Wirth、Ritchie 與 Stroustrup：簡潔、機器控制與相容性之間的三種系統語言倫理

Wirth, Ritchie, and Stroustrup: Three Ethics of Systems Language Design Across Simplicity, Machine Control, and Compatibility

v1.0 / 2026-07-30 / Neo.K / 公開版／第四部跨設計師正式比較研究

https://thisoneisneok.com/html/papers/pldst-023.html

英文名稱： Wirth, Ritchie, and Stroustrup: Three Ethics of Systems Language Design Across Simplicity, Machine Control, and Compatibility

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-023

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第四部跨設計師正式比較研究

SECTION

摘要

Niklaus Wirth、Dennis Ritchie 與 Bjarne Stroustrup 都設計過能處理系統級問題、重視效率且深刻影響工程實踐的語言。然而，若只以「Pascal／Oberon、C、C++」的功能表比較三人，就會忽略最重要的差異：他們對語言設計者應保護什麼、允許什麼、刪除什麼，以及把複雜度交給誰，具有三套不同倫理。

本文所謂「倫理」不是對個人人格作道德評價，而是指一種規範性責任配置：

> 當語言不可能同時最大化簡潔、效率、安全、相容、抽象、可攜與生態規模時，設計者認為哪些價值不可犧牲，哪些成本可以轉移，又由誰承擔後果？

Wirth 的核心要求是：

> 【整個語言與系統必須小到仍可被理解、實作、教授和重建】

他在 Modula-2／Oberon 回顧中把設計簡潔稱為最重要指導原則，並將概念清楚、功能節制、實作效率與可靠性視為其結果。設計 Oberon 時，其策略是先決定從 Modula-2 刪除什麼，再決定必須加入什麼，以提高能力並同時降低複雜度。[R1] Project Oberon 更由 Wirth 與 Jürg Gutknecht 從零建造語言、編譯器、作業系統和工作站環境，使完整系統可以在一本書中被說明和理解。[R2]

Ritchie 的核心要求是：

> 【保留系統程式所需的機器控制，
>
> 同時把大多數程式邏輯從單一硬體中解放】

C 從 BCPL 與 B 演化而來。Ritchie 保留 B 的小型表面，加入適合 PDP-11 的型別、陣列／指標規則、結構及編譯器，使 Unix 能在 1973 年由組合語言轉向主要以 C 實作。C 的抽象不是要隱藏機器，而是建立一層足以跨機器保存系統結構、又不阻止位元、位址、布局和外部程式互操作的薄介面。[R3][R4]

Stroustrup 的核心要求是：

> 【程式設計者不應被迫在高階抽象、機器控制、效率與既有程式之間四選一】

C++ 的原始目標是把 Simula 的程式組織能力與 C 的系統效率、彈性結合，並在真實專案中迅速可用。[R5] 後來的零額外成本原則要求：未使用的功能不支付成本；使用的抽象不應比合理手寫低階方案更昂貴。[R6] 但 C 相容、多範式和長期工業採用也使 C++ 保存大量歷史能力及交互複雜度。

本文以七個共同軸比較三人：

- 設計起點；
- 抽象高度；
- 硬體暴露；
- 安全與錯誤責任；

- 相容性；
- 完整系統驗證；
- 治理權力與長期演化。

本文提出的核心比較是：

> 【Wirth：以刪減保護可理解性;
> Ritchie：以薄抽象保護可攜控制;
> Stroustrup：以零額外成本擴張保護既有能力】

三者並不是線性進步關係，而是在不同制度條件下求解不同優化問題：

- Wirth 多在小型研究與教育團隊中，可以建立新語言和完整新系統；
- Ritchie 在 Bell Labs 小型研究共同體中，讓語言與 Unix 共演化，再由外部移植驗證；
- Stroustrup 面對龐大 C 生態、工業專案、多領域需求和國際標準，必須把相容視為政治與工程契約。

因此，三人的差異不能只歸因於品味，也由歷史可行空間決定。若忽略此點，便會得到錯誤結論：

- Wirth 看似只因偏愛簡潔而拒絕功能；
- Ritchie 看似只因信任高手而忽略安全；
- Stroustrup 看似只因無法拒絕功能而造成複雜。

更精確的結論是：

> 三人各自建立了一套「誰有權控制機器、誰應承擔錯誤、誰可以破壞相容、誰必須理解整個系統」的系統語言憲法。

關鍵詞： Niklaus Wirth、Dennis Ritchie、Bjarne Stroustrup、Oberon、C、C++、系統語言、簡潔、可攜性、零額外成本、相容性、PLDST

SECTION

一、倫理不是功能清單

語言功能表可以回答：

- 有沒有 Module；
- 有沒有 Pointer；
- 有沒有 Class；
- 有沒有 GC；
- 有沒有 Template。

  但它不能回答：

- 為什麼一項能力進入核心；
- 為什麼另一項被刪除；
- 誰應理解成本；
- 誰承擔不安全；
- 是否允許打破舊程式；
- 語言是否應統治完整系統；
- 設計者如何退出治理。

SECTION

二、責任配置向量

本文將系統語言倫理表示為：

> cal-E(L)=
> (
> B_u,
> B_c,
> B_r,
> B□,
> B_e,
> B_g

其中：

- B_u：使用者承擔的負擔；
- B_c：編譯器承擔的負擔；
- B_r：Runtime／OS 承擔的負擔；
- B□：工具鏈承擔的負擔；
- B_e：生態與 Library 承擔的負擔；
- B_g：治理和標準制度承擔的負擔。

  沒有語言能消除全部負擔，只能重新分配。

SECTION

三、價值向量

再定義：

> cal-V(L)=
> (
> U,
> M,
> A,
> S,
> P,
> K,
> G
>)

其中：

- U：Understandability，可理解性；
- M：Machine control，機器控制；

- A：Abstraction power，抽象能力；
- S：Safety enforcement，安全強制；
- P：Portability，可攜性；
- K：Compatibility，相容性；
- G：Governability，可治理性。

  三位設計者都重視這些價值，但排序不同。

SECTION

四、歷史約束集合

語言設計並非在真空中求最大值。

令：

> Ω=
> (
> TeamSize,
> InstalledBase,
> Hardware,
> UseCase,
> Institution,
> Time,
> MigrationCost
>)

同一設計者在不同 Ω 下也可能作出不同決定。

SECTION

五、Wirth：從概念過量開始

Wirth 的問題常是：

- 既有語言含有太多非本質功能；
- 學生和實作者無法完整掌握；
- Compiler、OS 與硬體之間失去可讀因果鏈；
- 大型系統的每一層都依賴無人能完整解釋的機制。

其第一問是：

> 哪些概念可以刪除，而不損失真正能力？

SECTION

六、Ritchie：從機器和系統缺口開始

Ritchie 的問題是：

- B 的無型別 Word 模型無法自然使用 PDP-11；
- Unix 若永久以組合語言實作，便無法跨機器保存；
- 高階語言若隱藏過多機器能力，又無法實作 Kernel、Driver、Compiler 和 Runtime。

其第一問是：

> 哪一層抽象足以讓 Unix 可攜，又不奪走系統程式設計需要的控制？

SECTION

七、Stroustrup：從抽象與效率斷裂開始

Stroustrup 的問題是：

- Simula 能組織大型程式，但當時的效能、實作和系統整合不足；
- C 能控制機器，但缺少大型程式抽象；

- 真實使用者已有大量 C 程式、Library、Compiler 和組織知識。

  其第一問是：

> 如何加入高階抽象，又不要求使用者放棄 C 世界或支付不必要成本？

SECTION

八、三種起點的形式差異

> Wirth ：ExistingConcepts - Nonessential;
> Ritchie ：MachineSpecificSystem + PortableStructure;
> Stroustrup ：ExistingSystemLanguage + AffordableAbstraction

SECTION

九、簡潔的地位

Wirth 將簡潔視為：

- 語言設計準則；
- 編譯器品質條件；
- 教育條件；
- 系統可靠性條件；
- 安全及審計條件。

  這不是表面字數，而是概念經濟。

SECTION

十、先刪後加

Oberon 的策略為：

先問 Modula-2 哪些功能可刪

再問缺少哪些必要能力

而非：

列出所有使用者可能想要的功能

再全部整合

SECTION

十一、提高能力同時降低複雜度

Oberon 的重要目標不是單純縮小，而是：

> Capabilityuparrow
>
> Complexitydownarrow

Type extension 是高覆蓋新增；多項低覆蓋或交互成本高的型別、模組選項和語法則被移除。

SECTION

十二、完整系統作為證明

Project Oberon 不以 Microbenchmark 或 Toy compiler 證明語言，而是建立：

- Compiler；
- Module loader；
- File；
- Text；
- Viewer；
- Network；
- Graphics；
- Operating system；
- 後來的 RISC processor。

因此：

> LanguageAdequacy
> ⇐
> WholeSystemImplementability

SECTION

十三、設計者的義務

在 Wirth 模型中，語言設計者應能：

- 清楚定義每項概念；
- 親自或以小團隊實作；
- 說明 Compiler；
- 建立真實系統；
- 提供可重建教材；
- 為每個功能的必要性辯護。

SECTION

十四、使用者承擔什麼

使用者仍需：

- 明示型別；
- 明示資料結構；
- 明示 Module；
- 明示控制流程；
- 在低階邊界管理資源；

- 接受語言不替每項便利建立內建功能。

SECTION

十五、Wirth 願意犧牲什麼

- 向上相容；
- 功能廣度；
- 既有生態；
- 多種寫法；
- Vendor extension 的便利；
- 全球平台的完整需求覆蓋。

SECTION

十六、Wirth 拒絕犧牲什麼

- 概念一致；
- 可教學；
- Compiler 可理解；
- 全系統因果鏈；
- 小型可靠實作；
- 設計者對功能的舉證責任。

SECTION

十七、C 的抽象不是隱藏機器

C 暴露：

- Pointer；
- Integer；
- Array；
- Structure；
- Bit operation；
- Explicit conversion；
- Control flow；
- External linkage。

  但把：

- Register allocation；
- Instruction selection；
- 大部分 Calling detail；
- 大部分硬體指令差異；

  交給 Compiler。

SECTION

十八、可攜性的正確形式

可攜不是：

> MachineSpecificity=0

而是：

> Program
> =
> PortableCore
> +

且：

> |PortableCore|
> ≫
> |MachineBoundary|

SECTION

十九、C 的設計單位

C 並不試圖成為完整封閉系統。

它接受：

- Assembly；
- Linker；
- Loader；
- libc；
- OS API；
- Preprocessor；
- Build tool；
- Debugger；
- Foreign language。

因此：

> Language
> ⊂
> Toolchain
> ⊂
> System

SECTION

二十、結構高於逐指令對應

C 的關鍵提升是：

- struct；
- 型別；
- Function；
- Scope；
- Expression；
- Separate translation。

  它讓 Unix 的 Inode、Process、Buffer 等系統概念能以程式結構表示，而不是以無名 Offset 和 Register 組成。

SECTION

二十一、實作與使用共同裁決

C 的早期規則常由：

- Compiler 能否實作；
- Unix 是否需要；
- Port 是否成功；
- Existing code 是否可轉換；
- 使用者是否會犯錯；

  共同決定。

  Ritchie 對 &&／　、Preprocessor、Pointer 和 Cast 歷史的回顧，甚至明確承認部分規則具有遷移折衷和後來看來不理想的結果。[R3]

SECTION

二十二、Ritchie 的設計者義務

設計者應：

- 提供可靠 Compiler；
- 使成本可預測；
- 保留硬體能力；
- 讓完整系統可實作；
- 以真實 Port 驗證可攜；
- 不把所有平台差異藏成虛構一致。

SECTION

二十三、使用者承擔什麼

使用者承擔：

- Allocation／Free；
- Bounds；
- Lifetime；
- Alias；
- Integer representation；
- Pointer validity；
- Synchronization；
- Platform-specific assumption；
- API 契約。

Library、Lint、Compiler warning 和組織規範只是補助。

SECTION

二十四、Ritchie 願意犧牲什麼

- 完整記憶體安全；
- 自動資源管理；
- 全面抽象封裝；
- 單一開發環境；
- 過度語言內建；
- 對初學者完全友善的型別表面。

SECTION

二十五、Ritchie 拒絕犧牲什麼

- 真實機器能力；
- 編譯效率；
- 系統互操作；
- 語言小型；
- 大部分系統邏輯可攜；
- 使用者建立基礎設施的自由。

SECTION

二十六、抽象不能強迫支付成本

零額外成本原則有兩層：

- 不使用的功能不支付其一般成本；
- 使用抽象時，不應比合理手寫低階方案更差。

這使高階機制能進入：

- Kernel-adjacent code；
- Embedded；
- Game engine；
- Finance；
- HPC；
- Browser；
- Database。

SECTION

二十七、User-defined type 與內建型別平權

Stroustrup 的核心目標是讓使用者定義：

- Vector；
- Complex；
- String；
- File；
- Lock；
- Iterator；
- Matrix；
- Domain value；

並使其在：

- 語法；
- 效率；
- 組合；
- 泛型算法；

  上接近內建型別。

SECTION

二十八、RAII：資源責任型別化

C 將資源釋放交給流程紀律。

C++ 以：

> ResourceLifetime
> =
> ObjectLifetime

建立：

- Deterministic destruction；
- Exception-safe cleanup；
- Lock guard；
- Smart pointer；
- File wrapper。

  這是對 C 責任配置的重大重寫，但仍保留 Raw escape。

SECTION

二十九、多範式不是無中心

C++ 支援：

- Procedural；
- Data abstraction；
- Object-oriented；
- Generic；
- Compile-time；
- Functional techniques。

  共同中心是：

> 高效、型別化、可組合的使用者定義抽象

SECTION

三十、能力不可因設計者無知而被刪除

Stroustrup 擔心語言設計者：

- 不理解所有領域；
- 以 Paternalism 移除必要能力；
- 強迫單一程式風格；
- 讓低階工作無法完成。

  因此 C++ 保留：

- Pointer；
- Cast；
- Manual allocation；
- Union；

- Inline Assembly／Interop；
- Unsafe operation。

SECTION

三十一、相容是社會部署技術

C 相容降低：

- 重寫成本；
- Tool replacement；
- Library loss；
- 組織知識遷移；
- 效能風險。

C++ 的成功不只在語言能力，也在能進入既有 C 專案。

SECTION

三十二、Stroustrup 的設計者義務

設計者應：

- 不讓抽象造成不必要 Runtime 成本；
- 不因個人偏好剝奪已證實能力；
- 支援真實專案；
- 讓高階和低階程式共存；
- 建立可逐步採用的語言；
- 在公共標準中協調多領域需求。

SECTION

三十三、使用者承擔什麼

使用者需理解：

- 多種資源模型；
- Value／Reference semantics；
- Template；
- Overload；
- Lifetime／Ownership 慣例；
- Raw／Safe abstraction 邊界；
- 多代 C++ Style；
- Build、ABI、Compiler 和 Library。

SECTION

三十四、Stroustrup 願意犧牲什麼

- 語言表面簡潔；
- 單一範式；
- 功能最小化；
- 教學的一致起點；
- 編譯器簡單；
- 完全消除歷史能力。

SECTION

三十五、Stroustrup 拒絕犧牲什麼

- 機器控制；
- 高階抽象；
- 零額外成本目標；
- 現有 C／C++ 程式；
- 多領域適用；
- 使用者選擇；
- 公共標準的長期可用性。

SECTION

三十六、Wirth：抽象應保持規則且可完整編譯

Wirth 不追求最高抽象，而追求：

- 清楚型別；
- Module；
- Type extension；
- 小 Compiler；
- 可預測機器映射。

低階能力應集中在 SYSTEM 等邊界，而非瀰漫全語言。

SECTION

三十七、Ritchie：抽象是可攜機器介面

C 的 Pointer、Array、Structure 與整數模型讓使用者持續看見機器。

其抽象距離最短，但仍比 Assembly 高出足以保存系統結構的一層。

SECTION

三十八、Stroustrup：抽象可以很高，只要不多付不必要成本

C++ 允許：

- Generic container；
- Iterator；
- Range；
- Coroutine；
- Class hierarchy；
- Compile-time algorithm。

  但設計目標仍要求成本能映射到機器。

SECTION

三十九、比較

c|c
設計者 抽象與機器關係;
hline
Wirth 少量規則抽象，完整可實作;
Ritchie 薄抽象，直接暴露表示與位址;
Stroustrup 高階抽象，但要求低階成本可承受

SECTION

四十、Wirth：以限制換取靜態清楚

主要手段：

- Strict type；
- Structured control；
- Module visibility；
- 明確 Import；
- 低階能力隔離；
- 小型語言。

設計者透過刪除危險交互降低錯誤空間。

SECTION

四十一、Ritchie：能力優先，安全外移

主要手段：

- 基本 Type；
- Structure；
- Cast；
- Compiler warning；
- Lint；
- Library discipline。

語言本身不阻止大量非法記憶體狀態。

SECTION

四十二、Stroustrup：建立安全抽象，但保留逃生

主要手段：

- Constructor／Destructor；
- RAII；
- Access control；
- Generic；
- Stronger type；
- Smart pointer；
- Container；
- Guideline；
- Static analysis。

  但仍保留 C 型能力和 Undefined behavior 邊界。

SECTION

四十三、安全責任分配

> c|c|c
> 安全主要放在哪裡 低階逃生;
> hline
> Wirth 語言限制＋Module＋小系統 集中、明示;
> Ritchie 程式設計者＋工具＋慣例 廣泛;
> Stroustrup 型別化抽象＋Library＋工具 保留且多層

SECTION

四十四、Wirth 的位置

Modula-2 早期因：

- 小記憶體；

- Real-time；
- Metadata 可靠性；

  拒絕一般 GC。

  Oberon System 後來可使用 GC，顯示其立場不是永遠反對自動回收，而是要求：

- 機制符合整體系統；
- 可理解；
- 可有效實作；
- 不破壞目標領域。

SECTION

四十五、Ritchie 的位置

C 提供：

- Static；
- Stack；
- malloc／free；
- Pointer；
- Library。

  資源生命期主要由人和流程管理。

SECTION

四十六、Stroustrup 的位置

C++ 以：

- Constructor；

- Destructor；
- RAII；
- Value semantics；
- Smart pointer；
- Container；

  把一般資源生命期放入型別和 Scope，同時保留 Manual control。

SECTION

四十七、比較

> Wirth：系統整體決定管理模型;
> Ritchie：程式流程直接管理資源;
> Stroustrup：型別封裝資源，但保留直接控制

SECTION

四十八、Wirth：允許另起新語言

Wirth 的路線：

> Pascal
> →
> Modula-2
> →
> Oberon
> →
> Oberon-07

核心一致高於 Source compatibility。

他願意讓使用者學習新語言，以修正舊核心。

SECTION

四十九、Ritchie：實作和使用形成漸進相容

C 早期可以變動，但：

- Unix Source；
- K&R；
- Compiler；
- Vendor；
- Hardware；

  快速形成既有實踐。

  後來標準化主要編碼共同實踐，而非另建新語言。

SECTION

五十、Stroustrup：相容從第一天即是原始條件

C++ 選 C 作基礎，本身就是：

- 採用策略；
- Interop 策略；
- Library 策略；
- 人才遷移策略。

  相容不是後來附加，而是核心政治。

SECTION

五十一、相容矩陣

設計者 破壞舊語言的容忍 主要理由

Wirth 高 修正核心、保持概念一致

Ritchie 中→低 早期演化，後期既有實踐與標準

Stroustrup 低 C／C++ 生態、工業遷移、ABI 與多領域

SECTION

五十二、相容不是純美德

相容保護：

- 使用者投資；
- Library；
- 產業；
- 知識；
- 供應鏈。

也保存：

- 語法歷史；
- 不安全能力；
- 不一致；
- 特徵交互；
- 教學負擔。

SECTION

五十三、Wirth：從語言到處理器的閉環

Project Oberon 的理想驗證：

Language

→ Compiler
→ OS
→ UI
→ Network
→ Processor
設計者可以追蹤完整因果鏈。

SECTION

五十四、Ritchie：以 Unix 作真實共演化平台

C 的驗證不是從零封閉建造所有硬體，而是：

C
↔ Unix Kernel
↔ Utilities
↔ Port
↔ Other OS environments
可攜成功比單一完整說明更重要。

SECTION

五十五、Stroustrup：以多領域專案和生態驗證

C++ 不由單一 Project C++ 證明。

它由：

- Telecom；
- Finance；
- Game；
- Browser；
- OS；
- Embedded；
- Science；

- Standard Library；

  共同驗證。

  這提高覆蓋，降低整體可理解性。

SECTION

五十六、三種證據標準

> c|c
> Wirth 完整小系統可被一體理解;
> Ritchie 真實 OS 可跨機器重建;
> Stroustrup 多領域大型生態可長期使用

SECTION

五十七、Wirth：集中設計與新版本替換

特徵：

- 個人設計權高；
- 小型研究團隊；
- 完整實作；
- 以新語言表達修正；
- 社群不直接決定核心。

  優勢是概念一致；風險是採用和接班。

SECTION

五十八、Ritchie：研究小組、文件與標準委員會

階段：

- Bell Labs 小組實作；
- K&R 公共文本；
- 多 Compiler 實踐；
- ANSI／ISO 委員會。

權力從個人和小組逐步轉移。

SECTION

五十九、Stroustrup：創始者方向與公共標準共存

階段：

- C with Classes 個人原型；
- Bell Labs 使用者；
- 商業 Compiler；
- ANSI／ISO WG21；
- 全球 Library／Tool／Vendor。

Stroustrup仍具高度影響力，但沒有最終單人裁決。

SECTION

六十、治理複雜度

$$
\begin{aligned}
&\text{GovernanceComplexity} \\
&\propto \\
&\text{InstalledBase} \\
&\times \\
&\text{StakeholderDiversity} \\
&\times \\
&\text{CompatibilityCost}
\end{aligned}
$$

這解釋了為何 C++ 的治理必然比 Oberon 複雜，而不必直接歸因於設計者性格。

SECTION

六十一、案例一：增加新抽象

Wirth

先問：

- 是否基本；
- 是否能取代多項現有功能；
- 是否能使系統更小；
- 是否能清楚編譯與教授。

Ritchie

先問：

- Unix／系統是否真正需要；
- 是否能有效映射多種硬體；
- 是否保持核心小；
- 是否能與現有 Toolchain 工作。

Stroustrup

先問：

- 是否支援已證明的程式技術；
- 是否不增加不必要成本；
- 是否可與現有 C++ 互操作；

- 是否能在標準和多實作中落地。

SECTION

六十二、案例二：歷史缺陷

Wirth

建立後繼語言，刪除錯誤。

Ritchie

若已形成大量實踐，透過 Warning、Cast、Lint、Standard 漸進收緊。

Stroustrup

很少移除；以：

- 新語法；
- Library；
- Guideline；
- Deprecation；
- Tool；
- Profile；

建立推薦子語言。

SECTION

六十三、案例三：初學者遇到低階成本

Wirth

應理解型別、資料結構和 Compiler，因為教育是系統能力的一部分。

Ritchie

應理解 Pointer、Memory 和表示，因為這些是系統工作的本體。

Stroustrup

應先使用高階 Container、RAII 和 Value type，⌧在需要時下降到 Raw control。

SECTION

六十四、Wirth 的主要保護對象

- 學習者；
- Compiler 實作者；
- 系統維護者；
- 希望理解完整系統的人；
- 小型高可靠工程團隊。

SECTION

六十五、Ritchie 的主要保護對象

- OS 實作者；
- Compiler／Runtime 作者；
- Driver／Embedded 開發者；
- 跨硬體移植者；
- 需要與外部系統共存的人。

SECTION

六十六、Stroustrup 的主要保護對象

- 已在 C／C++ 世界工作的產業程式設計者；
- 需要高階抽象又不能犧牲效能的人；
- 大型 Library 作者；
- 多領域基礎設施團隊；
- 不能一次重寫整個系統的組織。

SECTION

六十七、Wirth 路線可能犧牲

- 需要龐大 Library 生態的人；
- 依賴向後相容的組織；
- 需要快速納入大量領域功能的專案；
- 多方標準治理。

SECTION

六十八、Ritchie 路線可能犧牲

- 缺乏低階專業的使用者；
- 安全關鍵應用；
- 大規模自動驗證；
- 想要記憶體生命期由語言保證的人。

SECTION

六十九、Stroustrup 路線可能犧牲

- 需要小型完整語言模型的學習者；
- Compiler／Tool 實作者；
- 希望消除歷史不安全能力的人；
- 團隊之間的 Style 一致性；
- 快速、乾淨的核心重設。

SECTION

七十、若 Wirth 面對 C++ 的 Installed base

他很可能不會直接接受所有歷史能力，但也可能無法以 Oberon 式方式重建全球工業平台。

因此：

> WirthStyle
> +
> C++InstalledBase
> ≠
> OberonAtScale

SECTION

七十一、若 Ritchie 在今日設計 C

不能假設他會完全複製 1973 C。

現代條件包括：

- Security；

- Multicore；
- Optimizing compiler；
- Massive memory；
- Network；
- Supply chain；
- Formal analysis。

Ritchie 的穩定風格可能仍是薄層和機器透明，但具體安全機制可能不同。

SECTION

七十二、若 Stroustrup 沒有 C 相容

可能得到：

- 更乾淨型別；
- 更少語法化石；
- 更強安全；
- 更一致泛型。

也可能失去：

- 早期採用；
- C Library；
- 系統工程師；
- 真實專案；
- 商業 Compiler。

成功與複雜具有共同原因。

SECTION

七十三、不能說 Wirth 最簡單所以最好

因為：

- 覆蓋領域不同；
- 生態規模不同；
- 相容約束不同；
- 全球平台需求不同。

SECTION

七十四、不能說 C 最接近機器所以最有效

現代效能還受：

- Cache；
- Vectorization；
- Alias；
- Compiler；
- Parallelism；
- Data layout；
- Algorithm；

影響。

高階抽象有時能產生更好最佳化。

SECTION

七十五、不能說 C++ 功能最多所以最強

能力多也意味：

- 交互；
- 學習；
- 診斷；
- 不同 Style；
- 安全子集不明；
- Governance cost。

SECTION

七十六、不能把三人化成性格漫畫

Wirth = 極簡潔癖
Ritchie = 放任高手
Stroustrup = 不會拒絕功能
這些都忽略歷史可行空間和設計者的自我反省。

軸	Wirth	Ritchie	Stroustrup
首要問題	系統不可理解	系統綁定機器	抽象與效率分裂
核心策略	刪減、正交、完整實作	薄抽象、可攜 Compiler	零額外成本、多範式
硬體暴露	受控邊界	廣泛直接	可高可低
安全責任	語言限制與小系統	使用者、工具、慣例	型別化抽象＋Escape
Runtime	小且可解釋	最小、外部化	依抽象選擇，避免強制成本
相容性	可犧牲	早期變動、後期保守	核心契約
擴張方式	新語言／少量高覆蓋機制	Library／Toolchain	Language＋Template＋Library
驗證方式	完整可理解系統	Unix 與跨機器 Port	多領域工業生態
治理	集中設計	小組→文本→標準	創始者影響＋WG21
最大優勢	概念清楚	可攜控制	抽象與效能共存

最大代價　生態與相容　安全外移　語言與制度複雜

SECTION

七十七、三個頂點

定義：

- U：Whole-system understandability；
- M：Direct machine control；
- K：Compatibility-preserving abstraction growth。

  則三位設計者的重心可近似表示：

> Wirth → U

> Ritchie → M+P

> Stroustrup → K+A+M

其中 P 為可攜，A 為抽象。

SECTION

七十八、不可同時極大化

若：

> Uuparrow

通常需要：

- 刪除功能；
- 控制生態；

- 限制歷史相容。

  若：

> Muparrow

通常需要：

- 暴露表示；
- 增加使用者證明義務。

  若：

> K+Auparrow

通常需要：

- 保存歷史；
- 增加語言及工具交互。

SECTION

七十九、系統語言不可能三角

可提出：

> 【完整可理解
> +
> 最大機器自由
> +
> 長期相容的高階擴張】

難以同時極大化。

這不是形式不可能定理，而是從三個歷史案例抽取的工程張力模型。

SECTION

八十、先聲明保護對象

設計新系統語言前應明確回答：

- 保護初學者？
- 保護 Kernel 作者？
- 保護既有程式？
- 保護安全稽核者？
- 保護 Library 作者？
- 保護跨平台部署？
- 保護整個生態的遷移？

  沒有語言能同時將所有群體成本降到最低。

SECTION

八十一、把省略寫成正式決策

不加入某功能應記錄：

- 省略原因；
- 替代方法；
- 成本轉移；
- 未來重評條件。

  Wirth 的方法提醒：省略不是缺少設計。

SECTION

八十二、把低階能力限制在可審查邊界

Ritchie 的歷史證明低階能力不可避免；C 的安全代價則證明能力若在全語言無邊界擴散，後代會承擔巨大成本。

現代語言應區分：

- Safe default；
- Explicit unsafe；
- FFI boundary；
- Runtime primitive；
- Tool verification。

SECTION

八十三、相容必須有預算

每保留一項歷史行為，都應計算：

> Cost_(compat)
> =
> Teaching
> +
> Tooling
> +
> Interaction
> +
> Security
> +
> Specification
> +
> MigrationDeferred

Stroustrup 的案例表明相容是成功來源，也可能是最大複雜度來源。

SECTION

八十四、完整實作仍不可替代

即使語言最終服務全球生態，也應建立：

- Reference compiler；
- Standard library；
- OS／Runtime prototype；
- Large application；
- Toolchain；

  驗證設計。

  三人都不是只寫語法規格。

SECTION

八十五、治理應與 Installed base 一起設計

小型新語言可集中裁決。

成熟平台需要：

- 公開提案；
- 多實作；
- 相容測試；
- 版本政策；
- 退出和接班；
- 跨領域代表。

  Niklaus Wirth、Dennis Ritchie 與 Bjarne Stroustrup 代表的不是三種單純語言風格，而是三種對系統程式責任的回答。

Wirth 認為：

> 設計者首先有責任讓語言和完整系統仍能被人理解。若歷史功能阻礙這一點，就應刪除、重建或另立新語言。

Ritchie 認為：

> 系統語言必須誠實面對機器，保留真正需要的控制；但編譯器應把大部分程式結構從單一硬體中解放，使完整系統可以移植。

Stroustrup 認為：

> 使用者不應因追求抽象而失去效能和機器能力，也不應因設計者偏好而被迫放棄既有程式；抽象必須可負擔，相容必須被承擔。

本文對三者的最終 PLDST 判定為：

> 【Wirth ：Understandability-First Systems Ethic;
> Ritchie ：Portable Machine-Control Ethic;
> Stroustrup ：Compatibility-Constrained Abstraction Ethic】

三種倫理各自具有不可替代價值：

Wirth

提醒我們：

- 功能需要舉證；
- 完整理解是一種可靠性；
- 語言、Compiler、OS 和硬體應能形成清楚因果鏈；
- 有時另起新核心比永遠修補更誠實。

Ritchie

提醒我們：

- 系統抽象不能建立在對機器成本的幻想上；
- 可攜性不是隱藏全部差異，而是限制差異的範圍；
- 小型語言可支撐巨大外部工具世界；
- 真實系統使用是語言設計的最高壓力測試之一。

Stroustrup

提醒我們：

- 高階抽象與效能不必然對立；
- 資源責任可以進入型別；
- 使用者已有程式和能力是設計約束，不是雜訊；
- 語言治理必須面對產業、標準和數十年相容。

三者也各有不可忽視的代價：

> 【Wirth：以生態和相容換取清楚;
> Ritchie：以安全證明外移換取控制;
> Stroustrup：以語言和治理複雜換取能力保存】

最終，不存在脫離情境的最佳系統語言倫理。

真正可普遍化的原則是：

> 【先說明不可犧牲的價值，
>
> 再公開複雜度被轉移給了誰。】

若一門語言宣稱：

- 更簡單；
- 更快；

- 更安全；
- 更相容；
- 更高階；

卻不說明代價轉移位置，它就尚未完成設計論證。

Wirth、Ritchie 和 Stroustrup 的共同歷史價值，正是三人都以不同方式使自己的取捨接受了真實實作、完整系統和長期使用的檢驗。

Niklaus Wirth
保護對象：學習者、維護者、完整系統理解者
核心手段：刪減、型別、Module、完整實作
不可犧牲：概念清楚
主要代價：相容和生態
Dennis Ritchie
保護對象：系統實作者、移植者、硬體控制者
核心手段：薄抽象、Pointer、Structure、Compiler
不可犧牲：機器能力與可攜控制
主要代價：安全責任外移
Bjarne Stroustrup
保護對象：既有產業程式設計者、高效抽象使用者
核心手段：Class、RAII、Template、Zero-overhead、Standard
不可犧牲：能力、效能和相容
主要代價：語言、工具與治理複雜
以下不是排名，而是相對風格位置，範圍 1–5：

軸	Wirth	Ritchie	Stroustrup
完整可理解性優先	5	3	2
直接機器控制	3	5	5
高階抽象廣度	3	2	5
語言強制安全	4	1	3
向後相容優先	1	3	5
核心功能節制	5	5	2
Library／生態擴張	2	4	5
完整系統親自驗證	5	5	3

多方治理需求 1 3 5

此表是 PLDST 分析工具，不是客觀測量。

[R1] Niklaus Wirth, “Modula-2 and Oberon,” HOPL III, revised 2006.

— 設計簡潔、Modula-2／Oberon 演化、先刪後加、Type extension、完整實作與限制。

[R2] Niklaus Wirth and Jürg Gutknecht, Project Oberon: The Design of an Operating System, a Compiler, and a Computer, 1992／2013.

— 完整系統、可理解性、Compiler、OS、硬體及共同作者。

[R3] Dennis M. Ritchie, “The Development of the C Language,” HOPL II, 1993.

— BCPL／B／C、PDP-11、Array／Pointer、Structure、Preprocessor、Unix 重寫及直接貢獻者。

[R4] Dennis M. Ritchie, “The Evolution of the Unix Time-sharing System,” 1979／1984 editions.

— Unix 歷史、組合語言到 C、Port、社會條件及完整系統演化。

[R5] Bjarne Stroustrup, “A History of C++: 1979–1991,” HOPL II, 1993.

— Simula＋C、C with Classes、效率與彈性、真實專案、工具及標準化。

[R6] Bjarne Stroustrup, “Foundations of C++,” ETAPS, 2012.

— Hardware mapping、Zero-overhead、Type-rich programming、Resource management 及 Modern C++。

[R7] Niklaus Wirth, “A Plea for Lean Software,” 1995.

— 軟體膨脹、硬體進步、功能經濟與可靠性。

[R8] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, 1978／1988.

— C 公共文本、Library、Reference 及 K&R 共同文化。

[R9] Bjarne Stroustrup, The Design and Evolution of C++, 1994.

— 相容性、功能取捨、使用者能力、RAII、多範式和標準前史。

[R10] ANSI／ISO C and C++ Rationale／WG14／WG21 historical materials.

— 現有實踐、公共標準、多方治理和相容性制度。

[R11] S. C. Johnson and D. M. Ritchie, “Portability of C Programs and the UNIX System,” Bell System Technical Journal, 1978.

— 可攜性的相對定義、環境依賴、系統移植及 Portable C Compiler。

[R12] Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020,” HOPL IV, 2021.

— Problem-driven、Zero-overhead、Stability、Committee、Library 及長期相容演化。

[W-U] Wirth: whole-system understandability
[W-R] Wirth: reduction before addition
[W-I] Wirth: implementability ethics
[R-M] Ritchie: machine transparency
[R-P] Ritchie: portable systems layer
[R-E] Ritchie: externalized safety responsibility
[S-Z] Stroustrup: zero-overhead abstraction
[S-C] Stroustrup: compatibility politics
[S-M] Stroustrup: multi-paradigm capability preservation
[C-E] Comparative ethics
[C-B] Burden allocation
[C-G] Governance under installed-base pressure

SECTION

E.1 Wirth 並非「永遠反對垃圾回收」

第二輪重新核對 Modula-2／Oberon 回顧與 Project Oberon：

- Wirth 在早期 Modula-2 中拒絕一般 Garbage collector，理由包括 64K 等級記憶體、Real-time 可預測性，以及當時不安全語言和 Metadata 難以保證；
- 他同時承認手工 Deallocate 可能釋放仍可到達的物件，是嚴重錯誤來源；
- Oberon System 後來把 Garbage collector 作為 Background task；
- Wirth 的 Oberon 教材亦把 GC 視為實作機制，而不是必然屬於語言定義本身。

  因此本文將其立場校準為：

> Memory-management mechanism 必須符合目標系統、型別安全與可理解性

而不是「Wirth 的設計倫理永遠排斥 GC」。

SECTION

E.2 Project Oberon 的「完整」範圍

Project Oberon 的官方前言說明：

- 1986–1989 年由 Wirth 與 Jürg Gutknecht 共同進行；
- 目標是從零設計和實作一個可整體描述、解釋及理解的工作站系統；
- 作者不只提出概念，也實作書中系統；
- 後來 2013 版將簡潔延伸至 RISC5 處理器和更新 Compiler。

「完整系統」不表示：

- 覆蓋今日所有 Driver、Security、Internationalization、Cloud 或 Accessibility 需求；
- 系統沒有任何外部工具或硬體前提；
- 所有模組均由 Wirth 單人完成。

本文將其用作「完整因果鏈驗證」案例，而非全球平台等價物。

SECTION

E.3 Ritchie 對可攜性的正式定義

第二輪重新核對 Johnson／Ritchie 的 Portability 論文：

- 程式可攜，意指移往新環境所需工作顯著少於重新撰寫；
- 高階語言程式仍可能依賴 Word size、Character、OS、Library、Representation 和外部環境；
- C 與 Unix 的可攜性是經過修改、分離硬體邊界、Compiler 和 Library 工作逐步取得；
- 1976–1977 年的 Unix 可攜實驗是重要制度及工程驗證。

因此：

> Portability
> ≠
> ImmediateMachineIndependence

本文使用的「擴大可攜核心、縮小機器邊界」與原始定義一致。

SECTION

E.4 Ritchie 並非將所有安全問題都視為個人紀律

第二輪核對 C 歷史：

- Alan Snyder 建議以 &&／　消除 B 中依 Context 改變 &／ 的難解行為；
- Pointer／Integer 混用後來被收緊，Cast 用來明示轉換；
- Steve Johnson 建立 lint，用以偵測可疑構造及 Separate compilation 難以發現的介面不一致；
- Ritchie 直接承認 Preprocessor 與語言整合不完整、Operator precedence 等歷史決策存在缺陷。

因此，Ritchie 的倫理不是「只信任高手、不需要工具」，而是：

保留低階自由
+
以 Type、Compiler、Lint、Library 和慣例逐步補強
只是其語言強制安全程度仍明顯低於 Wirth 路線及現代安全系統語言。

SECTION

E.5 零額外成本的成本邊界

第二輪重新核對〈Foundations of C++〉：

- Light-weight abstraction 指 Space／Time overhead 不超過特定抽象之合理手工實作；
- 「不用不付費」和「使用時手寫也不能更好」是設計目標；
- 該原則主要描述 Runtime 時間、空間及機器映射；
- 它不自動涵蓋：

- Compile time；
- Error-message complexity；
- Language-learning cost；
- Binary-size duplication；
- Build-system cost；
- Specification／governance cost；
- 特定 Compiler 未成功最佳化的個案。

因此本文沒有以零額外成本否認 C++ 在其他維度的巨大複雜度。

SECTION

E.6 C++ 相容性不是偶然的後期負擔

Stroustrup 的早期 HOPL 回顧和 HOPL IV 長期回顧共同支持：

- C with Classes 自一開始就要求在真實 C 環境中迅速可用；
- C 的效率、彈性、Tool、Library 及使用者是原始採用條件；
- 長期 C++ 設計準則仍包括 Problem-driven、Efficient／Zero-overhead，以及 Stable／“Don’t break my code”；
- WG21 於 1990–1991 年形成國際標準制度，現代語言決策不再由 Stroustrup 單人裁決。

因此，C++ 複雜性同時來自：

> 原始多目標設計
> +
> C 相容
> +
> 工業領域異質性
> +
> 長期標準演化

不能只歸因於委員會，也不能只歸因於創始者個人。

SECTION

E.7 三人對安全的比較層級

本篇比較的是主要歷史語言及設計方向，而非宣稱：

- Oberon 完全安全；
- C 完全沒有靜態檢查；
- C++ 已達記憶體安全。

更精確的層級是：

Wirth：
以較小語言、嚴格型別、Module 與受控低階邊界減少錯誤空間
Ritchie：
提供基本型別與結構，但把 Bounds、Lifetime、Alias 等大量義務外移
Stroustrup：
在 C 能力上建立 RAII、Container、Template 和型別化抽象，
但為相容與系統控制保留 Raw／Unsafe 路徑

SECTION

E.8 三人的「完整系統驗證」不是同一種證據

第二輪校準後，本文保留三類不同證據：

Wirth

小型完整系統能否被整體說明和重建

Ritchie

真實作業系統能否以高階語言重寫並跨架構移植

Stroustrup

抽象機制能否在大量工業領域和多實作中維持效能與互操作
三者不能互相直接替代，也不構成單一分數排名。

SECTION

E.9 Installed base 是因果變量，不是免責理由

本文把 Installed base 寫入歷史約束：

> Ω=
> (
> TeamSize,
> InstalledBase,
> Hardware,
> UseCase,
> Institution,
> Time,
> MigrationCost
>)

它能解釋為何：

- Wirth 可用新語言修正核心；
- Ritchie 從快速演化轉向既有實踐和標準；
- Stroustrup 從一開始就受到 C 相容約束。

  但 Installed base 不是保存所有缺陷的自動正當化理由。設計者與治理機構仍需評估：

- Deprecation；
- Migration tooling；
- Safe subset；
- New edition／profile；
- Library replacement；
- 教育和診斷。

SECTION

E.10 「倫理」一詞的公開邊界

本文所稱系統語言倫理，是：

價值優先序
+
成本與風險配置
+
保護對象
+
權力及相容責任

它不是：

- 對三人私德的判斷；
- 宣稱某種語言使用者更有道德；
- 把技術失誤直接等同倫理失敗。

此詞用於讓被隱藏的責任轉移成為可討論對象。

SECTION

E.11 比較矩陣的信心層級

高信心：

- Wirth 的簡潔與先刪後加；
- Ritchie 的 C／Unix 共演化及可攜；
- Stroustrup 的 Simula＋C、零額外成本和相容；
- 三種不同治理路線。

中高信心：

- 三位設計者的主要保護對象；
- 「可理解性憲法／薄層控制憲法／能力保存憲法」的分類；

- PLDST 不可能三角。

  後者是本文根據多項決策的理論抽象，不是三位設計者本人正式提出的名稱或定理。