序

程式語言設計是個迷人的主題。許多程式設計師認為他們可以設計一種程式語言，勝過他們目前正在使用的語言，也有許多研究者相信他們可以設計出超越目前使用的語言。他們的抱負或許令人稱許，但極少數他們所設計的結果曾離開他們的抽屜，你將會發現這些人的故事出現在本書中。

程式語言設計是個嚴肅的工作。語言設計的小錯誤可導致軟體實作時更大的錯誤，甚至小錯誤也會產生極度嚴重的後果。廣泛採用的軟體常因反覆被惡意病毒攻擊而導致數以百萬計的財務損失，對世界經濟造成損害。因此，程式語言的安全性也是本書一再強調的主題。

程式語言設計之路是無法預期結果的探險歷程。程式語言原先設計用於廣泛的應用，即使有龐大組織贊助，最後難免僅用於特殊領域市場。相反的，程式語言設計初衷鎖定為地方性使用，反而贏得廣泛的顧客群，並擴展到不同環境或平台之應用，超乎原設計者們的想像。本書專注於介紹後者這類型的程式設計始祖們的故事。

成功的程式語言有個共同的特質：皆起始於某人或某個小團隊志同道合成員們最初的想法或計畫。這些程式設計的開山祖師們即為程式設計過程的首腦；他們擁有經驗、有遠景、有衝勁、有毅力，及天賦推動程式語言之初期執行與後續發展，並藉由實際使用與讓其合法存在，推動程式語言的標準化。

藉由本書，讀者將能直接接觸這一群程式語言的開山祖師們。他們每個人給予我們額外的訪談時間，與我們分享他們成功背後的因素與心路歷程，但不容否認，正確的決策加上好運相輔相成。最後，這些訪談對話過程內容之出版也讓我們對於這些程式語言創始者的人格特質、動機有深入了解的機會，這些特質與他們程式語言設計的作品本身一樣迷人。

— Sir Tony Hoare

Sir Tony Hoare，ACM 的得獎者，在 *50* 年的研究生涯中，是電腦演算法及程式語言研究之先驅，他的第一篇學術報告，撰寫於 *1969* 年，是關於探索如何驗證軟體之正確性，並建議程式語言設計的目標是讓撰寫正確軟體的工作更容易。他很高興他的觀念能逐漸傳佈至程式語言設計社群中。

前言

撰寫軟體程式是件難事—至少，撰寫經得起時間和環境考驗的軟體程式是困難的。過去百年間，不僅軟體工程領域對於如何讓撰寫軟體程式更容易絞盡腦汁，程式語言該如何設計的更簡單也是個挑戰。但究竟是什麼讓撰寫程式變得是件難事呢？

多數的書籍或學術報告聲稱強調解決此問題者，多著眼於架構、需求，和類似與**軟體**相關的主題。若是問題難在**撰寫**程式本身呢？換言之，如果，我們視程式設計師的工作為溝通軟體 — **語言** — 而非從工程的角度呢？

孩童在其出生初期幾年，學習如何說話，到五或六歲時，我們開始教導他們如何閱讀與寫作。我從沒認識任何偉大的作者是成年後才學習閱讀與寫作的。你曾認識任何偉大的程式設計師是在其生涯晚期學習寫程式嗎？

若孩童學習外語比成年人容易，以此類推，學習撰寫程式語言，不就像是學習一種新的外語嗎？

想像你正在學習某種外語，你不知道如何以外語表達某物，你只能以你所知道的字詞去描述它，希望他人能了解你所想表達的。這不就是我們每日開發軟體時所做的事嗎？我們試著以程式語言描述心中所構思的物件，希望程式的描述對編譯或直譯而言夠清楚。若轉譯的過程不順利，我們通常在心中重新構思，並試著了解是否遺漏什麼或是描述錯誤。

基於這些問題，我選擇開始一系列的調查，以了解為什麼某種程式語言會被創造出來，它如何在技術上發展成熟，如何被教導並學習，並如何隨著時間演化。Shane 和我有此榮幸讓 27 位程式設計師引領我們重溫他們的開發心路歷程，因此我們能為你們收集他們的智慧結晶與經驗。

在這本書中，你將發現開發一種成功的程式語言，某些思維與步驟是必要的，也會了解是什麼因素讓那些語言廣為接受，如何解決那些程式設計師目前面對的問題。因此，若你想學習更多關於成功的程式語言設計的秘訣，這本書絕對能幫助你。

若你正在尋覓關於軟體或程式語言領域具有啟發性的思維，你將會需要一支或兩支螢光筆，因為我相信你將會在本書中找到許多值得反覆思考的智慧結晶。

— *Federico Biancuzzi*

本書架構

本書的章節排列呈現多樣與前瞻性的觀點，細數並再三回味這些訪談內容，將讓你獲益良多。

第一章，*C++* 與 Bjarne Stroustrup 對話

第二章，*Python* 與 Guido van Rossum 對話

第三章，*APL* 與 Adin D. Falkoff 對話

第四章，*Forth* 與 Charles H. Moore 對話

第五章，*BASIC* 與 Thomas E. Kurtz 對話

第六章，*AWK* 與 Alfred Aho、Peter Weinberger 和 Brian Kernighan 對話

第七章，*Lua* 與 Luiz Henrique de Figueiredo 和 Roberto Ierusalimschy 對話

第八章，*Haskell* 與 Simon Peyton Jones、Paul Hudak、Philip Wadler 和 John Hughes 對話

第九章，*ML* 與 Robin Milner 對話

第十章，*SQL* 與 Don Chamberlin 對話

第十一章，*Objective-C* 與 Tom Love 和 Brad Cox 對話

第十二章，*Java* 與 James Gosling 對話

第十三章，*C#* 與 Anders Hejlsberg 對話

第十四章，*UML* 與 Ivar Jacobson、James Rumbaugh 和 Grady Booch 對話

第十五章，*Perl* 與 Larry Wall 對話

第十六章，*PostScript* 與 Charles Geschke 和 John Warnock 對話

第十七章，*Eiffel* 與 Bertrand Meyer 對話

本書於**貢獻者**（*Contributors*）一章列出所有貢獻者的自傳。

本書編排慣例

本書使用下列之排版標記

楷體字（*Italic*）

用來呈現新詞、網址、檔案名稱及實用功能。

定寬字（`Constant width`）

用來表示程式碼內容。

第一章

C++

C++ 在程式語言中佔有一席之地：它建立在 C 語言的基礎上，整合 Simula 提出的物件導向觀念；被世界標準組織標準化；且設計口號為：「你不須要對你不使用的東西付出代價」及「對使用者定義或內建類型的支援俱佳」。雖然 C++ 在 80 年代及 90 年代隨著物件導向及視覺化使用介面程式開發而盛行，其中一項主要貢獻是普及泛型的程式開發技術（pervasive generic programming techniques），其標準樣版程式庫（Standard Template Library）是最好的體現。較新的程式語言像是 JAVA 與 C#，曾試圖取代 C++，但即將來臨的 C++ 標準改版增加了令人期待已久的新功能。Bjarne Stroustrup 是 C++ 語言的創始者，也仍然是其最強而有力的倡導者之一。

設計的抉擇

為何您選擇擴展一種既有的程式語言，而非創造新的呢？

Bjarne Stroustrup：剛開始，早在西元 1979 年，我的目的是要幫助程式設計師開發系統。目標迄今未變。提供程式設計師解決問題的實質幫助，而非學術的練習，一種程式語言必須在應用領域有其完整的解決方案，即一種非研究性的語言，存在的目的在於解決真實世界的問題。我所強調的問題與作業系統設計、網絡，和模擬有關。我和我的同事們需要一種能表示軟體組成的語言，如同 Simula 所能達到的（或稱為物件導向程式設計），但同時亦能像 C 語言一樣，撰寫有效率的低階程式，在西元 1979 年，沒有任何一種程式語言能同時滿足這兩種需求。當時，我並不是特別想設計一種新的程式語言，而只是想要解決一些問題。

鑒於此，建立在既有的程式語言基礎上是個合理的選擇。從基礎的語言，你可獲得一種基本的語法或語義的結構，獲取有用的程式庫，並融入這個程式語言的部分文化。若不是從 C 開始，我也會讓 C++ 建立在其他程式語言的基礎上。但為何選擇 C 語言呢？我有不少在貝爾實驗室的電腦科學研究夥伴們，如：Dennis Ritchie、Brian Kernighan，都是 Unix 專家，這個問題似乎顯得多餘，但卻是我嚴肅看待的問題。

值得一提的是，C 語言的類型系統是非正式且非強制性的（如丹尼斯 • 瑞奇所言：「C 是一種強型態、弱檢查的程式語言」）。弱檢查機制（weakly checked）的部份令我擔心，且造成許多目前 C++ 程式設計師開發上的困擾。而且，C 已不再是現今廣泛使用的語言，將 C++ 建立在 C 的基礎上，表示對 C 語言基礎的信任與對 C 語言計算模式特色強型態（strongly typed）的信仰，選擇 C 是基於當時用於系統程式設計的多數高階程式語言知識。值得一提的是，這是個多數工作離硬體很近（close to the hardware）的時代，且需要達到如組合程式語言（assembler）所要求嚴謹的效能。從許多角度來看，Unix 是一種重要突破，包括使用 C 語言來進行要求高的系統程式設計工作。

所以，我選擇 C 語言的基本模式，而非那些擁有更佳錯誤檢查類型的系統。我真正想要的系統架構是如同 Simula 的類別（Classes），因此我將類別對應到 C 的記憶和運算模式中。結果極具表達性（expressive）及彈性，且在沒有快速運算支援系統的情況下，其運行速度足以能與組合語言匹敵。

為何您選擇支持多元程式設計典範？

Bjarne：因為組合各種程式設計型態能產生最好的程式碼，我所謂「最好」是指程式能最直接表達設計理念、更快速執行、最容易維護…等。當人們挑戰你的想法時，他們常藉由界定他們偏好的程式設計型態，來包含每個有用的構念（例如：泛型程式設計僅是一種物件導向開發形式），或藉由排除一些應用領域（例如，每個人都有 1GHz，1GB 規格的機器）來達到「最好」。

Java 僅著重於物件導向程式設計。您覺得這是否讓 Java 程式碼在某些情況下更複雜，而 C++ 可取而代之，運用泛型程式設計的優勢？

Bjarne：Java 的程式設計師，或者說 Java 的推廣者更貼切，過度強調物件導向至某種不合理的程度。當 Java 剛問世時，主張它的單純與簡潔。我個人預測，若成功的話，Java 會在市場規模及複雜度上大幅成長。而 Java 也確實成功了。

例如，當從容器（container）中取出一個值，使用類型轉型（casts）進行物件類型轉換（例如：`(Apple)c.get(i)`），是個無法陳述何種物件類型應在容器中的荒謬結果，這種程式語言描述方式不但冗贅且無效率。現在 Java 有了泛型（generics）的概念，因此顯得有點慢。其他 Java 漸增程式語言複雜度的例子還包括：列舉（enumerations）、反射機制（reflection）、內部類別（inner classes）。

簡而言之，複雜性將會在某些地方衍生，若不是在程式語言的定義，就是在數以千計的軟體應用與程式庫中。同樣的，Java 獨鍾將每個演算法放入一個類別的作法也會導致荒謬的情況，比如：類別中沒有資料只包含靜態函數。這也是為什麼數學使用 `f(x)` 與 `f(x,y)` 表達函數，而非 `x.f( ), x.f(y),` 和 `(x,y).f( )` 的原因：因為後者的表達方式是為了嘗試表達所謂「真正物件導向的方法」（truly object-oriented method），並避免 `x.f(y)` 內在的不對稱性。

C++ 藉由結合資料抽象性（data abstraction）與泛型程式設計（generic programming）的技術，強調許多物件導向觀念中邏輯性與程式符號標記（notational）的問題。一個經典的例子是向量 `vector<T>` 可以是任何類型，且可被複製，包括內建的、指標（pointers）到物件導向的階層，或是使用者定義的類型，例如：字串和複雜的數字。這些都可以被實現，而不需增加運行時間，不需對資料陳列增加額外的限制，或不需對標準的程式組件庫制定特別的規則。另一個不契合經典物件導向的單分派階層模式（single-

dispatch hierarchy model）例子是，一個運算（operation）需要使用兩個類別，例如：`operator*(Matrix,Vector)`，此運算並不屬於其中任何一種類別。

C++ 和 Java 本質上的差異之一在於指標（pointers）被執行的方式。某些方面來看，您可以說 Java 並沒有真正的指標，但到底這兩種途徑有什麼不同呢？

Bjarne: Java 當然有指標。實際上，Java 程式語言是種隱性的指標，只是它有不同的名稱，叫做*參考*（*references*）。隱性指標有優點也有缺點。例如：優點是它有 C++ 擁有的區域物件（local objects），但這也有缺點。

C++ 選擇支持各種堆疊配置（stack allocation）區域變數與真正成員變數，提供統一的語意（uniform semantics）、支援數值語義（value semantic）的概念、給予壓縮的資料儲存格式，並將存取成本最小化，且為 C++ 對一般資源管理之支援建立良好基礎。這是主要的，但 Java 對指標的普及和隱性使用（像是參考）將阻礙上述的一切應用。

考慮權衡 C++ 語言的排列（layout）規劃：在 C++ 裡，一個 `vector<complex>(10)` 可表示為在自由存儲區（free store）一個含有十個複數（complex）的陣列（array），裡面有二十五個字（words）：向量包含三個字，加上二十個字用來涵蓋複數，再加兩個字的標頭。同樣的狀況在 Java（使用者定義的容器，包含使用者定義的物件）則會有五十六個字：一個字作為該容器的參考，加上三個字為該容器本身，加十個字作為容器內物件的參考，再加二十個字供物件本身使用，還有二十四個字則供自由儲存區用來表示十二個獨立配置物件的標頭。顯而易見的，這些數字皆是近似值，因為自由儲存區的執行在兩種程式語言中定義。然而，結論很清楚：藉由使用參考無所不在且隱性的作法，Java 或許簡化了程式設計模式及物件回收的執行，但卻大量增加了所需的記憶體儲存空間，更提高記憶體存取的成本（需要更多間接的存取管道），及等比例的配置額外開支。

Java 所缺乏的 — 或許對 Java 是件好事 — 是 C 和 C++ 透過指標算數時誤用指標的可能性。寫的好的 C++ 程式碼並不會被這個問題所困擾，他們會運用高階抽象設計，例如：輸入輸出流（iostreams）、容器（containers）、演算法（algorithms），而不會浪費時間在亂用指標上。本質上，所有的陣列和多數的指標都涉及深度執行，多數的程式設計師並不需要去了解。不幸的是，仍到處可見許多寫的很差的、非必要低階設計的 C++ 程式碼。

然而，一個重要的時機，指標的運用還是有裨益的：當我們需要直接且有效率的表達資料結構。Java 的參考（references）在這方面表現是欠缺的，例如，你不能在 Java 裡表達一

個交換（swap）運算。另一個例子是，對每個系統，僅使用指標在低階直接存取記憶體之設計，有些程式語言需要這麼做，而那個語言通常是 C++。

當然，誤用指標將導致使用指標（以及 C 陣列）的「黑暗面」：緩衝區溢位（buffer overflow）、指標指向已刪除的記憶體、未初始化的指標…等。然而，寫的好的 C++ 程式碼，這些並不會是主要的問題。當指標及陣列被運用在抽象概念的類別（例如：`vector`、`string`、`map` 等），你也不會遭遇到這類的問題。範疇基礎的資源管理（scope-based resource management）能處理大多數的需要；智慧指標（smart pointers）和特殊識別碼（specialized handles）能被用來處理其他部分的需求。擁有 C 或舊式 C++ 開發經驗的程式設計師們覺得這難以置信，但範疇基礎的資源管理確實是非常強大的工具，配合適合的使用者定義操作比起過去那些不安全的作法，能透過更少的程式碼來解決經典的問題。例如：這是一個最簡單的經典緩衝區溢位的安全性問題：

```
char  buf[MAX_BUF];
gets(buf);  //  Yuck!
```

採用標準程式庫中的字串，這個問題就消失了：

```
string  s;
cin  >>  s;        //  read  whitespace  separated  characters
```

這些雖然是瑣碎的例子，但適合的字串和容器能被精巧的改造，以達成所有的需求，標準程式庫提供了一個好的工具集讓程式設計師們可從此處開始。

您所提到的數值語義（value semantics）和全面資源管理（general resource management）的意義是什麼呢？

Bjarne：數值語義通常是用於指涉物件的類別，當你複製類別的物件屬性時，你得到兩個獨立的副本（擁有相同的值），例如：

```
X  x1  =  a;
X  x2  =  x1;      //  now  x1==x2
x1  =  b;          //  changes  x1  but  not  x2
                   //  now  x1!=x2  ( provided  X(a)!=X(b) )
```

當然，我們有常見的數值類型，例如：整數、複數，或向量這種數學抽象概念。最有用的概念在於 C++ 支援內建的類型，及任何使用者定義的類型。Java 則不同，僅支援內建的類型，如整數、字符類型，但使用者定義的類型卻不支援，也無法支援。和 Simula 相同，Java 裡所有使用者定義的類型都是參考語意（reference semantics）。在 C++ 裡，不論是需要數值語義或參考語義，程式設計師皆可支援。C# 雖不完全依循 C++ 的原則，但基本上支援使用者定義的數值語義。

全面資源管理是指管理物件擁有的資源（例如：一個檔案的識別碼或機鎖）的普遍技術。若物件是個已界定範疇的變數，該變數的生命週期限制了其資源持有的最長時間。典型地，建構式（constructor）獲取資源，解構式（destructor）釋出資源。這樣的現象稱為「資源獲取即初始化」（Resource Acquisition Is Initialization, RAII），而且，它與異常處理機制完美的整合。顯然，並非每個資源都能以此方式處理，但許多可以這麼做，如此，資源管理技術變得隱含而有效率的。

「接近硬體」（close to the hardware）似乎是設計 C++ 的指導原則。比起其他許多程式語言，我們是否可說 C++ 的設計理念較具由下而上的精神，因為它們不像那些由上而下的程式語言，嘗試提供抽象的理性構件，並強制編譯器去讓它們配合可用的運算環境？

Bjarne：我認為由上而下或由下而上來描述這些程式設計決策的特性是錯誤的方式。無論是 C++ 或是其他程式語言，「接近硬體」指的是電腦機器的運算模式 ── 物件在記憶體及操作中的排列次序 ── 而非數學的抽象概念。這個觀念在 C++ 和 Java 都適用，但不適用其他函式語言。C++ 和 Java 不同之處在於其建立在真正的電腦機器上，而非抽象的電腦機器上。

真正的問題在於，如何將人類對問題與解決之道的構想實現於電腦機器的有限世界。你可以忽略人的考量，最後仍完成電腦程式碼（或是美其名程式碼，但卻寫的很糟的 C 語言程式碼）。你也可以忽略電腦機器，最後想出可解決任何問題的美麗抽象化表達方式，但這不儘耗費成本，也缺乏智識上的嚴謹。C++ 試圖在你需要時（如：指標與陣列的執行），給予直接存取硬體之能力，但同時也能提供廣博的抽象機制，容許高階的設計想法（如：類別階層與模板）被充分表達。

然而，對於 C++ 開發過程及其程式庫，諸如運行期和使用空間效能一直存在的顧慮，的確遍及基礎程式語言及抽象工具，這些並不是所有程式語言共通的問題。

使用此語言

您如何進行程式除錯？您有任何相關的建議提供給 C++ 程式設計師嗎？

Bjarne：透過自省式的觀察。我已研究程式撰寫相當長的時間，因此有足夠的知識理解基礎，對於程式錯誤具有敏感度而能提出合理的猜測。

測試是另一種情況，使錯誤最小化的設計也是另一途徑。我非常不喜歡排除程式中的錯誤，所以會盡可能避免。若我是某個軟體的設計師，會透過介面和系統佈局來設計，如此，不容易在編譯時出現嚴重失誤的程式或是運行錯誤。此外，我也盡力讓軟體可測試。測試是種系統化偵測錯誤之方法。但系統化測試結構不佳的系統是很難的，因此我再次推薦程式架構簡潔的重要性。測試是可自動化且是可重複的，除錯則不然。光是讓一群鴿子隨機地啄食螢幕，並觀察牠們是否能破壞圖形使用者介面設計的軟體，並不可能確保系統品質。

有何建議？我覺得很難給予一般性的建議，因為最好的技巧通常取決於某個特定系統，在特定開發環境下，哪些是可行的部分。然而，可以做的是：界定主要的、能被系統性測試的使用者介面，並寫出符合上述情境測試用的腳本。若可以的話，盡量自動化並經常執行自動化的測試。且盡量經常保持迴歸測試。確認每個系統的進入點及輸出點都能執行系統化的測試。善用具有品質的組件（components）來撰寫系統：龐大的程式對於理解和測試皆會造成非必要的困難。

在何種程度上，我們有必要改善軟體安全？

Bjarne：首先，安全性是系統問題。沒有局部或部分的解決之道可以獨立成功。需記得的是，即使你的程式碼非常完美，若我能竊取你的電腦或儲存備份的硬體設備，我仍可獲取你儲存在系統中的機密。其次，安全性是個成本效益比的遊戲：完美的安全性或許遙不可及，但能充分保護我的系統，讓壞人考慮將他們的時間花在其他較易入侵的系統上。實際上，我傾向不將重要的機密放在網路上，並把攸關重大的安全性問題留給專家。

但是關於程式語言與開發程式技巧呢？假設每行程式碼必須是安全的（無論安全的定義為何）是很危險的想法，即使假設不懷好意的某人蓄意破壞系統的某個部分也是如此。最危

險的情況其實是任由程式碼雜亂繁衍，也未經系統化的測試程式碼是否有能力防禦，讓系統不受安全威脅。同時這也讓程式碼不美觀、龐大且緩慢。不美觀的程式碼讓瑕疵有潛藏的空間，龐大的程式碼導致不完整的測試，緩慢的程式碼鼓勵捷徑或秘密伎倆的使用，招致更多安全的漏洞。

我認為安全性問題唯一恆久的解決之道，在於創造一個簡單的安全模式，有系統地應用到硬體和 / 或軟體的選取介面。在安全屏障後方應該要提供空間讓程式碼簡單、優雅、有效率的撰寫，無須顧慮某部分的程式碼濫用了其他部分的程式碼。只有那時我們才能專注於程式的正確性、品質，及關鍵的效能。任何人皆可提供不被信任的回呼（callback）、外掛（plug-in）、越權操控（overrider）等等是很愚蠢的想法，我們必須區別能捍衛詐騙行為的程式碼與僅能防止意外發生的程式碼。

我不認為人們可以設計出一種完全安全的程式語言，同時在系統開發上也是實用的。顯然，這取決於「安全」和「系統」的定義。可能可以透過領域專用的程式語言實現安全的目標，但我主要有興趣的領域是系統程式設計（廣義而言），包括嵌入式系統程式設計。我認為透過目前 C++ 所能做到的類型安全（type safety）可以也將會被改善，但這只是問題的一環：類型安全並不等同安全。撰寫 C++ 的人們使用很多未封裝陣列（unencapsulated arrays），型態轉換（casts），但缺乏結構的新增與刪除運算（new and delete operations），也徒增困擾。這些人們仍固守 80 年代的程式撰寫風格，要能善用 C++，你必須採用最小化違反類型安全原則的風格，並且以一種簡單而有系統的方式管理可用資源（包括記憶體）。

您會推薦實務開發者採用 C++ 於某些目前他們不願去應用的系統嗎？例如：系統軟體及嵌入式應用。

Bjarne：當然會，我確實會推薦，不是每個人都會不願意用。除了那些行之有年的組織，在嘗試新技術時內部產生的自然抗拒，我並沒有看到太多不願意嘗試的例子。相反的，我看到 C++ 的使用呈穩定及顯著的成長。例如：我協助 Lockheed Martin 撰寫 聯合攻擊戰鬥機（JSF）關鍵任務所需軟體的程式指導守則。那是一架完全用 C++ 寫成軟體的飛機。你可能並不熱衷於軍用飛機，但根據我的網站流量統計，JSF++ 程式撰寫規則，一年內被下載超過十萬次，且多數是非軍方嵌入式系統的開發者，如此使用 C++ 就和軍事的特殊性無關了。

自 1984 年以來，C++ 被應用於開發嵌入式系統，許多有用的小程式就是以 C++ 開發出來的，其使用也呈現快速增長的趨勢，包括 Symbian 或 Motorola 的行動電話、iPods、以及地理定位系統。我特別喜歡 C++ 在火星漫遊者上的使用：景象分析和自動駕駛子系統，多數的地表通訊系統，與影像處理。

那些深信 C 語言比 C++ 更有效率的人們或許會想看看我的研究報告“Learning Standard C++ as a New Language”[C/C++ *Users Journal*, May 1999]，描述一些程式設計哲學，並發表一些簡單實驗的結果。此外，世界標準組織 C++ 標準委員會提出一份關於 C++ 效能的技術報告，指出許多關於需要實現效能時使用 C++ 的問題和迷思（你可以在網路上輸入關鍵字“Technical Report on C++ Performance”搜尋此篇報告）[註1]，尤其，該報告特別強調嵌入式系統的議題。

Linux 和 BSD 的核心仍是用 C 語言撰寫。為何他們不改用 C++ 呢？因為 C++ 是物件導向典範的緣故嗎？

Bjarne：主要是保守主義和慣性的緣故。另外，GCC 成熟的速度很慢，在 C 語言社群的某些人，仰仗他們老成的經驗，似乎維持著故意的忽略。其他作業系統、許多系統程式設計，甚至難度高的即時、攸關安全的程式碼，幾十年來都是由 C++ 撰寫而成。考慮某些系統的例子：如 Symbian, IBM's OS/400 and K42, BeOS，和部分的 Windows。一般而言，還有許多 C++ 開放軟體程式碼（如：KDE）。

你似乎將 C++ 和物件導向（OO）的觀念等同看待。C++ 並不是也未曾企圖成為物件導向程式語言。我於 1995 年撰寫過一篇名為「為何 C++ 不只是一個物件導向程式設計語言」的報告，你可從站上下載[註2]。這個想法是為了支持多元程式設計風格（或可說是「典範」）及其不同的組合。與高效能（high-performance）和接近硬體的使用（close-to-the-hardware use）脈絡相關的其他典範則是泛型程式設計（縮寫為 GP）。國際標準組織 C++ 標準資料庫在設計其演算法架構和容器時（所謂 STL），較傾向泛型程式設計觀點，而非物件導向觀點。泛型程式設計是典型的 C++ 風格，大量仰賴模版（template），已被那些需要同時兼顧抽象表現和效能者廣泛採用。

註1 *http://www.open-std.org/JTC1/sc22/wg21/docs/TR18015.pdf*

註2 *http://www.research.att.com/~bs/oopsla.pdf*

我從未見過任何程式用 C 寫會比用 C++ 寫更好。我不認為這種程式存在。因此，你可以運用接近 C 風格的方式撰寫 C++。若非必要，你不需要狂熱地追逐使用異常（exceptions）、類別階層（class hierarchies），或模版（templates）。一個好的程式設計師使用更進階的功能來幫助他們更直接的表現想法，並減少可避免的管理費。

為何程式設計師將他的程式碼由 C 轉成 C++？使用 C++ 作為一種泛型程式語言有何優勢呢？

Bjarne：你的問題似乎假設程式碼當初是以 C 語言寫成的，而且程式設計師是從 C 語言開始他的程式設計生涯。對多數 C++ 程式和 C++ 程式設計師而言，這不是實際的狀況。不幸的是，「C 語言優先」（C first）途徑，仍停留在許多課程設計中，但現在已經不再是我們視為理所當然的事了。

某人可能會從 C 轉移至 C++，因為他們發現 C++ 對於程式設計風格的支持優於 C 語言。C++ 語言的型別檢查（type checking）更嚴格（你不能忘記宣告函式或其引述類型），而且對於常見的操作有型別安全的通知支持，例如：物件創造（包括初始化）和常數。我曾看過有些人那麼做，而且很開心能解決相關問題，這些能透過結合採用某些 C++ 函式庫而達成，例如：標準向量，圖形使用者介面庫，和某些特定應用的程式庫。

僅僅使用一個簡單的使用者自訂型別（user-defined type），例如向量、字串，或複數，並不需要典範移轉。人們可以選擇使用這些型別就如同它們是內建的型別。使用 `std::vector` 就等同於使用物件導向型別嗎？我並不認為。但若某人實際未加入新功能，而使用 C++ 開發圖形使用者介面，就是使用物件導向型別嗎？我傾向回答是的，因為使用者通常需要瞭解並使用繼承（inheritance）機制。

使用 C++ 作為「一種泛型程式設計開發語言」，你能取得標準的容器和演算法（作為標準庫的一部分），主要權衡在於多種應用和源自 C 語言對於抽象性的提高。除此之外，人們可以開始從程式庫（如：Boost）獲益，並開始欣賞某些承襲泛型程式設計的函式程式設計技巧。

然而，我認為這個問題稍微有些誤導。我不希望將 C++ 表示成一種「物件導向程式語言」（an OO language）或是一種「泛型程式設計語言」（a GP language），而希望它能是一種程式語言支持：

後記

只有一個字能描述我參與這個專案的主要愉悅 ── 熱情。每個受訪者提供你可能期待的獎賞 ── 深厚的知識、過去的知識傳統，與實務的洞察力 ── 但是，那都是因為他們對於語言設計、執行，與經驗累積成長的熱情，證明是具有感染力的。

例如，Anders Hejlsberg 和 James Gosling 重燃我對 C# 和 Java 的興奮之情。Chuck Moore 和 Adin Falkoff 說服我探索 Forth 和 APL，這兩種在我出生之前發明的語言。Al Aho 藉由描述他的編譯器課程而吸引我。每個我訪談的對象給予我多元的想法，我希望我有更多的時間去探索！

我心中充滿了感激，不僅是他們對於 Federico 和我所付出的時間，更是因為他們開闢小徑，而抵達豐饒和成熟的發明領域。最好的一課我已經從這個經驗中獲得：

- 永遠不要低估設計簡單性或執行的價值。你永遠能增加複雜性。大師會移除它。
- 熱情追求你的好奇心。當一個人在對的地點、對的時間，許多最好的發明和發現因而發生。準備好追求正確的答案。
- 熟知領域，它的過去與現在。每個受訪者都曾與其他聰明、辛勤工作的人們共事。我們的領域仰賴於這種資訊的分享交流。

當今的語言可能不斷地變化，但是，每位大師所面對的問題仍舊困擾著我們 — 而且，他們的答案仍然適用。你如何維護軟體？你如何找到最好的問題解決方案？你如何使你的使用者驚喜和快樂？你如何處理無可避免的渴望改變，而不致於採取必須繼續運作的破壞性解決方案？

對於那些問題，我現在有更好的答案。我希望這本書能夠協助你，陪伴你探尋智慧的旅程。

— *Shane Warden*