Bjarne Stroustrup：零額外成本、多範式與相容性政治

Bjarne Stroustrup: Zero-Overhead Abstraction, Multi-Paradigm Design, and the Politics of Compatibility

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-016.html

英文名稱： Bjarne Stroustrup: Zero-Overhead Abstraction, Multi-Paradigm Design, and the Politics of Compatibility

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-016

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

Bjarne Stroustrup 經常被描述成 C++ 的發明者、物件導向的推廣者，或者一門過度複雜語言的主要責任者。這些描述各有部分事實，卻容易忽略 C++ 原始問題的雙重約束：

- Simula 能提供 Class、Coroutine 及大型程式組織能力，但當時的實作成本、可攜性與系統接近性不足；
- C 能直接使用硬體、作業系統與既有工具鏈，但缺少適合大型系統的抽象和模組化能力。

  Stroustrup 於 1979 年在 Bell Labs 開始建立 C with Classes，目標不是把物件導向附加到 C 作為語法時尚，而是把 Simula 的程式組織能力帶入 C 的效率、可攜與系統程式世界。早期設計明確要求在 Runtime time、Code compactness 和 Data compactness 上與 C 匹配；即使曾出現約 3% 的系統性效率下降，也被視為不可接受並移除。[R1]

這形成後來著名的零額外成本原則：

不使用的功能，不應支付成本；
使用抽象的程式，不應比對應的精心手寫低階方案更差。

但此原則並不表示：

- Compiler 永遠能消除所有抽象；
- 編譯時間、錯誤訊息、二進位大小和語言學習成本皆為零；
- 所有 C++ 功能都符合相同成熟度；
- 使用者不需要理解資料表示及機器成本。

C++ 的第二個重要特徵是多範式。Stroustrup 從未把它限制成只使用 Class hierarchy 的物件導向語言。C++ 逐步結合：

- Procedural programming；
- Data abstraction；
- Object-oriented programming；
- Generic programming；
- Compile-time programming；
- Resource management；
- Value semantics；
- Functional techniques；
- Concurrency。

然而，多範式不是單純「什麼都加入」。其共同中心是：

> 【讓使用者建立與內建型別同等有效、同等可組合的抽象，
>
> 並保留對機器與資源的直接控制。】

C++ 的第三個特徵是相容性政治。Stroustrup 選擇 C，不是因為他認為 C 語法完美，而是因為：

- C 已被真實系統使用；
- C Compiler 和 Linker 廣泛存在；
- 使用者需要漸進遷移；
- C with Classes 必須與其他語言和系統組件共同連結；
- 設計者不能因自己的偏好剝奪使用者已有能力。[R1][R2]

  這個選擇促成成功，也建立長期負債。C 的 Declaration、Preprocessor、Implicit conversion、Raw pointer、Array model 和未檢查低階能力被保存；後來 C++ 又在 ANSI／ISO WG21 中由多國委員會、Compiler vendor、Library author、產業使用者及提案制度持續演化。今日 C++ 不再是 Stroustrup 可以單獨決定的語言。[R2][R3][R4]

  本文將 Stroustrup 的設計生涯分成六個相位：

- Simula—BCPL 經驗形成期：抽象與效率的斷裂；
- C with Classes 實用試驗期：Class、Constructor、Destructor、Static checking 與 C 相容；
- C++ 多範式擴張期：Virtual function、Operator、Exception、Template；
- 標準化與 STL 制度轉換期：WG21、Library、Generic programming 與相容政治；
- Modern C++ 再建模期：RAII、Value semantics、Move、Lambda、Concurrency、Concept；
- 後期方向與安全改革期：Profiles、Concept-based generic programming、靜態反射與 C／C++ 協調等持續提案。

  本文核心判斷為：

> 【Stroustrup 的核心風格不是功能堆疊，
>
> 而是拒絕在抽象能力、硬體控制、效率和既有世界之間只選一項。】

但他的最大爭議也來自相同立場：

> 【保留所有重要能力
> +
> 保持相容
> +

因此，C++ 的複雜性不能只歸因於 Stroustrup 個人，也不能完全推給委員會。原始「不剝奪能力、可漸進採用、與 C 共存、支援多種有效風格」的設計原則，本身就會在時間中生成龐大的交互與相容成本。

關鍵詞： Bjarne Stroustrup、C++、C with Classes、零額外成本、多範式、RAII、泛型程式設計、相容性、WG21、PLDST

SECTION

一、本文研究範圍

本文主要分析：

- Stroustrup 的 BCPL／Simula 經驗；
- C with Classes；
- 1979–1991 年 C++ 形成；
- 1985–1998 年公共化及標準化；
- STL／Generic programming 的整合；
- C++11 以後的 Modern C++ 方向；
- Stroustrup 對設計原則、教育、安全與後期演化的持續主張。

  本文不把每項 C++ 標準功能、Library 或 Compiler 全部歸於 Stroustrup。

SECTION

二、C++ 的創始權重

Stroustrup 具有高度可直接歸因的工作：

- 問題設定；
- C with Classes；
- 第一批語言定義；
- 第一個實作；

- Cfront；
- Class、Constructor／Destructor、Virtual function 等早期設計；
- 多項核心準則；
- 早期文件與使用者支援；
- 長期 Evolution Working Group 參與。

因此稱其為 C++ 創造者是合理的。

SECTION

三、C++ 不是單人長期作品

Stroustrup 的 HOPL 歷史反覆強調人、使用者、約束與標準制度。

重要共同影響者包括：

- Ole-Johan Dahl、Kristen Nygaard：Simula；
- Dennis Ritchie：C、系統語言及 Bell Labs 回饋；
- Douglas McIlroy：大量關鍵設計討論；
- Steve Johnson：Compiler 與語言經驗；
- Sandy Fraser、Sudhir Agrawal、Jonathan Shopiro：早期真實應用；
- Andrew Koenig、Stan Lippman、Jonathan Shopiro、Tom Cargill 等 Bell Labs 使用者與實作者；
- Alexander Stepanov、David Musser、Meng Lee：Generic programming 與 STL；
- ANSI／ISO WG21 各工作組與國家代表；
- Compiler、Library、Boost 及產業社群。[R1][R2][R3]

因此：

原始問題與核心準則：Stroustrup 極高
早期實作與文件：Stroustrup 高
應用回饋：Bell Labs 使用者共同

STL／Generic library：Stepanov、Lee、Musser 等核心
標準化後功能：WG21 多主體
當代 C++：制度、實作者與生態共同

SECTION

四、Simula 提供組織能力

Stroustrup 在劍橋研究分散式系統時，使用 Simula 經驗到：

- Class；
- Inheritance；
- Virtual procedure；
- Coroutine；
- Static type checking；
- 以抽象表示系統實體。

  它讓大型系統概念能直接反映在程式結構中。

SECTION

五、Simula 的實作代價

在當時硬體與 Compiler 條件下，Simula 的問題包括：

- 執行速度；
- Compiler 成本；
- Portability；
- 與 Unix／C Toolchain 整合；
- 低階硬體控制。

  它能描述 Stroustrup 想建造的分散式系統，卻難以成為真實平台工具。

SECTION

六、BCPL／C 提供機器能力

BCPL 與 C 能：

- 直接使用 Memory；
- 建立 OS；
- 連接 Library；
- 使用現有 Linker；
- 移植到不同機器；
- 生成小型高效程式。

  但它們缺乏高階程式組織機制。

SECTION

七、原始問題不是「增加物件導向」

Stroustrup 的問題可表述為：

> Need
> =
> SimulaOrganization
> +
> CSystemsControl
> +
> PortableImplementation
> +
> NoUnnecessaryOverhead

這個交集在當時沒有現成語言。

SECTION

八、初始使命

C with Classes 不是完整替代 C 的新世界，而是：

- 保留 C；
- 增加 Class；
- 改善程式組織；
- 保持性能；
- 可在半年內服務真實專案。

Stroustrup 強調，計算本身已由 C 解決，新的問題是組織。[R1]

SECTION

九、早期核心

1980 年左右的 C with Classes 已包含：

- Class；
- Derived class；
- Public／Private；
- Constructor；
- Destructor；
- Friend；
- Function argument type checking；
- 後續 Inline；
- Default argument；

- Assignment operator overload。[R1]

SECTION

十、Class 是使用者定義型別

Class 的核心意義不是建立 GUI Widget 階層，而是：

> Type
> =
> Representation
> +
> Invariant
> +
> Operations
> +
> Construction
> +
> Destruction

使用者可以建立與內建型別相近的：

- Stack；
- Vector；
- File；
- Task；
- Complex；
- Device abstraction。

SECTION

十一、Constructor／Destructor 與資源責任

Constructor 和 Destructor 不只管理 Memory。

它們可管理：

- Lock；
- File；
- Socket；
- Transaction；
- Temporary state；
- Device；
- Memory。

  這後來形成 RAII：

> ResourceLifetime
> =
> ObjectLifetime

使資源責任能由型別和 Scope 共同管理。

SECTION

十二、早期零額外成本壓力

C with Classes 明確要求：

- Runtime 與 C 匹配；
- Code compactness 與 C 匹配；
- Data compactness 與 C 匹配；
- 不在所有 Object 中加入不必要 Housekeeping data。

  這不是後來的宣傳口號，而是第一實作的工程准入條件。[R1]

SECTION

十三、不移除 C 的低階能力

Stroustrup 保留：

- Pointer；
- Cast；
- Bit operation；
- Manual memory；
- Union；
- C Linkage；
- Dangerous operation。

他的理由不是認為這些都安全，而是：

- C 已能解決重要系統問題；
- 新語言不能因設計者偏見使真實工作變得不可能；
- C++ 應減少使用不安全操作的必要性，而不是完全移除低階出口。[R1]

SECTION

十四、不是因為 C 最漂亮

Stroustrup 明確承認：

- C 不是最乾淨語言；
- Declaration syntax 令人痛苦；
- 某些語義和轉換不理想；
- 早期嘗試修正時受到使用者相容需求阻力。[R1]

SECTION

十五、C 的一階優勢

其選擇依據包括：

- Flexibility；
- Efficiency；
- Availability；
- Portability；
- Unix；
- Existing programmers；
- Linker；
- Libraries；
- Hardware access。

  這些現實優勢大於表面設計缺陷。

SECTION

十六、相容性作為採用技術

令轉換成本為：

> MigrationCost
> =
> LanguageLearning
> +
> CodeRewrite
> +
> ToolReplacement
> +

以 C 為基礎可降低：

- Code rewrite；
- Tool replacement；
- Library loss；
- Performance risk。

因此 C 相容是社會部署機制，不只是技術選擇。

SECTION

十七、「一門系統中的語言」

Stroustrup 接受 C Linker，並形成一項重要原則：

> C++ 是系統中的一門語言，不是完整封閉環境。

因此它必須：

- 與 C、Fortran、Assembly 等共同工作；
- 使用既有 OS；
- 連接外部 Binary；
- 接受部分程式資訊不可見；
- 不要求所有世界由 C++ Runtime 控制。[R1]

SECTION

十八、Virtual function 與真正 OOP

早期 Class 主要支援 Data abstraction。

加入 Virtual function 後，語言才能更直接支援：

- Interface-based polymorphism；

- Substitution；
- Dynamic dispatch；
- Simula 式 OOP。

  Stroustrup 自己不把最初 C with Classes 宣稱成完整 OOP 語言。[R1]

SECTION

十九、Operator overloading

Operator overloading 允許：

```
Complex a, b;
auto c = a + b;
```

其目標是讓使用者定義型別能以接近內建型別的表示工作。

優勢：

- 抽象自然；
- Generic algorithm 可統一；
- Value semantics。

  風險：

- 語義濫用；
- 成本不透明；
- 過度聰明的 DSL；
- 不同 Library 慣例衝突。

SECTION

二十、Exception 與 RAII

Exception 允許錯誤跨多層傳遞；RAII 使 Stack unwinding 期間資源自動釋放。

兩者形成：

> ErrorPropagation
> +
> DeterministicCleanup

但也帶來：

- ABI；
- Compiler；
- Exception safety；
- Hidden control flow；
- Real-time concern。

  C++ 並未要求所有領域都必須使用 Exception，而是保留多種錯誤模型。

SECTION

二十一、Template 與 Generic programming

Template 最初提供 Parameterized type／function。

後來經 Stepanov 等人的 Generic programming 理論和 STL 實踐，C++ 形成：

- Container；
- Iterator；
- Algorithm；
- Function object；
- Compile-time polymorphism；
- Specialization。

  這使 C++ 從「C 加 Class」轉向真正多範式。

SECTION

二十二、STL 不是 Stroustrup 單人設計

Alexander Stepanov 和 Meng Lee 的 Library、David Musser 等人的 Generic programming 工作，是 STL 的核心來源。

Stroustrup 和標準化制度的重要作用包括：

- 支持 Generic programming 進入主流；
- 將 STL 納入標準；
- 協調 Language 與 Library；
- 使 Template 成為一般抽象機制。

  但 STL 的具體算法—容器架構不可歸於 Stroustrup 單人。

SECTION

二十三、第一部分：不用不付費

若程式不使用：

- Virtual dispatch；
- Exception；
- RTTI；
- Thread；
- 某個 Library；

  不應被迫支付其一般 Runtime 成本。

SECTION

二十四、第二部分：使用時接近手寫最佳方案

若使用：

- vector；
- RAII；
- Template algorithm；
- Class abstraction；

  其性能目標是不遜於合理的手寫低階等價實作。

  比較基線必須明確：

> Cost(Abstract)
> ≤
> Cost(EquivalentHandwritten)
> +
> UnavoidableDifference

SECTION

二十五、不包括哪些成本

零額外成本不保證：

- Compile time 零增加；
- Binary size 不增加；
- Error message 簡單；
- Debug experience 無成本；
- 每個 Compiler 都最佳化成功；
- 所有 Debug build 都高速；
- 每個抽象都設計良好；
- 使用者不需理解表示。

SECTION

二十六、抽象失效時的責任

C++ 需要：

- Compiler explorer；
- Optimization report；
- Benchmark；
- Profile；
- Assembly／IR；
- Allocation analysis。

「理論上可以零成本」不等於特定程式已達成。

SECTION

二十七、Procedural

適合：

- 明確流程；
- 小型函式；
- 系統 API；
- C Interop。

SECTION

二十八、Data abstraction

適合：

- Invariant；
- Resource；
- User-defined type；
- Value。

  這是 C++ 最早核心。

SECTION

二十九、Object-oriented

適合：

- Dynamic interface；
- Heterogeneous collection；
- Plugin；
- Runtime polymorphism。

  不是所有問題都需要 Class hierarchy。

SECTION

三十、Generic programming

適合：

- Algorithm independent of representation；
- Static polymorphism；
- Container；
- Concept constraint；
- 高性能重用。

SECTION

三十一、Compile-time programming

Template metaprogramming、constexpr、Concept、Reflection 等使：

- 驗證；
- 生成；
- 計算；
- 介面適配；

  可移至編譯期。

  但也增加：

- Compile time；
- Tool complexity；
- Error；
- Code generation burden。

SECTION

三十二、Functional techniques

Lambda、Immutable value、Algorithm pipeline 和 Higher-order operation 可以在 C++ 中使用，卻不要求整個語言成為純函數式。

SECTION

三十三、共同中心：型別化抽象與資源

多範式不是無中心集合。

其共同核心可表示為：

> 【User-defined type
> +
> Value／Reference semantics
> +
> Resource lifetime
> +
> Static composition
> +
> Optional dynamic polymorphism】

SECTION

三十四、C++ 想消除「需要不安全」而非「可能不安全」

Stroustrup 的早期原則是：

- 保留 C 的低階能力；
- 建立更高階安全抽象；
- 讓不安全操作只在真正必要時出現；
- 不強迫單一 Style。

這是：

> UnsafeNeeddownarrow
>
> UnsafeCapabilityretained

SECTION

三十五、RAII 的安全作用

RAII 可排除大量：

- Leak；

- Missing unlock；
- Partial construction；
- Error-path cleanup；
- Manual ownership confusion。

但 Raw pointer、Alias 和跨 Thread 行為仍需規範。

SECTION

三十六、Type safety 的不完整性

C++ 提供：

- Static checking；
- Constructor invariant；
- Access control；
- Template constraints；
- Smart pointer；
- Bounds-aware Library 的可能性。

同時保留：

- C-style cast；
- Reinterpretation；
- Raw Array；
- Pointer arithmetic；
- Union；
- Unchecked indexing；

- Undefined behavior。

因此安全依賴使用者選擇的 C++ 子集和 Library。

SECTION

三十七、Profiles 與後期安全方向

Stroustrup 及 WG21 近年持續討論：

- Profiles；
- Lifetime analysis；
- Bounds safety；
- Type safety；
- C／C++ Liaison；
- 更安全 Library；
- Static analysis。

這反映一個後期修正：

> 在不破壞巨大相容基礎的條件下，試圖把 Modern C++ 的安全子語言制度化。

截至 2026 年，具體 Profiles 和安全方案仍處於持續提案、實驗與委員會演化中，不能當成全部已穩定落地。

SECTION

三十八、相容性不是單純技術保守

C++ 的相容對象包括：

- C Source；
- 舊 C++ Source；
- ABI；

- Linker；
- Vendor；
- Library；
- Hardware；
- 已部署產品；
- 組織知識。

  任何「清理語言」提案都可能讓不同群體支付成本。

SECTION

三十九、早期使用者已限制設計

Stroustrup 曾嘗試修正：

- Declaration；
- Narrowing；
- C syntax；
- Type checking。

  但真實 C 程式和使用者拒絕大規模不相容。

  這說明：

> CompatibilityLockIn
> 在正式標準前已形成

SECTION

四十、保留能力的政治立場

Stroustrup 擔心語言設計者因：

- Paternalism；
- 不理解領域；
- 偏愛某種風格；

  移除使用者需要的能力。

  因此，他偏好：

- 支援多種已證明有效 Style；
- 提供工具避免陷阱；
- 不把所有人限制在單一方法。

SECTION

四十一、相容的長期代價

保留 C 和舊 C++ 意味：

- 多套初始化；
- 多套 Cast；
- Raw 與 Smart pointer；
- Macro 與 Template；
- C Array 與 Container；
- Legacy I/O 與 Modern Library；
- 多代 Error model；
- 複雜 Overload resolution。

  新語言層必須與舊層交互。

SECTION

四十二、Modern C++ 是文化遷移而非刪除

由於舊功能不能輕易移除，C++ 常以：

- Guideline；
- Linter；
- Core Guidelines；
- Modern Library；
- Teaching；
- Profile；

  推動新 Style。

  因此實際推薦語言是：

> $$\text{Recommended}C++ \subset \text{Standard}C++$$

SECTION

四十三、WG21 的形成

ANSI／ISO C++ 標準化在 1990–1991 年前後正式制度化，目標包括：

- 公共規格；
- 多 Compiler；
- Library；
- 不由單一公司控制；

- 國際採用；
- 長期相容。

SECTION

四十四、Stroustrup 的權力變化

早期：

> DesignAuthority_(Stroustrup)
> ≈ High

標準化後：

> ProposalInfluence_(Stroustrup)
> remains high

但：

> FinalAuthority
> =
> WG21
> +
> NationalBodyVoting
> +
> ConsensusProcess

SECTION

四十五、提案不是個人意見即規格

Stroustrup 自己在 Committee paper 頁面提醒：

- 提案是特定時間、特定討論的探索；
- 不完整；

- 可能失敗；
- 不能單獨視為 C++ 規格。[R5]

因此後期個案研究必須區分：

Stroustrup proposal
WG21 adopted design
Compiler implementation
Community convention

SECTION

四十六、委員會的必要性

C++ 服務：

- Embedded；
- Finance；
- Games；
- Browser；
- OS；
- HPC；
- Automobile；
- Telecom；
- Scientific computing。

任何單一設計者都無法完整掌握全部限制。

委員會提供代表性和實作審查。

SECTION

四十七、委員會的代價

多方制度也可能造成：

- Feature coalition；
- 交互不足；
- 一致性降低；
- 語言和 Library 同時膨脹；
- 提案數量巨大；
- 實作落差；
- 教學跟不上標準。

Stroustrup 對 C++ 方向的長期工作，部分就是試圖讓制度仍保有共同設計原則。

SECTION

四十八、從 Class hierarchy 轉向 Value 與 Generic

C++11 以後的推薦 C++ 更強調：

- Value semantics；
- RAII；
- Move；
- Smart pointer；
- Range；
- Generic algorithm；
- Lambda；
- Concurrency；
- Type inference；

- constexpr。

  這不等於拋棄 OOP，而是恢復 C++ 從來具有的多範式方向。

SECTION

四十九、Move semantics

Move 使資源擁有型別能：

- 保持 Value interface；
- 避免不必要 Deep copy；
- 支援 Container；
- 表達 Ownership transfer。

  它解決的是：

> ValueAbstraction
> +
> ResourceEfficiency

之間的張力。

SECTION

五十、Concept

Concept 的目標是：

- 為 Generic parameter 表達語義需求；
- 改善 Overload；
- 改善 Error；
- 支援可組合 Generic programming；

- 讓 User-defined constraint 進入型別系統。

  其設計歷經多年提案、失敗版本和多位研究者貢獻，不能只歸於 Stroustrup。

SECTION

五十一、Concurrency

C++11 將：

- Thread；
- Atomic；
- Memory model；
- Synchronization；

  納入標準。

  這使 C++ 不再把並行完全交給平台，但也增加：

- 語言—硬體語義；
- Undefined behavior；
- Library；
- 教學；

  的巨大負擔。

SECTION

五十二、Simula／BCPL 經驗期

問題：抽象與系統效率分裂

SECTION

五十三、C with Classes 期

問題：C 無法組織大型系統
策略：Class、Construction、Static checking、零額外成本

SECTION

五十四、早期 C++ 期

問題：需要動態多型、Error、泛型
策略：Virtual、Exception、Template

SECTION

五十五、標準化／STL 期

問題：語言需成為多廠商公共平台
策略：WG21、STL、Library、相容規格

SECTION

五十六、Modern C++ 期

問題：Legacy C++ Style 不安全且冗長
策略：RAII、Value、Move、Lambda、Generic、Concurrency

SECTION

五十七、後期方向期

問題：語言巨大、安全弱、工具和教學分裂
策略：Concept、Guideline、Profile、Reflection、方向文件

SECTION

五十八、問題 framing

Stroustrup 的核心問題是：

> 如何讓系統程式設計者在最高可行抽象層工作，又不失去低階資源控制、可攜性、效率和既有系統？

SECTION

五十九、價值優先序

> V_(Stroustrup)
> ≈
> (
> Abstraction,
> Efficiency,
> Generality,
> TypeSafety,
> ResourceControl,
> Compatibility,
> RealWorldUse
>)

SECTION

六十、核心—擴張偏好

偏好：

- 一般機制；
- User-defined type；
- Library；
- Generic；
- 多範式；
- 不移除已證明重要能力。

  風險是核心和交互持續擴大。

SECTION

六十一、顯式—推導偏好

C++ 同時存在：

- Explicit resource；
- Static type；
- auto；
- Template deduction；
- Overload；
- ADL；
- Implicit conversion。

  Stroustrup 傾向讓機器推導重複型別，但保留成本和資源的可控制性。

SECTION

六十二、效率—可讀性偏好

目標是：

> HighestFeasibleAbstraction
>
> without unnecessary overhead

「可行」由 Compiler、Hardware 和真實應用決定。

SECTION

六十三、安全—自由偏好

偏好：

- 提供 Safe abstraction；
- 保留 Unsafe escape；
- 不強迫單一風格；

- 以 Library、Guideline 和分析降低危險。

  其弱點是 Safe subset 缺乏單一強制邊界。

SECTION

六十四、相容性偏好

相容性是 C++ 採用與存續的核心，但也被 Stroustrup 視為大量 Incidental feature 和教學複雜性的來源。

這不是無條件讚美，而是持續承擔的歷史契約。

SECTION

六十五、治理偏好

早期是集中原型；後期接受：

- 公開標準；
- 國際代表；
- 實作證據；
- Consensus；
- Library 與 Core 分組。

  同時持續主張需要共同方向，避免委員會僅累積功能。

SECTION

六十六、C++ 不是只為 OOP

Stroustrup 明確否認自己發明 OOP，也不把 C++ 限定為狹義物件導向。

C++ 的 Generic、Value、Procedural 和 Compile-time 模型同樣核心。

SECTION

六十七、零額外成本不是經驗保證

某個抽象可能因：

- Compiler；
- Build mode；
- ABI；
- Lost inlining；
- Allocation；
- Dynamic dispatch；

  產生成本。

  需要實際證據。

SECTION

六十八、C 相容不是完整 C 子集保證

C 與 C++ 在：

- Type；
- Keyword；
- Conversion；
- Linkage；
- Library；
- Compound literal；
- Variable length array；

- _Generic 等；

  存在差異。

  「C++ 以 C 為基礎」不能簡化成現代 C 程式皆為合法同義 C++。

SECTION

六十九、功能一般性不等於使用簡單

Template、Overload、Multiple inheritance、Concept 和 Coroutine 即使一般，也可能：

- 交互複雜；
- 診斷困難；
- 教學成本高；
- 形成多種 Style。

SECTION

七十、多範式可能變成語言聯合體

當共同中心不清楚時，使用者可能把 C++ 當成：

- Better C；
- Class-only OOP；
- Template metaprogramming；
- Functional-like；
- Embedded subset；

  不同團隊無法閱讀彼此程式。

SECTION

七十一、委員會不能吸收無限提案

標準每加入一項功能，都需：

- 多實作；
- Tool；
- Library；
- 相容；
- 教學；
- 與所有舊功能交互。

週期性標準發布可能提高演化速度，也放大吸收負擔。

SECTION

七十二、Stroustrup 的回顧具有作者立場

HOPL 與《The Design and Evolution of C++》是不可替代的一手來源，但仍是創始者敘事。

需與：

- WG21 文件；
- 其他參與者；
- Compiler history；
- Library history；
- 使用者資料；

交叉校對。

時期	問題	決策	複雜度去向	風格
1970s	Simula 抽象與 BCPL 效率分裂	結合兩者方向	Compiler／Language	問題綜合
1979–80	C 難組織分散式系統	C with Classes	Type／Class	實用抽象
1980s	需要動態多型與資源模型	Virtual、Constructor、Destructor	Compiler／ABI	零額外成本

1980s　使用者定義型別需自然表達　Operator、Exception、Template　語言交互　一般機制

1990s　多廠商語言需公共平台　WG21／ISO　委員會　制度化

1994–98　Generic Library 需要標準化　STL　Library／Template　多範式

C++11 後　Legacy Style 阻礙安全與可讀　Move、Lambda、RAII、Concurrency　Compiler／Library　現代化

近年　安全與複雜度壓力　Concept、Profile、Guideline　Tool／Governance　反身改革

SECTION

七十三、主要原型

Bjarne Stroustrup 同時屬於：

- 零額外成本抽象設計者；
- 多範式系統語言建築師；
- 使用者定義型別推動者；
- 相容性現實主義者；
- 長期標準治理參與者。

SECTION

七十四、不適合的簡單標籤

不應只稱：

OOP 語言設計者
C 加 Class 的作者
功能堆疊者
C++ 永久獨裁者
零成本承諾者

較精確的描述是：

> 試圖讓高階抽象、硬體控制和既有系統同時成立，並因拒絕簡單犧牲任何一端而承擔巨大語言與治理複雜度的設計者。

SECTION

七十五、最重要的連續性

1979 至今的共同方向是：

> 【以型別與 Library 提升抽象
>
> ∧
> 不失去機器和資源控制】

SECTION

七十六、最重要的制度轉換

C++ 從：

Stroustrup 的 C with Classes 原型
轉為：

Bell Labs 使用者塑造的 C++
再轉為：

WG21 多國標準和全球生態

SECTION

七十七、最重要的設計矛盾

C++ 希望同時：

- 保持 C；
- 變得更安全；
- 支援新範式；

- 不增加不必要成本；
- 服務所有系統領域；
- 不分裂生態。

這些目標在局部可協調，在數十年演化後必然產生張力。

Bjarne Stroustrup 的設計不能以「C++ 太複雜」一句話結案，也不能因 C++ 成功而把每一項歷史選擇合理化。

他的核心貢獻是證明：

- 系統語言可以支援高階抽象；
- User-defined type 可以接近內建型別的效率；
- Resource management 可以由型別和 Scope 系統化；
- Object-oriented 與 Generic programming 都能服務低階系統；
- Library 可以承擔大量語言能力；
- 高效能不必永遠以 Assembly-like code 換取；
- 語言可以漸進進入既有產業。

本文對 Stroustrup 的 PLDST 判定為：

> 【Zero-Overhead Abstraction Engineer
> →
> Multi-Paradigm Systems Architect
> →
> Compatibility-Constrained Language Statesman】

其核心優勢是：

- 抽象與效率不再被視為必然對立；
- RAII 建立一般資源責任模型；
- 多範式服務不同問題；

- C Toolchain 和生態能漸進遷移；
- 語言可用於極廣泛領域；
- 長期接受公共標準治理。

  其核心代價是：

- C 與早期 C++ Legacy 長期存在；
- 功能交互和教學負擔巨大；
- Safe subset 依賴文化和工具；
- Compiler、Build 和診斷成本上升；
- 委員會演化不易維持整體一致；
- 「保留選擇」本身成為使用者必須理解的複雜度。

  最終原則為：

> 【不要要求程式設計者在抽象與效率間二選一】

但這條原則必須配合另一個同樣重要的限制：

> 【每保留一種能力與相容路徑，
>
> 都必須計算其交互、教學、安全和治理成本。】

C++ 的歷史最終不是「一門完美語言如何誕生」，而是：

> 一個拒絕簡單犧牲真實需求的設計，如何成功成為全球基礎設施，又如何被自己所保留的世界持續約束。

人物：Bjarne Stroustrup
主要語言／系統：C with Classes、C++
核心時期：1979–至今
主要問題：高階程式組織與系統效率分裂
主要策略：Class、RAII、Template、Value、零額外成本
複雜度去向：Compiler、Language、Library、Committee
責任去向：型別管理資源，使用者保留低階控制

主要保護對象：系統及基礎設施程式設計者
主要限制：Legacy、交互、診斷、安全子集與治理規模
歸因信心：高

[R1] Bjarne Stroustrup, “A History of C++: 1979–1991,” HOPL II, 1993.

— C with Classes、Simula／C 來源、早期性能目標、使用者和設計決策。

[R2] Bjarne Stroustrup, The Design and Evolution of C++, 1994.

— 原始設計準則、相容、功能取捨及標準化前史。

[R3] Bjarne Stroustrup, “Evolving a Language in and for the Real World: C++ 1991–2006,” HOPL III, 2007.

— WG21、STL、Generic programming、C++98／C++0x 及多主體演化。

[R4] Bjarne Stroustrup, “Foundations of C++,” ETAPS Keynote, 2012.

— 基本構造、零額外成本、資源管理、型別安全與 Modern C++。

[R5] Bjarne Stroustrup, Official WG21 paper archive and publication list.

— 後期提案、設計空間、未採用方案與個人意見／正式標準邊界。

[R6] Computer History Museum, “Oral History of Bjarne Stroustrup,” 2015.

— Simula、C、Unix、Bell Labs、分散式系統動機與標準制度回顧。

[R7] ISO/IEC JTC1/SC22/WG21, Standing Documents and “Direction for ISO C++.”

— 委員會程序、Library／Language compatibility、Direction、Profiles 與安全改革。

[R8] Computer History Museum Software Preservation Group, C++ Historical Sources Archive.

— Cfront、早期文件、新聞、Compiler 及歷史保存。

[R9] Alexander Stepanov, Meng Lee, David Musser and related STL／Generic Programming historical documents.

— STL 和 Generic programming 的獨立來源與共同歸因。

[T-S] Simula／BCPL experience phase
[T-C] C with Classes phase
[T-E] Early C++ expansion phase
[T-W] WG21／STL standardization phase
[T-M] Modern C++ phase
[T-R] Safety／direction reform phase

[S-Z] Zero-overhead abstraction
[S-P] Multi-paradigm
[S-R] Resource management
[S-G] Generic programming
[S-C] Compatibility realism
[S-I] ISO governance

附錄 D　第二輪史實與歸因校對紀錄
D.1 C with Classes 的性能門檻
第二輪重新核對〈A History of C++: 1979–1991〉：
- C with Classes 的初始目標明確包括 Runtime time、Code compactness 與 Data compactness 應與 C 匹配；
- 某版實作曾造成約 3% 的系統性效率下降，Stroustrup 判定不可接受並移除；
- 此史實支持「零額外成本」在早期即是工程准入條件，而非 C++ 成功後才建立的宣傳敘事；
- 但該原則是設計目標與比較框架，不是每個程式、每個 Compiler、每種 Build mode 的逐案證明。

D.2 Simula、BCPL／C 與原始問題
已重新核對 Stroustrup 的 HOPL 論文與 CHM 口述歷史：
- Simula 提供 Class、Virtual procedure、Coroutine 和大型程式組織能力；
- BCPL／C 提供效率、可攜、Unix 工具鏈與低階控制；
- Stroustrup 的原始問題確實是讓兩者優勢同時成立；
- C with Classes 最初主要解決 Program organization，並非一開始就具有所有後來的 Object-oriented、Generic 或 Compile-time 特徵。
本文因此沒有把 1979 年 C with Classes 寫成已完成的現代 C++。

D.3 C 相容與「一門系統中的語言」
第二輪核對原始歷史：
- Stroustrup 選 C 的理由包含效率、彈性、可用性、可攜、Unix、既有使用者及 Linker；
- 他明確把 C++ 視為系統中的一門語言，而不是要求全世界由單一 C++ Runtime 管理；
- 與 C、Fortran、Assembly 及既有 Binary 共存是採用策略的一部分；
- 這不表示現代 C 是現代 C++ 的完整語法子集，兩者在型別、轉換、Keyword、Library 及後續標準功能上已存在差異。

D.4 低階能力與安全歸因
Stroustrup 的早期論文明確顯示：
- 他刻意保留 C 的低階及可能不安全能力；
- 理由是語言不能因設計者偏好剝奪使用者完成真實工作的能力；
- 同時希望 Class、Constructor、Type checking 和 Library 降低必須直接使用危險操作的頻率；
- `unsafe capability retained` 與 `unsafe operation recommended` 不是同一判斷。
本文把這一配置稱為「降低不安全需求、保留不安全能力」，而不是把 C++ 描述成安全語言或放任語言。

D.5 RAII 與資源模型
第二輪校對保留以下分層：
- Constructor／Destructor 是 C with Classes 的早期核心；
- 以物件生命期管理一般資源的 RAII 思想在 C++ 實踐中逐步成熟；
- Exception safety、Smart pointer、Container 和 Move semantics 共同擴大此模型；
- 不能把所有 RAII Library、Exception-safety rule 或 Smart pointer 設計歸為 Stroustrup 單人發明。

本文將其視為 Stroustrup 長期支持的資源責任模型，而非單一瞬間完成的功能。

D.6 Template、Generic programming 與 STL

已重新核對 HOPL III 及相關歷史材料：

- Template 是 C++ 語言核心的重要發展；
- Alexander Stepanov、David Musser、Meng Lee 等人對 Generic programming 和 STL 具有不可替代的直接貢獻；
- STL 的 Container／Iterator／Algorithm 分離不應歸入 Stroustrup 個人設計；
- Stroustrup 的作用包括支持一般抽象方向、協助語言能力及標準制度容納 STL。

本文因此區分：

Template language mechanism

Generic-programming theory

STL library architecture

WG21 adoption

D.7 C++ 標準化的權力轉移

第二輪核對 WG21 官方資料：

- ANSI／ISO C++ 標準化工作組於 1990–1991 年形成；
- 現代決策由 WG21 工作組、國家會員、提案、投票、實作及共識程序共同構成；
- Stroustrup 仍是高影響力設計者與提案作者，但沒有單人最終裁決權；
- 個人論文只能表示特定時間的建議或探索，不等於已採用標準。

D.8 Modern C++ 的多作者性

C++11 至 C++26 時期的：

- Move semantics；
- Lambda；
- Concurrency memory model；
- Concept；
- Coroutine；
- Module；
- Range；
- Format；
- Reflection；

均涉及不同提案作者、Library 團隊、Compiler 實作者及委員會決策。

本文只把「Value、RAII、Generic、零額外成本及高階抽象」的長期方向歸為 Stroustrup 持續倡議，不把所有現代功能回寫成其單人作品。

D.9 2026 年 Direction、Profiles 與安全提案

截至本文日期重新核對 WG21 文件：

- P2000R5《Direction for ISO C++》日期為 2026 年 2 月，是 Direction Group 的方向性文件，不是語言規格本身；
- P3984R0《A type-safety profile》日期為 2026 年 1 月，是 Stroustrup 等人的草案提案，不代表完整 Profiles 已成為穩定標準；
- 2026 年 WG21 Paper index 顯示安全、Contracts、Profiles、Reflection 及 C／C++ 協調仍在持續演化；

- 本文因此只把這些材料當成後期問題意識及方向證據，不聲稱安全改革已全部落地。

D.10 標準程式庫與語言規模

HOPL IV 對 2006–2020 年 C++ 演化的回顧指出：

- 現代標準程式庫在 C++20 規格中佔有極大篇幅；
- C++ 演化包含大量成功、失敗、撤回及延期方案；
- Library、Core language 和 Tooling 已形成多主體制度；
- 「C++ 的複雜性」不能只按 Keyword 或 Stroustrup 個人提案數量解釋。

本文將複雜度來源分成：

原始多目標原則

C／舊 C++ 相容

多領域需求

標準委員會與生態長期疊加

D.11 PLDST 推論邊界

下列名稱為本文分析原型，而非 Stroustrup 自稱的正式學派：

零額外成本抽象設計者

多範式系統語言建築師

相容性現實主義者

相容受限的語言政治家

其中「語言政治家」只表示其長期在國際標準、產業利益、相容與共同方向之間協調，不表示他具有單人統治權。