c++□□

c++□□

- □□□___
  - □□__
  - □□□□□_____
  - □□__
  - □□□□____





第一章

开始

本书用的是 ***visualstdio***软件进行编写代码与演示的

1.开头语句与固定格式

```
#include <iostream>
using namespace std;
int main()
{
  return 0;
}
```

如你所见，上面的这一段代码就是每一个 c++程序必须写的代码，下面我们将会一一解释其中的含义。

#include 这一行代码是用来引用头文件的，就像想要打扫房间就必须有工具一样，后面的: iostream就是c++所需的工具。

using namespace std;这一段先不用理解，后面的章节会讲到，先照抄就行。

int main () 这一段是最重要的，这个 main 函数里包括了程序开始时最先运行的代码，如果没有这一行，编译器会报错。代码在大括号中写。

return 0; 这一段是返回值，也是记住就行，后面会讲。

2.关于程序运行的顺序

几乎所有编译器的运行顺序是从上到下，从左到右。

在 c++中，每一个在函数内的语句结束后都要加分号。

第一章

输入输出流

在我们之前的章节里讲过，#include是用来引用头文件 iostream的，而我们今天讲的输入输出流就包括在了iostream里。

最基本的语法如下

```
//输出流
cout<<"Hello World"<<endl;
//输入流
int a;
cin>>a;
```

这就是最简单的一个输入输出流。

cout用来把后面的内容输出到控制台， cin用来把控制台用户输入的信息给到程序里。

1.cout

这里的双引号用来告诉编译器这是文字，不是函数名或变量名。

<<这个符号就像压缩机，把这一行的代码从右向左压成一个包，发给控制台。

endl的作用是换行。

2.cin

>>这个符号就跟 cout里的 <<作用相反。把控制台里用户输入的东西打包发给程序。

后面接的只能是变量名。

意思是把控制台内的用户输入的数字或字符传给程序内的变量，实现人机接口。

3.应用

现在让我们结合上面写一个简单的代码吧

```
#include <iostream>
using namespace std;
int main()
  {
  cout<<"Hello World"<<endl;
  int a;
  cin>>a;
  cout<<a<<endl;
  return 0;
  }
```

第一章

变量

C++ 变量教程

变量是程序中存储数据的具名内存单元。本教程从声明与初始化讲起，覆盖类型系统、初始化方式、作用域与生命周期、类型推导、存储类说明符，以及现代 C++ 的变量特性。

一、变量的声明、定义与初始化

1.1 三者的区别

概念 含义 示例 声明 告诉编译器 "有这么一个名字和类型 "，不分配内存 extern int x; 定义分配内存，可同时初始化（一个变量只能定义一次） int x = 5; 初始化 定义时赋初值 int x{5};

```
extern int g;       // 声明（通常在头文件）
int g = 10;         // 定义（在某个 .cpp 中）

int main() {
    int a;          // 定义但未初始化 → 值不确定（局部变量）
    int b = 5;      // 定义并拷贝初始化
    return 0;
}
```

⚠ 未初始化的局部变量值是不确定的（俗称 "垃圾值 "），读取它是未定义行为。全局 / 静态变量则会被零初始化。

1.2 初始化方式（ C++11 起）

C++ 提供多种初始化语法，推荐使用花括号初始化：

```
int a = 5;          // 拷贝初始化
int b(5);           // 直接初始化
int c{5};           // □□□□□□□□□□□□□□□ □□
int d = {5};        // 拷贝列表初始化

double e{5};        // 允许: 5 → 5.0
// int f{5.7};      // □ □□□□□□□□□□□□□□
int g(5.7);         // 允许但会截断为 5（不推荐）
```

花括号初始化的优势：

1. 禁止窄化转换（如 double → int、long → char），在编译期捕获精度丢失。
2. 适用于所有类型，语法统一。
3. 避免 "最令人烦恼的解析 "问题：

```
int x(10);          // 变量
int y();            // ⚠◌ 这是函数声明，不是变量！
int z{};            // 明确是变量，值为 0
```

默认初始化陷阱：

```
int a;              // 不确定值（局部变量）
int b{};            // 0
int c = int();      // 0
```

现代 C++ 核心准则建议：始终初始化变量，优先 {}。

二、基本数据类型

2.1 整型

类型 典型大小 范围（典型） short 2 字节 -32768 ~ 32767 int 4 字节 ≈ ±21 亿 long 4/8 字节 平台相关 long long 8 字节 ≈ ±9.2×10¹⁸ unsigned 各类型 同上 0 到 2^n-1

```
int i = 42;
unsigned int u = 42u;
long long big = 9'000'000'000LL;   // C++14 数字分隔符，提高可读性
unsigned char c = 255;
```

要点：

· int 至少 16 位，实际大小依赖平台；需要精确宽度用 中的 int32_t、uint64_t。 · 数字后缀： u（无符号）、l/ll（长整型）、 f（浮点）。

```
#include <cstdint>
int32_t a = 100;       // 精确 32 位有符号
uint64_t b = 1ULL << 40;  // 精确 64 位无符号
```

2.2 浮点型

类型 典型大小 精度 float 4 字节 约 ~~67~~ 位有效数字 ~~double 8 字节 约 1516~~ 位 long double 8/16 字节 平台相关

```
float f = 3.14f;
double d = 3.141592653589793;
long double ld = 3.14159265358979323 8L;
```

⚠ 浮点数不要用 == 比较，用误差范围（见第五节）。

2.3 字符与布尔

```
char ch = 'A';            // 单个字符，1 字节
bool flag = true;         // true / false
```

char 是有符号还是无符号取决于平台；需要明确时用 signed char / unsigned char。

2.4 sizeof 运算符

```
#include <iostream>
std::cout << sizeof(int) << "\n";       // 字节数
std::cout << sizeof(long long) << "\n";
std::cout << sizeof(std::string) << "\n";  // 对象大小，非内容长度
```

三、 const 与 constexpr

3.1 const 常量

```
const int MAX = 100;
// MAX = 200;   // □ □□□□

const double PI = 3.14159;
const std::string NAME = "C++";
```

const 与指针的组合（经典难点）：

```
int x = 10;
const int* p1 = &x;         // 指向常量的指针：不能改 *p1，能改 p1
int* const p2 = &x;         // 常量指针：能改 *p2，不能改 p2
const int* const p3 = &x;   // 都不能改
```

读法技巧：从右往左读。 const int* p → "p 是指针，指向 const int"。

3.2 constexpr（C++11）

表示 "编译期常量表达式"，可用于数组大小、模板参数等编译期场景。

```
constexpr int SQUARE(int n) { return n * n; }
constexpr int N = SQUARE(5);   // 编译期计算，N = 25

int arr[SQUARE(3)];            // 数组大小必须是编译期常量
```

const vs constexpr：

const constexpr 保证 运行期不可修改 编译期可求值（更强） 可用于数组大小 仅当初始化为编译期常量 □ 总是 函数 不能修饰为编译期函数 □ □□□□□

能用 constexpr 就用 constexpr，它隐含 const。

3.3 consteval 与 constinit（C++20）

```
consteval int cube(int n) { return n * n * n; }  // 必须在编译期求值

constinit int global = 42;   // 保证静态初始化（无动态初始化开销）
```

四、类型推导：auto 与 decltype

4.1 auto（C++11）

让编译器根据初始化表达式推导类型。

```
auto a = 42;           // int
auto b = 3.14;         // double
auto c = 'x';          // char
auto d = std::string("hi");  // std::string
auto e = 1 + 2.0;      // double
```

必须初始化，且推导时会丢弃顶层 const 和引用：

```
const int ci = 10;
auto x = ci;           // int（const 被丢弃）
auto& r = ci;          // const int&（引用保留 const）

int arr[3] = {1, 2, 3};
auto p = arr;          // int*（数组退化为指针）
```

典型用途：

```
// 迭代器类型冗长
for (auto it = v.begin(); it != v.end(); ++it) { ... }

// 范围 for 中
for (const auto& item : container) { ... }

// 复杂返回类型
auto result = computeSomething();
```

现代 C++ 核心准则：用 auto 避免重复类型名，但不要滥用导致可读性下降（如 auto x = getValue(); 看不出是什么类型时，可考虑显式写出）。

4.2 decltype（C++11）

推导表达式的类型，保留 const 和引用。

```
int i = 0;
const int ci = 0;
decltype(i) a = 5;      // int
decltype(ci) b = 5;     // const int
decltype((i)) c = i;    // int&（带括号表示左值引用）

// 常见用途：函数返回类型
template <typename T, typename U>
auto add(T a, U b) -> decltype(a + b) {   // C++11 尾置返回类型
    return a + b;
}
```

C++14 起函数返回类型可直接 auto 推导，简化上述写法。

4.3 结构化绑定（C++17）

一次性解包 pair、tuple、结构体、数组：

```
#include <map>
#include <tuple>

std::map<std::string, int> ages = {{"Alice", 25}};
for (const auto& [name, age] : ages) {
    std::cout << name << ": " << age << "\n";
}
```

```
auto [x, y] = std::make_pair(1, 2);
```

五、作用域与生命周期

5.1 作用域类型

作用域 说明 示例 块作用域 { } 内 函数体、 if/for 块内的变量 函数作用域 函数参数 void f(int x) 命名空间作用域 命名空间内（全局变量属于全局命名空间） namespace N { int x; } 类作用域 类成员 class C { int x; }; 全局作用域 所有命名空间外 int g;

```
int g = 1;                      // 全局

void f() {
    int a = 2;                  // 函数块
    {
        int b = 3;              // 内层块
        // 此处可访问 a、b、g
    }
    // 此处 b 已销毁
}
```

名字遮蔽（ shadowing）：

```
int x = 1;
void f() {
    int x = 2;        // 遮蔽全局 x
    std::cout << x;  // 2
    std::cout << ::x; // 1（用 :: 访问全局）
}
```

遮蔽易导致 bug，现代准则建议避免用同名变量遮蔽外层。

5.2 生命周期与存储期

存储期 说明 典型对象 自动存储期 进入作用域创建，离开销毁 局部变量 静态存储期 程序运行期间一直存在 全局变量、 static 局部变量 线程存储期 线程存活期间 thread_local 变量 动态存储期 由 new/delete 手动管理 堆对象

六、存储类说明符

6.1 static

在函数内：只初始化一次，生命周期延长到程序结束。

```
void counter() {
    static int count = 0;   // 只初始化一次
    ++count;
    std::cout << count << "\n";
}
// 多次调用依次输出 1, 2, 3...
```

在文件 /命名空间作用域：内部链接，仅当前翻译单元可见。

```
static int filePrivate = 42;   // 其他 .cpp 无法访问
```

在类中：属于类而非对象，所有实例共享。

```
class Widget {
    static int instanceCount;   // 声明
};
int Widget::instanceCount = 0;  // 定义（C++17 可用 inline static 直接在类内定义）
```

6.2 extern

声明在其他翻译单元中定义的变量：

```
extern int sharedVar;   // 声明，定义在别处
```

6.3 thread_local（C++11）

每个线程拥有独立副本：

```
thread_local int tlsCounter = 0;
```

6.4 register（C++17 已废弃）

曾用于建议放入寄存器，现代编译器自动优化，不要使用。

七、引用

引用是变量的别名，必须初始化且不能重新绑定。

```
int a = 10;
int& ref = a;       // ref 是 a 的别名
```

```
ref = 20;           // a 也变成 20
```

常量引用：可绑定临时值，常用于函数参数避免拷贝：

```
void print(const std::string& s) {   // 不拷贝，且不能修改
    std::cout << s << "\n";
}

print("hello");   // 临时 string 可绑定到 const 引用
```

引用 vs 指针：

引用 指针 必须初始化 □ □ □□□□□ □ □ □□□ □ □ □□ 像普通变量 需 * / &

优先用引用，需要 "可空 "或 "重新指向 "时才用指针。

八、类型别名

8.1 using（C++11，推荐）

```
using Age = int;
using StringList = std::vector<std::string>;
using Callback = void(*)(int);

Age a = 25;
StringList names;
```

8.2 typedef（旧式）

```
typedef int Age;
typedef std::vector<std::string> StringList;
```

现代 C++ 推荐 using，语法更清晰，且支持模板别名（ typedef 不支持）。

8.3 模板别名

```
template <typename T>
using Vec = std::vector<T>;

Vec<int> v;   // 等价于 std::vector<int>
```

九、类型转换

9.1 C 风格转换（不推荐）

```
double d = 3.7;
int i = (int)d;    // 难看出意图，且可能掩盖错误
```

9.2 四种具名转换（推荐）

```
// static_cast: 相关类型间的转换（数值转换、向上转型等）
double d = 3.7;
int i = static_cast<int>(d);        // 3（截断）

// dynamic_cast: 多态类型的安全向下转型（运行期检查）
Base* b = new Derived();
Derived* dp = dynamic_cast<Derived*>(b);    // 失败返回 nullptr

// const_cast: 去除/添加 const（谨慎使用）
const int ci = 10;
int* p = const_cast<int*>(&ci);

// reinterpret_cast: 底层重新解释（危险，平台相关）
int x = 42;
char* bytes = reinterpret_cast<char*>(&x);
```

优先 static_cast; reinterpret_cast 和 const_cast 仅在确实需要时使用。

9.3 隐式转换与窄化

```
int i = 3.99;          // 隐式截断为 3（可能告警）
char c = 1000;         // 窄化，值不确定
int j{3.99};           // □ □□□□□□□□□□□□□
```

十、常见陷阱与最佳实践

□ □□□□

错误 说明 int a; 后直接使用 值不确定，未定义行为 if (x = 5) 赋值而非比较 有符号与无符号混用比较 int i = -1; unsigned u = 1; i < u 为 true（i 被转换为无符号） 浮点数 == 比较 精度问题 返回局部变量的引用 /指针悬垂引用 全局变量滥用 难以维护、线程不安全、初始化顺序问题

有符号 /无符号陷阱示例：

```
std::vector<int> v = {1, 2, 3};
for (int i = 0; i < v.size(); ++i)   // ⚠◌ 有符号与 size_t 比较告警
    ;

for (size_t i = 0; i < v.size(); ++i)  // □
    ;
// 或使用范围 for
```

□ □□□□

1. 始终初始化变量，优先用 {}：

```
int count{};
std::string name{};
```

2. 用 const / constexpr 修饰不变量，让编译器帮忙检查。
3. 缩小变量作用域：在需要的地方才声明，而非集中在函数开头。
4. 优先 auto + 范围 for，减少类型重复和越界风险。
5. 避免全局变量；必须用时，放在命名空间中并加 const 或 static。
6. 浮点数比较用误差范围：

```
#include <cmath>
bool nearlyEqual(double a, double b) {
    return std::fabs(a - b) < 1e-9;
}
```

7. 函数参数用 const& 传大对象，避免拷贝。
8. 需要精确宽度时用 （int32_t、uint8_t 等），而非假设 int 大小。

十一、综合示例

一个综合运用变量的示例：

```
#include <iostream>
#include <string>
#include <vector>
#include <cstdint>

// 常量：编译期确定
constexpr double TAX_RATE = 0.08;
constexpr int MAX_ITEMS = 100;

// 类型别名
```

```
using Price = double;
using ItemList = std::vector<std::string>;

// □□□□□□
class Order {
public:
    explicit Order(std::string name) : customer_{std::move(name)} {
        ++totalOrders;
    }

    static int totalOrders;   // □□

private:
    std::string customer_;
};

int Order::totalOrders = 0;   // □□

int main() {
    ItemList items{"apple", "banana", "cherry"};
    Price subtotal{0.0};

    // □□ for + const auto&
    for (const auto& item : items) {
        constexpr Price unitPrice = 2.5;   // □□□□ 2.5 □
        subtotal += unitPrice;
    }

    const Price tax = subtotal * TAX_RATE;
    const Price total = subtotal + tax;

    std::cout << "□□□□" << items.size() << "□□□ " << MAX_ITEMS << "□\n";
    std::cout << "□□□" << subtotal << "\n";
    std::cout << "□□□" << tax << "\n";
    std::cout << "□□□" << total << "\n";

    // □□□□□
    Order o1{"Alice"}, o2{"Bob"};
    std::cout << "□□□□□" << Order::totalOrders << "\n";
```

```
    return 0;
}
```

十二、小结

主题	关键点
声明 vs 定义	定义分配内存，只能一次；声明可多次
初始化	优先 {}，禁止窄化，避免未初始化
类型	固定宽度用 ；浮点勿用 ==
const / constexpr	修饰不变量；能编译期求值用 constexpr
auto / decltype	简化类型，注意 auto 丢弃顶层
const	作用域 尽量缩小，避免遮蔽
static	局部：只初始化一次；全局：内部链接
引用	别名，必须初始化；const& 避免拷贝
类型别名	用 using，支持模板别名
转换	用 static_cast 等具名转换，慎用 reinterpret_cast

变量是程序的基础，理解类型、初始化、作用域和生命周期，才能写出正确、高效、可维护的 C++ 代码。建议结合编译器警告（ -Wall -Wextra）实践，很多变量相关的错误能被编译器提前捕获。

第一章

流程控制

程序由若干条语句组成，程序的执行顺序称为 流程。C++ 提供了三种基本流程结构，以及若干跳转语句来灵活控制程序走向：

- 顺序结构：语句自上而下依次执行。
- 选择结构（分支结构）：根据条件决定执行哪一条分支。
- 循环结构：在满足条件时重复执行某段代码。
- 跳转语句：`break`、`continue`、`goto`、`return`，用于改变流程走向。

一、顺序结构

顺序结构是最基本的流程结构，代码按照书写的先后顺序从上到下逐条执行。

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

int main() {
    int a = 10;
    int b = 20;
    int sum = a + b;          // 先计算
    cout << "sum = " << sum << endl;  // 再输出
    return 0;
}
```

输出：

```
sum = 30
```

二、选择结构

选择结构根据条件表达式的真假，选择性地执行不同的代码块。条件表达式中，**0** 表示假（**false**），非 **0** 表示真（**true**）。

2.1 if 语句

语法：

```
if (条件表达式) {
    // 条件为真时执行的语句
}
```

示例：

```
int score = 85;
if (score >= 60) {
    cout << "及格" << endl;
}
```

2.2 if-else 语句

语法：

```
if (条件表达式) {
    // 条件为真
} else {
    // 条件为假
}
```

示例：

```
int num = 7;
if (num % 2 == 0) {
    cout << num << " 是偶数" << endl;
} else {
    cout << num << " 是奇数" << endl;
}
```

2.3 if-else if-else 多分支

语法：

```
if (条件1) {
    // 条件1成立
} else if (条件2) {
    // 条件2成立
} else if (条件3) {
    // 条件3成立
} else {
    // 以上都不成立
}
```

示例（成绩等级判定）：

```
int score = 85;
if (score >= 90) {
    cout << "优秀" << endl;
} else if (score >= 80) {
    cout << "良好" << endl;
} else if (score >= 60) {
    cout << "及格" << endl;
} else {
    cout << "不及格" << endl;
}
```

> 注意：多分支按顺序自上而下判断，一旦某个条件成立，后面的分支将不再判断。因此条件的顺序很重要（例如先判断 `>= 90` 再判断 `>= 80`）。

2.4 嵌套 if

在一个 `if` 内部再写 `if`，即嵌套分支。示例：判断一个数是否为三个数中的最大值。

```
int a = 3, b = 8, c = 5;
if (a >= b) {
    if (a >= c) {
        cout << "最大值是 " << a << endl;
    } else {
```

```
        cout << "最大值是 " << c << endl;
    }
} else {
    if (b >= c) {
        cout << "最大值是 " << b << endl;
    } else {
        cout << "最大值是 " << c << endl;
    }
}
```

> 悬挂 **else** 问题：`else` 总是与它前面最近的、尚未配对的 `if` 配对。建议始终使用花括号 `{}` 明确代码块范围，避免歧义。

2.5 switch 语句

`switch` 适合根据一个整型或枚举表达式的 多个固定取值 进行分支。

语法：

```
switch (表达式) {
    case 常量1:
        // 语句
        break;
    case 常量2:
        // 语句
        break;
    default:
        // 其他情况
        break;
}
```

示例（根据数字输出星期）：

```
int day = 3;
switch (day) {
    case 1: cout << "星期一" << endl; break;
    case 2: cout << "星期二" << endl; break;
    case 3: cout << "星期三" << endl; break;
    case 4: cout << "星期四" << endl; break;
```

```
    case 5: cout << "星期五" << endl; break;
    case 6: cout << "星期六" << endl; break;
    case 7: cout << "星期日" << endl; break;
    default: cout << "非法输入" << endl; break;
}
```

switch 的要点:

- `switch` 的表达式只能是整型（ `int`、`char`、`enum` 等）或可隐式转换为整型的类型，不能是浮点数或字符串 。
- `case` 后面必须是 常量表达式 ，各 `case` 的值不能重复。
- `break` 的作用是跳出 **switch**。如果省略 `break`，程序会继续执行下一个 `case` 的语句，这称为 **case** 穿透（ **fall-through**）。有时可利用穿透实现多个值共享同一段逻辑:

```
char grade = 'B';
switch (grade) {
    case 'A':
    case 'B':
        cout << "良好及以上" << endl;  // A 和 B 都执行这里
        break;
    case 'C':
        cout << "及格" << endl;
        break;
    default:
        cout << "其他" << endl;
}
```

2.6 三目运算符 ?:

三目运算符是一个简洁的条件表达式:

```
结果 = 条件 ? 值1 : 值2;
```

条件为真取 `值1`，否则取 `值2`。示例:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;   // max = 20
cout << "较大值: " << max << endl;
```

if 与 switch 对比

特性	if / else if	switch
判断类型	任意布尔条件、范围判断	整型 /枚举的相等比较
表达式	可以是关系、逻辑表达式	只能是比较具体的值
可读性	范围判断清晰	多固定值分支更清晰
穿透行为	无	有（需 `break` 控制）

三、循环结构

循环结构用于重复执行某段代码，直到条件不再满足。 C++ 有三种循环语句： `for`、`while`、`do-while`。

3.1 for 循环

适合 已知循环次数 的场景。

语法：

```
for (初始化; 循环条件; 迭代更新) {
    // 循环体
}
```

执行流程：初始化 → 判断条件 → 执行循环体 → 迭代更新 → 再判断条件 ……

示例（求 1 到 100 的和）：

```
int sum = 0;
for (int i = 1; i <= 100; i++) {
    sum += i;
}
cout << "1+2+...+100 = " << sum << endl;  // 5050
```

3.2 while 循环

适合 循环次数不确定、依赖条件 的场景。先判断条件，再执行循环体。

语法：

```
while (循环条件) {
    // 循环体
}
```

示例（求一个整数的各位数字之和）：

```
int n = 12345;
int sum = 0;
while (n > 0) {
    sum += n % 10;   // 取个位
    n /= 10;         // 去掉个位
}
cout << "各位之和: " << sum << endl;  // 15
```

3.3 do-while 循环

先执行一次循环体，再判断条件。**无论条件是否成立，循环体至少执行一次。**

语法：

```cpp
do {
    // 循环体
} while (循环条件);   // 注意结尾的分号
```

示例（至少执行一次的输入校验）：

```
int input;
do {
    cout << "请输入一个正整数: ";
    cin >> input;
} while (input <= 0);
cout << "你输入的是: " << input << endl;
```

3.4 范围 for 循环（C++11）

用于遍历数组或容器中的每个元素，语法更简洁。

```
int arr[] = {1, 2, 3, 4, 5};
for (int x : arr) {
```

```
    cout << x << " ";
}
// 输出: 1 2 3 4 5
```

如果需要修改元素，需使用引用 `&`：

```
for (int &x : arr) {
    x *= 2;   // 每个元素翻倍
}
```

3.5 嵌套循环

一个循环内部再包含另一个循环。常用于多维数据处理，如打印图形、矩阵。

示例（打印九九乘法表）：

```
for (int i = 1; i <= 9; i++) {
    for (int j = 1; j <= i; j++) {
        cout << j << "*" << i << "=" << i * j << "\t";
    }
    cout << endl;
}
```

三种循环对比

循环	判断时机	最少执行次数	适用场景
`for`	先判断后执行	0 次	已知循环次数
`while`	先判断后执行	0 次	次数不定、依赖条件
`do-while`	先执行后判断	1 次	至少执行一次

四、跳转语句

4.1 break

- 在 循环 中：立即跳出当前层循环，不再继续。

- 在 **switch** 中: 结束 switch 分支。

示例（查找第一个能被 7 整除的数）：

```
for (int i = 1; i <= 100; i++) {
    if (i % 7 == 0) {
        cout << "第一个能被 7 整除的数: " << i << endl;  // 7
        break;   // 找到后立即跳出循环
    }
}
```

> `break` 只能跳出 当前一层 循环；嵌套循环中需配合其他手段跳出多层。

4.2 continue

跳过本次循环中 `continue` 之后的语句，直接进入下一次循环的条件判断。

示例（输出 1~10 中的奇数）：

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue;   // 偶数跳过
    }
    cout << i << " ";
}
// 输出: 1 3 5 7 9
```

4.3 goto

无条件跳转到同一函数内的某个标号处。 现代 **C++** 中一般不推荐使用，会破坏程序结构、降低可读性，仅在某些跳出多层循环的场景偶尔使用。

语法：

```
goto 标号;
...
标号:
    // 语句
```

示例：

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        if (i * j == 12) {
            cout << "找到: i=" << i << ", j=" << j << endl;
            goto end;   // 直接跳出两层循环
        }
    }
}
end:
    cout << "循环结束" << endl;
```

4.4 return

结束当前函数的执行，并将结果（若有）返回给调用者。在 `main` 函数中执行 `return 0;` 表示程序正常结束。

```
int add(int a, int b) {
    return a + b;   // 返回后函数立即结束
}
```

五、综合实例

实例 1：判断是否为素数

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

int main() {
    int n;
    cin >> n;
    bool isPrime = (n > 1);
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            isPrime = false;
```

```
            break;
        }
    }
    cout << n << (isPrime ? " 是素数" : " 不是素数") << endl;
    return 0;
}
```

实例 2：简单的菜单选择（switch + 循环）

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

int main() {
    int choice;
    do {
        cout << "\n=== 菜单 ===" << endl;
        cout << "1. 打招呼" << endl;
        cout << "2. 显示版本" << endl;
        cout << "0. 退出" << endl;
        cout << "请选择: ";
        cin >> choice;

        switch (choice) {
            case 1: cout << "你好，欢迎学习 C++！" << endl; break;
            case 2: cout << "版本 1.0" << endl; break;
            case 0: cout << "再见！" << endl; break;
            default: cout << "无效选项，请重新输入" << endl; break;
        }
    } while (choice != 0);
    return 0;
}
```

六、注意事项与最佳实践

1. 条件表达式：C++ 中 `0` 为假，非 `0` 为真；但判断相等时务必用 `==`，不要误写成赋值 `=`。
2. 浮点数比较：不要直接用 `==` 比较两个 `float`/`double`，由于精度问题易出错，应判断差值是否足够小，如 `fabs(a - b) < 1e-6`。

3. 避免死循环：`while`/`for` 循环体中要确保循环条件最终会变为假，否则会造成死循环。
4. 花括号习惯：即使循环体或分支只有一条语句，也建议使用 `{}`，可避免悬挂 `else` 等问题，也便于后期修改。
5. **switch** 记得 **break**：除非有意利用 case 穿透，否则每个 `case` 末尾都要写 `break`。
6. 少用 **goto**：优先使用结构化控制语句（`break`、`continue`、函数封装）替代 `goto`。
7. 循环变量作用域：在 `for (int i = ...)` 中声明的 `i` 只在循环内有效，这是推荐写法。

七、小结

结构	关键字	作用
顺序结构	—	按顺序依次执行
选择结构	`if` / `else` / `switch`	根据条件选择分支
循环结构	`for` / `while` / `do-while`	重复执行代码
跳转语句	`break` / `continue` / `goto` / `return	