现代 C++ 笔记

HongShiyu

2025 年 9 月 21 日

目录

前言	1
第一部分 C++ 基础知识	2
第一章 “Hello World” 程序: C++ 入门经典示例	3
1.1 程序概述	3
1.1.1 验证开发环境有效性	3
1.1.2 掌握 C++ 基础语法框架	3
1.2 完整代码示例	4
1.3 核心概念拆解	4
1.3.1 注释: 代码的“自我说明文档”	4
1.3.2 头文件	6
1.3.3 #include 预处理器指令	6
1.3.4 主函数: 程序的“唯一入口”	7
1.3.5 标准库组件: <iostream> 与流操作	8
1.4 跨平台构建: 使用 Xmake	9
1.4.1 xmake.lua 配置解析	9
1.4.2 终端操作流程	10
1.5 关键误区与实践推荐	11
1.5.1 std::endl vs \n: 别滥用刷新缓冲区	11

前言

本文档聚焦现代 C++（C++11 及后续标准）的核心知识体系，系统整理了基础知识等关键点。其定位作为学习过程中的参考资料，旨在帮助开发者梳理知识脉络、巩固核心概念，同时为日常编码中的技术决策提供依据。

第一部分

C++ 基础知识

第一章 “Hello World” 程序：C++ 入门经典示例

1.1 程序概述

“Hello World” 程序是 C++ 入门阶段的经典示例，核心功能是在控制台输出字符串 "Hello, World!"。尽管代码简洁却承载着 C++ 开发的两大核心价值。

1.1.1 验证开发环境有效性

作为最精简的可执行程序，“Hello World” 程序能快速验证 C++ 工具链是否配置成功：

- 编译器（GCC、Clang、MSVC 等）能否正常解析 C++ 源代码
- 链接器能否正确处理依赖并生成可执行文件
- 操作系统与工具链（编译器、链接器、调试器）配置是否兼容

1.1.2 掌握 C++ 基础语法框架

通过拆解代码结构，可掌握 C++ 的基础概念：

- 注释的书写方式
- 头文件的引入逻辑与作用
- 预处理器指令的执行机制
- 主函数的“程序入口”特殊地位
- 函数返回值的语义
- 标准输出流（`std::cout`）的控制台交互逻辑

1.2 完整代码示例

```
// 引入标准输入输出流库，提供控制台 I/O 功能
#include <iostream>

/**
 * @brief 程序唯一入口：主函数
 *
 * 作为操作系统启动程序时的首个调用函数，演示标准输出流的基础用法，
 * 向控制台输出 "Hello, World!" 字符串，并返回程序退出状态码。
 *
 * @return int 程序退出状态码：0 表示执行成功，非零表示异常
 */
int main() {
    std::cout << "Hello, World!" << std::endl;

    return 0;
}
```

1.3 核心概念拆解

1.3.1 注释：代码的“自我说明文档”

注释是不参与编译的辅助文本，用于解释设计意图，提升可维护性。C++ 支持两种注释方式：

1.3.1.1 单行注释 (//)

- 语法规则：从 // 开始，到当前行末尾结束
- 适用场景：单行代码说明、局部逻辑注释
- 示例：

```
// 引入标准输入输出流库，提供控制台 I/O 功能（单行注释说明头文件用途）
#include <iostream>
```

1.3.1.2 多行注释 (/* ... */)

- 语法规则: 从 /* 开始, */结束, 支持跨越多行
- 适用场景: 函数文档、复杂逻辑说明、代码块注释
- 延伸: 文档化注释 (如 Doxygen 风格), 通过 /* ... */ 和标签 (@brief、@return、...) 生成结构化文档
- 示例:

```
/**
 * @brief 程序唯一入口: 主函数 (多行注释说明主函数功能)
 *
 * 作为操作系统启动程序时的首个调用函数, 演示标准输出流的基础用法,
 * 向控制台输出 "Hello, World!" 字符串, 并返回程序退出状态码。
 *
 * @return int 程序退出状态码: 0 表示执行成功, 非零表示异常
 */
int main() {
    std::cout << "Hello, World!" << std::endl;

    return 0;
}
```

i 注记

多行注释不可嵌套 (即一个 /* ... */ 内部不能包含另一个 /* ... */), 否则会导致编译器解析歧义, 触发编译错误。

1.3.2 头文件

头文件是用于声明函数、变量、类及库依赖的“接口说明书”，让编译器提前知晓实体定义，确保后续代码正常编译。

1.3.3 #include 预处理器指令

#include 是 C++ 预处理器的核心指令，作用是在编译前将目标文件的全部内容嵌入当前源文件，是代码模块化与复用的基础。

1.3.3.1 #include 的两种形式（搜索路径差异）

特性	#include <path>	#include "path"
搜索路径优先级	优先检索系统标准库路径	先检索当前源文件目录，再查系统路径
适用场景	引入 C++ 标准库头文件	引入用户自定义头文件
示例	#include <iostream>	#include "test.h"

1.3.4 主函数：程序的“唯一入口”

main 函数是 C++ 可执行程序的唯一入口——操作系统启动程序时，会自动调用 main 函数，无 main 函数会导致编译失败（如 MSVC 会报 error: LINK : fatal error LNK1561: 必须定义入口点的错误）。

1.3.4.1 主函数的两种形式

C++ 支持以下两种定义：

特性	<code>int main</code>	<code>int main(int argc, char** argv)</code>
说明	不接收命令行参数	接收命令行参数
适用场景	简单程序（如 Hello World）	需要外部输入的程序

1.3.4.2 命令行参数解析

若需通过命令行传递参数（如 `./app arg1 arg2`），需用 `int main(int argc, char** argv)` 形式：

- `argc` (argument count): 参数总数（含程序名本身，最少为 1）
- `argv` (argument vector): 参数字符串数组，`argv[0]` 为程序名，`argv[1] ~ argv[argc - 1]` 为实际参数

示例（打印命令行参数）：

```
int main(int argc, char **argv) {
    std::cout << " 程序名: " << argv[0] << std::endl;
    std::cout << " 参数个数: " << argc - 1 << std::endl;

    for (int i = 1; i < argc; ++i) {
        std::cout << " 参数" << i << ": " << argv[i] << std::endl;
    }

    return 0;
}
```

1.3.4.3 主函数的返回值

返回类型固定为 `int`，用于表示程序退出状态：

- 返回零值：程序正常执行结束（可省略 `return 0;`，编译器自动补全）
- 返回非零值：程序异常结束（异常值的含义由开发者自定义，需在文档中说明）

1.3.5 标准库组件：<iostream> 与流操作

<iostream> 是 C++ 处理标准输入输出的核心库，基于“抽象流”模型封装底层硬件，提供跨平台统一接口。核心对象包括：

对象	含义	绑定设备	特征	示例
<code>std::cin</code>	标准输出流	键盘	有缓冲，支持格式化读取	<code>std::cout << "Hi";</code>
<code>std::cout</code>	标准输入流	显示器	有缓冲（提升性能）	<code>int x;</code> <code>std::cint >> x;</code>
<code>std::cerr</code>	标准错误流	显示器	无缓冲（即时输出错误）	<code>std::cerr << "Error!";</code>

i 注记

`std::cout` 用于正常输出，`std::cerr` 用于错误信息——前者可能因缓冲延迟输出，后者确保错误即时显示（如程序崩溃前的日志）。

1.4 跨平台构建: 使用 Xmake

传统 Makefile 配置复杂, 现代 C++ 开发可以使用 Xmake(跨平台构建工具), 通过简洁的 `xmake.lua` 配置实现一键构建。

```
.
├── src
│   └── main.cpp # 源代码文件
└── xmake.lua   # Xmake 配置文件
```

1.4.1 xmake.lua 配置解析

```
-- 构建模式配置: 支持 debug (调试) 与 release (发布) 两种模式
-- debug 模式: 保留调试符号, 禁用代码优化, 便于断点调试与内存诊断
-- release 模式: 剥离调试符号, 启用 O2 优化, 生成高性能可执行文件
add_rules("mode.debug", "mode.release")

-- 自动生成 compile_commands.json 到 .vscode 目录
-- 为 VSCode 等 IDE 提供语义分析, 智能补全、静态检查等功能
add_rules("plugin.compile_commands.autoupdate", {outputdir = ".vscode"})

-- 定义项目构建目标: 名称为 "app"
target("app")
    -- 启用 C++20 标准
    set_languages("cxx20")
    -- 指定输出类型为二进制可执行文件
    set_kind("binary")
    -- 添加源文件: 递归收集 src 目录下所有 .cpp 文件
    add_files("src/*.cpp")
```

1.4.2 终端操作流程

```
# 1. 配置构建模式 (默认 release, 调试时需显示切换)
$ xmake f -m debug # f=configure (配置), -m=mode (模式)

# 2. 执行构建 (自动识别编译器, 如 GCC、Clang、MSVC)
$ xmake           # 编译 + 链接, 生成可执行文件

# 3. 运行程序
$ xmake run       # 执行执行构建产物
# 预期输出: Hello, World!
```

1.5 关键误区与实践推荐

1.5.1 `std::endl` vs `\n`: 别滥用刷新缓冲区

初学者常混淆 `std::endl` 与 `\n`, 两者差异直接影响 I/O 性能:

特性	<code>std::endl</code>	<code>\n</code>
核心功能	插入换行符 <code>\n</code> 并强制刷新缓冲区	仅插入换行符 <code>\n</code>
缓冲区行为	立即刷新缓冲区 (<code>flush()</code>)	不触发刷新 (依赖缓冲策略)
性能影响	高频调用 (如循环输出) 显著降低 I/O 性能	无额外开销, 性能更优
适用场景	需即时输出的场景 (日志、调试信息)	普通换行 (如批量输出)

1.5.1.1 性能验证

通过执行 100000 次输出操作, 对比两者的时间消耗, 验证性能差异 (测试环境: MSVC19.44.35211 + Win11, 编译模式: debug)。

- 测试代码

```
#include <chrono>
#include <cstdint>
#include <fstream>
#include <iostream>

// 测试使用 \n 进行输出的性能
int64_t test_newline(int iterations) {
    std::ofstream file("test_newline.txt");
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < iterations; ++i) {
        file << "Line " << i << "\n";
    }

    auto end = std::chrono::high_resolution_clock::now();
```

```
        return std::chrono::duration_cast<std::chrono::microseconds>(end - start)
            .count();
    }

// 测试使用 std::endl 进行输出的性能
int64_t test_endl(int iterations) {
    std::ofstream file("test_endl.txt");
    auto start = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < iterations; ++i) {
        file << "Line " << i << std::endl;
    }

    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::microseconds>(end - start)
        .count();
}

// 对比 \n 和 std::endl 进行输出的性能
void compare_newline_and_endl_performance(int iterations) {
    std::cout << " 测试次数: " << iterations << "\n";

    // 测试
    auto newline_time = test_newline(iterations);
    auto endl_time = test_endl(iterations);

    // 输出结果
    std::cout << " 使用 \\n 的时间: " << newline_time << " 微秒\n";
    std::cout << " 使用 std::endl 的时间: " << endl_time << " 微秒\n";
    std::cout << " 性能差异: " << (endl_time - newline_time) << " 微秒\n";
    std::cout << "std::endl 比 \\n 慢: "
        << (static_cast<double>(endl_time) /
            static_cast<double>(newline_time))
        << " 倍\n";
}

int main() {
```

```
compare_newline_and_endl_performance(100000);  
  
return 0;  
}
```

- 实测结果

测试次数: 100000

使用 `\n` 的时间: 180761 微秒

使用 `std::endl` 的时间: 659990 微秒

性能差异: 479229 微秒

`std::endl` 比 `\n` 慢: 3.65117 倍

1.5.1.2 实践推荐

- 普通换行用 `\n` (如 `std::cout << "Hi\n";`)
- 仅在需即时输出时使用 `std::endl` (如调试日志)