Dennis Ritchie：可攜式系統語言、機器透明度與克制的抽象

Dennis Ritchie: Portable Systems Languages, Machine Transparency, and Restrained Abstraction

v1.0 / 2026-07-30 / Neo.K / 公開版／第三部設計師個案正式研究

https://thisoneisneok.com/html/papers/pldst-015.html

英文名稱： Dennis Ritchie: Portable Systems Languages, Machine Transparency, and Restrained Abstraction

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-015

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第三部設計師個案正式研究

SECTION

摘要

Dennis Ritchie 經常被描述為 C 語言的創造者與 Unix 的共同開發者。這個描述正確，卻不足以說明他的設計風格，也容易產生兩種相反誤讀：

- 把 C 描述成幾乎等同組合語言的低階工具；
- 把 C 的長期成功解釋成一套預先完整設計的普遍語言哲學。

  Ritchie 自己對 C 的歷史回顧顯示，C 並不是從空白紙上建立的純理論系統，而是在 BCPL、B、PDP-11、Unix 與實際編譯器之間反覆調整的工程結果。Ken Thompson 由 Martin Richards 的 BCPL 發展 B；Ritchie 在 1971 至 1973 年間保留 B 的大部分表面形式，加入型別、結構與適合 PDP-11 的資料表示，寫出第一個 C Compiler；其後 Alan Snyder、Steven C. Johnson、Michael Lesk、Thompson 等人也對語言與實作提供重

C 的設計價值不只在「接近機器」，而在於提供一個足夠薄、足夠可預測、又足以描述大型系統資料與控制結構的可攜層。它讓 Unix 可以由組合語言轉移到高階語言，使作業系統及其工具不再永久綁定單一處理器；同時，它沒有建立大型 Runtime、封閉物件世界或完整應用環境，而是接受作業系統、Linker、Library 與其他語言共同存在。[R1][R2][R3]

本文以 PLDST 方法將 Ritchie 的設計生涯分成五個相位：

- Multics／BCPL 學習期：大型系統、語言與作業系統的共同背景；
- B 到 C 的型別化重建期：PDP-11、Byte addressing、資料型別與 Compiler；
- Unix 共演化與可攜驗證期：以 C 重寫系統、跨機器移植及工具生態；
- K&R 與公共語言制度期：語言由研究小組慣例變成可教、可傳播的公共文本；
- 標準化與後創始者期：ANSI／ISO 委員會以現有實踐為基礎建立公共規格，C 不再由 Ritchie 個人治理。

本文核心判斷為：

> 【Ritchie 的主要抽象策略不是隱藏機器，
>
> 而是只隱藏那些會妨礙系統跨機器重建的偶然細節。】

C 的深層配置可表示為：

> 【具體資料表示
> +
> 直接資源存取
> +
> 小型語義核心
> +
> 可攜編譯
> +
> 外部系統互操作】

這種設計的成功與代價來自同一來源。C 讓程式設計者能建立 Kernel、Compiler、Runtime、Driver 和跨語言介面；但它也把陣列邊界、物件生命期、表示有效性、別名、整數轉換和大量資源正確性留給程式設計者、Library、工具與組織。後來標準化又將多種實作自由與未定義行為制度化，使 Compiler 能適應不同機器，也使安全分析承擔長期複雜度。

因此，Ritchie 不應只被分類為「低階語言設計者」。更精確的判定是：

> 他以克制的抽象把真實機器提升到可攜系統程式設計的最低充分層級，並以完整 Unix 系統證明這一層級足以支撐大型軟體。

關鍵詞： Dennis Ritchie、C、Unix、BCPL、B、系統程式設計、可攜性、機器透明度、K&R、標準化、PLDST

SECTION

一、本文研究範圍

本文主要分析：

- Multics 與早期 Bell Labs 系統背景；
- BCPL、B 與 C 的直接演化；
- 1971–1977 年間 C 的核心形成；
- C 與 Research Unix 的共同演化；
- 《The C Programming Language》；
- C89／C90 標準化前後的權力轉移；
- Ritchie 對軟體研究環境及 Unix 演化的回顧。

本文不把下列內容全部歸入 Ritchie：

- BCPL；
- B；
- Unix 的全部命令、工具與架構；
- PCC；
- ANSI C／ISO C 的全部決策；
- POSIX；

- C99、C11、C17、C23；
- Linux、BSD 及其他 Unix-like 系統；
- 所有 C Compiler 與 Library。

SECTION

二、C 的直接先行者

Ritchie 對 C 歷史的總結清楚區分：

Martin Richards：BCPL
Ken Thompson：B
Dennis Ritchie：由 B 發展 C、加入型別並寫第一 Compiler

BCPL 與 B 提供：

- 簡潔語法；
- 表達式導向；
- Pointer-like 操作；
- 系統程式傳統；
- Compiler portability 經驗。

C 並不是將這些全部捨棄，而是針對 PDP-11 與 Unix 的需求進行型別化重建。[R1]

SECTION

三、C 不是完全單人完成

Ritchie 將 1972–1977 年間的直接語言貢獻者列為：

- Dennis Ritchie；
- Ken Thompson；
- Alan Snyder；

- Steven C. Johnson；
- Michael Lesk。

其中：

- Johnson 的 Portable C Compiler 對 C 的跨平台傳播極為重要；
- Thompson 的 B 與 Unix 需求直接塑造 C；
- Unix 使用者和系統工具持續暴露語言缺陷；
- Brian Kernighan 在教學、文件、命名、範例及 K&R 公共化方面具有獨立作用。[R1][R4][R5]

因此：

C 核心創始設計：Ritchie 極高
第一 Compiler：Ritchie 高
前代語言：Richards、Thompson
語言細節與早期實踐：Bell Labs 小組共同
Portable Compiler：Johnson 關鍵
公共教學與 Reference：Kernighan＋Ritchie
標準化後演化：X3J11／WG14

SECTION

四、Unix 也不是雙人神話

Ken Thompson 是早期 Unix 的主要創建者；Ritchie 很快加入，並對語言、Kernel、工具、可攜和系統演化作出核心貢獻。

但 Research Unix 還包括：

- Douglas McIlroy；
- Joe Ossanna；
- Brian Kernighan；
- Robert Morris；
- Steven Johnson；
- Michael Lesk；

- Lee McMahon；
- 其他 Bell Labs 研究者與使用者。

  Pipe、Shell、Text processing、Tool philosophy、Library、Port 等成果需逐項歸因。

SECTION

五、Multics 的雙重影響

Multics 提供：

- Time-sharing；
- Hierarchical file system；
- Security；
- Dynamic linking；
- 大型系統模組；
- 高階語言實作作業系統的經驗。

  Bell Labs 退出 Multics 並不表示研究者拒絕其全部目標。Ritchie 的 Unix 歷史回顧顯示，早期 Unix 同時受到：

- 對 Multics 延遲與規模的失望；
- 對互動式計算環境的持續追求；

  兩種力量影響。[R2]

SECTION

六、BCPL 的吸引力

BCPL 是：

- 小型；

- 可自舉；
- 適合 Compiler 與系統工具；
- 對機器表示保持接近；
- 較容易移植。

其弱點也很清楚：

- 單一 Word-oriented representation；
- 缺乏適合 PDP-11 Byte-addressed memory 的型別；
- 資料結構及對齊需人工處理；
- 新硬體特性無法自然表達。

SECTION

七、設計問題來自使用而非分類學

Ritchie 的問題不是：

> 應建立一門低階還是高階語言？

而是：

> 如何保留 B 的小型、Compiler 友善與系統能力，同時讓它能有效表示 PDP-11 上的 Character、Word、Pointer、Array、Structure，並用於重寫 Unix？

這是一個具體工程約束集合。

SECTION

八、PDP-11 改變了語言

PDP-11 提供：

- Byte addressing；

- 不同大小的資料單位；
- 適合系統程式的 Addressing mode；
- 較好的 Stack 與 Procedure 支援。

  B 的無型別 Word model 無法自然利用這些特性。

  因此，C 的型別不是純粹安全工程，而同時是：

> Type
> =
> Representation
> +
> OperationSet
> +
> MachineMapping

SECTION

九、型別化重建

C 逐步加入：

- char；
- int；
- Pointer；
- Array；
- Structure；
- Union；
- Function type；
- 更完整 Declaration。

這些機制使程式能描述：

- Byte buffer；
- Device register；
- File structure；
- Process table；
- Tree；
- Linked structure；
- Compiler syntax object。

SECTION

十、Array 與 Pointer 的結合

C 將 Array indexing 與 Pointer arithmetic 緊密連接：

> a[i]
> ≡
> *(a+i)

這提供：

- 接近機器的資料遍歷；
- 小型語言規則；
- 高效實作；
- Array／Pointer API 互操作。

但也造成長期代價：

- Array parameter 容易退化成 Pointer；
- Bounds 不在一般 Pointer 型別中；

- Length 與地址分離；
- 安全責任落在呼叫者與 Library。

SECTION

十一、Structure：比組合語言高出關鍵一層

C 的真正提升不只是語法簡短，而是允許程式設計者為系統中的概念建立結構化表示：

```
struct inode
struct process
struct buffer
```

這讓大型系統不必只由無名 Offset 和 Word 組成。

因此：

> MachineTransparency
> ≠
> RepresentationAnonymity

接近機器不表示拒絕資料抽象。

SECTION

十二、Declaration 與表達式的妥協

C 的 Declaration syntax 受到「Declaration mirrors use」的思想影響：

```
int *p;
int a[10];
int (*f)(int);
```

它使型別構造與使用形式有直接關係，但複雜型別會形成高解析成本。

這種設計代表：

- 優先保持表面規則與使用相近；
- 接受複雜 Declaration 的教學負擔；
- 沒有建立完全獨立的型別表達語法。

SECTION

十三、語言由真實系統壓力塑形

C 不是完成後才用於 Unix。

更接近：

> C_(t+1)
> =
> Refine(
> C□,
> UnixNeeds□,
> CompilerExperience□,
> Hardware□
>)

Unix 需要：

- Character handling；
- File system structure；
- Process；
- Device；
- Pointer；
- Bit operation；
- Separate compilation；
- Assembly interoperation。

  這些需求反覆測試 C。

SECTION

十四、以 C 重寫 Unix

Unix 早期以 Assembly 為主。1973 年前後，大部分 Kernel 被改寫為 C，只保留少數硬體相關部分。

此舉的重要性是：

- 系統邏輯與單一處理器指令分離；
- Compiler 成為硬體遷移層；
- 作業系統可跨機器重建；
- 系統來源更容易閱讀與修改；
- 語言本身得到完整壓力測試。[R2][R3]

SECTION

十五、可攜不是「零修改」

早期 Unix Port 仍需要處理：

- Word size；
- Byte order；
- Device；
- Assembly startup；
- Memory management；
- Compiler；
- ABI；
- Alignment。

因此：

> Portable
> ≠
> MachineIndependent

C 所提供的是：

> 把大多數系統邏輯移出硬體專屬 Assembly，只把不可避免差異保留在受限區域。

SECTION

十六、C 的可攜性是成本分區

令系統程式為：

> Program
> =
> PortableCore
> +
> MachineSpecificBoundary

C 的價值在於使：

> |PortableCore|
> ≫
> |MachineSpecificBoundary|

而不是完全消除後者。

SECTION

十七、何謂機器透明

C 讓使用者相對容易推理：

- Object representation；

- Pointer；
- Integer size；
- Memory layout；
- Function call；
- Bit；
- Address；
- Control flow。

  但規格仍描述抽象機器，Compiler 可以：

- Register allocate；
- Reorder；
- Fold；
- Inline；
- Eliminate；
- 使用不同指令。

  所以來源不是 Assembly 的逐字轉錄。

SECTION

十八、克制抽象

Ritchie 的抽象策略通常避免：

- 大型隱式 Runtime；
- 強制物件系統；
- 單一封閉環境；

- 語言內建所有 OS 能力；
- 將特定應用模型寫入核心。

C 提供：

- Function；
- Structure；
- Pointer；
- Arithmetic；
- Control；
- Separate translation；
- Library boundary。

其餘由系統和 Library 建立。

SECTION

十九、語言只是系統的一部分

C 接受：

- Linker；
- Assembly；
- OS；
- Library；
- Foreign code；
- Toolchain；

共同存在。

這種設計與 Smalltalk／Lisp 式完整環境不同：

> Language
> ⊂
> System

而不是：

> LanguageRuntime
> ≈
> WholeSystem

SECTION

二十、核心小不代表實際系統小

C 的核心相對小，但真實開發需要：

- Preprocessor；
- Header；
- Linker；
- Build；
- ABI；
- libc；
- OS API；
- Debugger；
- Static analyzer；
- Coding standard。

  因此：

> SmallLanguageCore
> not⇒
> SmallEffectiveEnvironment

SECTION

二十一、C 禁止的錯誤有限

C 的型別系統能檢查：

- 基本 Declaration；
- Function type；
- Structure member；
- 部分轉換；
- 表達式合法性。

但一般不保證：

- Bounds；
- Lifetime；
- Null absence；
- Integer overflow safety；
- Data-race freedom；
- Initialization；
- Pointer provenance 可由使用者直接理解；
- Resource release。

SECTION

二十二、低階能力的必要性

系統語言需要：

- Memory-mapped I/O；
- ABI；
- Device register；
- Allocator；
- Kernel data；
- Foreign interface；
- Runtime implementation。

完全禁止 Raw operation 會使語言無法實作自己的基礎。

Ritchie 的配置是保留能力，並將正確性責任放給：

- 程式設計者；
- API；
- Review；
- Tool；
- Platform convention。

SECTION

二十三、自由不是免費

此配置的長期代價包括：

- Buffer overflow；

- Use-after-free；
- Invalid conversion；
- Data race；
- Uninitialized data；
- Security exploit；
- Platform-dependent behavior。

因此：

> Freedom_(representation)
> uparrow
> ⇒
> ProofObligation_(human)
> uparrow

SECTION

二十四、不能把後來標準全部回寫給 Ritchie

「Undefined behavior」的完整現代制度由：

- 早期實作；
- K&R；
- ANSI C；
- ISO C；
- Compiler optimization；
- Defect reports；

共同形成。

Ritchie 的原始設計保留大量實作自由，但今天 Pointer provenance、Sequence、Effective type 等複雜標準問題不能全部視為他的直接個人選擇。

SECTION

二十五、Brian Kernighan 的作用

Brian Kernighan：

- 撰寫早期 Tutorial；
- 以清楚範例整理慣用法；
- 與 Ritchie 共同撰寫《The C Programming Language》；
- 使語言具有可傳播的公共敘述；
- 將 Language、Library 與 Style 一起教給外部使用者。[R4][R5]

SECTION

二十六、K&R 不只是教材

第一版 K&R 的附錄長期充當：

- 語言 Reference；
- Compiler 共同基線；
- 使用者預期；
- 標準化 Base document。

C89 Rationale 說明，C89 大部分語言與 K&R 第一版附錄及當時大多數 Translator 的共同實踐一致。[R6]

SECTION

二十七、文體也是設計介面

K&R 的短小、範例導向與逐步展示，強化 C 的文化風格：

- 小範例；
- 可直接執行；
- 少量核心；
- 不依巨大框架；
- 以真實程式解釋規則。

  但「K&R 風格」是 Kernighan 與 Ritchie 的共同公共成果，不應全歸 Ritchie。

SECTION

二十八、從 De facto 到 De jure

早期 C 的主要事實來源是：

- Bell Labs Compiler；
- K&R；
- Unix code；
- 不同 Vendor implementation。

  方言增長後，需要：

- 清楚規格；
- 跨 Compiler 一致；
- Library 定義；
- 可攜承諾；
- Defect resolution。

SECTION

二十九、X3J11／WG14 的任務

C89 委員會的基本任務是：

- 編碼共同現有實踐；
- 在既有先例清楚時尊重先例；
- 正式化已證明有價值的改進；
- 促進不同 C 環境的程式可攜性。[R6]

  這不是讓委員會自由重新設計一門理想語言。

SECTION

三十、Ritchie 不再是唯一決策者

標準化後：

> Authority_(Ritchie)
> downarrow
>
> Authority_(Committee)
> uparrow

Ritchie 的原始設計與 K&R 仍具有歷史權重，但：

- 新型別；
- Library；
- Memory model；
- Atomics；
- Unicode；

- Defect interpretation；

  由後續制度決定。

SECTION

三十一、相容性政治

委員會面對：

- 已有 Source；
- Compiler；
- Hardware；
- ABI；
- Vendor extension；
- K&R code；
- 效能需求。

因此，C 的現代不一致往往是：

> OriginalCore
> +
> ExistingPractice
> +
> Compatibility
> +
> Optimization
> +
> CommitteeCompromise

共同結果。

SECTION

三十二、Ritchie 對 Bell Labs 環境的回顧

在圖靈獎演講中，Ritchie 把 Unix 的形成與下列條件連結：

- 小型互信團隊；
- 使用者與開發者重疊；
- 可以使用真實系統；
- 可自行選擇問題；
- 同事具有互補能力；
- 系統逐步形成而非依巨大預先計畫。[R7]

SECTION

三十三、實際使用者回饋

Unix 小組自己每天使用系統。

因此：

> Design
> →
> Use
> →
> Irritation
> →
> Revision

循環很短。

這與先完成巨大規格、再交給外部使用者不同。

SECTION

三十四、克制不是沒有理想

Unix 與 C 具有強烈方向：

- 小型工具；
- 可組合；
- 文字介面；
- 可重寫；
- 可移植；
- 清楚資料表示。

  但它們不是透過完整形式藍圖一次完成，而由實作與使用逐步收斂。

SECTION

三十五、Multics／BCPL 期

問題：大型互動系統如何由高階語言建造
所得：系統願景、Compiler 與語言經驗

SECTION

三十六、B→C 期

問題：B 無法自然映射 PDP-11 資料
策略：加入型別、Structure、Byte 與新 Compiler

SECTION

三十七、Unix 共演化期

問題：作業系統綁定 Assembly
策略：以 C 重寫大部分 Kernel

SECTION

三十八、K&R 公共化期

問題：語言靠口傳與單一實作傳播
策略：教程、範例、Reference manual

SECTION

三十九、標準化期

問題：方言、Vendor 和既有實踐分裂
策略：委員會編碼共同實踐

SECTION

四十、後期系統研究期

Ritchie 後來參與：

- Streams；
- Plan 9；
- Inferno 相關系統研究；
- 軟體研究管理。

這些工作延續：

- 分散式系統；
- 簡化界面；
- 系統可組合；
- 研究原型。

但各專案具有新的主要設計者和團隊，不能直接歸為 Ritchie 的個人語言作品。

SECTION

四十一、問題 framing

Ritchie 的核心問題可表述為：

> 如何用一門足夠接近機器、又足以表達大型資料結構的語言，讓系統軟體從單一硬體中解放？

SECTION

四十二、價值優先序

> V_(Ritchie)
> ≈
> (
> Portability,
> Implementability,
> MachineAccess,
> Compactness,
> Interoperability,
> SystemUsefulness,
> Clarity
>)

SECTION

四十三、核心—擴張偏好

核心偏好：

- 小；
- 一般；
- 與機器表示相容；
- 不建立封閉系統；
- 以 Library 和 Tool 擴張。

SECTION

四十四、顯式—推導偏好

偏好明示：

- Pointer；
- Type；
- Layout；
- Control；
- Conversion；
- Resource operation。

Compiler 自動處理：

- Register；
- Instruction selection；
- Calling detail；
- 大部分機器映射。

SECTION

四十五、效率—可讀性偏好

理想是：

> ReadableSystemStructure
> +
> PredictableMachineCost

而不是：

- 完全隱藏成本；
- 每行等同 Assembly。

SECTION

四十六、安全—自由偏好

C 保留高度表示自由，安全主要不是由語言全面保證。

因此 Ritchie 的風格不是現代安全優先，而是：

> 以最低抽象成本保留系統實作能力，並接受使用者必須理解機器和表示。

SECTION

四十七、相容性偏好

早期 C 可以快速修改；K&R 後，既有實踐迅速形成相容負擔。

標準化後的強相容性主要是制度結果，不能全部回寫為 Ritchie 的終身個人風格。

SECTION

四十八、C 不是「可攜 Assembly」

C 提供：

- 型別；
- Structure；
- Expression；
- Function；
- Abstract machine。

Compiler 並不保證逐句映射。

「可攜 Assembly」可作隱喻，不能作精確技術定義。

SECTION

四十九、C 的成功不只來自語言品質

還包括：

- Unix source dissemination；
- Bell Labs 研究文化；
- K&R；
- PCC；
- 大學採用；
- Hardware vendor；
- 標準化；
- ABI；
- Compiler availability。

SECTION

五十、小核心不等於低複雜度

C 將大量複雜度移到：

- Preprocessor；
- Header；
- Linker；
- Build；
- Undefined／implementation-defined behavior；

- Library；
- Coding standard；
- Static analysis；
- Human review。

SECTION

五十一、機器接近性不是安全理由

能理解 Hardware 並不表示能避免所有：

- Alias；
- Lifetime；
- Concurrency；
- Optimizer；
- Spec ambiguity。

現代 Compiler 和 Hardware 使「憑直覺理解 C」比早期更困難。

SECTION

五十二、Ritchie 的簡潔不應浪漫化

C 的某些歷史設計：

- Null-terminated string；
- Array／Pointer boundary；
- Declaration；
- Preprocessor；
- Implicit conversion；

確實形成長期工程成本。

成功不表示所有設計都應被複製。

SECTION

五十三、現代 C 不等於 1973 C

現代 C 包含：

- 標準 Library；
- Prototype；
- const；
- Wider integer model；
- Atomics；
- Thread memory model；
- 新語法及屬性；
- 大量 Defect interpretation。

人物分析必須分期。

時期 問題 決策 複雜度去向 風格

1960s 大型系統語言與 Compiler 參與 Multics／學習 BCPL 語言與系統研究 系統導向

1969–70 Unix 需要較高階工具 Thompson 建立 B Compiler 小型系統語言

1971–73 B 無法映射 PDP-11 資料 C 型別化與第一 Compiler Type／Compiler 表示導向

1973 Unix 綁定 Assembly 大部分 Kernel 改寫 C Compiler／邊界 Assembly 可攜分區

1970s 語言靠小組傳播 K&R 公共文件 制度化

1980s 方言與實作分裂 ANSI C 委員會 相容標準

後期 Unix 模型需再研究 Plan 9 等團隊研究 新系統團隊 持續實驗

SECTION

五十四、主要原型

Dennis Ritchie 同時屬於：

- 可攜系統語言建築師；
- 機器透明抽象設計者；
- 語言—作業系統共演化工程師；
- 小型研究系統實作者；
- 克制式語言核心設計者。

SECTION

五十五、不適合的簡單標籤

不應只稱：

低階語言設計者
可攜 Assembly 發明者
Unix 唯二作者之一
K&R 單一作者
現代 ISO C 的唯一設計者
較精確的描述是：

> 將 B 的簡潔系統傳統重新型別化，使它足以表達 Unix，又不把機器和外部系統完全藏起來的設計者。

SECTION

五十六、最重要的連續性

從 C 到 Unix 可攜：

> 【保留必要機器控制
>
> 移除不必要硬體綁定】

SECTION

五十七、最重要的制度轉換

C 從：

Ritchie 與 Bell Labs 小組的實作語言
轉為：

K&R 定義的公共實踐
再轉為：

ANSI／ISO 委員會治理的標準語言

SECTION

五十八、最重要的責任代價

C 把低階能力普遍化，也把大量安全證明義務普遍化。

這是其力量與長期風險的共同來源。

Dennis Ritchie 的設計生涯顯示，一門系統語言可以同時：

- 對機器保持誠實；
- 對不同機器保持可攜；
- 具有足以組織大型系統的資料抽象；
- 不建立巨大 Runtime；
- 與 Assembly、Library、OS 及其他語言共存；
- 由真實完整系統反覆驗證。

  本文對 Ritchie 的 PLDST 判定為：

> 【Machine-Aware Language Engineer
> →
> Portable Systems Architect
> →
> Restrained Abstraction Designer】

其核心優勢是：

- 型別與資料表示緊密連接；
- 語言核心小而可自舉；
- Compiler 承擔大部分硬體差異；
- C 與 Unix 形成極強共演化證據；
- 語言不壟斷完整系統；
- 公共文件清楚、易於實作和傳播。

其核心代價是：

- 安全責任高度外移；
- Array、Pointer、Length 與 Lifetime 缺少統一保證；
- 小核心催生龐大工具及規範層；
- 相容性保留早期折衷；
- 現代最佳化使機器透明度不再完全直觀；
- 標準語言的複雜性已遠超早期個人設計。

最終原則為：

> 【抽象只提升到足以獲得可攜和結構的高度
>
> ∧
> 不隱藏系統實作者必須控制的資源】

但今天重新採用此原則時，還必須補上一條早期 C 未完整提供的要求：

> 若語言把低階能力交給一般使用者，就必須同步提供更強的邊界型別、診斷、分析與安全封裝制度，而不能只依靠熟練者的直覺。

人物：Dennis Ritchie
主要語言／系統：C、Unix

核心時期：1968–1980s
主要問題：系統軟體綁定 Assembly 與單一機器
主要策略：小型 Typed system language、Compiler、可攜邊界
複雜度去向：Compiler、Library、Programmer、Toolchain
責任去向：機器映射交給 Compiler，安全大量交給使用者
主要保護對象：OS／Compiler／Tool 實作者
主要限制：Memory safety、Undefined behavior、工具鏈與相容負擔
歸因信心：高

[R1] Dennis M. Ritchie, “The Development of the C Language,” HOPL II／Bell Labs, 1993.

— BCPL、B、C、PDP-11、型別、Compiler 及直接貢獻者。

[R2] Dennis M. Ritchie, “The Evolution of the Unix Time-sharing System,” 1984.

— Multics 退出、PDP-7／PDP-11、Unix 技術與社會演化。

[R3] Dennis M. Ritchie and Ken Thompson, “The UNIX Time-Sharing System,” Bell System Technical Journal, 1974／1978 editions.

— Unix 架構、系統功能、C 實作與可攜性。

[R4] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, 1978／1988.

— 公共教程、語言 Reference、Library 及 K&R 實踐。

[R5] Computer History Museum, Oral History of Brian Kernighan, 2017; Dennis Ritchie memorial materials.

— K&R 形成、早期 Tutorial、Bell Labs 團隊與公共傳播。

[R6] ANSI X3J11／ISO WG14, Rationale for the C89／C99 International Standard.

— K&R、共同現有實踐、相容、標準化與委員會治理。

[R7] Dennis M. Ritchie, “Reflections on Software Research,” ACM Turing Award Lecture, 1984.

— Bell Labs 研究條件、Unix 團隊、小型使用者—開發者循環及軟體研究制度。

[R8] ACM, Dennis M. Ritchie A.M. Turing Award materials.

— C、Unix 及泛用作業系統的歷史定位。

[R9] Nokia Bell Labs, Dennis Ritchie archive and biography.

— 原始歷史文件、職業資料及 Bell Labs 保存版本。

[T-M] Multics／BCPL learning phase

[T-C] B-to-C reconstruction phase
[T-U] Unix co-evolution phase
[T-K] K&R public-language phase
[T-S] Standards／post-founder phase
[S-P] Portability
[S-M] Machine transparency
[S-R] Restrained abstraction
[S-I] Implementability
[S-X] External-system interoperability
[S-C] Compiler-centered hardware mapping

附錄 D　第二輪史實與歸因校對紀錄
D.1 BCPL、B 與 C 的直接繼承
第二輪重新核對 Ritchie 的〈The Development of the C Language〉：
- Martin Richards 設計 BCPL；
- Ken Thompson 從 BCPL 發展 B；
- Ritchie 在 1971 至 1973 年間將 B 逐步改造成 C；
- 主要改動包括型別、資料表示、Structure 與適合 PDP-11 的編譯器；
- Ritchie 明確把 1972 至 1977 年間對語言細節有直接影響的人列為自己、Ken Thompson、Alan Snyder、Steven C. Johnson 與 Michael Lesk。
本文因此沒有使用「C 完全由 Ritchie 從零發明」的敘述。

D.2 Unix 以 C 重寫的時間與範圍
已重新核對 Ritchie 的 Unix 歷史回顧及 Ritchie／Thompson 的系統論文：
- Unix 早期主要以 PDP-11 組合語言實作；
- 1973 年前後，Kernel 的大部分被改寫為 C；
- 仍需保留啟動、陷阱、設備及其他機器專屬邊界；
- 當時 C 版本的系統體積約比原本組合語言版本大三分之一，但可理解性、可修改性及後續可攜性收益被認為值得；
- 「以 C 重寫」不等於作業系統從此完全不含組合語言。
本文因此將可攜性表述為「擴大可攜核心、縮小機器專屬邊界」，而不是機器獨立。

D.3 C 與抽象機器
第二輪核對 WG14 Rationale：
- C 標準定義的是抽象機器及可觀察行為；
- 實作可依 as-if 原則選擇指令、暫存器配置及轉換；
- 來源中的 Pointer、Object、Integer 與控制結構比許多高階語言更接近機器成本，但仍不是固定硬體指令序列；
- 「Portable assembly」只能作歷史隱喻，不能作精確規格描述。

D.4 K&R 的共同作者與公共化作用
已重新核對 Brian Kernighan 的口述歷史與 K&R：
- Kernighan 曾撰寫早期 C Tutorial；
- 《The C Programming Language》是 Kernighan 與 Ritchie 的共同作品；
- 書中的範例、文體、Library 教學與 Reference appendix 共同塑造公共 C；
- 第一版附錄長期成為標準化前的 De facto 參考；
- 因此「K&R 風格」與語言公共傳播不能只歸於 Ritchie。

D.5 ANSI／ISO 標準化的目標
第二輪核對 C89／C99 Rationale：
- 委員會目標是建立清楚、一致且無歧義的標準；
- 主要基礎是 K&R Appendix A 與當時大多數 Translator 的共同現有實踐；
- 同時允許把已被證明有價值、具有先例的改進正式化；
- 委員會明確追求跨環境的程式可攜；
- 這不是重新設計一門與既有 C 無關的理想語言。
本文將標準化描述為從小組語言到公共制度的權力轉移。

D.6 現代 C 的治理邊界
截至本文日期：
- 現行 ISO C 版本為 C23，於 2024 年完成國際標準採納；
- C23 與後續 Defect report、Implementation 和 Library 工作由 WG14 及各國標準機構治理；
- Ritchie 已不可能對這些後期功能、記憶體模型或解釋擁有直接裁決權；
- 現代 C 的安全、Pointer provenance、Atomic、Unicode 及屬性等議題不能全部回寫成 Ritchie 的個人設計。

D.7 未定義行為的歸因邊界
C 從早期即保留大量實作自由，以適應不同硬體和高效 Compiler。
但現代 Undefined behavior 制度由下列因素共同形成：
早期實作慣例

K&R

ANSI／ISO 規格

Compiler 最佳化

Defect report

硬體與 ABI

本文因此只將「保留機器與實作自由」歸為 Ritchie 的深層傾向，不把每一項現代 UB 規則視為他的直接決策。

D.8 Bell Labs 團隊與研究環境
第二輪核對 Ritchie 的圖靈獎演講及 Unix 歷史：
- Unix 的形成仰賴小型、互信、使用者與開發者重疊的研究環境；
- Ken Thompson 對早期 Unix 和 B 的權重極高；
- Douglas McIlroy、Joe Ossanna、Brian Kernighan、Steven Johnson、Michael Lesk 等人對工具、系統及公共文化具有獨立貢獻；
- 本文把「語言—作業系統共演化」視為 Ritchie 的風格證據，不把全部 Unix philosophy 歸於個人。

D.9 PLDST 推論邊界
下列名稱是本文分析原型，而非 Ritchie 自稱的正式學派：
可攜系統語言建築師

機器透明抽象設計者

克制式語言核心設計者

它們由 C 的型別化、Unix 重寫、外部互操作與小型核心等跨決策證據推導。
「克制」不表示 C 沒有歷史缺陷，也不表示所有省略的安全機制在今日仍然合理。