程序设计与计算思维

第 24 讲：C++ 语言基础

韩文弢

清华大学

2026 年 5 月

本讲内容

问题与目标

问题引入

编程语言的发展有一条清晰的主线：**早期语言为机器服务，后来的语言为人服务**。

1970 年代，计算机极其昂贵且缓慢。程序员的目标是让程序**在有限的硬件上跑起来**。C 语言诞生于这个时代——它比汇编语言方便，但仍然贴近硬件：手动管理内存、指针算术、没有运行时安全检查。C++ 在 C 的基础上加入了面向对象等特性，但继承了“零开销抽象”的哲学——不为没用到的功能付出运行时代价。

问题引入

1990 年代，硬件速度已经提升了上千倍。Guido van Rossum 设计 Python 时的出发点完全不同：让编程**对人类更友好**。动态类型、自动内存管理、简洁的语法——这些便利以运行速度为代价，但机器已经快到可以承受。

同一个问题，两种风格截然不同。计算 1 到 100 的和：

Python

```python
total = sum(range(1, 101))
print(total)
```

问题引入

C++

```cpp
#include <iostream>

int main() {
  int sum = 0;
  for (int i = 1; i <= 100; ++i) {
    sum += i;
  }
  std::cout << sum << std::endl;
}
```

问题引入

Python 一行搞定，C++ 需要写类型、写 main 函数、编译后才能运行。为什么这么麻烦？

本讲将学习 C++ 的基本语法，掌握 C++ 的程序的写法，理解 C++ 的设计哲学和使用场景。

本讲目标

1. 了解 C++ 程序的编译与运行方式
2. 掌握 C++ 的基本数据类型、运算符、控制流和函数
3. 理解指针和引用的概念
4. 掌握数组、`std::string` 和 `std::string_view` 的基本用法
5. 能够用 C++ 实现简单的数值计算程序

Hello, C++!

语言发展历史

时间	C++	Python
1970s	Dennis Ritchie 发明 C 语言	–
1979	Bjarne Stroustrup 在 Bell 实验室开始 “C with Classes”	–
1985	正式命名为 C++，发布第一个商业版本	–
1991	–	Guido van Rossum 发布 Python 0.9.0
1998	C++98：第一个 ISO 标准	–

语言发展历史

时间	C++	Python
2011	C++11：现代 C++ 的起点（auto、lambda、range-for）	Python 3 发布
2014–17	C++14/17：结构化绑定、string_view	Python 3.6（f-string）
2020	C++20：concepts、ranges、modules	Python 3.9

- C++ 从 C 语言演化而来，强调**性能**和**底层控制**
- Python 强调**简洁**和**开发效率**，用 C 语言实现的
- Python 解释器本身（CPython）就是用 C 编写的

为什么学 C++?

Python	C++
解释型语言	编译型语言
动态类型	静态类型
开发效率高	运行效率高
适合快速原型	适合性能关键场景
自动内存管理	手动内存管理
NumPy/SciPy 生态	模板元编程 / STL

- 本课程以 Python 为主，C++ 为辅
- 学 C++ 的目的：理解编译型语言，为后续课程和科研工程打基础

第一个 C++ 程序

```cpp
#include <iostream>

int main() {
  std::cout << "Hello, world!" << std::endl;
}
```

编译并运行:

```
// 单文件编译
g++ -o hello hello.cpp

// 开启优化（运行性能更好，但编译时间更长）
g++ -O2 -o hello hello.cpp
```

第一个 C++ 程序

```
// 使用 C++23 标准
g++ -std=c++23 -o hello hello.cpp
```

与 Python 的对比：

1. C++ 需要 `#include` 导入库，Python 用 `import`
2. C++ 程序从 `main()` 函数开始，Python 从脚本顶部开始
3. C++ 需要**编译**成可执行文件再运行，Python **解释执行**
4. C++ 用花括号 `{ }` 定义代码块（换行和空格都无关紧要），Python 用缩进
5. C++ 语句以 `;` 结尾，Python 不需要

编译过程

```
源代码（.cpp）
    ↓  预处理器（preprocessor）
展开后的源代码
    ↓  编译器（compiler）
汇编代码（.s）
    ↓  汇编器（assembler）
目标文件（.o）
    ↓  链接器（linker）
可执行文件（a.out / .exe）
```

- **预处理**：处理 `#include`、`#define` 等指令（展开头文件）
- **编译**：将 C++ 源代码翻译为汇编代码
- **汇编**：将汇编代码翻译为机器码（目标文件 `.o`）
- **链接**：将多个目标文件和库合并为可执行文件

数据类型与变量

基本数据类型

类型	大小	说明	Python 对应
`int`	4 B	整数	`int`
`double`	8 B	双精度浮点数	`float`
`bool`	1 B	布尔值	`bool`
`char`	1 B	字符	`str`（单字符）
`std::string`	–	字符串	`str`

C++ 是**静态类型**语言：变量类型在编译时确定，不能随意改变。

bool 类型与真值

关于 bool 类型：值为 true（真）或 false（假）。在需要布尔值的上下文中，**零**被当作 false，**非零**被当作 true：

```
bool flag = true;
int n = 42;
if (n) { ... }          // n != 0, 相当于 true
if (n - 42) { ... }     // 0, 相当于 false
```

Python 也有类似规则：0、0.0、""、None、空容器等被视为 False，其余为 True。

变量声明与 auto

```cpp
#include <iostream>
#include <string>

int main() {
  int n = 42;
  double x = 3.14;
  std::string s = "hello";
  bool flag = true;

  std::cout << "int:    " << n << std::endl;
  std::cout << "double: " << x << std::endl;
  std::cout << "string: " << s << std::endl;
  std::cout << "bool:   " << std::boolalpha << flag
            << std::endl;
```

变量声明与 auto

```
    auto pi = 3.14159;
    auto count = 100;
    std::cout << "auto:    " << pi << " " << count
              << std::endl;

}
```

- C++ 变量必须先声明类型，再使用
- auto 关键字让编译器**自动推导**类型（C++11 引入）
- const 表示常量，初始化后不能修改

运算符

算术与关系运算符

C++ 的运算符大部分与 Python 相同：

类别	Python	C++
算术	`+ - * / // % **`	`+ - * / %`（整数除法自动截断）
自增/自减	`i += 1`	`++i` / `i++`（无 Python 对应）
比较	`== != < > <= >=`	`== != < > <= >=`（完全相同）
逻辑	`and or not`	`&& || !`
位运算	`& | ^ ~ << >>`	`& | ^ ~ << >>`（完全相同）

算术与关系运算符

- C++ 没有 `**` 幂运算，用 `std::pow(x, n)` 或 `x * x`
- C++ 的除法根据两侧的类型决定行为，整数除法**自动截断**（`7 / 2 == 3`），Python 用 `/` 和 `//` 分别表示真除法和整数除法
- C++ 的自增/自减运算符 `++` 和 `--` 有前置和后置两种形式（`++i` vs `i++`）
  - 前置形式（`++i`）先增加后使用
  - 后置形式（`i++`）先使用后增加
  - 例如：`i = 1, j = i++` 后 `i == 2, j == 1`；而 `i = 1, j = ++i` 后 `i == 2, j == 2`

逻辑运算符的短路求值

C++ 和 Python 的逻辑运算符都具有**短路求值**（short-circuit evaluation）特性：

1. `&&`（and）：如果左侧为 false，右侧**不再求值**，整个表达式直接为 false
2. `||`（or）：如果左侧为 true，右侧**不再求值**，整个表达式直接为 true

C++ 中的副作用差异：

```cpp
int x = 5;
// 因为 x > 10 为 false，++x 不会执行
bool result = (x > 10) && (++x > 0);
```

逻辑运算符的短路求值

```cpp
std::cout << x;  // 5（没有变！）

// 因为 x > 0 为 true，++x 不会执行
result = (x > 0) || (++x > 10);
std::cout << x;  // 5（仍然没有变！）
```

Python 中同样的短路行为：

```python
x = 5
result = (x > 10) and (x := x + 1) > 0
print(x)  # 5，右侧没有执行

result = (x > 0) or (x := x + 1) > 10
print(x)  # 5，右侧没有执行
```

逻辑运算符的短路求值

- 短路求值可以用来**避免错误**：`if (p != nullptr && *p > 0)` 先检查指针非空再解引用
- 但也要注意：不要在逻辑运算符右侧放置**有副作用的表达式**（如 `++x`、赋值），因为它们可能不会执行

控制流

条件与循环

C++ 的控制流与 Python 非常相似，只是语法细节不同：

```cpp
#include <iostream>

int main() {
  int score = 85;

  if (score >= 90) {
    std::cout << "A" << std::endl;
  } else if (score >= 80) {
    std::cout << "B" << std::endl;
  } else {
    std::cout << "C" << std::endl;
  }
```

条件与循环

```
    for (int i = 0; i < 5; ++i) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    int sum = 0;
    int i = 1;
    while (i <= 100) {
        sum += i;
        ++i;
    }
    std::cout << "sum = " << sum << std::endl;
}
```

条件与循环

与 Python 的对比：

特性	Python	C++
代码块	缩进	{ } 大括号
条件语句	`if x > 0:`	`if (x > 0) {`
for 循环	`for i in range(5):`	`for (int i = 0; i < 5; ++i) {`
while 循环	`while x > 0:`	`while (x > 0) {`
逻辑运算	`and, or, not`	`&&, ||, !`

函数

函数定义

```cpp
#include <cmath>
#include <iostream>

double sigmoid(double x) {
  return 1.0 / (1.0 + std::exp(-x));
}

void print_hello(const std::string& name) {
  std::cout << "Hello, " << name << "!" << std::endl;
}

int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
```

函数定义

```
}

int main() {
  std::cout << "sigmoid(0) = " << sigmoid(0.0) << std::endl;
  print_hello("C++");
  std::cout << "5! = " << factorial(5) << std::endl;
}
```

与 Python 的对比：

1. C++ 需要声明参数类型和返回类型
2. C++ 用 { } 包裹函数体，Python 用缩进

C++ 函数声明和定义可以分开（头文件 .h + 源文件 .cpp）

数组

原生数组

C++ 的原生数组是固定大小的，声明时必须指定长度：

```cpp
#include <iostream>

int main() {
  int a[5] = {10, 20, 30, 40, 50};

  for (int i = 0; i < 5; ++i) {
    std::cout << "a[" << i << "] = " << a[i] << std::endl;
  }

  int b[3][2] = {{1, 2}, {3, 4}, {5, 6}};
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 2; ++j) {
```

原生数组

```
            std::cout << b[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

- 数组大小**编译时确定**，不能动态改变
- 数组索引从 0 开始，与 Python 相同
- 越界访问**不会报错**（未定义行为，可能会引起程序崩溃），这是 C++ 的常见 bug 来源
- 实际开发中推荐使用 `std::vector`（第 26 讲将详细介绍）。

指针与引用

什么是地址?

计算机的内存可以视为一个平坦的字节数组，每个字节都有一个唯一的**地址**。

每个变量在内存中可以用它开始字节的**地址**来表示它的位置。

上一节介绍了数组，而数组与地址有着密切的联系——数组名本质上就是一个指向首元素的指针。

```
#include <iostream>

int main() {
  int x = 42;
  int* p = &x;
```

什么是地址?

```
    std::cout << "value of x:    " << x << std::endl;
    std::cout << "address of x: " << &x << std::endl;
    std::cout << "pointer p:     " << p << std::endl;
    std::cout << "value via *p: " << *p << std::endl;

    *p = 100;
    std::cout << "after *p=100, x = " << x << std::endl;
}
```

- `&x`：取变量 x 的地址（取址运算符）
- `*p`：通过地址访问变量（解引用运算符）
- 指针存储的是**内存地址**，不是值本身

指针与数组

数组名就是一个指向首元素的指针：

```cpp
int a[5] = {10, 20, 30, 40, 50};
int* p = a;          // p 指向 a[0]
std::cout << *p;           // 10
std::cout << *(p + 2); // 30 (即 a[2])
```

- `a[i]` 等价于 `*(a + i)`，也可以写成 `i[a]`：下标运算本质上是指针算术
- 这解释了为什么 C++ 数组越界不会报错——它只是访问指针偏移后的内存
- 将数组传给函数时，它退化为指针，丢失长度信息

指针与引用

指针：存储另一个变量的地址。**引用**：给变量取一个别名。

```
#include <iostream>

void swap_by_value(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

void swap_by_pointer(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
```

指针与引用

```
}

void swap_by_reference(int& a, int& b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int x = 1, y = 2;

  swap_by_value(x, y);
  std::cout << "after swap_by_value:        x=" << x
            << " y=" << y << std::endl;
```

指针与引用

```
    swap_by_pointer(&x, &y);
    std::cout << "after swap_by_pointer:      x=" << x
              << " y=" << y << std::endl;

    swap_by_reference(x, y);
    std::cout << "after swap_by_reference:    x=" << x
              << " y=" << y << std::endl;
}
```

- **值传递**：函数内修改不影响外部（Python 中不可变类型同理）
- **指针传递**：传递地址，通过 * 间接修改（&x 取地址）
- **引用传递**：直接绑定到原变量，语法更简洁（推荐使用）

字符串

std::string

std::string 是 C++ 的字符串类型，对应 Python 的 str。理解了引用之后，就能更好地理解字符串参数传递的效率问题：

```cpp
#include <iostream>
#include <string>
#include <string_view>

int main() {
  // std::string: 可变字符串
  std::string s1 = "Hello";
  std::string s2 = "World";

  std::string s3 = s1 + ", " + s2 + "!";
  std::cout << s3 << std::endl;
```

```
    std::cout << "length: " << s3.size() << std::endl;
    std::cout << "substr: " << s3.substr(0, 5) << std::endl;
    std::cout << "find: " << s3.find("World") << std::endl;

    // std::string_view: 不可变视图
    // (C++17)
    std::string_view sv = s3;
    std::cout << "view: " << sv << std::endl;
    std::cout << "prefix: " << sv.substr(0, 5) << std::endl;
}
```

std::string_view（C++17）

std::string_view 是对字符串的**不可变视图**，不拥有数据，不产生拷贝：

特性	std::string	std::string_view
拥有数据	是	否（只是视图）
可修改	是	否
拷贝开销	深拷贝	浅拷贝（仅指针+长度）
适用场景	需要构造/修改字符串	只读访问已有字符串

函数参数推荐使用 const std::string&（引用避免拷贝）或 std::string_view（更通用的视图）。

字符串与 Python 对比

操作	Python `str`	C++ `std::string`
拼接	`s1 + s2`	`s1 + s2`
长度	`len(s)`	`s.size()`
子串	`s[0:5]`	`s.substr(0, 5)`
查找	`s.find("abc")`	`s.find("abc")`
格式化	f-string / `format()`	`std::format`（C++20）
不可变	是	否（`std::string` 可修改）

文件输入输出

读写文件

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main() {
  std::ofstream out("output.txt");
  out << "Hello from C++!" << std::endl;
  out << "The answer is 42." << std::endl;
  out.close();

  std::ifstream in("output.txt");
  std::string line;
  while (std::getline(in, line)) {
    std::cout << line << std::endl;
```

读写文件

```
    }
    in.close();
}
```

- `std::ofstream`：写文件（output file stream）
- `std::ifstream`：读文件（input file stream）
- Python 中用 `open()` + `with` 语句，C++ 用流对象

实战：蒙特卡罗方法求 π

问题回顾

在第 10 讲中用 Python 实现了蒙特卡罗方法估算 π：随机向单位正方形内撒点，统计落入四分之一圆内的比例。

```python
import numpy as np

N = 1_000_000
rng = np.random.default_rng(42)
x = rng.random(N)
y = rng.random(N)
count = np.sum(x**2 + y**2 < 1.0)
pi = 4.0 * count / N
print(f"pi ≈ {pi}")
print(f"error = {abs(pi - np.pi):.6f}")
```

C++ 实现

现在用 C++ 实现同样的算法，对比两种语言的写法。

```cpp
#include <cmath>
#include <iostream>
#include <random>

int main() {
  const int N = 1000000;
  std::mt19937 rng(42);
  std::uniform_real_distribution<double> dist(0.0, 1.0);

  int count = 0;
  for (int i = 0; i < N; ++i) {
    double x = dist(rng);
```

C++ 实现

```
      double y = dist(rng);
      if (x * x + y * y < 1.0) {
        ++count;
      }
    }

    double pi = 4.0 * count / N;
    std::cout << "pi ≈ " << pi << std::endl;
    std::cout << "error = " << std::abs(pi - M_PI)
              << std::endl;
}
```

Python 与 C++ 写法对比

Python	C++
`import numpy as np`	`#include <random>`
`rng = np.random.default_rng(42)`	`std::mt19937 rng(42);`
`x = rng.random(N)`	`double x = dist(rng);`（逐个生成）
`np.sum(x**2 + y**2 < 1)`	`if (x*x + y*y < 1.0) ++count;`
`4.0 * count / N`	`4.0 * count / N`（完全相同）

- Python 的 NumPy 可以**向量化**运算，C++ 需要**显式循环**
- C++ 随机数引擎比 NumPy 更底层，但功能完整

本讲小结

知识要点

1. **编译与运行**：C++ 需要先编译（g++）再运行，Python 解释执行
2. **数据类型**：C++ 是静态类型，变量类型编译时确定；`auto` 可自动推导
3. **运算符**：与 Python 大部分相同；逻辑运算符 `&&`/`||` 具有短路求值特性，注意副作用
4. **控制流**：`if`/`for`/`while` 逻辑与 Python 相同，语法用 `{ }` 而非缩进
5. **函数**：需要声明参数和返回类型，支持声明与定义分离
6. **指针与引用**：指针存储地址，引用是别名；引用传递可以修改外部变量
7. **数组**：固定大小，与指针密切相关；推荐使用 `std::vector`

知识要点

8. **字符串**：`std::string` 对应 Python 的 `str`（但可修改），`std::string_view` 提供零拷贝只读视图
9. **文件 I/O**：`std::ofstream` 写文件，`std::ifstream` 读文件

设计哲学对比

C++ 的设计动机是**高性能系统编程**（操作系统、游戏引擎、嵌入式），Python 的设计动机是**易学易用**（教学、脚本、快速原型）。这一根本差异体现在方方面面：

设计选择	C++ 的做法	Python 的做法	背后的动机
类型系统	静态类型，编译时检查	动态类型，运行时检查	C++ 追求零开销：运行时不做类型检查
内存管理	手动 new/delete，智能指针	自动垃圾回收	C++ 需要**精确控制**内存布局和生命周期

设计哲学对比

设计选择	C++ 的做法	Python 的做法	背后的动机
错误处理	数组越界不报错（未定义行为）	越界抛 IndexError	C++ 不为安全检查付出运行时代价
函数参数	区分值传递/指针/引用	统一引用语义	C++ 让程序员**选择**最小开销的传递方式
字符串	`std::string` 可变，`string_view` 零拷贝	`str` 不可变，统一接口	C++ 为性能提供多种选择

设计哲学对比

设计选择	C++ 的做法	Python 的做法	背后的动机
编译模型	编译为机器码，需要头文件	解释执行，import 即用	C++ 编译时已知一切，才能充分优化

- 没有**绝对更好**的语言，只有**更适合场景**的选择
- Python 的“简洁”以运行速度为代价；C++ 的“麻烦”换来的是对硬件的完全掌控
- 在科学计算中，两者常常**配合使用**：Python 做高层逻辑，C++ 做底层计算