第 1 章 变量、数据类型与基本输入输出

> 本文是《XCPCer 从零开始的 C/C++ 入门指南（算法竞赛向）》系列第 1 章，系列训练大纲见本文。

1.0 环境搭建：Windows + Algo Bootstrap + MinGW

1.0.1 我们要准备的三个东西

写 C++ 程序只需要三样东西，各司其职：

东西	作用	类比
g++（编译器）	把人能读懂的 `.cpp` 源代码翻译成电脑能执行的 `.exe`	翻译官
VS Code（编辑器）	用来打字、改代码的工具，带语法高亮和报错提示	纸和笔（高级版）
Code Runner / cph（运行插件）	自动完成编译和运行，提供按钮、输入区与结果面板	一键操作按钮

市面上常见的 IDE 还有 Visual Studio、CLion、Code::Blocks 等。本讲义在 Windows 上统一使用 **Algo Bootstrap 一键配置的 MinGW（MinGW-w64）与 VS Code**，安装依赖和配置环境都由它一次完成。

1.0.2 第一步：用 Algo Bootstrap 一键配置

1. 打开 Algo Bootstrap 官网，下载 Windows 版本并运行安装程序；
2. 按安装界面的提示选择 **C++ / MinGW 与 VS Code 环境**，点击一键配置。不同版本的按钮文字可能略有变化，以官网"安装教程"为准；
3. 配置完成后，编译器通常位于 `C:\Algo Bootstrap\mingw64\bin\g++.exe`。如果安装时选择了其他位置，以你的实际安装目录为准；
4. **关闭已经打开的 VS Code 和终端窗口**，重新打开一个 PowerShell，依次输入：

   ```
   g++ --version
   where.exe g++
   ```

   第一条会显示 g++ 的版本号，第二条会显示编译器的位置。本文验证时使用的是 MinGW-Builds g++ 16.1.0，路径为 `C:\Algo Bootstrap\mingw64\bin\g++.exe`。你的版本号和构建信息不同也没关系，只要能正常显示即可。

> 如果提示"g++ 不是内部或外部命令"，先确认上述 `g++.exe` 文件确实存在，再重开 VS Code 和终端。仍不行时，检查 PATH 是否包含 `C:\Algo Bootstrap\mingw64\bin`；不要急着重新安装，也不要同时安装另一套编译器。

1.0.3 第二步：初始化项目并安装运行插件

1. 打开 Algo Bootstrap，选择 **新建或初始化旧项目**；
2. 选择一个用于存放代码的文件夹，例如 `D:\code`，确认后会为你初始化项目并打开 VS Code；
3. 在 VS Code 中按 `Ctrl + Shift + X` 打开扩展面板，搜索并安装 **Code Runner**（发布标识为 `formulahendry.code-runner`）；
4. 确认扩展面板里还有 Microsoft 的 **C/C++** 扩展。前者负责"点按钮编译运行"，后者负责代码高亮、错误提示和跳转，两者作用不同。

以后把 `.cpp` 文件放在这个项目文件夹里。我们先只用 Code Runner；cph 留到 1.0.6，避免一开始同时背两套界面。

1.0.4 第三步：让 Code Runner 使用本讲编译参数

第一次使用 Code Runner 前，只需要配置一次。它以后会自动完成编译，不需要你手敲编译命令。

1. 按 `Ctrl + ,` 打开 VS Code 设置，搜索 `code-runner.executorMap`；
2. 点击 **Edit in settings.json**（在 settings.json 中编辑）；
3. 如果文件是空的，填入下面内容。如果已经有其他设置，只添加这些项目，或在 `executorMap` 中替换 `cpp` 这一项，**不要覆盖其他设置，注意各项之间的逗号**：

```
{
    "code-runner.runInTerminal": true,
    "code-runner.saveFileBeforeRun": true,
    "code-runner.ignoreSelection": true,
    "code-runner.executorMap": {
        "cpp": "cd \"$dir\" ; g++ -std=c++17 -O2 -Wall \"$fileName\" -o
\"$fileNameWithoutExt.exe\" ; if ($?) { & \".\\$fileNameWithoutExt.exe\" }"
    }
}
```

这几项分别表示：

- `runInTerminal`：程序在 TERMINAL（终端）面板中运行，以后可以在这里给 `cin` 输入数据；**编译由插件自动完成，不需要你输入命令**；
- `saveFileBeforeRun`：点击运行前自动保存当前文件，避免运行到旧版本；
- `ignoreSelection`：即使选中了一部分代码，也运行整个文件；
- `executorMap`：C++ 文件的编译方式。`g++` 后面的 `-std=c++17` 表示使用 C++17，`-O2` 表示开启优化，`-Wall` 表示尽量显示可疑写法的警告。

上面的 `cpp` 配置使用 PowerShell 语法。如果 VS Code 默认终端不是 PowerShell，请在终端右上角的下拉菜单中新建一个 **PowerShell** 终端，再点击 Run Code。

如果保存后配置报错，先检查 JSON 的引号、花括号和逗号，不要急着重装插件。

1.0.5 第一个程序：点击 Run Code

在 VS Code 里新建文件 `main.cpp`，输入下面的代码（**所有的标点都是英文半角**）：

```cpp
#include <iostream>   // 示例 1-1：Hello World
using namespace std;

int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

保存后，点击编辑器右上角的三角形 **Run Code** 按钮；也可以在代码区域右键，选择 **Run Code**。插件会自动编译并运行当前文件，稍等片刻，终端面板就会显示：

```
Hello, World!
```

如果程序要求输入，直接在 TERMINAL 面板中键入数据并按回车，不需要手动编译。若点击按钮后没有任何输出或异常退出，可能是多套 MinGW 的运行库冲突：先执行 `where.exe libstdc++-6.dll`，若第一条不是 Algo Bootstrap 的路径，就在 Windows 环境变量中把 `C:\Algo Bootstrap\mingw64\bin` 移到其他 MinGW（例如 Git 自带的 `C:\Program Files\Git\mingw64\bin`）前面，**完全退出并重新打开 VS Code**，再点击 Run Code。

恭喜，你的第一个 C++ 程序跑起来了。现在**逐行拆解**这段程序骨架，之后每一章的每个程序都由这几部分组成：

代码	作用
`#include <iostream>`	**包含头文件**。`iostream` 是输入输出流库，`cin`、`cout` 都住在里面。想用什么库功能，就先包含对应的头文件，`#include` 必须写在程序最顶上。
`using namespace std;`	**使用标准命名空间**。标准库的东西都带 `std::` 前缀，这句相当于声明"后面写的 `cin`、`cout` 都指标准库的"，省得每次都写 `std::cout`。
`int main()`	**主函数，程序的入口**。电脑执行你的程序时，永远从 `main` 的第一行开始。`int` 表示它结束时返回一个整数。
`{ }`	**花括号成对出现**，括起来的部分叫函数体，是 `main` 里要执行的语句。写代码时养成"先写一对括号再往里填"的习惯，可以避免大量低级错误。
`cout << "Hello, World!\n";`	**输出语句**。`cout` 是标准输出（屏幕），`<<` 是"送出"方向，把右边的内容依次打印到屏幕；`\n` 是换行；行尾的分号 `;` 表示语句结束，**每条语句末尾都要有**。
`return 0;`	**返回 0，表示程序正常结束**。写在 `main` 的最后。

如果编译没有通过，面板会打印错误信息：**从第一条错误看起**，多半是上一行的标点写错了（分号、引号、括号），本章 1.9 节会专门讲最常见的三种错误。

> 程序里的双斜杠 `//` 是**注释**：从 `//` 到这一行结尾的内容只是给人看的说明，编译器完全不理会。善用注释标记自己的思路。

1.0.6 【可选】用 cph 管理题目和样例

当你已经能用 Code Runner 跑通程序，再考虑是否使用 **Competitive Programming Helper（cph）**。它更适合算法竞赛：可以为一道题保存多组输入和期望输出，并自动判断通过与否。

1. 在扩展面板搜索并安装 **Competitive Programming Helper (cph)**，发布标识为 `DivyanshuAgrawal.competitive-programming-helper`；
2. 在 VS Code 设置中搜索 `cph.language.cpp.Args`，填入 `-std=c++17 -O2 -Wall`。`cph.language.cpp.Command` 保持默认的 `g++` 即可；
3. 打开一个 `.cpp` 文件，点击 **Run Testcases**，或按 `Ctrl + Alt + B`，手动添加输入与期望输出，再运行并查看每组样例的结果。

支持的 OJ 还可以配合浏览器扩展 Competitive Companion 导入题目和样例；无法导入时，像上面这样手动添加即可。**刚开始学习时只用 Code Runner 就够了，不必同时配置 cph。**

1.1 导语

程序处理的数据，必须先放进**变量**才能参与运算。变量不是一种通用容器——每一种变量都有确定的**类型**，类型决定了它能存什么、能存多大、参与运算时遵循什么规则。

类型选错是算法竞赛中最隐蔽的错误来源之一：算法完全正确，仅仅因为变量类型装不下答案，评测结果就是 Wrong Answer（WA，表示程序输出与标准答案不符）。更常见的情况是：程序在本地小数据上一切正常，一提交就在大数据上出错。

所以本章的主线只有两件事：**搞懂几种基本类型**，以及**学会让程序读入数据、输出答案**。

1.2 变量的声明、初始化与命名

1.2.1 声明

声明一个变量的语法为：

```
类型 变量名;
```

示例 2-1：一次声明多个变量。

```
int a, b, c;
int totalScore, solvedCount, wrongAnswers;
```

一条语句可以声明同类型的多个变量，以逗号分隔。算法竞赛代码中常见 `int a, b, c;` 的紧凑写法，但初学阶段建议**一行一个变量、名字取完整**，可读性更好。

1.2.2 初始化

声明的同时给定初值称为初始化，常见写法有三种：

```
int a = 0;    // 赋值式，我们最常用的是这一种
int b(0);     // 括号式
```

```
int c{0};    // 花括号式（C++11）
```

必须养成的习惯：定义变量后立即初始化。 函数内部的局部变量如果不初始化，它的值不是 0，而是一块**不确定的旧数据（垃圾值）**——使用它是未定义行为，程序不会报错，但结果不可预测。

示例 2-2：未初始化局部变量的错误示范。

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

int main()
{
    int x;          // 声明了，但没有初始化
    cout << x << '\n';   // 输出的是垃圾值，不可预测
    return 0;
}
```

用 `-Wall` 编译时，编译器会警告 `'x' is used uninitialized`。即使强行运行（本次运行恰好输出 `0`），也**纯属巧合**——下次编译、换台电脑、改一行代码，都可能变成乱码。这类错误的危险之处就在于：它不报错，甚至经常"碰巧"正确。

唯一的例外是**全局变量**（定义在所有函数之外的变量）：不初始化时会被自动清零。这一点在数组上尤其有用，将在 1.8 节展开。

1.2.3 命名

语法层面的规则只有四条：

1. 只能由字母、数字、下划线组成；
2. 不能以数字开头；
3. 不能使用 C/C++ 关键字（如 `int`、`return`）；
4. 区分大小写，`sum` 与 `Sum` 是两个变量。

命名建议：用**有意义的英文单词或拼音**，`sum`、`score`、`nianFen` 都比 `a1`、`xx` 好；多个单词用驼峰或下划线连接。

> **避开库名**是竞赛经验的另一条红线：`y1`、`j1`、`rank`、`index`、`time` 等名字容易与标准库或评测环境里的名称冲突，引发难以理解的编译错误，建议改写为 `yy1`、`rk`、`idx`。

1.3 基本数据类型

1.3.1 类型总表

类型	字节	表示范围	printf / scanf	字面量后缀	典型场景
`bool`	1	`true` / `false`	`%d`	—	标记、flag

类型	字节	表示范围	printf / scanf	字面量后缀	典型场景
`char`	1	是否有符号由实现决定；至少能保存基本字符集	`%c`	—	单个字符
`int`	4	-2,147,483,648 ~ 2,147,483,647（约 $\pm 2.1\times10^9$）	`%d`	—	默认整数
`unsigned int`	4	0 ~ 4,294,967,295（约 4.2×10^9）	`%u`	`1u`	慎用，见 1.5 节
`long long`	8	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807（约 $\pm 9.2\times10^{18}$）	`%lld`	`1LL`	大整数、多数答案
`unsigned long long`	8	0 ~ 18,446,744,073,709,551,615（约 1.8×10^{19}）	`%llu`	`1ULL`	位运算、哈希
`float`	4	有效数字约 7 位	`%f`	`1.0f`	少用
`double`	8	有效数字约 15~16 位	`%f`（输出）/ `%lf`（输入）	—	默认浮点
`long double`	实现相关	精度不低于 `double`，具体取决于平台	`%Lf`	`1.0L`	需要额外精度时先确认评测环境

注：表中的整数大小是常见竞赛评测环境的典型值。`char` 的符号性与 `long double` 的大小、精度由编译器和平台决定；需要依赖它们的具体范围时，应使用 `sizeof` 和 `<limits>` 实测，而不是凭操作系统猜测。

1.3.2 整型的选择

两条基本原则：

1. **默认用 `int`**，预估结果可能超过 2.1×10^9 时直接用 `long long`，不要抱侥幸心理；
2. **算法竞赛通常不写 `long`**。`long` 在 Windows 上常为 4 字节、在 64 位 Linux 评测机上常为 8 字节，容易产生跨平台误解；需要大整数时使用 `long long`，它在常见竞赛环境中保证至少 8 字节。

1.3.3 浮点型：float、double 与 long double

浮点数除了表示范围，还要特别关注**有效数字位数**：常见环境中 `float` 约 6~7 位，`double` 约 15~16 位。这意味着一个 9 位整数（如 123456789）用 `float` 存储就可能开始丢精度。

示例 3-1：float 存储大整数丢精度。

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

```cpp
int main()
{
    float f = 123456789.0f;   // 后缀 f 表示 float 字面量
    double d = 123456789.0;
    cout << fixed << setprecision(0);   // 按整数格式输出（用法见 1.6.5）
    cout << f << '\n';   // 输出 123456792，末三位已经不是原数
    cout << d << '\n';   // 输出 123456789，分毫不差
    return 0;
}
```

运行结果：

```
123456792
123456789
```

同一个数，装进 `float` 再取出来就变成了 `123456792`——`float` 只有约 7 位有效数字，装不下 9 位的精确值，只能保存最接近它的可表示值。而 `double` 有 15~16 位有效数字，通常可以精确表示绝对值不超过 2^{53} 的整数。

因此算法竞赛中的浮点数**默认使用 `double`，`float` 基本可以忽略**；涉及计算几何或累加大量小数时，可在确认评测环境后考虑 `long double`。

输入输出有一个易错点：`printf` 输出 `float` 和 `double` 都用 `%f`，但 `scanf` 读入 `double` 必须用 `%lf`；写错格式符属于未定义行为，使用字面格式串并开启 `-Wall` 时编译器通常会警告。`long double` 的读写统一用 `%Lf`。

1.3.4 字符型 char

`char` 的本质是一个 1 字节的整数，里面存的是字符编码。在 ASCII 编码中，常用字符按顺序连续编号：`'A'` 是 65，`'B'` 是 66，……，`'a'` 是 97，`'0'` 是 48。字符参与算术运算前会先提升为 `int`，所以 `'A' + 1` 的值是整数 `66`。

示例 3-2：字符与它的编码可以互相换算。

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

int main()
{
    char c = 'A';
    cout << c << '\n';            // 输出 A
    cout << c + 1 << '\n';        // 输出 66：字符先转成编码 65，再加 1
    cout << char(c + 1) << '\n';  // 输出 B：66 转回字符
    cout << 'C' - 'A' << '\n';    // 输出 2：两个字符编码相减
    return 0;
}
```

逐行演算：`c` 的值是 `'A'`（编码 65）；`c + 1` 是 `65 + 1 = 66`；`char(66)` 是 `'B'`；`'C' - 'A'` 是 `67 - 65 = 2`。运行结果：

```
A
66
B
2
```

“字符减字符得到距离”是oj习题中一定会用到的技巧，第 2 章还会用它处理字母表问题。书写字符字面量用单引号 `'A'`；双引号 `"A"` 是字符串，两者不是一回事。

1.3.5 布尔型 bool

`bool` 只有两个值：`true` 和 `false`。与整数互换的规则是：`true` 转为整数是 1，`false` 是 0；反过来，**任何非零整数**赋给 `bool` 都是 `true`，只有 0 是 `false`。

示例 3-3：bool 与整数的互转规则。

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

int main()
{
    bool a = true;   // 显式的真
    bool b = 42;     // 非零整数 → true
    bool c = 0;      // 只有 0 是 false
    cout << a << ' ' << b << ' ' << c << '\n';   // 输出 1 1 0
    return 0;
}
```

算法竞赛中 `bool` 最常见的用途是标记（visited、flag），常常配合数组使用。

1.4 溢出：合理选择数据范围的原因

1.4.1 什么是溢出

整数类型的存储空间是固定的，当运算结果超出类型能表示的范围时，就发生溢出。这里必须区分两种情况：

- **有符号整数溢出是未定义行为**。例如计算 `INT_MAX + 1` 后，C++ 不保证得到某个固定结果，开启优化后甚至可能出现更意外的行为；
- **无符号整数按模运算回绕**。例如 `unsigned int` 的最大值再加 1，会确定地回到 0。

因此，竞赛代码不能依赖 `int` “从最大值绕到最小值”。需要更大结果时，应在运算发生前改用更宽的类型：

示例 4-1：让加法在 `long long` 里完成。

```cpp
#include <climits>
#include <iostream>
```

```cpp
using namespace std;

int main()
{
    int x = INT_MAX;              // int 能表示的最大值 2147483647
    long long safe = 1LL * x + 1; // 先转成 long long 再加
    cout << safe << '\n';
    return 0;
}
```

演算：x 是 2147483647；1LL * x 把它先提升成 long long；加 1 得到 2147483648，超出了 int 上限但在 long long 中安然无恙。运行结果：

```
2147483648
```

1.4.2 更隐蔽的：中间结果溢出

一个常见误区是"把结果存进 long long 就安全了"。看下面的写法：

示例 4-2：int 乘 int，赋值给 long long 仍然溢出。

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

int main()
{
    int a = 100000, b = 100000;
    long long wrong = a * b;        // 危险：乘法先在 int 里做
    long long right = 1LL * a * b; // 正确：先提升成 long long 再乘
    cout << wrong << '\n';
    cout << right << '\n';
    return 0;
}
```

运行结果（g++ -O2）：

```
1410065408
10000000000
```

我们来逐步模拟一下：

1. a * b 的两个操作数都是 int，于是乘法**在 int 的世界里完成**：100000 × 100000 = 10000000000，远远超过 int 上限 2147483647，溢出已经实实在在发生了（未定义行为）；
2. 溢出的残值（本环境下是 1410065408）被赋给左边的 long long——但错误已经铸成，long long 再大也只能忠实收下这个残值；

3. `1LL * a` 先得到一个 `long long`，再乘 `b` 时整个表达式按 `long long` 运算，`10000000000` 被完整保留。

总结：**表达式的类型由参与运算的操作数决定，与赋值给谁无关。** 防止中间溢出的唯一办法，是在运算发生之前完成类型提升。

算法竞赛中有一条朴素而有效的经验：看数据范围时，**拿不准就用 `long long`**。

1.4.3 数据范围 → 类型选择

比赛中看题的第一步永远是**读数据范围**，然后对照下表选类型：

答案与中间运算的最大值	选择的类型
不超过 $\pm 2.1\times10^{9}$	`int`
不超过 $\pm 9.2\times10^{18}$	`long long`
更大	`__int128`（见 1.4.4）；再大需要手写高精度，本系列不展开

判断依据是“**答案与中间运算可能到达的最大值**”，而不只是输入的范围。比如本次训练中有一道求和题：输入的 `n` 本身不超过 10^6，用 `int` 绰绰有余；但累加过程中答案约 3.3×10^{17}，而且单独一项 `i²` 最高就有 10^{12}——两者都远超 `int`，必须全程使用 `long long`。

拿不准时先估算最大值；来不及精算且范围允许时，可以保守使用 `long long`，但仍要检查乘法等中间结果是否可能超过它。

1.4.4 【拓展】__int128

g++ 内置了 128 位整数类型 `__int128`，可表示约 $\pm 1.7\times10^{38}$ 的整数，作为乘法可能溢出 `long long` 时的中间计算工具非常顺手。需要注意它没有配套的 `scanf`/`printf` 格式符，读写需要自行转换。初学阶段只需知道它的存在，具体用法在需要时再查即可。

1.5 类型转换

1.5.1 隐式转换

混合类型参与运算时，编译器会自动完成类型转换，规则有三条：

1. **小类型向大类型转换**。`int` 与 `long long` 混合运算，`int` 先转为 `long long`；整型与浮点型混合运算，整型先转为浮点型。1.4.2 节“表达式前乘 `1LL`”利用的正是这条规则。
2. **整型提升**。`char`、`bool`、`short` 参与任何算术运算之前，先提升为 `int`。因此 `'A' + 1` 的类型是 `int`，值为 `66`；两个 `char` 相加，结果也是 `int`。
3. **有符号与无符号混合时要格外小心**。在常见的 `int` 与 `unsigned int` 混合比较中，`-1` 会转成一个很大的无符号数，导致 `-1 < 1u` 为 `false`。初学阶段不必背完整规则，先做到不主动混用即可。

1.5.2 强制转换

需要主动转换类型时，使用强制转换运算符，有两种等价写法：

```
(int)x              // C 风格 （最常用同时最显然）
static_cast<int>(x) // C++ 风格 （不推荐在算法竞赛中使用）
```

示例 5-1：两种写法效果相同。

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

int main()
{
    cout << static_cast<int>(3.9) << ' ' << (int)3.9 << '\n';   // 输出 3 3
    cout << static_cast<double>(7) / 2 << '\n';                 // 输出 3.5
    return 0;
}
```

在本节这种普通数值转换中，两者会得到相同结果。大部分情况下请选择C风格的类型转换，这是经验之谈。

1.5.3 三个经典场景

示例 5-2：整数除法、截断与四舍五入。

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

int main()
{
    // 场景一：整数除法转浮点
    cout << 7 / 2 << '\n';            // 输出 3：整数除法直接舍去小数
    cout << 7.0 / 2 << '\n';          // 输出 3.5：至少一个操作数是浮点
    cout << (double)(7 / 2) << '\n';  // 输出 3：先算完 7/2=3，再转已经晚了

    // 场景二：浮点转整型的截断规则
    cout << (int)3.9 << ' ' << (int)-3.9 << '\n';   // 输出 3 -3：向零取整

    // 场景三：四舍五入
    cout << round(3.4) << ' ' << round(3.5) << ' ' << round(-3.5) << '\n';  // 输出 3 4 -4
    return 0;
}
```

三个场景的演算要点：

- **场景一**：`7 / 2` 中两个操作数都是 `int`，结果也是 `int`（3.5 的小数被直接舍去，不是四舍五入）；`7.0 / 2` 中 `7.0` 是 `double`，`2` 被自动提升，`3.5` 完整保留；而 `(double)(7 / 2)` 的括号让除法**先**完成，转换成 `double` 的只是结果 `3`。

- **场景二**：浮点转 `int` 一律**向零取整**（直接砍掉小数部分），对负数也是如此：`-3.9` 变成 `-3`，而不是数学上的向下取整 `-4`。需要真正的向下取整，请使用 `<cmath>` 中的 `floor`。
- **场景三**：`round` 按"四舍五入到最近整数，恰好一半时远离零"处理，所以 `round(3.5)` 是 `4`、`round(-3.5)` 是 `-4`。对非负浮点数，`(int)(x + 0.5)` 是等价的常用技巧。

1.6 输入与输出

1.6.1 默认路线：cin / cout

C++ 程序有两套输入输出工具，本讲义的默认路线是 **`cin` / `cout`**（头文件 `<iostream>`）：它们自动识别变量类型，不用记格式符，对初学者最友好。

另一套是 C 风格的 `scanf` / `printf`（头文件 `<cstdio>`），用格式符精确控制读写。**什么时候需要它？** 当你要精确控制输出格式时——比如保留几位小数、不足几位补前导零——`printf` 的格式串往往一行搞定，`cin`/`cout` 则需要额外的工具（1.6.5、1.6.6 会给出两套对照写法）。两套工具都会讲，建议先以 `cin`/`cout` 为主练熟，再用 `printf` 补齐格式控制。

初学阶段最稳妥的规则是：**同一个程序只选一套**，不要混用。混用本身不是语法错误，但会让缓冲与输出顺序变得难以理解；一旦使用 `ios::sync_with_stdio(false)` 解除同步（进阶写法，现在不需要用），就更不应再混用。

1.6.2 读入数据

`cin` 用 `>>` 把输入读进变量，一次可以连读多个：

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

int main()
{
    int a;
    long long b;
    double c;
    cin >> a >> b >> c;   // 依次读入，用空格、换行或 Tab 分隔都可以
    cout << a << ' ' << b << ' ' << c << '\n';
    return 0;
}
```

示例输入与输出：

```
（输入）3 123456789012 2.5
（输出）3 123456789012 2.5
```

`cin` 遇到空白字符就切分，所以输入写成 `3␣123456789012␣2.5`、一行一个数、或者 Tab 分隔，效果完全一样。输入时程序会**停在 `cin` 处等待**，你在终端敲完一行按回车，它才继续往下走。

如果题目用 C 风格，对应写法是 `scanf("%d%lld%lf", &a, &b, &c);`——格式符与变量一一对应，`%d` 配 `int`、`%lld` 配 `long long`、`%lf` 配 `double`。它最大的特点（也是最大的坑）是**严格按你写的格式串匹配输

入：格式串里写什么，输入就必须原样出现什么，少一个字符都匹配不上。竞赛输入绝大多数是空白分隔的数字，所以 `cin` 连写是最省心的默认路线；格式串主要在输出侧发挥威力。

1.6.3 读入字符的坑（scanf 路线）

默认的 `cin >> ch` 会**自动跳过空白字符**，读字符没有坑。但如果走 `scanf` 路线要注意：`%c` **不跳过空白字符**，上一行末尾残留的换行会被它读走，而不是你想要的字符。解决办法是读字符统一写成 `scanf(" %c", &ch)`——格式串里 `%c` 前面的那个空格表示"先跳过所有空白"。更多字符处理技巧在第 4 章字符串中展开。

1.6.4 输出：普通输出与转义字符

`cout` 用 `<<` 依次拼接要打印的内容，字符串原样输出，变量的值自动替换：

```
int a = 7;
long long sum = 123456789012;
cout << "a = " << a << ", sum = " << sum << "\n";
```

输出：

```
a = 7, sum = 123456789012
```

换行用 `'\n'`（`cout` 也可以用 `endl`，它会强制刷新输出缓冲、明显拖慢速度，竞赛中统一写 `'\n'`）。字符串里以反斜杠 `\` 开头的字符序列称为**转义字符**，用来表示无法直接书写的特殊字符：

转义字符	含义
`\n`	换行
`\t`	水平制表（Tab）
`\\`	反斜杠本身
`\"`	双引号
`\'`	单引号
`\0`	空字符，字符串的结束标志（第 4 章会用到）

容易出错的是后三个：要输出一个反斜杠必须写 `\\`，要在字符串中输出双引号必须写 `\"`，否则编译器会把它们当成转义序列的开始，导致格式串解析混乱。

示例 6-1：转义字符的输出效果。

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

int main()
{
    cout << "line1\nline2\n";
```

```
    cout << "a\tb\n";
    cout << "C:\\Users\\name\n";
    cout << "She said: \"Hello\"\n";
    return 0;
}
```

运行结果（第二行中 a 与 b 之间是一个制表位）：

```
line1
line2
a   b
C:\Users\name
She said: "Hello"
```

> 小提示：竞赛题的输出一般是数字、英文和标点，不用中文；如果自行尝试输出中文，Windows 终端可能显示乱码，那是终端编码问题，与程序正确性无关，评测系统不受影响。

1.6.5 定点小数：保留几位小数

输出浮点数经常需要指定位数，这是 printf 的主场，但 cin/cout 配合 <iomanip> 同样能做到：

```
printf("%.2f\n", x);                    // C 风格：保留两位小数
cout << fixed << setprecision(2) << x;  // C++ 风格：需头文件 <iomanip>
```

- %.2f 按**四舍五入**保留两位小数，%.nf 即保留 n 位；
- fixed 表示"固定小数点"输出（总是带小数部分），setprecision(2) 表示保留 2 位小数；两者**一旦设置，对之后所有的浮点输出持续生效**。

示例 6-2：两种写法的对照。

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

int main()
{
    double pi = 3.1415926;
    double x = 2.5;

    cout << pi << '\n';                                 // 默认 6 位有效数字
    cout << fixed << setprecision(2) << pi << '\n';     // 四舍五入保留两位
    cout << fixed << setprecision(2) << x << '\n';      // 2.5 → 2.50
    printf("%.2f\n", pi);                               // printf 写法
    return 0;
}
```

运行结果：

```
3.14159
3.14
2.50
3.14
```

演算要点：直接 `cout << pi` 默认只显示 **6 位有效数字**（`3.14159`），这不是"保留六位小数"，而是"从左边第一个非零数字算起一共六位"；`fixed << setprecision(2)` 之后才是我们常说的"保留两位小数"，并且 `2.5` 会补齐成 `2.50`。

1.6.6 前导补零：宽度与填充

有些题目要求输出定宽格式，比如时间要写成 `09:05` 而不是 `9:5`，名次要输出 `007`。两套写法对照：

- `printf("%02d", x)`：宽度至少 2 位，不足时在**左侧补 0**；
- `cout << setw(2) << setfill('0') << x`：`setw` 设置宽度，`setfill` 设置填充字符。

示例 6-3：时间与定宽编号。

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

int main()
{
    int hour = 9, minute = 5;
    printf("%02d:%02d\n", hour, minute);                 // 09:05
    cout << setw(2) << setfill('0') << hour << ':'
         << setw(2) << setfill('0') << minute << '\n';   // 09:05

    int rank = 7;
    printf("%03d\n", rank);                              // 007
    cout << setw(3) << setfill('0') << rank << '\n';     // 007
    return 0;
}
```

运行结果：

```
09:05
09:05
007
007
```

一个关键细节：`setw` **只对紧接着的一次输出有效**，所以 `hour` 和 `minute` 前面要各写一次；`setfill` 设置后持续生效。与之相关的还有对齐宽度 `%5d`（至少占 5 个字符宽、右对齐，`%-5d` 为左对齐），打印表格类对齐输出

时非常有用。

1.6.7 scanf / printf 格式符速查

格式符	用途
`%d`	`int`
`%lld`	`long long`
`%u` / `%llu`	`unsigned int` / `unsigned long long`
`%f`	输出 `float`/`double`；`scanf` 中仅用于 `float`
`%lf`	`scanf` 读入 `double`
`%Lf`	`long double` 读写
`%c`	单个字符
`%s`	字符串（字符数组，读入时不加 `&`）
`%nd`	至少占 n 个字符宽，右对齐（左对齐用 `%-nd`）
`%0nd`	宽度 n，不足时左侧补 0
`%.nf`	保留 n 位小数输出
`%%`	输出百分号本身

`cin`/`cout` 不需要格式符，类型自动匹配。再次提醒三个易错点：`scanf` 读普通变量必须加 `&`（`scanf("%d", &a);`，`cin` 不需要）；读入 `double` 必须用 `%lf` 而不是 `%f`；`%c` 不跳过空白（1.6.3）。

1.7 常量：const 与 #define

1.7.1 为什么需要常量

程序中不应出现来路不明的“魔法数字”——一个没有名字的字面量，读者无法知道它代表什么。把有固定含义的值命名为常量，既让代码自解释，也让修改只需改动一处。

C++ 中定义常量的两种方式：`const` 和 `#define`，算法竞赛代码中两种都很常见，都需要掌握。

1.7.2 const 定义常量

```
const int MAXN = 100005;
```

`const` 定义的常量有明确的类型，遵守普通变量的作用域规则，定义之后不允许再赋值（再赋值会直接编译报错）。它在本质上是一个“只读变量”。

1.7.3 #define 宏定义

```
#define MAXN 100005
```

`#define` 是预处理指令，在编译之前把代码中的名字**原样替换成后面的文本**。它没有类型、没有作用域、也不占内存。两个使用要点：行尾**不加分号**；替换是纯粹的文本操作，因此宏的内容若涉及运算，参与表达式时可能因优先级产生意外。

示例 7-1：const、#define，以及宏的经典陷阱。

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

const int MAXN = 100005;
#define MOD 1000000007
#define A 1 + 2    // 反例：内容没有整体加括号

int main()
{
    cout << MAXN << ' ' << MOD << '\n';    // 输出 100005 1000000007
    cout << 2 * A << '\n';                 // 输出 4，而不是 6！
    return 0;
}
```

演算陷阱所在：`2 * A` 被原样替换成 `2 * 1 + 2`，按乘法优先级先算 `2 * 1`，再加 2，得到 `4`；而你可能期望的是 `2 * (1 + 2) = 6`。稳妥做法是宏的内容涉及运算时整体加括号：`#define A (1 + 2)`。

> 本系列的使用习惯：数组容量等简单竞赛常量用 `#define`，需要类型与作用域语义时用 `const`，见 1.7.5 与第 3 章。

1.7.4 【拓展】眼熟即可：INF = 0x3f3f3f3f

在之后的数组、图论章节里，你会经常看到下面这个常量：

```
const int INF = 0x3f3f3f3f;
```

它表示“无穷大”，用来标记“还没算出来”或“走不到”。`0x3f3f3f3f` 是一个十六进制写法（十六进制本身在第 6 章位运算中系统讲解），数值约 10.6 亿。现在**不需要理解它为什么长这样**，更不必使用；等学到相关章节再回来看，理由会水到渠成。

1.7.5 【拓展】constexpr

C++11 引入了 `constexpr`，表示“编译期就能确定值的常量”，约束比 `const` 更强，常用于数组长度等必须是编译期常量的场合。初学阶段遇到时能认识即可。

1.8 作用域与全局变量

1.8.1 三层作用域

变量的**作用域**是它从定义到失效的有效范围，C++ 中有三层：

1. **全局变量**：定义在所有函数之外，整个程序中都可以访问；
2. **局部变量**：定义在函数内（如 `main` 函数内），只在该函数中有效；
3. **块级变量**：定义在某一对花括号内（如 `for` 循环体、`if` 分支内），离开这对花括号立即失效。

一个典型例子是循环变量：`for (int i = 0; ...)` 中定义的 `i` 只在循环体内有效，循环结束后即失效，因此两个并列的 `for` 循环可以各自定义自己的 `i` 而互不干扰。

1.8.2 同名遮蔽

内层作用域可以定义与外层同名的变量，此时内层变量**遮蔽**外层变量——在内层作用域中访问该名字，看到的是内层的值；离开内层后，外层变量恢复可见。

示例 8-1：三层作用域与遮蔽。

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

int g = 10;   // 全局变量

int main()
{
    cout << g << '\n';   // 输出 10：此时只有全局的 g
    int g = 20;          // 局部 g 遮蔽了全局 g
    cout << g << '\n';   // 输出 20
    {
        int g = 30;      // 块级 g 又遮蔽了局部 g
        cout << g << '\n';   // 输出 30
    }                    // 块级 g 在这里失效
    cout << g << '\n';   // 输出 20：局部 g 恢复可见
    return 0;
}
```

遮蔽规则保证了局部命名的自由，但也容易写出“以为改的是全局、实际改的是局部”的错觉代码。**建议避免重名变量**；如果全局变量就叫 `g`，局部就不要再用这个名字。

1.8.3 全局变量自动清零

回忆 1.2.2 节“局部变量不初始化是未定义行为”。全局变量（以及所有函数外定义的变量）则不同：**不加初始化时，它们会被自动清零**——数值型为 0，`bool` 为 `false`，`char` 为 `'\0'`。

这条规则最实用的场景是数组：把数组定义为全局变量，所有元素自动为 0，免去了手动清零的循环；标记数组、计数数组在算法竞赛中几乎都这样使用。至于“大数组为什么建议开全局”（函数栈空间的限制），将在第 3 章数组中展开。

1.9 常见错误对照

评测系统对一份提交只会给出有限几种结果，先记住它们的名字，后面每一章都会用到：

结果	全称	含义
AC	Accepted	通过，输出全部正确
WA	Wrong Answer	答案错误：能跑完，但输出不对
RE	Runtime Error	运行崩溃：越界、非法内存访问等
CE	Compile Error	编译失败：语法有错，根本没生成程序
TLE	Time Limit Exceeded	超时：程序跑得太慢
MLE	Memory Limit Exceeded	超内存

下面按新生最高频的三个错误逐一对照。

错误一（CE）：中文标点混入

示例 9-1：使用了中文分号的代码。

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

int main()
{
    int a = 1;        // ← 这个分号是全角的
    cout << a << '\n';
    return 0;
}
```

现象：编译失败，错误信息指向的行号和内容看起来莫名其妙，往往和真正的错误位置差了好几行。

原因：从聊天软件、网页或全角输入法状态下复制的代码混入了全角字符（中文分号 ； 、中文引号 “” 、中文括号 （） 等）。编译器只认识半角符号，全角字符会把后续的解析全部打乱。本例编译时，g++ 会报出一串完全看不懂的错误，例如 `error: unable to find numeric literal operator 'operator""\U0000ff1b'`——那个 `\U0000ff1b` 其实就是中文分号的编码；如果全角分号跟在类型后面（如 `int a；`），报的则是 `error: expected initializer before 'cout'`，指向的甚至是**下一行**。具体措辞随编译器版本不同（旧版本常报 `stray ‘\343’ in program`），但共同点是：报错内容看不懂、指的位置不靠谱。

修复：在报错位置附近仔细检查标点是否为全角，逐个重打。预防办法：永远在 VS Code 里手敲代码，不要从聊天记录里复制。

错误二（RE / 未定义行为）：scanf 漏写 &

示例 9-2：`scanf` 漏写 `&` 的代码。

```
#include <cstdio>
int main()
{
    int a = 0;
```

```
    scanf("%d", a);   // 漏了 &：把 a 的值当成了地址
    printf("%d\n", a);
    return 0;
}
```

现象：开启 `-Wall` 编译时，编译器会警告 `format '%d' expects argument of type 'int *', but argument 2 has type 'int'`。如果无视警告强行运行，属于未定义行为：在本文的验证环境中，输入一个数后它直接崩溃（RE）；换种环境它也可能"看起来没事"地跑出错误数据——不保证每次都一样。

原因：`scanf` 需要的是变量的**地址**（往哪里写），漏写 `&` 传入的却是变量的**值**（当前内容）。`scanf` 会把这个值当作内存地址使用，等于让程序往一个来路不明的位置写数据。

修复：`scanf` 中所有普通变量前加 `&`；唯一的例外是字符数组，数组名本身就是地址，不需要加（第 4 章会见到）。用 `cin` 则完全没有这个坑，这也是默认路线推荐 `cin` 的原因之一。

错误三（WA）：int 溢出

示例 9-3：大数累加仍用 `int` 的代码。

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

int main()
{
    int n;
    cin >> n;
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i + i * i;   // i*i 与累加都可能超出 int 的范围
    }
    cout << sum << '\n';
    return 0;
}
```

现象：输入 `3`，程序正确输出 `20`；输入 `100000`，它输出 `-1963344448`（溢出是未定义行为，这个残值换台机器可能不同，但总之是错的）——而正确答案是 `333343333400000`。程序逻辑一个字符都没错，小样例全对，大数据全错，全程没有任何报错，评测结果就是 WA。

原因：答案与中间运算都超出了 `int` 的表示范围（约 $\pm 2.1\times10^9$），触发 1.4 节讲的有符号整数溢出。它属于未定义行为，残值是多少取决于编译器和平台，不能指望"绕回负数"这种固定规律帮你定位问题。

修复：按 1.4.3 的对照表把相关变量升级为 `long long`，并且让**每一次可能超界的运算**都在 `long long` 中进行。修好的完整程序见 1.10 的示例 10-1（它正是 A 题「逐级加训」的解法）：改完后输入 `100000`，输出 `333343333400000`，与正确答案一致。

排错口诀

- **CE**：从编译信息的第一条错误看起，看不懂先查标点；
- **RE**：先查数组越界，再查 `scanf` 的 `&`；

- **WA**：先查数据范围与变量类型是否匹配，再怀疑算法逻辑。

1.10 本章小结与配套习题

本章知识路线

```
程序骨架：#include → using namespace std → int main()
    ↓
变量 = 类型 + 名字，定义后立即初始化
    ↓
默认 int / double / char / bool，可能超界就 long long
    ↓
溢出看“中间运算”，不是只看输入——先乘 1LL 再算
    ↓
cin >> 读、cout << 写；精确格式用 printf（%.nf、%0nd）
    ↓
常量起名字，作用域想清楚，写完对照 CE / RE / WA 三查
```

必须记牢的关键点：

- `int` 约 $\pm 2.1\times10^{9}$、`long long` 约 $\pm 9.2\times10^{18}$，看题先读数据范围；中间运算可能超界就先乘 `1LL`，别让 `int` 乘法先发生；
- 整数除法 `7 / 2` 得 `3`：要小数，先让某个操作数变成浮点；保留 n 位小数用 `fixed << setprecision(n)` 或 `%.nf`，定宽补零用 `%0nd`；
- `cin >>` 读、`cout <<` 写是默认路线；走 `scanf` 记得加 `&`、`double` 用 `%lf`、读字符用 `" %c"`；
- 转义字符 `\n \t \\ \"`；字符本质是编码（`'A'` 是 65），`char` 与 `int` 可以互相换算；局部变量不初始化是垃圾值，全局变量自动清零；
- CE 先查标点、RE 先查越界与 `scanf` 的 `&`、WA 先查数据范围与变量类型。

配套习题

本次训练题目已发布在 SYUCTOJ：2026 SYUCTACM 新生训练 第一次，请注册账号后加入比赛。**先读数据范围，再选类型，最后写代码。**

【基础练习】

需要的技能	对应小节
连读多个数据、原样输出	1.6.2
输出表格/多行文本（`\n`、`\t`、`\\`、`\"`）	1.6.4
保留两位小数（`fixed << setprecision(2)` 或 `%.2f`）	1.6.5
不足位补前导零（`%0nd` 或 `setw + setfill`）	1.6.6
整数商与小数商（`7 / 2` 与 `7.0 / 2`）	1.5.3
字符与编码互换（`'A' + 1`、`'C' - 'A'`）	1.3.4
答案或中间结果超界（`long long`、先乘 `1LL`）	1.4