数据结构与算法（草稿）

Data Structures and Algorithms

(C++11)

程帆　沈赟

chengfan85@gmail.com　yun.shen3@gmail.com

2026 年 2 月 8 日

前言

著名计算机科学家 Niklaus Emil Wirth (图灵奖得主, Pacal 语言发明者, 15/02/1934 – 01/01/2024) 曾经说过一句名言: 程序 = 算法 + 数据结构。

对于一个初学者而言，计算机专业的最大的挑战就是编程能力。除了 TCS 的一些科目，计算机方向的原理都不算太难理解。很多情况下，初学者往往都是知道问题的解法，但是无法把想法落实到程序，有心无力。编程能力是一个计算机专业的学生的核心素养，也是初学者面前最大阻碍。提升编程能力，那就需要在算法和数据结构上下大功夫。这也是最高效的途径。

本教材是《计算导论》的后续，读者需要有基本的编程经验，熟练使用一门高级编程语言，例如 Python。本教材的主要内容是用 C++来讲授基本的数据结构与算法。基本工作环境是基于 Linux 命令行下的 GCC 编译器。语言上从基本的 C 开始，逐步讲授 C++（包括现代 C++11）的常用语法。数据结构的内容覆盖 C++的 STL 中的常用数据结构的原理与实现。算法部分则包含了《算法导论》中的经典算法。在此基础上，读者被要求去解决大量的算法 (Leet code, Online Judge 等等) 练习题。

虽然现在 AI 可以帮我们写代码，但是我们还是需要具备相当的编码能力和经验，这样才能更好的利用 AI 这个能力放大器。尤其是编程经验，这只能通过亲自动手解决各种复杂的问题来积累。本教材会帮助读者进行专业的算法和数据结构训练，从而彻底解决编码能力低的问题。

本教材是基于作者本人的计算机学习和教育的经验编写的，虽然没有公开在正式课堂上使用过，但是私下提供给一些交大同学内部训练过，反馈良好。基于计算机学科开放共享的精神，我决定将此教材公布在 github 公开使用。特此邀请沈赟博士从工业界的视角提供协助，一起完成该书的撰写。由于个人能力的缺陷，相信本书还有相当的漏洞，也欢迎读者不吝指出。在 github 评论，或者发邮件 chengfan85@gmail.com.

程帆

2025 年 12 月 20 日

目录

第 1 章　内容简介

1.1　C 与 Unix 系统内核

1969 年，AT&T 贝尔实验室的 Ken Thompson 和 Dennis Ritchie 等人启动了 Unix 操作系统的开发。最初，Unix 内核完全基于汇编语言构建，这虽然赋予了内核极高的运行效率，却也带来了巨大的工程挑战。由于当年的计算机架构正处于爆发期，不同机型之间的指令集（ISA）互不兼容。这意味着每当 Unix 需要迁移到新硬件上时，工程师们都必须使用该机型的汇编语言从零开始重写内核。面对 Unix 逐渐增长的代码量与逻辑复杂性，这种依靠人力维持的“硬件绑定”模式在开发成本上已难以为继。这正是计算机科学中一个永恒的主题：开发效率与运行效率的权衡（Trade-off）。

为了打破“硬件绑定”的枷锁，Ken Thompson 创造了 B 语言，尝试以此重写 Unix 内核。其核心策略是引入一个关键的抽象中间层：通过编写 B 语言编译器，将同一套高层逻辑映射为不同平台的特定汇编代码。在工程量上，针对新架构开发一个编译器，远比用汇编语言重写整个操作系统内核要轻量得多。这种“一次编写，处处编译”的雏形，不仅极大地提升了 Unix 的开发与迁移效率，更确立了软件工程中的核心范式——通过抽象层隔离底层硬件的复杂性。这一“中间层”思想不仅成就了后来的 C 语言，更深远地影响了后续数十年虚拟机（JVM）、中间语言（LLVM IR）以及现代跨平台系统的架构逻辑。

遗憾的是，由于 B 语言设计上的局限（尤其是其“无类型”特性难以适配新兴硬件的字节寻址），使用它重写 Unix 内核的尝试最终半途而废。在此基础上，Dennis Ritchie 引入了丰富的类型系统并创造了 C 语言，最终于 1973 年成功完成了 Unix 内核的重写。这一壮举在当时具有革命性意义：它彻底打破了“操作系统必须与特定硬件绑定”的铁律。

C 语言的成功在于它找到了一处绝佳的平衡点：

- 向下：它足够“薄”，能够直接映射硬件指令，提供媲美汇编的运行效率。
- 向上：它足够“通用”，通过编译器屏蔽了底层架构的差异，赋予了代码卓越的可移植性。

这种平衡使 C 语言成为了构建数字世界的“工业母机”。从统治服务器市场的 Linux 内核，到支撑现代互联网的 Nginx、处理全球视频数据的 FFmpeg，乃至 JVM 和 CPython 等核心运行时环境，其底层无一不跳动着 C 语言的脉搏。

尽管 Unix 曾因版权纠纷和闭源限制逐渐淡出大众视野，但它作为现代操作系统的“灵魂”，其设计哲学早已跨越了代码本身，深远地塑造了 macOS、Linux 以及 Android 等后世巨擘。

Unix 留给世界最宝贵的遗产是其简洁而强大的工程哲学：

- “只做一件事，并把它做好”（Do one thing and do it well）：强调模块的原子性与纯粹。

- “组合的力量”（The power of pipes）：通过简单的接口将各模块拼接，实现 “1+1 > 2” 的复杂功能。
- “一切皆文件”（Everything is a file）：将硬件、内存、进程抽象为统一的流，极大地简化了系统架构。

这种 “大道至简” 的思想在数十年后依然回响。当我们翻阅 *Zen of Python*，看到 “简洁胜于复杂”、“代码的可读性重于一切” 等准则时，不难发现，这正是 Unix 精神在现代编程语言中的跨时空共鸣。

Linux 在设计之初就严格遵循了 Unix 的哲学与标准，尤其是通过兼容 POSIX（可移植操作系统接口）标准，实现了与 Unix 生态的深度对接。这使得原本运行在 Unix 上的大量基础软件（如 `ls`、`grep`、`bash` 等）能够以极低的成本移植到 Linux 环境中，极大地丰富了 Linux 早期的软件生态。

而 macOS 与 Linux 之间的 “神似”，则源于更近的血缘关系。macOS 的内核 Darwin 深度借鉴了 Unix 的 BSD（Berkeley Software Distribution）分支。正因如此，尽管 Linux 和 macOS 的内核架构截然不同（一个是宏内核，一个是混合内核），但在用户层面的命令行工具、权限管理和系统接口上，两者展现出了高度的兼容性。这种基于 Unix 传统的共性，也是现今广大开发者青睐这两种系统的重要原因。

1.2 C 语言简介

C 语言在诞生之初被冠以 “高级语言” 之名，但这主要是在与汇编语言及机器指令的 “肉搏” 中赢得的称谓。若将其置于 C++、Java 或 Python 的坐标系下，C 语言的原始性便暴露无遗。在当前的计算机专业教育中，若仅将 C 作为一门常规高级语言进行授课，往往会陷入 “只看语法、不看底层” 的误区，从而彻底剥离了它的精髓。从语法特质来看，C 语言更像是一门 “带语法的汇编” 或 “高级汇编语言”。它的力量不在于复杂的抽象机制，而在于它对计算机架构近乎透明的映射。只有当你意识到 C 语言中的指针就是地址、结构体就是内存布局、函数调用就是堆栈跳转时，你才能真正领悟它的灵魂。将 C 脱离硬件背景孤立使用，无异于买椟还珠。

作为一门经久不衰的系统级编程语言，C 语言以其极简的语法构建了通往硬件底层的直梯。其核心哲学在于 “信任程序员”（Trust the programmer）：它拒绝任何隐式的行为兜底，坚决不让运行性能为潜在的编码错误买单。这种哲学将极致的控制权悉数交予开发者，使 C 语言在资源受限或性能敏感的场景下拥有无可比拟的优势。然而，权力的背面是责任。在 C 的世界里，程序员必须化身为严谨的资源管理者，亲手操纵内存、句柄与并发。这种 “所见即所得” 的特性，既成就了 C 语言如 “锋利手术刀” 般的精准与高效，也使其成为了初学者的 “双刃剑”——稍有疏忽，便可能被内存溢出或悬空指针所伤。深入理解 C 语言的演进史与其底层特质，不仅是为了写出更高效的代码，更是为了建立起一种深邃的系统思维，在享受极致掌控感的同时，凭借工程经验规避其原始性带来的缺陷。

1978 年，由 Brian Kernighan 与 Dennis Ritchie 合著的经典巨作《The C Programming Language》（简称 K&R）正式出版。此书的问世不仅确立了 C 语言在程序员心中的地位，更成为了当时 C 语言的 “事实标准”。然而，随着 C 语言的广泛传播，各硬件厂商开始根据自身需求衍生出功能各异的版本，导致代码移植性陷入混乱。

为了终结这种割裂，标准化进程应运而生：

- C89（ANSI C / ISO C90）：1989 年发布的这一标准统一了语法，标志着“经典 C 语言”时代的开启。
- C99：1999 年发布，引入了如单行注释 //、变长数组（VLA）及布尔类型等实用特性，是目前工业界使用最广、最平衡的标准之一。
- 现代标准（C11/C17/C23）：进入 21 世纪后，C 语言在保持底层定位的基础上，针对并发支持（C11）和语法冗余（C23）进行了微调与加固。

由于 C 语言承担着数字世界底层基石的重任，其语法核心早已趋于稳定。本教材将主要基于 C99 标准进行教学，因为它在保留 C 语言纯粹性的同时，提供了最契合现代编程习惯的语法支持。

虽然 C 为了极致的兼容性舍弃了诸多高层抽象（类、继承、模板、异常、RTTI、重载等），但其图灵完备（即在计算能力上等价于一台图灵机）的特性决定了它在计算能力上与 C++、Python 等后来者平起平坐，且在底层互操作性上拥有不可撼动的统治地位。C 在工程实践中拥有一项 C++ 难以企及的核心优势：极高的 ABI（应用程序二进制接口）稳定性。

严格来说，C 标准本身并不定义 ABI，但由于 C 语法简洁、底层透明，它早已成为各大操作系统与硬件架构（如 x86-64、ARM64）下事实上的二进制标准。在同一平台下，无论是 GCC、Clang 还是 MSVC，其生成的二进制文件在函数调用约定（Calling Convention）和数据内存布局上都高度一致且长期预测。

相比之下，C++缺乏统一的 ABI 标准。其复杂的语言特性（如 Name Mangling 名字修饰、虚函数表管理、模板实例化等）导致不同编译器、甚至同一编译器的不同版本之间都难以实现二进制兼容。

这种“大道至简”的特性，让 C 成为了计算机世界的“标准螺纹”：

- 跨语言之桥：Rust、Go、Python、Java 等几乎所有高级语言，都将 C 接口（FFI）作为跨语言绑定的基础协议。
- 底层基石：像 Linux 内核、CPython 解释器、JVM 虚拟机等需要极致兼容性的系统级项目，至今仍将 C 作为核心开发语言。

C 的简单反而赋予了它最强大的互操作性。它就像工业界的通用标准件，无论生产商是谁，都能严丝合缝地拧在一起；而 C++ 虽然功能强大，却更像精密但规格各异的定制件，互相之间往往难以直接适配。

1.3 C++语言简介

随着问题复杂性的提升，代码量也随之激增，此时需要类、包等语法机制来隔离并管控复杂性。C 在大型软件工程中也面临同样的挑战，C++ 正是在此背景下应运而生。1979 年至 1983 年间，贝尔实验室的 Bjarne Stroustrup 在 C 的基础上开发了 *C with Classes*（即 C++ 的前身）。他借鉴了 Simula 语言中“类”的思想，旨在为 C 引入高级抽象能力。

在标准演进上，C++几乎完全兼容 C89/C90 语法，通常可以将 C 视为 C++的一个功能子集。尽管 C90 之后的 C 语言标准独立演进（如 C11、C17），但现代 C++依然保持了对这些新版本的高度兼容，确保了 C 生态中的大量遗留代码能顺畅地迁移至 C++环境中。

1998 年，C++发布了第一个 ISO 国际标准（C++98），其中里程碑式的改进便是引入了 STL（标准模板库）。STL 的出现让 C++从一门“带类的 C”真正蜕变为现代通用编程语言。它首次将*泛型编程*（*Generic Programming*）带入主流视野，并构建了“容器 + 迭代器 + 算法”这一黄金架构。通过迭代器这一桥梁，STL 成功实现了算法与数据结构的解耦：算法不再关心数据的存储细节，数据结构也不再关心如何处理数据。

这也是为什么 C++ 是学习“数据结构与算法”的绝佳工具：

1. 性能与抽象的平衡：它既保留了贴近硬件的高性能，又通过 STL 提供了高度抽象的现成工具。

2. 符合认知规律：初学者在尚未掌握语言基本语法时，若被强行要求手动实现链表或红黑树，会造成巨大的认知负荷（Cognitive Load）。C++ 允许学习者先通过 STL 以“黑盒”方式调用成熟的数据结构，在解决实际问题的过程中建立成就感，待时机成熟后再深入底层探究其实现原理。

相比之下，C 语言不提供标准数据结构，迫使开发者在“学走路”的阶段就得先“造轮子”。这种违反认知心理的路径，虽然磨练意志，却极大地拉高了普通人的学习门槛。

在经历了长达十余年的沉寂后，C++ 从 2011 年起迎来了“文艺复兴”。随着 C++标准委员会确立了每三年发布一个新版本的节奏，C++正式跨入“现代 C++” (Modern C++) 时代。正如社区常说的，“C++11 像是一门全新的语言”，它彻底重构了开发者的思维模式。而随后的 C++14/17/20/23 则在此基础上不断精进，持续强化了语言的表达力、安全性和并发处理能力。

现代 C++ 展现出了极强的包容性与进化力：

- 吸收先进范式：现代 C++ 展现出了极强的演进动力，通过博采众长大幅提升了开发者体验：它既借鉴了 Python 等语言的易用性，引入如 `std::println()`（C++23）这样简洁直观的 IO 风格；又深度拥抱函数式编程范式，通过 `std::optional`（C++17）提供安全的值处理，并利用 Ranges 库（C++20）的管道操作符实现类似 Haskell 或 Scala 的流式声明式编程。这种进化使 C++ 在保持底层高性能的同时，拥有了足以媲美现代高级语言的优雅表达力与工程安全性。

- 内存安全风险的缓解：针对 C/C++ 长期被诟病的内存错误（由于缺乏自动垃圾回收），现代 C++ 并不通过引入 GC（垃圾回收），而是通过 智能指针（Smart Pointers）、移动语义（Move Semantics）和 RAII（资源获取即初始化）机制，实现了极低开销甚至是零开销的内存安全管控，极大地缓解了悬空指针和内存泄漏的问题。

这种进化让 C++成功保住了“底层高性能”的护城河，同时在“高层开发效率”上缩小了与后来者的差距。

C 与 C++虽然血脉相连，但在设计哲学、应用场景及工程范式上已演化为两门完全不同的语言，并在各自的领地各司其职。长期以来，国内外的教学体系往往陷入一个误区：过度

强调 C++ 的面向对象（OOP）特性，仿佛 C++ 仅仅是“带类的 C”。事实上，C++ 是一门极具包容性的多范式语言（Multi-paradigm Language），它完美融合了过程式、面向对象、泛型编程以及现代函数式编程。与其称之为“应用级语言”或“高级 C”，不如将其定义为“多层次系统语言（Multi-level System Language）”：

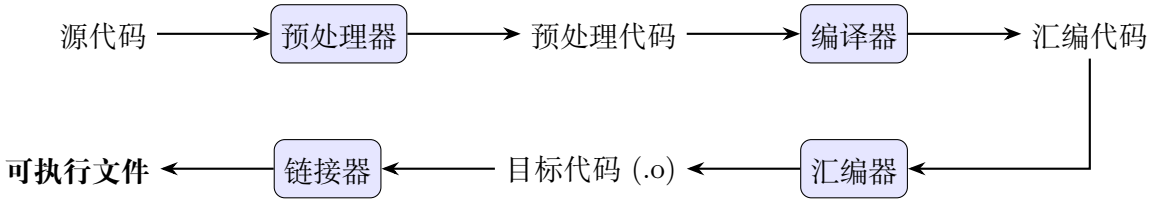
- 向下看：程序员可以像写 C 一样，直接操纵内存、精细控制底层硬件指令。
- 向上看：程序员可以利用 RAII、元编程（TMP）、Ranges 等高阶抽象来管控复杂逻辑。

这一切抽象的核心都在于 Bjarne Stroustrup 坚持的基石信条：“不为不使用的东西买单”（You only pay for what you use）。在理想情况下，C++ 的高层抽象在编译后会被转化为与手动编写的 C 代码等效的机器指令，从而实现“零运行时开销”。这种在极致性能与高层表达力之间反复横跳的能力，正是 C++ 跨越 40 年依然无可替代的生命力所在。

向后兼容性（Backward Compatibility）是 C++ 的立身之本，亦是其最沉重的历史枷锁。面对全球数十亿至上百亿行的遗留代码，C++ 标准委员会必须恪守“不破坏旧代码”的底线。这种承诺确保了软件资产的长期价值，却也迫使 C++在面对设计缺陷时无法推倒重来（如 `std::vector<bool>`），只能通过不断叠加新特性（如 `std::span`）来提供替代方案。这种“只增不减”的策略导致 C++ 变得愈发繁杂，宛如一座在旧地基上不断扩建的宏伟宫殿。程序员在享受现代特性带来的表达力时，必须时刻警惕脚下那些为了兼容性而保留的历史陷阱。

1.4 编译器

传统意义上，C 和 C++ 属于典型的编译型语言。其核心特征是在程序执行前，必须经历一条完整的流水线转换：



这种模式奠定了它们“静态”的特质：所有复杂的类型检查和逻辑优化都在 编译阶段（*Compile-time*）完成，从而确保了运行阶段（*Runtime*）的极致速度。

虽然现代技术如 JIT（即时编译）模糊了编译与解释的界限，但理解 C 语言这种“先编译、后执行”的传统模式，依然是掌握系统级资源管控和理解计算机底层运作逻辑的基石。

常见 C/C++ 编译器与开发环境

目前，C/C++ 领域主要由三大编译器阵营统治，分别对应不同的操作系统生态：

- GCC (GNU Compiler Collection)：Linux 平台的绝对霸主。诞生于 1987 年，是 Richard Stallman 主导的 GNU 项目的核心产物。它不仅是一套自由软件，更是开源世界的基石。
- MSVC (Microsoft Visual C++)：Windows 平台的标准配置。它深度集成的二进制接口（ABI）确保了 Windows 程序的稳定运行。

- Clang/LLVM：macOS 的默认选择。作为 GCC 的后起之秀，Clang 凭借 LLVM 架构的模块化设计，在编译速度、错误提示的友好度以及静态分析能力上实现了“弯道超车”。

历史的余晖与工业的选择

曾经叱咤风云的 Turbo C/C++ 和 Borland C++ 如今已退居二线。值得一提的是，Borland 的核心灵魂 Anders Hejlsberg 后来加入微软，主导开发了 C# 和 TypeScript，深刻影响了现代编程语言的形态。而在自动驾驶、气象预测等高性能计算（HPC）领域，Intel C++ Compiler 和 NVIDIA NVCC 则是追求极致算力的首选。

Table 1.1: 集成开发环境（IDE）建议

平台	推荐 IDE	特点
Windows	*Visual Studio*	调试功能极其强大，适合大型工程。
macOS	*Xcode*	开发苹果生态（iOS/macOS）应用的必选。
跨平台	*VS Code*	轻量、插件丰富，配合 `C/C++ Extension` 极为灵活。

1.5 回归最纯粹的编程范式：Linux + GCC

本教材选择“Linux 命令行 + GCC 编译器”作为核心开发工具。

我们建议学习者拥抱这种看似“传统”实则“高效”的编程方式。在算法与数据结构的学习中，脱离笨重的图形化界面（IDE），能让你更专注地理解编译流程与内存细节。

编辑器选型：Vim 的力量

在文本编辑器方面，我们强烈推荐 Vim。它不仅是一个工具，更是计算机专业的必备技能点。

- 策略建议：不要试图一次性背诵所有快捷键。先掌握基础的“增删查改”，然后在编写代码的过程中，通过 AI 工具或手册按需扩展。这种“在战争中学习战争”的路径，是掌握 Vim 的捷径。

跨平台方案：顺滑的迁移体验

- macOS 用户：你们拥有天然的类 Unix 环境。虽然你在终端输入 gcc 时，系统可能默认调用的是 Clang，但对于本教材的 C99 标准代码，两者的表现基本一致。
- Windows 用户：推荐使用 WSL2 (Windows Subsystem for Linux)。这是微软近年来最受开发者好评的功能，它允许你在不损失 Windows 便利性的前提下，运行一个完整的、高性能的 Linux 内核。

技术补充：为什么命令行对算法学习有益？

1. 强制性代码记忆：离开 IDE 的自动补全，能让你对标准库（如 <stdio.h>）中的函数签名印象更深。

2. Makefile 与工程化：掌握命令行编译后，你可以更容易地理解大型项目是如何通过 make 或 cmake 进行管理的。

3. 调试深度：配合命令行调试器 gdb，你能以更接近硬件的视角去观察内存中的数据结构（如链表的指针指向）。

WSL 的安装与检查

在 Windows 终端（PowerShell 或 命令提示符）中输入：

WSL 检查 *Terminal*

```
wsl --list --online
```

如果返回了分发版列表，说明 WSL 核心已就绪。若提示命令不存在，请按以下步骤快速安装：

1. 以管理员身份打开 PowerShell。

2. 输入以下命令（适用于 Windows 11 或更新版本的 Windows 10）：

   WSL 安装 *Terminal*

   ```
   wsl --install -d Ubuntu-24.04
   ```

3. 安装完成后，请根据提示重启计算机。

C++ 交互式环境

C++ 也可以像脚本语言一样使用。在 Ubuntu 下安装 Cling 后即可解释执行 C++ 代码。对于喜欢交互式编程的同学，还可以配置 Jupyter Notebook 结合 Cling 内核，实现类似 Python 的即时反馈体验。交互式环境非常适合用来观察数据结构（如链表、树）在每一步操作后的内存状态变化。

如果你希望免安装快速上手，可以使用在线编译器：https://www.onlinegdb.com/。

gcc 与 g++

编译器驱动程序：gcc 与 g++

简单来说，gcc 和 g++ 并不是两个独立的编译器，它们都是 GNU 编译器套件（GNU Compiler Collection, GCC）里的“驱动程序”（Drivers）。

gcc (小写)：C 语言编译器，默认链接 C 标准库（libc）。

g++：C++ 编译器，默认链接 C++ 标准库（libstdc++）。

虽然 gcc 也可以编译 C++ 代码，但由于它默认不链接 C++ 标准库，往往会导致链接错误。因此，本书严格遵循：C 代码用 gcc 编译，C++ 代码用 g++ 编译。

环境准备

在 Ubuntu (含 WSL) 下，可以通过以下命令安装完整工具链：

gcc/g++安装 *Terminal*

```
sudo apt update
sudo apt install build-essential -y
```

编译示例

helloworld.c

```c
#include <stdio.h>

int main(void){
    printf("hello world, C!\n");
    return 0;
}
```

假设有 C 源文件 helloworld.c，编译与运行指令如下：

gcc 编译与运行 *Terminal*

```
gcc helloworld.c -o helloworld
./helloworld
```

参数详解：

- `gcc`：编译器命令。
- `helloworld.c`：源文件名（C++ 建议使用 `.cpp`，C 建议使用 `.c`）。
- `-o helloworld`：指定输出（output）文件名。若省略，Linux 下默认生成 `a.out`。
- `./`：表示运行当前目录下的可执行文件。

1. 开启编译警告（-Wall）

在算法学习中，C 语言对内存的操作非常直接，但也容易出错（如指针未初始化）。开启警告能让编译器帮你发现这些低级错误。

在编译时加上 -Wall 参数（Warning all），编译器会开启几乎所有常用的语法警告。强烈建议在学习阶段始终带上这个参数，它能帮你提前发现诸如“数组越界风险”、“变量定义了但没使用”等隐患。

带警告的 C 编译 *Terminal*

```
gcc -Wall helloworld.c -o helloworld
```

2. 指定 C 标准（-std=c11 或 -std=c17）

C 语言虽然古老，但也有不同的标准（如 C89, C99, C11）。如果不指定，旧版编译器可能会默认使用 C89，这会导致你在函数中间定义变量时报错。

现代 C 语言编程推荐使用 C11 或 C17 标准。为了防止编译器因标准版本过旧而不支持某些语法（例如在 for 循环内定义变量），建议显式指定：

指定标准编译 *Terminal*

```
gcc -std=c17 -Wall helloworld.c -o helloworld
```

3. C 语言编译的四个阶段（原理补充）

前面我们介绍过编译器如何把“源码如何变成程序”，具体到 gcc/g++编译器而言。

预处理 (Preprocessing): 处理以 # 开头的指令（如 #include），把头文件内容插入代码。

编译 (Compilation): 将 C / C++ 代码翻译成汇编语言。

汇编 (Assembly): 将汇编语言转换成机器能懂的二进制目标文件（.o 或 .obj）。

链接 (Linking): 将你的目标文件与系统库文件（如 stdio 库）合并，生成最终的可执行文件。

以上四个阶段都可能报告错误，要进行针对性分析处理。

4. 针对 Mac 平台的 C 编译器补充

在 Mac 上，情况稍有不同：

在 Mac 的终端输入 gcc，系统调用的实际上是 Apple 的 Clang 编译器。虽然它也支持上述的 -Wall 和 -std=c11 参数，但如果你在运行结果中看到 Apple clang version... 字样，请不要感到惊讶。对于本课程的算法练习，Clang 与 GCC 的表现几乎完全一致。

5. 综合编译指令推荐

我们推荐读者使用如下“全能命令”来编译课本中的 C 程序：

gcc 推荐编译方式 *Terminal*

```
gcc -std=c17 -Wall -O2 helloworld.c -o helloworld
```

参数补充说明：

-O2: 优化开关。它会让编译器对你的算法进行优化，使程序运行得更快。

6. 从 gcc 到 g++

以上 gcc 编译 C 源代码的原理对于 g++ 编译 C++ 源文件也同样成立，除了将 gcc 替代为 g++, 文件名从 .c 替代为 .cpp , 以及 c 参数替代为 c++ 外。

g++推荐编译方式 *Terminal*

```
g++ -std=c++17 -Wall -O2 helloworld.cpp -o helloworld
```

读者可以测试如下 C++ 代码。

helloworld.cpp

```cpp
#include <iostream>

int main(){
  std::cout << "hello world, C++!" << std::endl;
  return 0;
}
```

Table 1.2: C 与 C++ 编译参数对比

特性	C 编译 (gcc)	C++ 编译 (g++)
源文件后缀	`.c`	`.cpp` / `.cc` / `.hpp`
核心标准	`-std=c17`	`-std=c++17`
标准库链接	默认 `libc`	默认 `libstdc++` (Linux) / `libc++` (Mac)
常用优化	`-O2`	`-O2`

Make 与 CMake

当项目涉及多个源文件时，手动编译不仅低效且极易出错。为此，我们需要引入自动化的构建系统：

- **Make 与 Makefile**：`Make` 是一个构建工具，它根据 `Makefile` 文件中定义的规则，自动判断哪些文件需要重新编译并执行链接。它的特点是直接、底层且精确，但手动编写复杂的 `Makefile` 往往非常痛苦。
- **CMake**：是一个更高层的“构建系统生成器”（Meta-build System）。它不直接编译代码，而是通过读取 `CMakeLists.txt` 配置文件，自动生成对应平台的构建脚本（如 Linux 下的 `Makefile` 或 Windows 下的 `Visual Studio` 项目文件）。

简单来说：CMake → 生成 Makefile → Make 执行构建。这种组合既保持了构建的跨平台性，又极大减少了开发者的手动维护工作。

1.6 数据结构与算法

$$\text{Programs} = \text{Algorithms} + \text{Data Structures}$$

过去的教材经常将算法与数据结构割裂开来，这无形中增加了学习的门槛。本质上，两者是不可分割的统一体。将它们融合讲解，不仅符合逻辑直觉，更能体现现代软件开发的实战思路。

- **语言基础**：本教材首先夯实 C/C++ 语法，确保读者拥有稳固的编码底座。
- **以 STL 为桥梁**：数据结构的内容在 C++ 标准模板库（STL）中已有成熟实现。我们将采用“从黑盒到白盒”的策略：先掌握其用法与复杂度特性，再深入剖析底层原理，最后指导读者手写实现。这种方式不仅能学会数据结构，更能深度理解 C++ 编程精髓。

- **经典算法回归**：我们将以《算法导论》（CLRS）为蓝本，精选其实用、经典的算法进行深度拆解，并补充现代工程中的新进展。
- **实战评测（Online Judge）**：每章配有严选的在线题库。这些题目不仅考查逻辑，更有严苛的时空复杂度限制。通过在 OJ 上的反复磨练，大家不仅能积累调试经验，更能培养严谨的算法思维。

没有算法与数据结构的程序是没有灵魂的！

1.7 语言之魂与系统之基

软件工程的奠基者：Ken Thompson & Dennis Ritchie

1983 年图灵奖（A.M. Turing Award）获得者

表彰他们在“操作系统通用理论，及尤其是对 *Unix* 操作系统的实现”方面所做出的决定性贡献。

这对“黄金搭档”在贝尔实验室（Bell Labs）的合作，被公认为是现代信息时代的开端。

- **Ken Thompson**：Unix 的开拓者。他不仅发明了 B 语言，还创造了 UTF-8 编码，甚至是 Go 语言的核心设计师。
- **Dennis Ritchie**：C 语言之父。他创造了 C 语言并用它重写了 Unix，奠定了系统的可移植性基础。

> “学习一门新编程语言的唯一方法就是用它编写程序。”
> — The only way to learn a new programming language is by writing programs in it.

C 家族的拓荒者：Bjarne Stroustrup

作为 C++ 的创造者，**Bjarne** 始终致力于在极致性能与高级抽象之间寻找平衡。他对 C 和 C++ 差异的精准评价已成为经典：

> “*C* 让误伤自己变得容易；*C++* 虽让这变难了，但一旦发生，它会轰掉你整条腿。”
> — C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

工具与架构的大师：Anders Hejlsberg

他的一生横跨了三个极其成功的商业编译器与语言：

Turbo Pascal / Delphi ： 将快速应用开发（RAD）推向巅峰。

C# ： 为微软帝国打造了工业级的生产力工具。Anders 凭借深厚的架构功底，成功将 C++ 的严谨与 Delphi 的开发效率在 C# 中完美统一，从而在‘极致控制’与‘快速生产’之间为开发者开辟了第三条道路。

TypeScript ： 用“静态类型”的思想拯救了混乱的 JavaScript 体系。

大师启示录

与其仰望星空，不如躬身入局。正如丹尼斯 · 里奇所言，学习编程唯一的捷径就是“写程序”。

当你第一次在 Linux 终端敲下 gcc，你不仅是在编译代码，更是在与那些伟大的灵魂进行一场跨越时空的对话。

带上这把名为“C 语言”的手术刀，去构建属于你的数字世界吧。

1.8 检查清单

1. 安装 Linux
2. 安装 GCC，确保 gcc，g++可运行
3. 学会基本的 Vim 操作：打开、编辑、保存、关闭文件。在三种模式中切换。