匹配情况；
4. 区间判断（例如分数等级）仍然更适合 `if / else if`。

2.3.7 条件运算符（三目运算符） `?:`【主线】

简单的二选一可以写成一个表达式：

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

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

```
    int larger = (a > b) ? a : b;
    cout << larger << '\n';
    return 0;
}
```

输入 `8 5`，输出 `8`。其含义是：条件 `a > b` 成立时取 `a`，否则取 `b`。三目运算符适合短小表达式，不要把许多层 `?:` 套在一起，那会让代码像谜语。

2.4 循环：让同一段代码工作很多次【主线】

循环由三件事组成：

1. **初始状态**：从哪里开始；
2. **继续条件**：什么时候还要做；
3. **状态更新**：每次做完后怎样接近结束。

少了第三项，循环条件可能永远为真，程序就会卡在死循环里。写任何循环前，先在纸上用中文把这三件事各写一句，这是对付循环题最便宜也最有效的习惯。

2.4.1 while：次数未知，条件明确

`while` 会先检查条件，条件为真才执行循环体。所以它可能一次也不执行。

下面把一个正整数的十进制数位倒序输出：

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

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

    if (n == 0)
    {
        cout << 0;
    }
    while (n > 0)
    {
        cout << n % 10;
        n /= 10;
    }
    cout << '\n';
    return 0;
}
```

输入：

```
12030
```

输出：

```
03021
```

每轮用 `n % 10` 取末位输出，再用 `n /= 10` 删除末位。循环次数取决于数字有多少位，事先不必知道。逐轮看一遍 `n` 的变化（这是本章最重要的读代码方法，叫**逐轮执行表**）：

轮次	进入循环时 `n`	输出 `n % 10`	轮末 `n /= 10`
1	12030	0	1203
2	1203	3	120
3	120	2	12
4	12	1	1
5	1	1	0
—	条件 `n > 0` 不成立，循环结束		

2.4.2 for：次数或范围明确

`for` 把"初始化、继续条件、更新"放在一行：

```
for (初始化; 继续条件; 更新)
{
    循环体
}
```

例如计算 `1 + 2 + ... + n`：

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

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

    long long sum = 0;
    for (int i = 1; i <= n; i++)
    {
        sum += i;
    }
```

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

输入：

```
100
```

输出：

```
5050
```

逐轮执行（`n = 3` 的缩小版）：

轮次	判断时 `i`	`i <= n`	循环体动作	轮末 `sum`	更新后 `i`
1	1	真	`sum += 1`	1	2
2	2	真	`sum += 2`	3	3
3	3	真	`sum += 3`	6	4
结束	4	假	—	6	—

2.4.3 do-while：至少执行一次【拓展】

> 第二次题单用不到 `do-while`，拓展阅读即可。

`do-while` 先执行循环体，再检查条件，因此无论条件如何，循环体至少执行一次。末尾的分号不能漏。

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

int main()
{
    int n = 0;
    do
    {
        cout << "Please enter a positive integer: ";
        if (!(cin >> n))
        {
            return 0;
        }
    }
    while (n <= 0);

    cout << "accepted " << n << '\n';
```

```
    return 0;
}
```

一次可能的运行过程：

```
Please enter a positive integer: -2
Please enter a positive integer: 0
Please enter a positive integer: 7
accepted 7
```

在线评测题一般不会要求反复提示用户，所以 `do-while` 在竞赛代码中不如 `for`、`while` 常见，但"菜单至少显示一次""输入至少处理一次"时很自然。示例中的 `if (!(cin >> n))` 用来处理读取失败（例如文件结束）：一旦读取失败就直接结束，避免输入流一直处于失败状态而形成死循环。正式 OJ 通常会保证输入格式合法，但知道"读取也能成为条件"很有用——这正是 2.7.2 多组输入的理论基础。

2.4.4 for 与 while 可以互相改写

`for` 的三个部分拆开写就是 `while`：`int i = 1; while (i <= n) { ...; i++; }` 与 2.4.2 的 `for (int i = 1; i <= n; i++) { ... }` 完全等价，不必重复整套程序。经验上：

- "从 1 到 n""重复 t 次"优先用 `for`；
- "直到某条件不成立""不断拆数位"优先用 `while`；
- 必须先做一次再判断时用 `do-while`（2.4.3，拓展）。

选择能让意图最明显的写法。

2.4.5 循环边界：`< n` 还是 `<= n`

边界错误是初学者最常见的 WA。写循环前先用中文说清楚要遍历的集合：

- 遍历 `1, 2, ..., n`：`for (int i = 1; i <= n; i++)`；
- 执行恰好 `n` 次，以 0 开始计数：`for (int i = 0; i < n; i++)`；
- 遍历 `l, l+1, ..., r`：`for (int i = l; i <= r; i++)`；
- 倒序遍历 `n, n-1, ..., 1`：`for (int i = n; i >= 1; i--)`。

不要凭手感改 `<` 和 `<=`。拿 `n = 1` 在纸上走一遍，往往立即能看出到底少一轮还是多一轮。

2.4.6 倒序循环与 unsigned 陷阱

倒序循环的控制变量优先使用 `int` 或 `long long`：

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

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

```
    for (int i = n; i >= 0; i--)
    {
        cout << i << (i == 0 ? '\n' : ' ');
    }
    return 0;
}
```

输入 `5`，输出：

```
5 4 3 2 1 0
```

这里 `(i == 0 ? '\n' : ' ')` 是 2.3.7 的条件运算符：最后一个数后换行，其余用空格隔开。如果把 `i` 写成 `unsigned int`，条件 `i >= 0` 永远成立——`i` 在 0 之后不会变成 -1，而会回绕到一个巨大的正数，循环无法正常结束。这正是第 1 章建议初学阶段慎用无符号类型的原因之一。

2.4.7 死循环不一定是错

`while (true)` 和 `for (;;)` 都表示条件永远为真。若循环内存在明确的 `break` 出口，它可以是合法写法；若没有出口，程序就会跑满时间限制（TLE）甚至被判运行时错误（RE）。

不过，如果题目要求"有多少组就读多少组，直到文件结束（EOF）"，不需要写死循环再手动判断，直接使用 `while (cin >> n)` 即可，详见 2.7.2。

2.5 break 与 continue：改变本轮流程【主线必学】

2.5.1 break：立刻结束当前循环

下面判断 `n` 是否为质数。只要找到一个因数，就已经能确定"不是质数"，没有必要继续检查。

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

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

    bool isPrime = (n >= 2);
    for (int divisor = 2; 1LL * divisor * divisor <= n; divisor++)
    {
        if (n % divisor == 0)
        {
            isPrime = false;
            break;
        }
    }
```

```
    if (isPrime)
    {
        cout << "prime\n";
    }
    else
    {
        cout << "not prime\n";
    }
    return 0;
}
```

输入：

```
97
```

输出：

```
prime
```

为什么只检查到 `divisor * divisor <= n`？如果 `n = a * b` 且 `a`、`b` 都大于 `sqrt(n)`，乘积就会大于 `n`，矛盾。因此合数一定存在一个不超过平方根的因数。这条剪枝让循环次数从 `n` 降到 `sqrt(n)`：对 `n = 1e6`，只需检查 1000 轮而不是 100 万轮——2.13 会专门练习从数据范围估算这个账。

条件中写 `1LL * divisor * divisor` 是为了避免两个 `int` 相乘先溢出。

逐轮执行（`n = 12` 的缩小版）：

轮次	divisor	1LL*d*d <= 12	12 % d	动作
1	2	真	0	isPrime = false，break
—	循环已结束			

而 `n = 97` 时，2、3、…、9 都除不尽，`divisor` 一路加到 10，`10 * 10 <= 97` 不成立，循环自然结束，`isPrime` 保持 `true`。

2.5.2 continue：跳过本轮剩余部分（含一个危险代码）

下面计算 `1` 到 `n` 中所有奇数的和：

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

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

```
    long long sum = 0;
    for (int i = 1; i <= n; i++)
    {
        if (i % 2 == 0)
        {
            continue;
        }
        sum += i;
    }

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

输入 `7`，输出：

```
16
```

遇到偶数时，`continue` 跳过本轮的 `sum += i`，然后进入下一轮。注意 `for` 执行 `continue` 后，仍会先执行更新表达式 `i++`，再判断条件——所以上面这段是安全的。

危险写法：continue 跳过了 while 的更新。 同样的问题改用 `while` 写，就很容易翻车：

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

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

    long long sum = 0;
    int i = 1;
    while (i <= n)
    {
        if (i % 2 == 0)
        {
            continue; // 危险：i++ 在 continue 后面，永远执行不到
        }
        sum += i;
        i++;
    }

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

输入 `n = 5` 时的逐轮执行表：

轮次	判断时 `i`	`i <= n`	`i % 2 == 0`	发生的事	`sum`	轮末 `i`
1	1	真	否	`sum += 1`，然后 `i++`	1	2
2	2	真	是	`continue`，`sum += i` 和 `i++` 都被跳过	1	**2**
3	2	真	是	同上，永远停在这里	1	**2**
…	2	真	是	死循环	1	2

第 2 轮 `i` 是偶数，`continue` 直接跳回条件判断，`i++` 没机会执行；`i` 永远是 2，循环永远出不去——这就是典型的"忘记更新循环变量"死循环，提交上去就是 TLE。

修正方法一：重组条件，让每条路都走到更新。 把"遇到偶数跳过累加"改成"只有奇数才累加"：`if (i % 2 != 0) sum += i;`，然后把 `i++` 挪到 `if` 外面——无论是否累加，更新都一定执行。输入 `5` 时输出同样是 `9`。

修正方法二：把更新挪到 continue 之前。 如果逻辑上确实需要"先跳过、后处理"，就让更新先执行：

```
    while (i <= n)
    {
        int current = i;
        i++; // 更新放在 continue 之前，保证一定执行
        if (current % 2 == 0)
        {
            continue;
        }
        sum += current;
    }
```

两种修正都合法。原则是：`while` 的更新语句必须放在循环体中**每条执行路径都能到达**的位置。`for` 的更新写在括号里，天然不会被 `continue` 跳过，这也是处理"跳过某些轮"时 `for` 更不容易出错的原因。

2.5.3 break 只结束最近一层循环

在嵌套循环里，`break` 只会退出它直接所在的那一层。想同时退出两层，可以使用一个 `bool` 标记：

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

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

    bool found = false;
    int answerA = 0;
    int answerB = 0;

    for (int a = 0; a <= 9 && !found; a++)
```

```
    {
        for (int b = 0; b <= 9; b++)
        {
            if (10 * a + b == target)
            {
                answerA = a;
                answerB = b;
                found = true;
                break;
            }
        }
    }

    if (found)
    {
        cout << answerA << ' ' << answerB << '\n';
    }
    else
    {
        cout << "not found\n";
    }
    return 0;
}
```

输入 `42`，输出：

```
4 2
```

外层循环条件中的 `&& !found` 让它在找到答案后也结束。内层 `break` 只负责退出内层；外层的退出靠的是条件里的标记。

2.6 嵌套循环：循环里面再放循环【主线】

2.6.1 先看执行顺序

嵌套循环的执行方式是：外层每进行一轮，内层都完整执行一遍。

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

int main()
{
    for (int row = 1; row <= 2; row++)
    {
        for (int column = 1; column <= 3; column++)
        {
            cout << '(' << row << ',' << column << ") ";
        }
```

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

运行结果：

```
(1,1) (1,2) (1,3)
(2,1) (2,2) (2,3)
```

逐轮执行：外层 `row = 1`，内层 `column` 从 1 跑到 3；然后外层 `row = 2`，内层又把 1 到 3 跑一遍。外层 2 轮，每轮内层 3 轮，总共输出 `2 * 3 = 6` 次。

2.6.2 打印直角三角形

遇到带图形的题不要盯着整幅图发呆。把问题拆开：一共有多少行？第 `i` 行输出多少个字符？图里的字符有什么规律？

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

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

    for (int row = 1; row <= n; row++)
    {
        for (int count = 1; count <= row; count++)
        {
            cout << '*';
        }
        cout << '\n';
    }
    return 0;
}
```

输入：

```
4
```

输出：

```
*
**
```

```
***
****
```

外层控制行号 `row`，内层输出恰好 `row` 个星号（第 1 行 1 个、第 2 行 2 个……），内层结束后再换行。行数与列数都由 `n` 决定时，循环边界用 `<=` 还是 `<` 尤其要在纸上验证一遍。

2.6.3 九九乘法表

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

int main()
{
    for (int i = 1; i <= 9; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            cout << j << '*' << i << '=' << i * j << '\t';
        }
        cout << '\n';
    }
    return 0;
}
```

运行结果的前四行（`\t` 是制表符，不同终端的对齐宽度可能略有区别）：

```
1*1=1
1*2=2   2*2=4
1*3=3   2*3=6   3*3=9
1*4=4   2*4=8   3*4=12  4*4=16
```

内层只到 `j <= i`，所以是下三角形状。如果题目要求严格对齐（题单里的九九乘法表就要求用 `%-3d` 补位），请跳到 2.11.3 学格式化输出，再回来重做这个例子。

2.7 多组输入与 EOF【主线】

多组输入是第二/三次题单的常客。这类题的输入分三种样子，本讲义统一成三套写法，选定一套就一路用到底，不要在同一道题里混用：

1. **题目给出测试组数 `t`**：先读 `t`，再 `while (t--)`（2.7.1）；
2. **题目不给组数，要求读到文件结束（EOF）**：`while (cin >> n)`（2.7.2）；
3. **题目给出明确的结束标记**：普通 `while` 配合 `break`（2.7.3）。

竞赛题保证输入合法、组数与格式和题面一致，所以直接读即可，不需要额外检查每次读取是否成功。

2.7.1 给组数：while (t--)

下面对每个非负整数 `n` 分别计算 `1 + 2 + ... + n`：

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

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

    while (t--)
    {
        int n;
        cin >> n;

        long long sum = 0;
        for (int i = 1; i <= n; i++)
        {
            sum += i;
        }
        cout << sum << '\n';
    }
    return 0;
}
```

输入：

```
3
3
0
5
```

输出：

```
6
0
15
```

`t` 的逐轮变化：`t = 3` 进入第 1 组，判断后减成 2；第 2 组后变 1；第 3 组后变 0——这一轮判断时 `t` 已经是 0（假），循环结束。每完成一组，`t` 减少 1，直到变成 0，`while (t--)` 的妙处就是把"判断"和"减一"合并在一行里。循环结束后 `t` 的值是 `-1`，不要再拿它表示原来的测试组数。

`sum` 必须定义在 `while (t--)` 内部（像上面这样），或在每组开始时重新赋为 0。若把它放在外面又忘记清零，后一组就会带着前一组的答案继续累加——这是多组输入最常见的 WA 来源。

2.7.2 不给组数：while (cin >> n)

有的题目不说有几组，只让你"一直读到输入结束"。在线评测的输入是一个文件，读到文件末尾的状态叫 **EOF（End Of File）**。`cin >> n` 这整个表达式在"成功读到数"时可以视为真，读到 EOF 会变为假，所以可以直接当作循环条件：

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

int main()
{
    int sum = 0;
    int n;

    while (cin >> n)
    {
        sum += n;
    }

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

输入（评测时文件到此为止）：

```
3 5 -2
```

输出：

```
6
```

本地自测时，Windows 终端里输入数据后按 `Ctrl + Z` 再回车表示 EOF；Linux / macOS 是 `Ctrl + D`。如果每组包含多个字段，可以写成 `while (cin >> n >> m)`，确保一整组的组头读完后再进入循环。逐轮看：读到 3、5、-2 各进一轮；再次尝试读取时遇到 EOF，条件为假，循环自然结束。

2.7.3 给结束标记：普通 while 配 break

有的题既不给你组数，也不靠 EOF，而是约定一个"结束信号"。题单里的"数字翻转"就是这类格式：**当 `a`、`b` 都为 0 时输入结束**，且 `a`、`b` 都为 0 的这一组不需要处理。这里先用"每组输出两数之和"演示读法，题目本身的反转逻辑见 2.9.3：

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

int main()
{
    while (true)
```

```
    {
        int a, b;
        cin >> a >> b;
        if (a == 0 && b == 0)
        {
            break; // 读到结束标记，不再处理
        }
        cout << a + b << '\n';
    }
    return 0;
}
```

输入：

```
12 34
99 1
0 0
```

输出：

```
46
100
```

这里的 `while (true)` 是 2.4.7 说的合法死循环：`break` 就是明确的出口。三种写法再汇总一遍：

题面说法	推荐写法
"第一行一个 `t`，表示 `t` 组数据"	`cin >> t;` + `while (t--)`
"多组输入，读到文件结束" / 没有组数	`while (cin >> n)`
"当 …… 时输入结束"（哨兵值）	`while (true)` + 读入 + `if (哨兵) break;`

2.8 循环四大套路：累加、计数、累乘、最值【主线必学】

第二次题单的半壁江山都是下面四种套路的直接应用。它们共用同一个骨架：先想好"答案变量"的初值，再让循环每轮更新它。请对每一个例子都动手把逐轮执行表补完——考场上 debug 靠的就是这张表。

2.8.1 累加：答案从 0 开始

题目（题单"循环练习 3"）：给定 $n(1 \le n \le 1000)$，计算 1 到 n 的平方和。样例：$n = 3$，输出 14（$1^2 + 2^2 + 3^2 = 14$）。

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

int main()
```

```cpp
{
    int n;
    cin >> n;

    long long sum = 0;
    for (int i = 1; i <= n; i++)
    {
        sum += 1LL * i * i;
    }

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

输入 3，逐轮执行：

轮次	i	sum += i² 之前	之后
1	1	0	1
2	2	1	5
3	3	5	14
结束	4	条件不成立	输出 14

n = 1000 时答案约为 3.3e8，仍在 int 范围内；但养成"求和用 long long"的习惯几乎零成本，能避开大量溢出 WA。被加的东西本身可能溢出时（如此处的 i * i），记得先转 long long 再加。

2.8.2 计数：数出满足条件的个数

题目（题单"循环练习 C"）：第一行输入 n，第二行输入 n 个分数，统计大于等于 60 分的人数。样例：6 个数 10 20 30 40 50 60，输出 1。

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

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

    int count = 0;
    for (int i = 1; i <= n; i++)
    {
        int score;
        cin >> score;
        if (score >= 60)
        {
            count++;
        }
    }
```

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

输入：

```
6
10 20 30 40 50 60
```

输出：

```
1
```

`count` 从 `0` 开始，每有一个分数达标就 `+1`。注意循环体和 `if` 里各管一件事：读入 → 判断 → 累加，三件事拆开写，错了也只在一件事上错。

2.8.3 累乘：答案从 1 开始

求和从 `0` 开始，因为 `0` 加任何数不变；求积要从 `1` 开始，因为 `1` 乘任何数不变。

题目（题单"循环练习 4"）：给定 $n(1 \le n \le 20)$，输出 $n! = 1 \times 2 \times \ldots \times n$。样例：$n = 3$，输出 6。

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

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

    long long product = 1;
    for (int i = 1; i <= n; i++)
    {
        product *= i;
    }

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

输入 `20`，输出：

```
2432902008176640000
```

`20! ≈ 2.4e18`，恰好还在 `long long`（约 ±9.2e18）范围内；`21!` 就爆了。这正是"先看数据范围再选类型"的又一个例子：如果题目把 `n` 给到 100，循环本身没问题，但答案类型必须升级（第 1 章提过的 `__int128` 可以算到 `33!` 左右，再大就需要专门的大数处理方法了）。顺便记住几个常见阶乘的数量级：`10! = 3628800`，`15! ≈ 1.3e12`，`20! ≈ 2.4e18`。

2.8.4 边读边处理：不存数组也能求最值

如果数据只需要使用一次，就可以每读入一个数立刻更新答案，不必等学完数组。下面同时计算一批数的总和、正数个数和最大值：

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

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

    long long x;
    cin >> x;

    long long sum = x;
    long long maximum = x;
    int positiveCount = (x > 0);

    for (int i = 1; i < n; i++)
    {
        cin >> x;
        sum += x;
        if (x > 0)
        {
            positiveCount++;
        }
        if (x > maximum)
        {
            maximum = x;
        }
    }

    cout << "sum " << sum << '\n';
    cout << "positive " << positiveCount << '\n';
    cout << "maximum " << maximum << '\n';
    return 0;
}
```

输入：

```
5
7 -2 0 12 4
```

输出：

```
sum 21
positive 3
maximum 12
```

重点在最大值的初始化：不能随手写成 `maximum = 0`，否则输入全是负数时会得到一个输入中根本不存在的答案。这里先读第一个数作为初值，再处理剩余 `n - 1` 个数。题单里的"找最大值"还更进一步——连 `n` 都不给，直接读到 EOF，把本节的骨架套进 2.7.2 的 `while (cin >> x)` 即可。

2.9 数位拆分与反转【主线必学】

题单里的水仙花数、数字翻转、快乐数、牛牛数数，考的都是同一个小动作：**把整数按十进制一位一位拆出来**。武器只有两个：`n % 10` 取末位，`n /= 10` 删末位。2.4.1 已经用它做过倒序输出，这里把套路补全。

2.9.1 拆末位：数位和

读入一个非负整数，求它的各位数字之和。拆位套路和 2.4.1 的倒序输出是同一个，核心只有三行：

```
    int digitSum = 0;
    while (n > 0)
    {
        digitSum += n % 10; // 取出末位，加进总和
        n /= 10;            // 删除末位
    }
```

以 `1234` 为例：末位依次取出 4、3、2、1，`digitSum` 依次变成 4、7、9、10，`n` 减到 0 后循环结束，输出 `10`。

只想要某一位（比如末位）时，`n % 10` 就够了；想"翻转""按位判断"时，就配一个"不断乘 10 往高位垒"的变量 `reversed`，或者逐位分支处理。

2.9.2 水仙花数：按位拆分 + 立方和

题目：输出 100～999 中所有满足"各位数字立方和等于它本身"的数（如 $153 = 1^3 + 5^3 + 3^3$）。样例输出：`153`、`370`、`371`、`407`。

思路分两层：外层用循环枚举每个三位数；内层把当前数拆位、求立方和、比较。拆位时**不要真的修改循环变量本身**——用临时变量 `rest` 拷贝一份再拆：

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

int main()
{
    for (int number = 100; number <= 999; number++)
    {
        int rest = number;      // 拷贝一份用于拆位，number 本身保持不变
        int digitSumOfCubes = 0;
        while (rest > 0)
        {
            int digit = rest % 10;
            digitSumOfCubes += digit * digit * digit;
            rest /= 10;
        }

        if (digitSumOfCubes == number)
        {
            cout << number << '\n';
        }
    }
    return 0;
}
```

运行结果：

```
153
370
371
407
```

以 153 为例的内层逐轮：rest = 153 → 末位 3，立方和 27，rest = 15 → 末位 5，立方和 152，rest = 1 → 末位 1，立方和 153 → 与 number 相等，输出。

一个常见的错误写法是直接在 while 里拆 number，拆完 number 变成 0，后面的比较就全错了。记住：**拆位用拷贝**。

2.9.3 数字反转：翻转后比较

题目（题单"数字翻转"）：多组数据，每组两个正整数 a、b（都为 0 时输入结束）。若"a 的反转 + b 的反转"等于"a + b 的反转"，输出 a + b，否则输出 NO。样例：12 34 输出 46（21 + 43 = 64，恰好是 46 的反转）；99 1 输出 NO。

反转本身和 2.9.1 同构：反复取末位、往高位垒：

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

```cpp
int reverseNumber(int n)
{
    int reversed = 0;
    while (n > 0)
    {
        reversed = reversed * 10 + n % 10;
        n /= 10;
    }
    return reversed;
}

int main()
{
    while (true)
    {
        int a, b;
        cin >> a >> b;
        if (a == 0 && b == 0)
        {
            break;
        }

        int sum = a + b;
        if (reverseNumber(a) + reverseNumber(b) == reverseNumber(sum))
        {
            cout << sum << '\n';
        }
        else
        {
            cout << "NO\n";
        }
    }
    return 0;
}
```

输入：

```
12 34
99 1
0 0
```

输出：

```
46
NO
```

`reverseNumber` 是第 1 章之后第一次出现的"自己命名的代码块"，也就是函数。这里只需把它当"打包好的工具"用：给它一个整数，它返回反转结果；怎么定义的可以先不求甚解，第 7 章会系统学。对 `12` 的逐轮演算：

`reversed = 0*10+2 = 2`，`n = 1`；`reversed = 2*10+1 = 21`，`n = 0`，返回 21。

2.10 上取整：ceil 的整数写法【主线】

题目（题单"only one！"）：数轴上 `x` 点有一颗糖，`n` 只蚂蚁想吃到它。蚂蚁 `i` 在位置 `pos`、速度 `v`，爬到糖所需时间为"距离 ÷ 速度**向上取整**"，例如距离 7、速度 3 → `7 / 3 = 2.33...`，向上取整为 3。问哪只蚂蚁最先吃到糖；同时到达则取编号最小者。

向上取整（ceil）的意思是"只要有零头就多算一个"。C++ 的整数除法天然向下取整，要向上取整，正整数情形下有一个非常常用的公式：

```
ceil(a / b) = (a + b - 1) / b        // a ≥ 0，b > 0，均为整数
```

推导（值得看懂，以后到处都是）：设 `a = q * b + r`，其中 `0 ≤ r < b`，即 `q = a / b`（整数除法，向下取整）。

- 若 `r = 0`：`a` 恰好是 `b` 的倍数，`(a + b - 1) / b = q + (b - 1) / b = q + 0 = q`，正确；
- 若 `r ≥ 1`：`a + b - 1 = q*b + r + b - 1 = (q + 1) * b + (r - 1)`，而 `0 ≤ r - 1 < b`，所以 `(a + b - 1) / b = q + 1`，也正确。

为什么不用浮点？ 距离最大约 200、速度最小为 1 时，`dist / v` 用 `double` 算再 `ceil` 当然也行；但数据更大时浮点会丢精度（回想第 1 章），而整数公式永远精确。竞赛中对"正整数的上取整"一律推荐这个写法。

另一个前置小问题：蚂蚁位置可能小于糖的位置，距离是两者之差的绝对值。还没学 `<cstdlib>` 的 `abs`，就先手写：

```
int distance = pos - x;
if (distance < 0)
{
    distance = -distance;
}
```

完整程序：

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

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

    while (t--)
    {
        int n, x;
        cin >> n >> x;
```

```
        int bestIndex = 0;      // 目前最优蚂蚁编号（从 1 开始）
        long long bestTime = 0; // 目前最短用时
        for (int i = 1; i <= n; i++)
        {
            int pos, speed;
            cin >> pos >> speed;

            int distance = pos - x;
            if (distance < 0)
            {
                distance = -distance;
            }
            long long time = (distance + speed - 1) / speed; // 向上取整

            if (bestIndex == 0 || time < bestTime)
            {
                bestIndex = i;
                bestTime = time;
            }
        }
        cout << bestIndex << '\n';
    }
    return 0;
}
```

输入（第二组样例）：

```
2
3 0
1 1
-1 1
-2 2
5 7
-6 1
99 10
-99 20
18 1
50 20
```

逐轮执行第一组（`n = 3`，糖在 0）：

蚂蚁 `i`	`pos`	`distance`	`time = (d + v - 1) / v`	是否更新最优
1	1	1	`(1 + 0) / 1 = 1`	是（首次，`bestIndex 0 → 1`）
2	-1	1	`(1 + 0) / 1 = 1`	否（`1 < 1` 不成立，编号小者保留）
3	-2	2	`(2 + 1) / 2 = 1`	否

输出 `1`。第二组算出来最小用时是第 5 只蚂蚁的 3 天，输出 `5`。

输出：

```
1
5
```

两处细节：`bestIndex == 0` 作为"还没有最优"的标记（正式比赛编号从 1 开始，0 一定不是合法答案）；并列时条件是严格的 `<`，所以先到的编号自然留在答案里。

2.11 格式化输出：printf 精确控制格式【主线】

第 1 章学过两套输入输出：`cin / cout` 自动匹配类型，`scanf / printf` 用格式符精确控制。平时的读入求和用哪套都行，但遇到"保留两位小数""每个数占 3 格宽度""不足两位补前导零"这类要求，`printf` 往往更顺手。本节格式符速查：

格式符	含义	示例（x 的值）	输出
`%d`	普通 `int`	42	`42`
`%lld`	`long long`	—	—
`%c`	字符	`'A'`	`A`
`%.2f`	保留 2 位小数（四舍五入）	3.14159	`3.14`
`%5d`	宽度 5，右对齐，空格补齐	42	`___42`
`%-5d`	宽度 5，**左**对齐	42	`42___`
`%05d`	宽度 5，右对齐，**0 补齐**	42	`00042`
`%02d`	宽度 2，0 补齐（宽度 2 的常用简写）	7	`07`

（表中 _ 代表空格。）`printf` 不会自动换行，需要换行时在格式串末尾写 `\n`。使用 `printf` 记得 `#include <cstdio>`；并且遵守第 1 章的规矩：**同一个程序只选一套 IO**，所以本节的完整例子统一用 `scanf` 读入、`printf` 输出，不掺 `cout`。

2.11.1 保留两位小数：平均分

题目（题单"精神分裂症的小明"）：`n` 个评委打分（0～10），去掉一个最高分和一个最低分（并列只去一个），求剩下分数的平均分，保留 2 位小数。样例：5 个分数 `9 5 6 8 9`，去掉一个 9 和一个 5，剩 `6 8 9`，平均分 `23 / 3 ≈ 7.67`。

用 2.8.4 的边读边处理套路：读一个就更新和、最高、最低。`%.2f` 负责最后的四舍五入：

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

int main()
{
```

```
    int n;
    scanf("%d", &n);

    int sum = 0;
    int maximum = -1; // 分数范围 0~10，初值取范围外即可
    int minimum = 11;

    for (int i = 1; i <= n; i++)
    {
        int score;
        scanf("%d", &score);
        sum += score;
        if (score > maximum)
        {
            maximum = score;
        }
        if (score < minimum)
        {
            minimum = score;
        }
    }

    double average = (sum - maximum - minimum) / double(n - 2);
    printf("%.2f\n", average);
    return 0;
}
```

输入：

```
5
9 5 6 8 9
```

逐轮执行（前 3 轮）：

轮次	读入	sum	maximum	minimum
1	9	9	9	9
2	5	14	9	5
3	6	20	9	5
…	…	37	9	5

`sum - maximum - minimum = 23`，`n - 2 = 3`，`23 / 3` 写成 `23 / double(3)` 保证是浮点除法（否则整数除法得到 7，就全完了）。输出：

```
7.67
```

注意 `(sum - maximum - minimum) / double(n - 2)`：`double(...)` 是第 1 章的强制转换，让除法在浮点下进行。

2.11.2 前导零：数字三角形

题目（题单"循环练习 5"）：输出直角边长为 `n` 的数字三角形，所有数字都用两位表示，不足两位补前导 0。样例 `n = 5`：

```
0102030405
06070809
101112
1314
15
```

观察规律：第 1 行有 `n` 个数，第 2 行 `n - 1` 个……数字从 1 开始连续递增。用一个计数器 `number` 边打边加，`%02d` 负责补零：

```cpp
#include <cstdio>

int main()
{
    int n;
    scanf("%d", &n);

    int number = 1;
    for (int row = 1; row <= n; row++)
    {
        for (int count = 1; count <= n - row + 1; count++)
        {
            printf("%02d", number);
            number++;
        }
        printf("\n");
    }
    return 0;
}
```

输入 `5`，对第 1 行的逐轮演算：`number` 依次输出 `01`、`02`、`03`、`04`、`05` 后变成 6；第 2 行输出 `06`～`09`，依此类推，得到上面的样例。`%02d` 对已经两位的数（如 `10`）原样输出，不会截断。

2.11.3 左对齐补位：对齐的九九乘法表

题单的九九乘法表要求"空格用 `%-3d` 补位"。程序其余部分和 2.6.3 完全一样，只需把内层的输出语句换成定宽左对齐：

```cpp
printf("%d*%d=%-3d", j, i, i * j); // 每个算式的结果占 3 格、左对齐
```

第 4 行输出形如 `1*4=4 2*4=8 3*4=12 4*4=16` ：结果不足 3 格的在右侧补空格，`12`、`16` 这类两位结果只占一格尾巴，列与列自然对齐。有些评测对每行末尾的空格不敏感，但养成"想清楚再输出"的习惯总是好的；若题目严格禁行尾空格，可以在 `j == i` 时改用 `%d` 去掉最后一项的补位。

2.11.4 `cout` 的等价写法【拓展】

`printf` 能做的事 `cout` 基本也能做，只是要请 `<iomanip>` 里的工具出场：

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

int main()
{
    double pi = 3.14159;
    cout << fixed << setprecision(2) << pi << '\n'; // 3.14

    int x = 7;
    cout << setw(2) << setfill('0') << x << '\n';   // 07
    return 0;
}
```

`fixed << setprecision(2)` 等价于 `%.2f`；`setw` 设宽度、`setfill` 设补齐字符，等价于 `%02d` 的合体。左对齐要再加 `left`。第 1 章已经见过 `fixed << setprecision(2)`，这里补上对照，想用哪套随你——但一个程序里别混。

2.12 简单过程模拟：把题面"过一遍"【主线】

有一类题不需要任何算法，只要把题目描述的过程原样写成代码，让程序替你"走一遍"。这类题叫**模拟**，是新手最该拿下的题型：读题仔细 + 变量轨迹表清楚，就能 AC。

做模拟题的三步：

1. 从题面里圈出**状态**：哪些量在过程中变化？（手头现金、已存金额、摘到的苹果数……）
2. 写出**每个时间单位**发生什么：月初？读到一个苹果？一个月？
3. 决定**循环变量**：一共要走多少步？答案在什么时刻输出？

2.12.1 逐条判定：摘苹果

题目（题单"[NOIP2005] 陶陶摘苹果"的改编教学版）：陶陶伸手能够到的高度为 `hand`，她还有一个 30 厘米的板凳；给定 `n` 个苹果各自的高度，求她能摘到几个（高度 ≤ 手高 + 30 就能摘到）。原题固定 10 个苹果且高度写在手高之前，学完数组可以回去重做一次；这里把顺序调整为先读 `n` 和 `hand`，方便"边读边处理"：

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

int main()
{
```

```
    int n, hand;
    cin >> n >> hand;

    int reach = hand + 30;
    int picked = 0;
    for (int i = 1; i <= n; i++)
    {
        int height;
        cin >> height;
        if (height <= reach)
        {
            picked++;
        }
    }

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

输入（对应原题样例：10 个苹果，手高 110，即 `reach = 140`）：

```
10 110
100 200 150 140 129 134 167 198 200 111
```

逐轮执行（前 5 个苹果）：

苹果	高度	height <= 140	picked
1	100	是	1
2	200	否	1
3	150	否	1
4	140	是（注意等于也算）	2
5	129	是	3
…	…	…	…

10 个苹果走完，输出：

```
5
```

这类"每读一个数就判定一次"的模拟，骨架和 2.8.2 的计数完全一样：**读入** → **判断** → **更新计数**，只是判断条件换成了题面句子。

2.12.2 逐月模拟：储蓄计划

题目（题单"[NOIP2004] 津津的储蓄计划"）：津津每月月初得到 300 元，当月要花掉预算 `budget`；如果月初（手里的钱 + 300）**不够**当月预算，计划在第 `month` 个月破产，输出 `-month`。否则她会在花钱前把手里整百的钱存到妈妈那里，年末妈妈返还存款并多加 20% 利息。给定 12 个月的预算，求年末津津手里的总钱数。

题面那段"11 月初有 83 元、妈妈给 300、预算 180、存 200、留 183"就是极好的逐轮样例。把过程拆成每月五步：得钱 → 判断够不够 → 花钱 → 存整百 → 剩余。状态两个：`balance`（手头现金）、`saved`（已存总额）：

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

int main()
{
    int balance = 0; // 手头现金
    int saved = 0;   // 存在妈妈那里的整百总额

    for (int month = 1; month <= 12; month++)
    {
        int budget;
        cin >> budget;

        balance += 300; // 月初妈妈给 300

        if (balance < budget) // 存的取不出，不够用
        {
            cout << -month << '\n';
            return 0;
        }

        balance -= budget;                // 花掉本月预算
        int deposit = balance / 100 * 100; // 把整百部分存起来
        saved += deposit;
        balance -= deposit;
    }

    cout << balance + saved + saved / 5 << '\n'; // 20% 利息 = 存款 / 5
    return 0;
}
```

用题面例子的口径手工验证样例一（前 7 个月预算 `290 230 280 200 300 170 340`）：

月	预算	得钱后	够花?	花后	存	手头剩	累计存款
1	290	300	是	10	0	10	0
2	230	310	是	80	0	80	0
3	280	380	是	100	100	0	100
4	200	300	是	100	100	0	200
5	300	300	是	0	0	0	200

月	预算	得钱后	够花?	花后	存	手头剩	累计存款
6	170	300	是	130	100	30	300
7	340	330	**否**	输出 `-7`			

第 7 个月得钱后只有 330，预算 340 花不起，输出 `-7`。样例二把第 7 个月的 340 换成 330，就不破产；程序对 12 个月走完，输出 `balance + saved + saved / 5`。用第二组样例跑完（手工或交给程序）：

```
1580
```

两处容易写错的细节：一是 `deposit = balance / 100 * 100`，整数除法先丢掉个位十位，正好表示"整百部分"；二是利息 `saved / 5` 恰好整除（`saved` 一定是 100 的倍数），所以结果仍是整数。

这种"按月推进"的模拟和"逐条判定"的模拟覆盖了题单里模拟题的基本盘；更花哨的过程（赏月、快乐数）只是把"月"换成"天"或"一轮替换"，骨架不变。

2.13 枚举与复杂度直觉【主线】

枚举不是某个新语法，而是一种解题思路：把可能的候选按某种顺序逐个尝试，检查哪些满足条件。当候选数量不大时，枚举往往是最稳的正解，而不是"不会算法才暴力"。写枚举前回答三个问题：

1. 枚举什么变量?
2. 每个变量的范围是什么?
3. 怎样检查一个候选是否合法?

2.13.1 单变量枚举：找约数

枚举候选约数 `1` 到 `n`，用 `n % divisor == 0` 检查整除。试除循环的完整写法见 2.5.1（判断质数），把其中的 `break` 版检查换成逐个输出即是本题：

```
    for (int divisor = 1; divisor <= n; divisor++)
    {
        if (n % divisor == 0)
        {
            cout << divisor << ' ';
        }
    }
    cout << '\n';
```

输入 `12`，输出 `1 2 3 4 6 12`（最后还有一个换行）。这里最多循环 `n` 轮。

2.13.2 双变量枚举：百钱买百鸡（含上界推导）

题目：100 元钱，公鸡 5 元一只，母鸡 3 元一只，小鸡 1 元三只。恰好买 100 只且花完 100 元，求所有方案。

三种鸡要不要三层循环？先推导再上代码。**推导上界**：

- 公鸡 5 元一只，100 元全买公鸡也只能买 `100 / 5 = 20` 只，所以 `rooster ≤ 20`；
- 母鸡 3 元一只，同理 `hen ≤ 100 / 3 = 33.33`，取整得 `hen ≤ 33`；
- 小鸡不用枚举：总数固定 100 只，定下前两种后 `chick = 100 - rooster - hen` 直接算。

上界从 100 缩到 20 和 33，循环总轮数从 $100^3 = 1e6$ 降到 $21 \times 34 = 714$——这就是"先推导、再枚举"的价值。再检查两条约束：`chick` 不能为负，且必须能被 3 整除（小鸡 1 元三只，按整组买）：

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

int main()
{
    for (int rooster = 0; rooster <= 20; rooster++) // 上界：100 / 5
    {
        for (int hen = 0; hen <= 33; hen++)         // 上界：100 / 3
        {
            int chick = 100 - rooster - hen;
            if (chick >= 0
                && chick % 3 == 0
                && 5 * rooster + 3 * hen + chick / 3 == 100)
            {
                cout << rooster << ' '
                     << hen << ' '
                     << chick << '\n';
            }
        }
    }
    return 0;
}
```

运行结果：

```
0 25 75
4 18 78
8 11 81
12 4 84
```

本可以直接枚举 `rooster` 到 100、`hen` 到 100，反正答案一样；但推导上界是竞赛的核心习惯——数据范围稍大，不加推导的枚举就会从"能跑"变成"TLE"。

2.13.3 枚举时避免重复

若要枚举满足 `1 <= a < b <= n` 的数对，内层不必从 1 开始：

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

int main()
```

```
{
    int n;
    cin >> n;

    for (int a = 1; a <= n; a++)
    {
        for (int b = a + 1; b <= n; b++)
        {
            cout << a << ' ' << b << '\n';
        }
    }
    return 0;
}
```

输入 `4`，输出：

```
1 2
1 3
1 4
2 3
2 4
3 4
```

这样既不会出现 `(a, a)`，也不会把 `(1, 2)` 和 `(2, 1)` 重复计算。题单"幸运 UID"那道"六位编号中有多少个各位和能被 `n` 整除"，就是外层枚举 1～999999、内层拆数位的双层结构，总量约 `1e6` 轮，轻松跑完。

2.13.4 一秒能跑多少次：复杂度直觉

复杂度的系统知识会在后续章节展开，本章先建立最重要的直觉：**不要只看有几层循环，要数循环体总共执行多少次。** 假设 `n` 是输入规模：

代码形态	执行次数直觉	常写作
从 1 遍历到 n	约 n 次	`O(n)`
两层都从 1 到 n	约 n^2 次	`O(n²)`
每轮把 n 除以 2	约 $\log_2 n$ 次	`O(log n)`
三层都从 1 到 n	约 n^3 次	`O(n³)`

这里的大 O 只关心数据变大时的增长趋势，会忽略常数和较低次项。在普通 C++ 竞赛环境中，`1e8` 次**极简单**操作常被当作一秒量级的粗略上限，但这不是保证：取模、除法、输入输出比加法慢，评测机和时限也各不相同。入门阶段可用下面的量级帮助排除明显错误：

- `n <= 1e5`：`O(n)` 通常轻松，`O(n²)` 往往不行；
- `n <= 5000`：`O(n²)` 已经要看常数和时限；
- `n <= 500`：简单 `O(n³)` 有时可行；
- `n <= 20`：甚至可能允许枚举所有子集，这会在以后学习。

若 `n = 100000`，两层各跑 `n` 次就是 `1e10` 次，不要等提交后的 TLE 来告诉你它太慢。反过来，看到 `n ≤ 1000` 就该放心：`1e6` 轮随便跑（题单里的平方和、质数判断都属于这种）。

2.13.5 倍增循环通常是 O(log n)

下面的循环变量不断翻倍：

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

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

    int count = 0;
    for (long long value = 1; value <= n; value *= 2)
    {
        count++;
    }

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

输入 `100` 时，`value` 依次是 `1, 2, 4, 8, 16, 32, 64`，只循环 7 次，输出：

```
7
```

每翻倍一次就更接近 `n`，执行次数约为 $\log_2 n$。循环变量使用 `long long`，这样即使 `n` 接近 `int` 上限，最后一次翻倍也不会先在 `int` 中溢出。判断复杂度既要看变量怎样变化，也要检查更新表达式会不会越过类型范围。

2.13.6 用数学公式替代循环

`1 + 2 + ... + n` 可以循环求和，也可以用公式 `n * (n + 1) / 2`。即使 `n` 是 `1e9`，逐项循环也要跑 10 秒往上，公式却只需常数次运算，而且结果仍能由 `long long` 保存：

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

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

    long long answer;
    if (n % 2 == 0)
    {
```

```
        answer = (n / 2) * (n + 1);
    }
    else
    {
        answer = n * ((n + 1) / 2);
    }

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

输入 `1000000`，输出：

```
500000500000
```

先除以 2 再相乘，比直接计算 `n * (n + 1)` 更不容易溢出。题单"小 B 的米饭"（`n ≤ 1e9`，用总和公式 `n * (n + 1) / 2` 减去碗里剩余的）就是这一思想配上 `long long` 的应用；如果数学结果本身超过 `long long`，就需要更大的类型。

2.14 常见错误对照：CE、RE、WA、TLE【主线】

写循环题时，四类判题结果各有典型原因。先按这张表自查，再考虑"是不是算法不对"。

2.14.1 CE：编译错误

常见原因一：条件后误加分号。 `if (x > 0);` 中的分号代表一个空语句，后面的花括号不再受 `if` 控制。它有时能编译，但逻辑已经错了；若后面还有 `else`，甚至可能直接 CE。

常见原因二：漏括号、漏分号、符号不成对。 出现大量报错时，先看编译器给出的**第一条**错误。后续报错往往只是第一处语法破坏引发的连锁反应。

常见原因三：把 `=`、`==` 或中文符号写混。 赋值与相等判断含义不同；中文括号、分号、引号也不是合法 C++ 标点。

2.14.2 RE：运行时错误

本章还没有数组，最常见的 RE 是整数除以 0 或取模 0：

```
a / 0
a % 0
```

这两个操作都是未定义行为，可能直接导致程序异常退出。把"除数不为 0"写在短路条件左侧（2.1.6），或先用 `if` 单独检查。另一个风险是有符号整数溢出，它同样属于未定义行为，不能假设程序总会得到某个固定错误值。

2.14.3 WA：答案错误

循环题的 WA 优先检查：

1. `=` 是否误写成 `==`，或反过来；
2. `&&` 与 `||` 是否写反；
3. 边界应该是 `<` 还是 `<=`；
4. 累加器、计数器是否初始化；
5. 多组数据时，变量是否在每组开始前重置；
6. `int` 是否装得下答案和中间结果；
7. 题目要求的输出格式是否完全一致（空格、换行、小数位数、前导零）。

自测时至少试：最小合法输入、最大合法输入、刚好跨过分支边界的值、0、负数（若允许）和一个普通值。**交题之前，先拿样例在本地跑一遍**——样例都不过的代码没有资格见评测机。

2.14.4 TLE：超出时间限制

循环题的 TLE 常见于三类原因：

- 忘记更新循环变量，写出了死循环；
- `continue` 跳过了 `while` 中的更新语句（见 2.5.2）；
- 程序确实会结束，但复杂度太高，例如 `n = 1e5` 时仍写 $O(n^2)$，或该推导上界时直接大力枚举（见 2.13.2）。

判断方法：先问"它能不能结束"，再问"结束前总共执行多少轮"（见 2.13.4）。

2.14.5 一张表总结

结果	检查
CE	括号、分号、花括号、中文标点、`else` 是否有匹配的 `if`
RE	除数是否为 0、有符号溢出等未定义行为
WA	条件逻辑、循环边界、初始化、类型、输出格式
TLE	死循环、更新语句被跳过、枚举次数过多

编译器警告不一定意味着编译错误，但它经常能抓到赋值写进条件、变量未使用、符号类型混合等问题。警告算是免费的人工审查，不要主动忽略。

2.15 拓展选读【拓展】

> 下面几节对应题单里位置靠后的几道题。它们不代表"必须现在就学"：主线刷完后，按兴趣挑着读即可。拓展节可以引用主线没教过的工具，遇到陌生的名字先查目录或等后续章节。

2.15.1 复杂图形打印：找规律，不要背答案

题单里那道用大写字母 `A` 庆祝 AK 的题，要求输出 `n` 行、`2n + 5` 列、只含 `*` 和 `.` 的图案。这类复杂图形的通用做法：

1. 先数行数、列数，确认和外层循环变量的关系（这里 `n` 行，每行 `2n + 5` 个字符）；
2. 把每行切成几段"随 `row` 变化的量"，例如"A"字的尖顶、横杠、两侧斜线各自占哪些列；

3. 对每行用内层循环逐列判断"该列是不是 `*`"，规律通常是一次函数（列号 = 行号 ± 常数）加几个特殊行。

逐行抄写样例、硬编码 `if (n == 8)` 之类的做法在数据范围只有 8 和 11 的样例上看似可行，但评测数据一变就全线崩溃。把图案规律写成通式才是正路。这套"观察 → 分段 → 写通式"的流程，练一道就够，不必沉迷刷图形题。

2.15.2 大数据模拟：小 B 跑马拉松

题面：小 B 和小 K 轮流写代码，小 B 第 1、3、5…天每天写 `x` 千行，小 K 第 2、4、6…天每天写 `y` 千行，先让累计量**超过** `a + 0.5` 千行（即达到 `a + 1`）者胜出。`x, y, a ≤ 1e9`。

第一反应是"逐天模拟"，但天数最多约 `1e9` 轮（2.13.4 说一秒只有约 `1e8` 次运算），模拟必然 TLE。这题的正确姿势是**用数学代替循环**：设小 B 在第 `2k - 1` 天后累计写了 `k * x` 千行，小 K 在第 `2k` 天后累计 `k * y` 千行。小 B 在第 `2k - 1` 天胜出，当且仅当：

- `k * x ≥ a + 1`（他这一天达标了）；
- `(k - 1) * y ≤ a`（在此之前小 K 还没达标）。

其中最小的 `k = ceil((a + 1) / x)`，直接用 2.10 的上取整公式 `(a + x) / x` 算出，整个问题变成常数次整数运算。对照样例：`x = 1, y = 2, a = 4` 时 `k = 5`，`(k-1) * y = 8 > 4`，小 K 先到，输出 `YES`；`x = 2, y = 1, a = 4` 时 `k = 3`，`(k-1) * y = 2 ≤ 4`，小 B 先到，输出 `NO`。

这类"看起来要模拟、其实有公式"的题是竞赛常客。判断线索就是数据范围：`1e9` 级别的规模几乎总是在暗示"别模拟，找规律"。

2.15.3 很大很大的数：完全数

题单里那道完全数题把 `x` 给到 `1e80`，远超 `long long` 的 `9.2e18`。两个层次的解法：

- 如果只是想通过本题：读成字符串（第 4 章正式学，`cin >> s` 读 `string` 即可），先比长度、再逐位比字典序，与已知完全数列表比对即可；
- 想理解原理：欧几里得-欧拉定理指出，偶完全数恰好是 `2^(p-1) * (2^p - 1)` 且 `2^p - 1` 为梅森素数。`1e80` 以内的偶完全数一共只有 12 个，全部打表比对绰绰有余（是否存在奇完全数是著名未解问题，但目前已知的完全数全是偶数，打表法在题目数据范围内成立）。

这也预告了之后的两个知识点：字符串（第 4 章）与打表（把少量已知答案写死在程序里）。

2.15.4 复杂贪心方向：文明 2

"完美结局"那题：每个国家 `i` 有 `a_i` 块土地，选 `k` 个国家两两建交需要 `k * (k - 1)` 块别国大使馆土地加 `k` 块备用土地，即每个入选国家实际贡献 `min(a_i, k)` 块，求最大的 `k` 使 `Σ min(a_i, k) ≥ k * k`。直接的 `O(n²)` 枚举会超过时限，正解方向有二：对每个 `a_i` 分类讨论的数学解法，或"猜一个 `k`、检查可行性、按结果调整"的二分套路。两者都超出本章范围，知道"这题存在更快的做法"即可，做完主线再来挑战。

2.15.5 控制流综合：二分猜数字

假设答案位于 `[1, n]`。每次猜中间值，系统只告诉你答案更大、更小或猜中。下面用一个确定的答案模拟这个过程，展示条件、循环和"范围减半"如何连在一起：

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

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

    if (n < 1 || target < 1 || target > n)
    {
        cout << "invalid input\n";
        return 0;
    }

    int left = 1;
    int right = n;
    int guesses = 0;

    while (left <= right)
    {
        int middle = left + (right - left) / 2;
        guesses++;

        if (middle == target)
        {
            cout << "found " << middle << '\n';
            cout << "guesses " << guesses << '\n';
            break;
        }
        else if (middle < target)
        {
            left = middle + 1;
        }
        else
        {
            right = middle - 1;
        }
    }
    return 0;
}
```

输入：

```
100 73
```

输出：

```
found 73
guesses 6
```

每轮都会排除当前范围的一半，因此即使 `n = 1e9` 也只要约 30 次（`2^30 ≈ 1e9`，见 2.13.5 的倍增直觉）。这就是二分查找的核心思想；数组版本会在排序与查找章节正式学习，本例只当控制流综合训练。`middle = left + (right - left) / 2` 的写法是为了避免 `left + right` 在数据大时溢出。

2.16 代码习惯与本章小结

2.16.1 代码习惯

1. `if` 和循环统一写花括号，即使当前只有一行；
2. 变量先初始化：累计和常从 `0` 开始，累乘常从 `1` 开始，求最值用第一个数据或范围边界当初值；
3. 条件复杂时拆成有名字的 `bool`，例如 `bool leap = ...;`；
4. 循环前明确区间是闭区间还是半开区间；
5. 能提前确定答案时用 `break`，但不要为了少跑几轮把代码写得晦涩难懂；
6. 写完后拿小数据在纸上走一遍变量变化（逐轮执行表）；
7. 交题之前先在本地过样例。

2.16.2 本章小结

本章看起来知识点很多，实际上可以压缩成下面这张路线图：

```
变量与常量
    ↓ 通过运算符组合
表达式（产生一个值）
    ↓ 关系 / 逻辑表达式产生 bool
条件判断（选择一条路）
    ↓ 把判断与更新反复执行
循环（处理一批候选）
    ↓ 多个循环组合
枚举（系统尝试所有可能）
    ↓ 数总执行次数
复杂度入门
```

需要真正记住的关键点：

- 相等判断是 `==`，赋值是 `=`；
- 区间条件要用 `&&` 连接，C++ 不能写数学式 `1 <= x <= n`；
- 题面写 `10^8`，代码写 `1e8`；`^` 在 C++ 里不是乘方；
- `&&` 和 `||` 会短路，危险运算应放在保护条件之后；
- `for` 适合范围明确，`while` 适合结束条件明确；
- `break` 结束当前循环，`continue` 跳过当前这一轮，但 `while` 里要保证更新不被跳过；
- 多组输入三套写法：`while (t--)`、`while (cin >> n)`、哨兵 + `break`，选定一种用到底；
- `int * int` 可能在赋给 `long long` 前就溢出，必要时乘 `1LL`；
- 正整数上取整用 `(a + b - 1) / b`；

- 写循环必须同时想清楚起点、终点和更新；数据范围决定可接受的枚举次数。

2.17 配套练习

本章配套训练为 **2026 级 SYUCT-ACM 新生第二次训练（循环）**：

https://oj.syuctacm.cn/contest/6ac5b694753773230ebbcb8f

建议顺序：先按下面的对照表把主线题刷完，再回头挑战拓展题。标注"综合"的题会同时用到两三节内容，卡住时回到对应小节抄一遍骨架再改。

对应小节	题单中的题（知识点）
2.1 表达式	小 B 的米饭（等差数列求和 +`long long`）、精神分裂症的小明（去最值求平均，配 2.11）
2.3 条件判断	奇偶分离、牛牛数数（含 4 的数位判断，配 2.9）
2.4 循环基础	循环练习 1（输出奇数）、小 B 的乘法口诀、找最大值
2.6 嵌套循环	九九乘法表（配 2.11.3）、M ? ACM !（数字矩阵 + 多组）、abc（周期字符串 + 多组）
2.7 多组输入	以上带"多组"的题，以及数字翻转（哨兵结束）
2.8 四大套路	循环练习 3（平方和）、循环练习 4（阶乘）、循环练习 C（计数）、找最大值
2.9 数位处理	水仙花数、快乐数（综合）、数字翻转（综合）
2.10 上取整	only one！（蚂蚁吃糖）
2.11 格式化输出	循环练习 B（分数数列，保留两位小数）、循环练习 5（数字三角形）、牛牛学数列 3（保留三位小数）
2.12 模拟	陶陶摘苹果、津津的储蓄计划、cayun 日常之赏月（分支 + 多组）
2.13 枚举与复杂度	幸运 UID（六位枚举）、奶龙的质数判断、小明的口袋（质数 + 累加）、快乐数
【拓展】	A（复杂图形）、小 B 跑马拉松（大数据公式）、开拓者的奇妙冒险 2（大数完全数）、栗酱的文明 2（复杂贪心）

做题时记住：**条件、边界、更新**。