程序设计与计算思维

第 26 讲：C++ 泛型编程与标准库

韩文弢

清华大学

2026 年 6 月

本讲内容

问题与目标

问题引入

在前两讲中，我们为每个具体类型分别写了函数和类。比如 `max_val` 要写 `int` 版本和 `double` 版本，`Stack` 要写 `Stack<int>` 和 `Stack<string>`。有没有一种方式，写一次代码就能适用于多种类型？

这正是**泛型编程（Generic Programming）**要解决的问题。Python 因为动态类型天然就支持“泛型”，而 C++ 通过模板机制在**编译时**生成针对不同类型的代码。

本讲目标

1. 掌握 C++ 函数模板和类模板的基本用法
2. 了解 C++ 标准库容器：`vector`、`map`、`set` 等
3. 掌握标准库算法：`sort`、`accumulate`、`count_if` 等
4. 理解 lambda 表达式在 C++ 中的用法
5. 了解现代 C++（C++11/14/17）的关键特性
6. 理解智能指针的使用场景和最佳实践
7. 了解多文件工程的组织方式和 Makefile 的基本写法
8. 对比 Python 和 C++ 的性能差异

函数模板

泛型函数

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

template <typename T>
T max_val(T a, T b) {
  return (a > b) ? a : b;
}

template <typename T>
void print_vector(const std::vector<T>& v) {
  for (const auto& x : v) {
    std::cout << x << " ";
  }
```

泛型函数

```
    std::cout << std::endl;
}

int main() {
    std::cout << max_val(3, 7) << std::endl;
    std::cout << max_val(3.14, 2.72) << std::endl;
    std::cout << max_val<std::string>("abc", "xyz")
              << std::endl;

    std::vector<int> vi = {1, 2, 3};
    std::vector<double> vd = {1.1, 2.2, 3.3};
    print_vector(vi);
    print_vector(vd);
}
```

泛型函数

- `template <typename T>` 声明一个类型参数 `T`
- 编译器根据调用时的参数类型**自动推导** `T`，也可以显式指定 `max_val<string>(...)`
- Python 不需要模板——动态类型天然支持泛型

Python 的“泛型”

特性	Python	C++ 泛型
类型检查时机	运行时	编译时
类型错误发现	运行时报错	编译时报错
代码膨胀	无	每种类型生成一份代码
性能	较慢（动态派发）	快（静态生成）
灵活性	任意类型	需要满足操作约束

类模板

泛型类

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

template <typename T>
class Stack {
 private:
  std::vector<T> data_;

 public:
  void push(const T& value) { data_.push_back(value); }

  void pop() { data_.pop_back(); }
```

```
  T top() const { return data_.back(); }

  bool empty() const { return data_.empty(); }

  size_t size() const { return data_.size(); }
};

int main() {
  Stack<int> si;
  si.push(10);
  si.push(20);
  si.push(30);
  while (!si.empty()) {
    std::cout << si.top() << " ";
```

泛型类

```
    si.pop();
  }
  std::cout << std::endl;

  Stack<std::string> ss;
  ss.push("hello");
  ss.push("world");
  std::cout << ss.top() << std::endl;
}
```

- `Stack<int>` 和 `Stack<string>` 是两个**完全不同**的类
- 编译器根据模板参数生成对应的类代码
- 对应 Python 中不需要类型参数的 `Stack` 类（动态类型天然适配）

标准库中的类模板

C++ 标准库的核心就是泛型类模板：

容器	说明	Python 对应
`vector<T>`	动态数组	`list`
`map<K,V>`	有序键值对（红黑树）	`dict`（无序）
`unordered_map<K,V>`	哈希表	`dict`
`set<T>`	有序不重复集合	`set`
`stack<T>`	栈（LIFO）	`list`（模拟）
`queue<T>`	队列（FIFO）	`collections.deque`

标准库容器

std::vector：动态数组

std::vector 是 C++ 中最常用的容器，相当于 Python 的 list。它在堆上分配连续内存，支持随机访问、动态扩容：

```cpp
#include <iostream>
#include <vector>

int main() {
    // 创建
    std::vector<int> v = {10, 20, 30};

    // 添加元素
    v.push_back(40);
    v.push_back(50);
```

std::vector：动态数组

```cpp
// 访问元素
std::cout << v[0] << std::endl;      // 10
std::cout << v.at(2) << std::endl;   // 30（带越界检查）

// 删除末尾元素
v.pop_back();

// 大小与容量
std::cout << "size: " << v.size() << std::endl;

// 遍历：下标
for (int i = 0; i < (int)v.size(); ++i) {
  std::cout << v[i] << " ";
}
```

std::vector：动态数组

```
    std::cout << std::endl;

    // 遍历：range-for
    for (const auto& x : v) {
        std::cout << x << " ";
    }
    std::cout << std::endl;
}
```

常用操作与 Python 对比：

操作	Python `list`	C++ `std::vector`
创建	`v = [1, 2, 3]`	`std::vector<int> v = {1, 2, 3};`

std::vector：动态数组

操作	Python list	C++ std::vector
追加	`v.append(4)`	`v.push_back(4);`
删除末尾	`v.pop()`	`v.pop_back();`
长度	`len(v)`	`v.size()`
下标访问	`v[i]`（越界抛异常）	`v[i]`（越界未定义）或 `v.at(i)`（抛异常）
遍历	`for x in v:`	`for (const auto& x : v)`
拼接	`v1 + v2`（创建新列表）	`v1.insert(v1.end(), v2.begin(), v2.end())`

- `push_back` 在末尾追加，均摊 O(1)
- `v[i]` 不检查越界（性能优先），`v.at(i)` 检查越界（安全优先）
- 遍历时用 `const auto&` 避免**不必要的拷贝**

std::unordered_map：哈希表

std::unordered_map 对应 Python 的 dict，基于哈希表实现，查找平均 O(1)：

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

int main() {
  // 创建与插入
  std::unordered_map<std::string, int> scores;
  scores["Alice"] = 90;
  scores["Bob"] = 85;
  scores.insert({"Charlie", 78});
```

std::unordered_map：哈希表

```cpp
// 访问（不存在的 key 会自动插入默认值 0）
std::cout << scores["Alice"] << std::endl;

// 查找（不触发自动插入）
auto it = scores.find("David");
if (it != scores.end()) {
  std::cout << it->second << std::endl;
} else {
  std::cout << "not found" << std::endl;
}

// 判断 key 是否存在
int cnt = scores.count("Bob");
std::cout << "Bob count: " << cnt << std::endl;
```

std::unordered_map：哈希表

```
    // 遍历
    for (const auto& [name, score] : scores) {
        std::cout << name << ": " << score << std::endl;
    }

    // 删除
    scores.erase("Bob");
    std::cout << "size: " << scores.size() << std::endl;
}
```

常用操作与 Python 对比：

std::unordered_map：哈希表

操作	Python dict	C++ unordered_map
创建/插入	`d["key"] = val`	`m["key"] = val;` 或 `m.insert({"key", val})`
访问	`d["key"]`（不存在则 KeyError）	`m["key"]`（不存在则**插入默认值**）
安全查找	`d.get("key")`	`m.find("key")`（返回迭代器）
判断存在	`"key" in d`	`m.contains("key")`
删除	`del d["key"]`	`m.erase("key");`
遍历	`for k, v in d.items():`	`for (const auto& [k, v] : m)`
大小	`len(d)`	`m.size()`

std::unordered_map：哈希表

- `m["key"]` 在 key 不存在时会**自动插入默认值**，这是与 Python 最大的不同（类似 `collections.defaultdict`）
- 如果只想查询，用 `m.find()` 或 `m.count()`，避免意外插入
- `std::map` 是有序版本（红黑树），查找 O(log n)；`unordered_map` 更快，对应 Python 的 `dict`

其他容器一览

容器	说明	Python 对应
`map<K,V>`	有序键值对（红黑树），查找 O(log n)	`dict`（无序）
`set<T>`	有序不重复集合	`set`
`stack<T>`	栈（LIFO）	`list`（模拟）
`queue<T>`	队列（FIFO）	`collections.deque`
`priority_queue<T>`	优先队列（最大堆）	`heapq` 模块

迭代器与范围

迭代器：指针的一般化

在第 24 讲中学过，原生数组的指针可以做 `*(p + i)` 访问元素。**迭代器（Iterator）**是对指针的**一般化**——它提供统一的“遍历”接口，不仅适用于数组，也适用于 `vector`、`map`、`set` 等所有容器：

```
std::vector<int> v = {10, 20, 30};
auto it = v.begin();    // 指向第一个元素
*it;                    // 10
++it;                   // 移动到下一个
*it;                    // 20

// 遍历：从 begin() 到 end()
for (auto it = v.begin(); it != v.end(); ++it) {
```

迭代器：指针的一般化

```
    std::cout << *it << " ";
}
```

- `begin()` 返回指向首元素的迭代器，`end()` 返回指向**末尾之后**的迭代器
- `[begin(), end())` 构成一个**左闭右开**区间，与 Python 的切片 `v[0:len]` 约定一致
- `*it` 解引用（像指针）、`++it` 前进（像指针）、`it != end()` 判断结束

迭代器让标准库算法**不关心底层容器类型**——同一个 `std::sort` 对 `vector`、`array` 甚至 C 风格数组都能工作。

标准库算法

常用算法

标准库提供了大量通用算法，通过迭代器（或范围）指定操作区间：

```cpp
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>

int main() {
  std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6};

  // sort: 排序
  std::sort(v.begin(), v.end());
  for (int x : v) std::cout << x << " ";
  std::cout << std::endl;
```

常用算法

```cpp
// sort: 自定义降序
std::sort(v.begin(), v.end(),
          [](int a, int b) { return a > b; });
for (int x : v) std::cout << x << " ";
std::cout << std::endl;

// accumulate: 求和
int sum = std::accumulate(v.begin(), v.end(), 0);
std::cout << "sum = " << sum << std::endl;

// count_if: 条件计数
int cnt = std::count_if(v.begin(), v.end(),
                        [](int x) { return x > 3; });
```

```
std::cout << "count(>3) = " << cnt << std::endl;

// find: 查找元素
auto it = std::find(v.begin(), v.end(), 5);
if (it != v.end()) {
  std::cout << "found 5 at index " << it - v.begin()
            << std::endl;
}

// transform: 变换
std::vector<int> doubled(v.size());
std::transform(v.begin(), v.end(), doubled.begin(),
               [](int x) { return x * 2; });
for (int x : doubled) std::cout << x << " ";
```

```
    std::cout << std::endl;
}
```

算法与 Python 对比

功能	Python	C++ 标准库
排序	`sorted(v)` 或 `v.sort()`	`std::sort(v.begin(), v.end())`
自定义排序	`sorted(v, reverse=True)`	`std::sort(begin, end, lambda)`
求和	`sum(v)`	`std::accumulate(begin, end, 0)`
条件计数	`sum(1 for x in v if x>3)`	`std::count_if(begin, end, lambda)`
变换	`[x*2 for x in v]`	`std::transform(begin, end, out, lambda)`
查找	`v.index(target)`	`std::find(begin, end, target)`

算法与 Python 对比

功能	Python	C++ 标准库
最大/最小	`max(v)` / `min(v)`	`std::max_element` / `std::min_element`

- 算法不直接操作容器，而是通过**迭代器**操作元素区间——同一算法适用于所有容器
- `std::sort` 默认升序；传 lambda 自定义比较规则
- Python 的列表推导（`[x*2 for x in v]`）对应 C++ 的 `std::transform`

Lambda 表达式

C++ 中的 Lambda

C++11 引入了 lambda 表达式，语法与 Python 有所不同：

```
auto f = [](int x) { return x * 2; };
std::cout << f(21) << std::endl;  // 42
```

- `[]`：捕获列表，决定如何访问外部变量
- `(int x)`：参数列表
- `{ return x * 2; }`：函数体

捕获列表

捕获方式	含义
[ ]	不捕获任何外部变量
[=]	以值方式捕获所有外部变量（拷贝）
[&]	以引用方式捕获所有外部变量
[x]	只以值捕获 x
[&x]	只以引用捕获 x
[=, &x]	默认值捕获，但 x 以引用捕获

- Python 的 lambda 隐式以引用方式捕获所有外部变量
- C++ 的捕获列表让程序员**精确控制**变量的访问方式

智能指针

裸指针的问题

第 25 讲介绍了 RAII 原则：用构造/析构管理资源。但 `new` 返回的裸指针不遵守 RAII——如果函数中途 `return` 或抛出异常，`delete` 就会被跳过：

```
void foo() {
  Node* p = new Node(42);
  if (error) return;     // 忘记 delete → 内存泄漏！
  delete p;
}
```

智能指针是 RAII 的完整解决方案：它包装裸指针，在析构时**自动** `delete`。

unique_ptr 与 shared_ptr

```cpp
#include <iostream>
#include <memory>
#include <vector>

class Node {
 public:
  explicit Node(int v) : value_(v) {
    std::cout << "Node(" << value_ << ") created"
              << std::endl;
  }
  ~Node() {
    std::cout << "Node(" << value_ << ") destroyed"
              << std::endl;
  }
```

unique_ptr 与 shared_ptr

```
    int value() const { return value_; }

  private:
    int value_;
};

int main() {
    // unique_ptr: 独占所有权
    auto p1 = std::make_unique<Node>(1);
    std::cout << p1->value() << std::endl;

    // shared_ptr: 共享所有权（引用计数）
    auto p2 = std::make_shared<Node>(2);
    auto p3 = p2;  // 引用计数 = 2
```

unique_ptr 与 shared_ptr

```
    std::cout << "use_count: " << p2.use_count() << std::endl;

    // 容器中存储 shared_ptr
    std::vector<std::shared_ptr<Node>> nodes;
    nodes.push_back(p2);
    nodes.push_back(std::make_shared<Node>(3));
}
```

特性	unique_ptr	shared_ptr
所有权	独占（唯一）	共享（引用计数）
拷贝	**禁止**（只能移动）	允许（引用计数 +1）
开销	零（与裸指针相同）	有（引用计数原子操作）
创建方式	make_unique<T>(args)	make_shared<T>(args)

unique_ptr 与 shared_ptr

特性	unique_ptr	shared_ptr
Python 类比	无（所有对象都是共享引用）	最接近 Python 的变量语义

- unique_ptr：轻量、零开销，表达“这个资源**只属于我**”
- shared_ptr：多个指针共享同一对象，最后一个离开作用域时自动释放
- make_unique/make_shared 比直接 new 更安全（避免内存泄漏的极端情况）

最佳实践

1. **优先用栈对象**：能用局部变量就不用指针（`std::string` 优于 `std::string*`）
2. **必须用指针时，用智能指针**：`auto p = make_unique<T>(...)` 或 `make_shared<T>(...)`
3. **默认用 `unique_ptr`**：除非需要共享所有权，否则用零开销的 `unique_ptr`
4. **不要混用裸 `delete`**：智能指针负责释放，手动 `delete` 会导致双重释放

对比 Python：Python 中所有对象都是引用计数（类似 `shared_ptr`），程序员不需要关心。C++ 让你**选择**是否需要引用计数的开销。

多文件工程与 Makefile

为什么要分文件?

实际项目由**几十到上百个源文件**组成。分文件的好处:

1. **模块化**:每个 .cpp 文件负责独立功能,团队协作时减少冲突
2. **增量编译**:只重新编译修改过的文件,其余复用已编译的 .o 文件
3. **接口与实现分离**:头文件(.h)声明接口,源文件(.cpp)隐藏实现

以第 24 讲的 stats 项目为例,它由三个文件组成:

为什么要分文件?

stats.h（头文件）

```
#ifndef STATS_H_
#define STATS_H_

#include <vector>

double mean(const std::vector<double>& data);
double stddev(const std::vector<double>& data);

#endif  // STATS_H_
```

为什么要分文件?

stats.cpp（实现）

```cpp
#include "stats.h"

#include <cmath>

double mean(const std::vector<double>& data) {
  double sum = 0.0;
  for (double x : data) {
    sum += x;
  }
  return sum / data.size();
}
```

为什么要分文件?

```
double stddev(const std::vector<double>& data) {
  double m = mean(data);
  double s = 0.0;
  for (double x : data) {
    s += (x - m) * (x - m);
  }
  return std::sqrt(s / data.size());
}
```

为什么要分文件?

main.cpp（主程序）

```cpp
#include "stats.h"

#include <iostream>
#include <vector>

int main() {
  std::vector<double> v = {1.2, 3.4, 5.6, 7.8};
  std::cout << "mean:   " << mean(v) << std::endl;
  std::cout << "stddev: " << stddev(v) << std::endl;
}
```

头文件与宏守卫

头文件（.h）声明接口，源文件（.cpp）提供实现。这是 C++ 实现**接口与实现分离**的标准方式：

1. stats.h 只声明函数签名，不包含实现细节
2. stats.cpp 通过 #include "stats.h" 声明这些函数，并进行实现
3. main.cpp 通过 #include "stats.h" 声明这些函数即可使用，无需知道实现

宏守卫：防止头文件被重复包含

```
#ifndef STATS_H_          // 如果没有定义 STATS_H_
#define STATS_H_          // 定义它

// 头文件内容...

#endif  // STATS_H_
```

- 第一次 `#include "stats.h"` 时，`STATS_H_` 未定义，内容被包含
- 之后再次 `#include` 时，`STATS_H_` 已定义，整个文件被跳过
- 没有宏守卫，重复包含会导致“重复定义”编译错误
- C++20 可用 `#pragma once` 替代，但宏守卫是更通用的写法

Makefile：自动化编译

当文件数量增多，手动输入 g++ 命令变得繁琐且容易出错。

Makefile 用规则描述文件间的依赖关系，make 工具自动判断哪些文件需要重新编译：

```
CXX = g++
CXXFLAGS = -std=c++17 -Wall -O2

TARGET = stats_demo
SRCS = main.cpp stats.cpp
OBJS = $(SRCS:.cpp=.o)

all: $(TARGET)
```

Makefile：自动化编译


```
$(TARGET): $(OBJS)
  $(CXX) $(CXXFLAGS) -o $@ $^

%.o: %.cpp
  $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
  rm -f $(OBJS) $(TARGET)

.PHONY: all clean
```

Makefile：自动化编译

关键概念：

1. **变量**：CXX（编译器）、CXXFLAGS（编译选项）、SRCS（源文件）、OBJS（目标文件）
2. **规则**：目标：依赖 → 命令，描述如何从依赖生成目标
3. **自动变量**：$@（目标名）、$<（第一个依赖）、$^（所有依赖）
4. **模式规则**：%.o: %.cpp 描述所有 .cpp 到 .o 的通用编译方式
5. **增量编译**：make 比较文件时间戳，只重新编译比目标**更新**的依赖

只需执行 make 即可编译整个项目，make clean 清理生成文件。大型项目还可以使用 **CMake** 生成 Makefile，此处不再展开。

性能对比

C++ vs Python vs NumPy

用同样的任务对比三种实现的性能：计算 10^7 个点的 sin 值。

Python + NumPy：

```python
import numpy as np
import time

N = 10_000_000
x = np.linspace(0.0, 3.14, N)

t1 = time.perf_counter()
y = np.sin(x)
t2 = time.perf_counter()

print(f"sin({N} points): {t2 - t1:.6f} s")
print(f"y[0]={y[0]:.6f}, y[N//2]={y[N//2]:.6f}")
```

C++ vs Python vs NumPy

C++ 原生循环：

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

std::vector<double> linspace(double start, double end,
                             int n) {
  std::vector<double> v(n);
  double step = (end - start) / (n - 1);
  for (int i = 0; i < n; ++i) {
    v[i] = start + i * step;
```

C++ vs Python vs NumPy


```
    }
    return v;
}

void compute_sin(const std::vector<double>& x,
                 std::vector<double>& y) {
    for (size_t i = 0; i < x.size(); ++i) {
        y[i] = std::sin(x[i]);
    }
}

int main() {
    const int N = 10000000;
    auto x = linspace(0.0, 3.14, N);
```

C++ vs Python vs NumPy


```
    std::vector<double> y(N);

    auto t1 = std::chrono::high_resolution_clock::now();
    compute_sin(x, y);
    auto t2 = std::chrono::high_resolution_clock::now();
    double elapsed =
        std::chrono::duration<double>(t2 - t1).count();

    std::cout << "sin(" << N << " points): " << elapsed
              << " s" << std::endl;
    std::cout << "y[0]=" << y[0] << ", y[N/2]=" << y[N / 2]
              << std::endl;
}
```

性能对比结果

实现	时间 (量级)	特点	开发难度
Python 循环	约 1.5 s	逐元素解释执行	低
NumPy 向量化	约 110 ms	C 库批量运算	低
C++ 循环	约 230 ms	编译优化，直接执行	中
C++ -O2 优化	约 50 ms	编译器自动向量化	中

- **NumPy 之所以快**：底层调用 C/Fortran 实现的数学库
- **C++ 之所以快**：编译器直接生成机器码，-O2 还能自动向量化
- Python 循环比 C++ 慢约 **5-30 倍**，但 NumPy 接近 C++

为什么 NumPy 这么快?

NumPy 将 Python 的“慢循环”下推到 C 层：

层次	执行方式
Python 层	`y = np.sin(x)` 一行调用
NumPy C 层	遍历数组，逐元素调用 C 的 `sin()`
BLAS/LAPACK	硬件优化的线性代数库

- Python 负责高层逻辑，C 负责底层计算
- 这就是为什么学 C++ 有意义：理解“快”的本质

本讲小结

知识要点

1. **函数模板**：`template <typename T>` 让函数适用于多种类型，编译时自动推导
2. **类模板**：`Stack<T>` 为不同类型生成不同的类，标准库容器都是类模板
3. **标准库容器**：`vector`（动态数组）和 `unordered_map`（哈希表）最常用，分别对应 Python 的 `list`/`dict`
4. **标准库算法**：`sort`、`accumulate`、`count_if`、`transform` 等通用算法，通过迭代器适用于所有容器
5. **Lambda 表达式**：`[capture](params) { body }` 定义匿名函数，捕获列表精确控制变量访问

知识要点

6. **现代 C++**：`auto`、`range-for`、结构化绑定等让 C++ 更接近 Python 的简洁性
7. **智能指针**：`unique_ptr` 独占所有权（零开销），`shared_ptr` 共享所有权（引用计数）；优先栈对象，其次 `unique_ptr`
8. **多文件工程**：头文件声明接口，源文件实现功能；Makefile 自动化增量编译
9. **性能**：C++ 原生性能接近 NumPy；Python 循环慢 100–300 倍

C++ 设计思维：泛型编程 vs OOP

C++ 同时支持两种代码复用范式，它们的思路截然不同：

对比	OOP（第 25 讲）	泛型编程（本讲）
复用单位	类与继承层次	函数模板与类模板
多态方式	运行时多态（虚函数 + vtable）	编译时多态（模板实例化）
类型约束	基类接口（`virtual` 函数）	隐式约束（操作合法即可）
代码生成	一份代码，运行时分发	每种类型生成一份代码
开销	虚函数调用有间接查表开销	编译时展开，零运行时开销
错误发现	运行时（动态绑定）	编译时（实例化失败即报错）

C++ 设计思维：泛型编程 vs OOP

对比	OOP（第 25 讲）	泛型编程（本讲）
典型场景	“is-a” 关系（Dog is an Animal）	算法适用多种类型（sort<T>）

- OOP 通过**继承**和**虚函数**实现接口统一，适合有层次关系的类型
- 泛型编程通过**模板**实现编译时多态，适合算法与数据结构解耦
- 标准库是泛型编程的典范：容器、迭代器、算法三者通过模板连接，互不依赖
- **实际开发中两者常常结合使用**：OOP 定义领域模型，泛型编程提供通用算法