前言

Is it really punk rock
Like the party line?

— Wilco, "*Too Far Apart*"

C 語言是龐克搖滾

C 語言沒有太多的關鍵字與過多的修飾，充滿著搖滾風味。C 語言能完成各種工作，就像吉他的 C、G、D 和弦，雖然是能夠很快學會的基本和弦，卻得花上一輩子練習精進；不了解的人害怕它的強大，覺得太過危險不夠安全，不論在任何標準下，在沒有任何企業與組織投入行銷費用的情況下，C 語言一直都是最受歡迎程式語言[1]。

同時，這個程式語言已經 40 歲，以人類而言已步入中年。它一開始是一小群人為了對抗管理階層而開發，完全符合龐克搖滾的精神，但那是 1970 年代的事，這個程式語言已經進入主流很長一段時間了。

當龐克搖滾成為主流時，人們有什麼反應？在 1970 年代，龐克來自非主流：The Clash、The Offspring、Green Day 以及 The Strokes 在全球賣出上百萬上的專輯（這還只是一小部分），筆者曾在自家附近的超市聽到垃圾搖滾（grunge）演奏版。垃圾搖滾是龐克搖滾的分支；Sleater-Kinney 的前主唱現在有一個經常嘲諷龐克搖滾歌手的喜劇小品[2]。對於不斷演進的反應是訂出嚴格標準，將原始作品稱為龐克搖滾，其他則是迎合

---

1 這篇前言是向廣受注目，同時有許多爭論的《*Punk Rock Language: A Polemic*》致敬，作者是 Chris Adamson，*http://pragprog.com/magazines/2011-03/punk-rock-languages*

2 像「Can't get to heaven with a three-chord song」這樣的歌詞，可能會讓 Sleater-Kinney 被歸類在後龐克時期？不幸的是，沒有 ISO 龐克標準為各種音樂類型提供精確的定義。

大眾的龐克流行樂。傳統主義者仍然播放著 70 年代的專輯，即使唱片磨損也可以下載數位版本，還可以購買雷蒙斯樂團（Ramones）的衣服給小寶寶穿。

圈外人不懂，總會有人一聽到龐克，腦袋裡就會浮現出 1970 年代的特定影像 — 當時一部分年輕人的確做了許多異於常規的事。傳統龐克愛好者仍然喜歡播放 1973 年 Iggy Pop 的黑膠唱片，這更強化了龐克已死，不再是主流的印象。

回到 C 語言的世界，其中有揮舞著 ANSI '89 大旗的傳統主義者，他們持續寫出各種程式，完全不知道所寫的程式碼根本無法在 1990 年的環境編譯或執行。圈外人無法分辨兩者的不同，仍然讀著 1980 年發行的書籍，看著 1990 年上線的線上教學，聽著死硬派傳統主義者堅持在現代社會中遵循古法，完全不知道程式語言與其他使用者一直不斷演進，真是可惜，他們錯過了許多美好的事物。

本書主題是打破傳統讓 C 語言繼續搖滾，筆者並不想將本書中的程式碼與原始 K&R 1978 年書中的 C 語言規格比較，筆者的手機已經有 512 MB 記憶體了，為什麼 C 語言課本還要花費好幾頁的篇幅說明如何減少幾 K 的執行檔大小？本書是在一部功能陽春的紅色小筆電上完成，小筆電的執行速度已經達到每秒 3,200,000,000 個指令；為什麼還要在意運算是以 8 位元還是 16 位元執行？寫程式時應該著重在能夠快速完成他人容易理解的程式碼。我們用的可是 C 語言，即使以可讀性不那麼最佳化的方式撰寫，仍然會比其他笨重程式語言寫出來的程式快上好幾個數量級。

問與答（或，本書的參數）

問：這本 C 語言的書與其他有何不同？

答：不論寫得好壞或是否易於閱讀，C 語言的教學書籍都十分類似（筆者大部分都讀過，包含《*C for Programmers with an Introduction to C11*》、《*Head First C*》、《*The C Programming Language, 1st Edition*》、《*The C Programming Language 2nd Edition*》、《*Programming in C*》、《*Practical C Programming*》、《*Absolute Beginner's Guide to C*》、《*The Waite Group's C Primer Plus*》以及《*C Programming*》），大多數都是在 C99 標準簡化了許多使用方式之前完成，可以從某些新版本只是增加一些註記說明相異之處，而不是重新思考語言使用方式看得出來。這些書都會提到能在程式中使用函式庫，但這些書籍完成時，缺少能大幅改善函式庫可靠性與可攜性的現代化安裝工具與生態系，這些書籍的內容仍然有其價值，但現代化的 C 語言程式碼與書中的範例大不相同。

本書挑選了它們缺漏的部分，重新考慮程式語言及其生態系。主題是使用提供鏈結串列（linked list）以及 XML 剖析器函式庫，而不是從頭自行撰寫，著重於撰寫高可讀性且具備對使用者友善函式介面的程式碼。

問：本書的目標讀者？是否需要是程式高手？

答：讀者必須要有使用程式語言的經驗，也許是 Java 或是 Perl 之類的命令稿式程式語言。筆者不會解釋為什麼應該用許多較小的子函式取代一個巨大的函式等概念。

本書預期讀者從實際撰寫 C 語言程式的經驗中學會一些基本概念，如果已經忘了細節或完全不懂 C 語言，附錄 A 提供的基本 C 語言介紹，是針對只有 Python 或 Ruby 等命令稿式語言基礎的讀者所撰寫。

筆者曾經寫過一本與統計以及科學計算相關的書《*Modeling with Data*》【Klemens 2008】，除了許多處理數值資料以及透過統計模型描述資料的詳細介紹之外，還包含了一個獨立章節介紹 C 語言，筆者認為這份介紹避免了許多陳舊的 C 語言教學中的不足。

問：我只是個應用程式設計師，不是核心駭客，為什麼需要使用 C 語言，而不是使用其他寫起來更快的命令稿語言，如 Python？

答：對於應用程式設計師而言，這本書更為合適，許多人認為 C 語言是個系統程式語言，但這對筆者而言不夠龐克 — 誰允許那些人限制我們哪些事情能做，哪些不能做呢？

「這個程式語言的執行速度幾乎與 C 一樣快，可是卻好寫得多」這樣的說法十分常見，都快變成陳腔濫調了。當然 C 語言一定和 C 語言一樣快，本書的目的正是要告訴讀者，C 語言寫起來比起過去幾個世代的教科書上所說的還要容易得多。比起對 1990 年代的系統程式設計師而言，只有不到一半的情況需要呼叫 `malloc` 透過管理記憶體展現男性魅力，現在有更方便處理字串的機制，甚至核心語法也演進得能提高程式碼的可讀性。

筆者開始使用 C 語言是為了加快以 R 命令稿語言撰寫的模擬程式的執行速度，如同其他的命令稿語言，R 也有 C 語言介面並鼓勵使用者在主語言太慢時使用這些介面。最後，筆者擁有太多從 R 命令稿跳轉到 C 程式碼的函式，以致於完全拋棄主語言。

問：來自命令稿語言世界的應用程式設計師會喜歡這本書是件好事，但我是個核心駭客，在五年級就自學 C 語言了，連作夢都能寫出可以正確編譯的程式碼，這本書裡有些什麼新東西？

答：C 語言在過去 20 年間有了很大的改變，這點稍後會再討論，所有編譯器共同支援的功能也隨著時間有所不同，這得感謝在久遠之前，最初的 ANSI 標準定義了 C 語言之後的兩個後續標準。也許可以先翻翻第 10 章，看看有沒有什麼沒看過的新東西，本書的一些章節，像是釐清指標常見誤解的第 6 章的內容，則是打從 1980 年之後就沒什麼變化。

此外，環境也隨之進步，讀者可能會熟悉筆者介紹的一些工具，例如 `make` 或除錯器，但筆者認為其他的一些工具就不那麼廣為人知。Autotools 完全改變了程式碼派送的方式，而 Git 更是改變了協同合作的方式。

問：我注意到本書有三分之一的內容幾乎沒有任何 C 語言的程式碼。

答：本書希望涵蓋其他 C 語言教材沒有提到的部分，其中最重要的就是工具與環境，如果沒有使用過除錯器（不論是單獨執行或在 IDE 中使用），都會讓日子更加難過。教材一般會忽略除錯器的介紹，就算有提到也只是在最後簡單帶過。與其他人分享程式碼則需要另一組工具，包含了 Autotools 與 Git，程式碼無法單獨存在，要是這本書只是另一本假裝讀者只需要知道語言的語法，就能夠有足夠的生產力，那筆者的罪過就大了。

問：有許多能夠開發 C 語言的工具，本書怎麼挑選介紹的工具？

答：C 語言社群對互通性有很高的要求，許多 C 語言的擴充來自於 GNU 環境、Windows 獨有的 IDE 以及只具備 LLVM 擴充功能的編譯器，這也許是大多數教材都略過介紹工具的原因。但是在這個時代，有一些系統能夠廣泛執行在一般認知的電腦上，其中大多數都來自於 GNU；LLVM 及其相關工具很快的佔有一席之地，但還不夠普遍，不論讀者使用什麼樣的電腦，也許是 Widnows 主機，又或是 Linux 主機，甚至只是連到雲端供應商提供的虛擬主機，都能夠簡單快速的安裝本書所介紹的工具。筆者也的確提到了一些平台特有的工具，但在介紹的時候都會明確的指出工具適用的情況。

本書並沒有介紹任何整合開發環境（integrated development environments，IDEs），這些工具大都無法在所有平台上執行（可以試試在 Amazon Elastic Compute Cloud 的主機上安裝 Eclipse 和 C 語言 plug-ins），同時 IDE 的選擇大都有很強的主觀喜好，一般而言 IDE 都會提供專案建置系統，但這些系統通常無法與其他 IDE 的專案建置系

統相容。因此，除了強制要求所有人使用相同 IDE 的情況下（例如課堂上、辦公室或特定的計算平台），IDE 的專案檔很少被用來作為專案共享的機制。

問：現在是網際網路時代，只要一、兩秒就能夠查到命令和語法的細節，為什麼還需要讀這本書？

答：的確如此，只要在 Linux 或 Mac 的終端機輸入 `man operator` 就會列出詳細的運算子說明表，為什麼還要在書裡放一個？

筆者跟讀者一樣都有使用網際網路，也花了許多時間閱讀網路上的資訊，很清楚的知道網路上資訊缺少的部分，這也是本書的主要內容。介紹 `gprof` 或 `gdb` 等新工具時，會提供足夠的資訊讓讀者有基本知識，讓讀者有能力透過搜尋引擎解決其他的相關問題，或是其他書籍中缺少的部分（這部分還不少）。

標準：眾多的選擇

除非特別說明，本書範例都符合 ISO C99 與 C11 標準，為了說明這些標準的意義，並提供讀者一些歷史背景，以下介紹主要的 C 語言標準（略過較小的改版以及修正）。

K&R（約 1978）

> Dennis Ritchie、Ken Thompson 等人在建立 Unix 作業系統的過程中完成了 C 語言，Brian Kernighan 與 Dennis Ritchie 最後在合著的書中完成了第一版的語言描述，也成為實質的標準 [Kernighan 1978]。

ANSI C89

> 貝爾實驗室將語言規範送交美國國家標準局（American National Standards Institute），在 1989 年公布了一份標準，在 K&R 的基礎上做了改善，第二版的 K&R 書中包含完整的語言規格，也代表成千上萬的程式設計師桌上都有一份 ANSI 標準 [Kernighan 1988]。這份 ANSI 標準在 1990 年被 ISO 採用為標準，但 *ANSI'89* 是比較常見的說法（同時也是很好的 T-shirt 標語）。

經過一個世代，每部 PC 上的基礎程式多少都有使用 C 語言，而且所有的網際網路伺服器都是以 C 語言開發，在這個角度上 C 語言成為了主流，近乎人類努力所能達到的極限。

這段期間，出現了 C++ 並獲得很大的注意（雖然實際上並沒有那麼顯著），C++ 是 C 語言史上發生過最好的事件，當時其他程式語言為了趕上物件導向熱潮或實現語言作

者的新想法，加入許許多多額外的語法，C 語言卻緊守標準。需要可靠度與可攜性的人們持續使用 C 語言，想要更多樣功能的人們則是把大量的經費投注在 C++，滿足了所有的人。

ISO C99

一個世代之後，C 語言標準迎接另一次主要的版本更新，這次針對數值與科學計算作了改進，標準型別中加入了複數以及一些泛用型別（type-generic）函式。納入一些來自 C++ 在方便性上的改良，如單行式註解（源自於 C 語言的前身 BCPL），以及能夠在 for 迴圈的開頭宣告變數，針對結構宣告與初始化的改進以及表示上的改良讓結構更易於使用。語言本身的現代化反應出對安全性的重視，以及體認到並非所有人都是使用英語。

體認到 C89 所帶來的影響以及世界各地都在執行 C 語言程式，很難想像 ISO 能夠產生一份沒有受到嚴重質疑的標準 — 即使是單純拒絕任何改變的漫罵。事實上，這份標準的確存在爭議。複數有兩種常見的表示法（直角座標以及極座標），ISO 標準到底採用了哪一種？為什麼需要提供動態參數巨集的機制？所有運作良好的程式碼都沒有需要這個功能？換句話說，基本教義派質疑 ISO 屈服於壓力加入了更多功能。

本書撰寫時，大多數編譯器都支援 C99，但在一些需要注意的地方上稍作增減，然而在這個廣泛的共識中有個值得注意的例外，微軟公司目前拒絕在 Visual Studio C++ 編譯器中支援 C99。在第 6 頁「在 Windows 環境下編譯 C 語言」中介紹了一些在 Windows 下編譯 C 語言程式碼的方式，不使用 Visual Studio 最多只是有些不方便，標準的主要制定人之一告訴我們不要使用 ANSI 或 ISO 標準的 C 只是讓標準更有龐克搖滾的風格罷了。

C11

受到大量指責之後，ISO 在第三版標準中作了許多重大的改變，支援泛用型別函式，許多功能也更加現代化，以進一步回應對安全性以及其他語言使用者的重視。

C11 標準是在 2011 年 12 月公布，但編譯器開發人員對標準的實作速度驚人的快，目前許多主流編譯器都已經宣稱幾乎完全符合標準。然而，標準同時定義了編譯器與標準函式庫的行為，其中函式庫的支援程度，例如多緒與基元（atomics）則會依系統有所不同，在某些系統上已經完備，而另外的系統上則仍有待努力。

POSIX 標準

C 語言本身的狀況大約就是如此，然而這個語言是與 Unix 作業系統一起演進，本書中會一再看到兩者的交互關係對日常工作的重大影響。如果某些工作在 Unix 命令列上十分容易達成，很可能是因為相同的工作也很容易用 C 語言完成；Unix 工具通常都是用 C 語言撰寫而成。

Unix

> C 與 Unix 都是在 1970 年初期由貝爾實驗室設計，在 20 世紀大多數時候，貝爾實驗室都投資在壟斷性的作法，它與美國聯邦政府的合約中同意貝爾實驗室不會將觸角延伸到軟體領域。因此，免費將 Unix 提供給研究人員研究與重建，Unix 是個商標，一開始由貝爾實驗室擁有，稍後則像球員卡般在幾家公司間交易。

隨著許多駭客研究程式碼、重新實作並用各種不同方式改善，延伸出許多不同的 Unix。只需要少許不相容就會讓應用程式或命令稿失去可攜性，因此需要儘快建立標準。

POSIX（*Portable Operating System Interface*）

> 這個標準最初是由電機電子工程師學會（Institute of Electrical and Electronics Engineers，IEEE）在 1988 年建立，提供 Unix 類作業系統共通的基礎，其中指定了 shell 的運作方式，`ls`、`grep` 等命令列工具的預期結果，以及一些應該提供的 C 語言函式庫。例如在命令列環境串接命令的管線（pipe）就有詳細的說明，這表示 C 語言的 `popen`（開啟管線，pipe open）是個 POSIX 標準，而非 ISO C 標準。POSIX 標準有過多次改版，本書撰寫時的版本是 POSIX:2008，也是本書中提到 POSIX 標準時的標準。符合 POSIX 標準的系統必須提供 C 語言編譯器，指令的名稱是 `c99`。
>
> 本書會使用 POSIX 標準，使用時會特別說明。
>
> 除了微軟作業系統家族的許多成員之外，幾乎所有市面上叫得出名號的作業系統都是建立在 POSIX 相容的基礎上：Linux、Mac OS X、iOS、webOS、Solaris、BSD，甚至 Windows 伺服器都提供 POSIX 子系統，對於堅持使用 Windows 作業系統的讀者，「在 Windows 環境下編譯 C 語言」一節中介紹了如何安裝 POSIX 子系統。

最後，有兩種最常見的 POSIX 實作值得一提，兩者都十分盛行也都有很大的影響力：

BSD

在貝爾實驗室將 Unix 提供給研究人員之後，在加州大學柏克萊分校的一群人做了許多的改良，最後重寫了整個 Unix 程式碼，建立了 Berkeley Software Distribution。如果讀者使用蘋果公司的電腦，其作業系統就是個加上引人目光的圖形前端的 BSD，BSD 在某些方面超越了 POSIX，本書會提到幾個不屬於 POSIX 標準的函數，但太過重要無法略過（最明顯的就是能節省大量時間的 `asprintf`）。

GNU

是 GNU's Not Unix 的縮寫，另一個獨立重新實作改良 Unix 的重要成功案例，絕大多數 Linux 發行版本都使用了 GNU 提供的工具，有很大的機會在讀者的 POSIX 主機上擁有 GNU Compiler Collection（`gcc`），即使是 BSD 也使用了 GNU。同時，`gcc` 在 C 語言與 POSIX 的一些改進也成為實質的標準，本書使用到這些擴充時，筆者會特別說明。

就法律上而言，BSD 授權比 GNU 授權更為寬鬆，由於某些單位對於授權在政治與商業上的影響有較多考量，大多數的工具都同時存在著 BSD 以及 GNU 兩個版本，例如，GNU Compiler Collection（`gcc`）與 BSD 的 `clang` 都是最好的 C 編譯器，兩個開發團隊的成員都緊盯著對方，學習對方的成果，可以預期兩者的差異會隨著時間愈來愈小。

法律小常識

除了少數例外，美國法律不再採用登記制的版權系統，任何人只要寫下東西，立刻就擁有版權。

當然，派送函式庫需要在硬碟間複製資料，有些常用的機制能在最小的爭議下授權複製擁有版權的產品。

- *GNU 通用公共授權（GNU Public License，GPL）*

  允許無限制複製與使用原始碼及其可執行檔，唯一的要求是：如果派送的程式或函式庫是以 GPL 程式碼為基礎，就必須同時提供程式的原始碼；注意如果程式只供內部使用，這個條件就不成立，也沒有義務提供程式碼。執行 GPL 授權的程式，例如使用 `gcc` 編譯程式，並不需要同時提供你的程式碼，因為程式的輸出（例如編譯產生的可執行檔）並不被認為是以 `gcc` 為基礎或其延伸產品。例如：GNU Scientific Library。

- *GNU* 較寬鬆公共授權（*Lesser GPL*，*LGPL*）

  LGPL 與 GPL 類似，但特別規定如果只是以共用函式庫的方式連結 LGPL 函式庫，就不將你的程式碼視為延伸作品，也沒有提供原始碼的義務。也就是說，可以派送封閉原始碼但連結到 LGPL 函式庫的產品。例如：GLib。

- *BSD* 授權

  要求使用者維持 BSD 授權原始碼原有的版權聲明以及免責聲明，但不要求同時提供你的原始碼。例如：Libxml2 授用類似 BSD 的 MIT 授權。

讀者必須特別注意以下的免責聲明：筆者並非律師，這段小常識只是幾份完整法律文件的簡單說明，讀者如果無法判斷所處情況的相關細節，請閱讀原始文件（*http://opensource.org/licenses*）或請教律師。

關於第二版

以往筆者總是自私的認為那些寫第二版的人只是為了擾亂初版書的二手市場，但這本書的第二版的確只能夠在第一版之後才問市（幸好大多數讀者都是讀數位版）。

第二版主要新增的部分是並行執行緒（concurrent thread），也就是平行（parallelization），內容著重在 OpenMP 與基元變數（atomic variable）以及結構（struct），OpenMP 並不是 C 語言標準的一部分，但的確是 C 語言生態系裡可靠的成員，很適合納入本書的內容。基元變數在 2011 年 12 月的修訂才納入 C 語言標準，也就是說，本書的第一版在修訂後不到一年就上市，市面上也幾乎沒有支援這項功能的編譯器。到了現在，已經能夠同時在理論面與實務面呈現這個功能，提供經過真實世界驗證的程式碼，詳細內容請參第 12 章。

初版得利於許多帶有書呆子氣息的讀者，他們找到可能被認為是臭蟲的所有一切，從筆者提到在命令列用破折號到一些在特殊情境可能會誤解的句子。當然，總是會有臭蟲存在，但藉助於這許許多多傑出讀者的回饋，本書更加正確，也能夠提供更多的協助。

本書增訂的部分如下：

- 附錄 A 為來自其他程式語言的讀者提供了一份簡單的 C 語言介紹，由於市面上已經有許多 C 語言的入門書籍，筆者很勉強的在初版中包含這個部分，但這部分的確讓本書更加有用。
- 基於廣大讀者的要求，對於偵錯器（debugger）的討論內容大幅增加，參看「使用除錯器」一節的介紹。
- 初版中介紹了如何撰寫能夠接受各種數量變數函式的作法，能夠在程式中合法的使用 `sum(1, 2.2)` 與 `sum(1, 2.2, 3, 8, 16)`，但要是想要傳入的是多個串列，例如 `dot((2,4), (-1,1))` 或 `dot((2, 4, 8, 16), (-1, 1, -1, 1))` 這樣能夠求取兩個任意長度向量內積的函數該怎麼做？「多個串列」一節介紹了這個部分。
- 我重寫了第 11 章關於使用新函數擴展物件。 主要添加是虛擬表的實作。
- 增加了一些前置處理器的內容，在「測試甲巨集」一節提到了測試巨集面臨的困境以及巨集的使用方式，同時也提到 `_Static_assert` 關鍵字。
- 筆者在本書繼續堅守不在書中討論正規表示式剖析方式的自我期許（因為市面上與網路上已經有太多相關的資訊），但筆者的確在「剖析正規表示式」一節中，使用 POSIX 的正規表示式剖析函式製作了一個範例，只是與其他程式語言提供的剖析器相比，這個範例十分的簡陋。
- 初版中對字串處理的討論都是圍繞著 `asprintf`，這個 `sprintf` 式的函式能夠在寫入字串前先配置字串需要的記憶體，雖然在許多環境裡都能夠找到由 GNU 所提供的版本，但許多讀者因為一些限制無法使用這個版本，因此，筆者加入了範例 9-3，示範用 C 語言提供的標準，實作出功能相同的函式。
- 第 7 章的一大主題是說明微觀管理數值型別可能產生的問題，因此，在初版中完全沒有提到 C99 中新加入的數值型別，如 `int_least32_t`、`uint_fast64_t` 等（C99 §7.18，C99 §7.20），許多讀者建議至少介紹一些較常用的型別，如 `intptr_r` 與 `intmax_t` 等，筆者也從善如流，在時機適當時提到相關的型別。

環境

在命令稿語言（scripting language）花園圍牆外的荒野，存在能夠解決 C 語言最惱人問題的豐富工具，必須自己去尋找，我的意思是「必須」：其中有許多撰寫 C 語言程式不可或缺的基本工具。少了除錯器（debugger，不論是獨立執行或內嵌在 IDE 中），必然會讓自己暴露在未知的困境當中。

還有許多等著被使用的函式庫，讓開發人員能夠集中精力處理手頭上的問題，而不是浪費時間重新實作鏈結串列、剖析器等基本工具，必須要盡量簡化編譯使用外部函式庫的程式碼。

以下是第一部分的簡介：

第 1 章介紹建立基本環境，包含設定套件管理工具以及使用套件管理工具安裝必要的工具。這是往後樂趣的基礎，能夠編譯使用外部函式庫的應用程式，這些設定與安裝方式十分標準化，只需要設定一些環境變數與程序。

第 2 章介紹除錯、文件與測試工具；畢竟需要經過除錯、建立文件並通過測試之後，才能顯現出程式碼良好的一面。

第 3 章介紹 Autotools，這是個打包程式碼以供派送的系統，採取按部就班的介紹方式，過程中需要撰寫 shell 命令稿以及 makefile。

人是最大的變數，因此第 4 章會介紹 Git，這個系統能追蹤專案團隊各成員硬碟上的微小更動，盡可能簡化合併不同版本異動的過程。

現代 C 語言環境中，其他程式語言扮演了重要的角色，許多程式語言都提供了 C 語言的介面，第 5 章會提供撰寫介面的一般性建議，並提供 Python 的延伸範例。

第一章

簡化編譯過程的設定

> Look out honey 'cause I'm using technology.
>
> — Iggy Pop, "Search and Destroy"

C 語言標準函式庫並不足以解決所有的問題。

相反的，C 語言生態系延伸到標準之外，如果想要解決比課本習題更複雜的問題，就必須知道如何方便地呼叫常用的非 ISO 標準函式。如果想要處理 XML 檔案、JPEG 影像或 TIFF 檔案，就會需要 libxml、libjpeg 或是 libtiff，這些都是能夠自由使用的函式庫，但都不屬於標準的一部分。可惜大部分教科書都略過這個部分，由讀者自行探索，造成許多貶低 C 語言的人會有些刺耳的言論，「*C 語言是個 40 多年的老舊語言，開發人員得從頭撰寫許多必要的函式*」，他們從來不知道如何連結外部函式庫。

以下是本章的主題：

設定基本工具

比起需要自行拼湊各種元件的黑暗時代，如今已經簡單得多，只需要 10 到 15 分鐘就能夠建立完整的建置系統，還可以加上許多裝飾（當然得再加上下載工具所需要的時間）。

編譯 *C* 語言

是的，讀者已經知道該怎麼做，但我們還需要能夠掛載函式庫與函式庫所在位置的設定；單單輸入 `cc myfile.c` 已經不敷使用了，*Make* 幾乎是最簡單的編譯程式工具，很適合作為討論的基礎。筆者將提供一個擁有良好成長空間，最簡化的 makefile。

設定變數與加入函式庫

所有的系統都需要設定環境變數，因此會介紹環境變數的作用與設定方式，完成這些繁複的設定之後，只需要稍稍調整原有的環境變數，就能夠輕易地加入新的函式庫。

設定編譯系統

除此之外，還能夠利用以上設定的環境，建立一個十分簡單的編譯系統，在命令列環境中剪貼程式碼。

IDE 使用者提醒：即使不使用 `make` ，本節內容仍然與各位息息相關，因為 `make` 編譯程式碼的每個步驟，在 IDE 都有對應的功能。了解 `make` 的運作方式，就更能夠調校個人使用的 IDE。

使用套件管理工具

如果從來沒有用過套件管理工具，就錯過太多東西了！

特別提出套件管理工具的理由如下：首先，部分讀者可能不曾安裝過，本書針對這些讀者提供了一個小節的內容，讀者需要儘快取得這些工具；良好的套件管理工具能讓讀者快速地建立完整 POSIX 子系統、編譯各式各樣的程式語言、提供大量的遊戲、常見的辦公室生產力軟體，以及成千上萬的 C 語言函式庫。

其次，對 C 語言的使用者而言，套件管理工具是取得函式庫協助日常工作的主要途徑。

第三點，如果想要從下載套件的使用者身分轉換為套件開發者，本書能協助讀者作好準備，展示讓套件易於安裝的方法，如此一來，當套件儲存庫（package repository）管理員決定納入讀者開發的套件時，就能夠正確的建立最終的套件。

Linux 使用者的電腦上已經擁有套件管理工具，大都知道安裝軟體的過程十分方便。針對 Windows 使用者，會詳細介紹 Cygwin（*http://cygwin.com*）；Mac 使用者有幾個選擇，例如 Fink（*http://finkproject.org*）、Homebrew（http://brew.sh）以及 Macports（*http://macports.org*）。Mac 套件管理工具都必須使用 Apple 的 Xcode 套件（依據 Mac 年份的不同），其能夠從作業系統安裝光碟、安裝程式目錄、Apple App Store 或註冊 Apple 的開發者專案取得。

需要哪些套件呢？以下很快的介紹基本 C 語言開發環境，因為各系統使用不同的組織方式，套件的組織方式會根據系統而有所不同，可能包含在預設基礎套件當中或是用奇怪的名稱。對套件內容有所懷疑的時候，先安裝再說，現在已經不再是那個安裝太

多東西會讓系統不穩定或是效能變差的時代了。然而，仍然可能因為頻寬（或是磁碟空間）無法安裝系統提供的所有套件，這時需要有所取捨，萬一遺漏套件，總是能夠透過套件管理工具安裝。必要安裝的套件如下：

- 編譯器，必然是安裝 `gcc`，可能也會有 `clang` 可供安裝
- GDB，除錯器
- Valgrind，檢查 C 程式的記憶體使用錯誤
- `gprof`，效能評測器（profiler）
- `make`，不再需要直接呼叫編譯器
- `pkg-config`，尋找函式庫使用
- Doxygen，產生文件
- 文字編輯器，有成千上百的編輯器可供選擇，以下是筆者主觀的建議：
  - Emacs 與 vim 是硬派 geek 的最愛，Emacs 納入了各式各樣的功能（*E* 代表**擴充性**，*extensible*），vim 則刻意的縮限，只提供最基本的功能，非常適合鍵盤愛好者使用。如果打算花個上百個小時盯著編輯器工作，值得花點時間從這兩個編輯器找一個來學。
  - Kate 十分友善也很吸引人，提供語法標示等許多程式設計師需要的方便功能。
  - 最後，試看看 nano，十分簡單的文字模式編輯器，即使沒有 GUI 也能夠正常使用。
- 如果喜歡使用 IDE，就挑一個或幾個，同樣有許多選擇，以下是筆者的推薦：
  - Anjuta：屬於 GNOME 家族，對 GNOME GUI builder，Glade，十分友善。
  - KDevelop：屬於 KDE 家族。
  - XCode：Apple 公司為 OS X 環境提供的 IDE。
  - Code::blocks：十分簡單，能夠在 Windows 下使用。
  - Eclipse：能夠跨平台執行（也是一台擁有許多杯架和按鈕的豪華房車）。

在後續章節中，會使用以下這些重火力工具：

- Autotools：Autoconf、Automake、libtool
- Git

- 其他的 shell，例如 Z shell

當然，還有許許多多能省下大量重複發明輪子時間的 C 函式庫（或是，更正確的比喻應該是重新發明火車頭）。讀者可能會想要更多，以下是本書內容會用到的函式庫：

- libcURL
- libGLib
- libGSL
- libSQLite3
- libXML2

函式庫套件並沒有一致的命名規則，讀者必須找出所使用的套件管理工具拆解函式庫的方式，一般會由一個供使用者使用的套件，加上一個供開發人員專案使用的函式庫套件組成，因此要特別注意除了基本套件外，還要安裝 `-dev` 或 `-devel` 套件。某些系統還會將文件分離到獨立的套件，某些則需要另外下載除錯用的符號表（symbol），GDB 應該會在第一次遇到缺少除錯符號的時候引導讀者下載需要的符號表。

如果使用的是 POSIX 系統，在安裝完所有需要的項目之後，就擁有完整的開發系統，能夠開始寫程式了。對於 Windows 使用者，接下來需要稍稍繞個路，了解設定工具與 Windows 主系統溝通的方式。

在 Windows 環境下編譯 C 語言

在大多數系統上，C 語言是主要、VIP 級的程式語言，所有的工具都以 C 語言為首要考量；但 Windows 系統很特別地忽略了 C 語言。

因此筆者需要花些時間說明如何設定 Windows 主機，建立 C 語言的開發環境。如果讀者使用的不是 Windows 環境，可以略過這部分，跳到「連結函式庫的方式？」一節。

Windows 下的 POSIX

由於 C 語言與 Unix 是一起演進，很難將兩者分開討論。從 POSIX 開始應該比較容易些，對於想在 Windows 主機上編譯其在其他平台撰寫的程式的讀者而言，這似乎是最自然的做法。

就筆者所知，擁有檔案系統的東西主要分成兩大陣營（兩者稍有重疊）：

- POSIX-相容系統
- Windows 家族作業系統

POSIX 相容並不表示系統的外觀（look and feel）像是 Unix 主機，例如，大部分 Mac 使用者完全不知道自己使用的是搭配了吸引人前端的標準 BSD 系統，但瞭解的人可以從應用程式→工具程式目錄啟動終端機（Terminal）程式，盡情執行 `ls`、`grep` 或 `make` 等各種工具。

此外，並非所有系統都 100% 符合 POSIX 標準的要求（例如提供 Fortran '77 編譯器），就本書的目的，需要能夠有類似基本 POSIX shell 的 shell、一些工具（`sed`、`grep`、`make` 等等）、C99 編譯器、以及 `fork` 與 `iconv` 等標準 C 語言函式庫之外的函式庫，可以作為主系統的擴充。套件管理工具底層的命令稿、Autotools 以及所有想要提供具可攜性程式碼的使用者都需要依賴這些工具，因此，即使不願意整天盯著命令列提示符號，仍然值得安裝這些方便的工具。

在伺服器等級的作業系統以及完整版 Windows 7 中，微軟公司提供了以往稱為 INTERIX、現在稱為 Subsystem for Unix-based Application（SUA）的子系統，這個子系統提供了常用的 POSIX 系統呼叫、Korn shell 以及 `gcc`。這個子系統預設不會安裝，是需要另行下載的元件，目前其他版本的 Windows 並不提供 SUA，Windows 8 也是如此，這也表示無法依賴微軟為自家作業系統提供 POSIX 子系統。

也就是需要 Cygwin。

如果想要從頭開始自行開發 Cygwin，可以參考以下簡單的介紹：

1. 為 Windows 撰寫 C 函式庫，提供所有的 POSIX 函式。這需要弭平 Windows/POSIX 系統間的差異，例如 Windows 系統使用 *C:* 的方式表示不同的磁碟機，POSIX 系統則使用統一的檔案系統（unified filesystem），針對這種情況，可以為 *C:* 建立 */cygdrive/c*、為 *D:* 建立 */cygdrive/d* 等別名。
2. 現在可以利用連結到前一個步驟開發的函式庫的方式，編譯 POSIX 標準程式，產生 Windows 版本的 `ls`、`bash`、`grep`、`make`、`gcc`、`X`、`rxvt`、`libglib`、`perl`、`python` 等等。
3. 建置好這許許多多的程式與函式庫之後，接著需要建立套件管理工具，讓使用者能夠選擇自己想要安裝的元件。

作為 Cygwin 的使用者，只需要從 Cygwin 網站（*http://cygwin.com*）下載套件管理工具，選擇要安裝的套件。當然包括了之前列出的清單，再加上一個上得了檯面的終端程式（可以試試 mintty 或安裝 X 子系統和使用 xterm，這兩個終端機程式都比 Windows 的 `cmd.exe` 更加友善），讀者可以發現開發系統需要的各種豪華工具都在其中。

稍後在「路徑」一節裡會介紹影響編譯的各個環境變數，包含尋找檔案的路徑，不是只有 POSIX 環境提供環境變數，Windows 平台同樣也有環境變數，如果將 Cygwin 的 `bin` 路徑（可能是 `c:\cygwin\bin`）加到 Windows 的 `PATH` 環境中，Cygwin 會更加方便。

接下就是開始編譯 C 語言程式了。

搭配 POSIX 編譯 C 語言

微軟 Visual Studio 提供的 C++ 編譯器擁有 C89 相容模式（儘管目前 ANSI 標準是 C11，C89 一般仍然被稱為 *ANSI C*），這是目前微軟公司提供唯一編譯 C 語言程式碼的方式。該公司的許多代表很明確的表示不會提供 C99 支援（更別說是 C11 了），Visual Studio 是唯一一個還停留在 C89 的主流編譯器，我們需要尋求其他的替代方案。

當然，Cygwin 提供了 `gcc`，讀者如果依照安裝步驟裝好 Cygwin，也就擁有了完整的建置環境。

預設情況下，在 Cygwin 編譯的程式碼會使用到 *cygwin1.dll* 函式庫提供的 POSIX 函式，也就是會使用到（不論程式碼中是否直接呼叫 POSIX 函式），*cygwin1.dll* 在有安裝 Cygwin 的主機上執行這些程式不會有任何問題，使用者可以點擊執行檔執行程式，系統應該能夠順利的找到需要的 Cygwin DLL。如果想在沒有安裝 Cygwin 的主機上執行以 Cygwin 編譯的程式，就必須同時提供執行檔與 *cygwin1.dll*。在筆者的主機上，檔案所在的位置是：*/bin/cygin1.dll*，*cygwin1.dll* 使用的是類 GPL 授權（參看「前言」中的「法律小常識」介紹），也就是說，要是你將這個 DLL 抽離 Cygwin 單獨散布，就必須要發佈你的應用程式的原始程式碼[3]。

如果這是問題，就需要透過其他方式重新編譯，讓程式不會相依於 *cygwin1.dll*，也就是表示程式碼中不能使用 POSIX 專屬的函式（如 `fork` 與 `popen` 等），必須改用 MinGW

[3] Cygwin 是紅帽公司（Red Hat, Inc.）的專案，能夠透過購買其他形式的授權，不再受限於 GPL 提供原始碼的要求。

（稍後會介紹 MinGW），可以使用 cygcheck 找出程式碼需要的 DLL 檔案，藉此檢查執行檔是否連結到 *cygwin1.dll*。

檢查程式或動態連結函式庫所使用的其他函式庫：

- Cygwin: `cygcheck libxx.dll`
- Linux: `ldd libxx.so`
- Mac：`otool -L libxx.dylib`

不搭配 POSIX 編譯 C 語言

如果程式不需要 POSIX 函式（例如 fork 或 popen），就可以使用 MinGW（Minimalist GNU for Windows），MinGW 提供標準 C 編譯器以及基本工具，MSYS 能夠提供 MinGW 環境下的其他的輔助工具，如 shell。

MSYS 提供了 POSIX shell（可以在 mintty 或 RXVT 終端下執行），也可以完全拋棄命令提示列改用 Code::blocks（*http://www.codeblocks.org/*），這是個利用 MinGW 在 Windows 上編譯的 IDE。Eclipse 是擴充性更高的 IDE，也能夠支援 MinGW，但需要更多設定。

如果讀者比較習慣 POSIX 命令列，仍然可以安裝 Cygwin；在 Cygwin 環境下搭配 MinGW 版本套件的 gcc，用 MinGW 版本的 gcc 取代 Cygwin 預設使用 POSIX 連結的 gcc。

對於沒用過 Autotools 的讀者，稍後就會介紹這個工具，使用 Autotools 建置套件的特徵是使用以下三命令安裝：`./configure && make && make install`。MSYS 提供了足夠的機制讓這些套件有很高的機會能順利在 MinGW 環境下安裝；否則就得下載套件從 Cygwin 命令列建置，但必須使用以下命令設定套件，透過 Cygwin 的 Mingw32 編譯器建立不使用 POSIX 的程式碼：

```
./configure --host=mingw32
```

接著同樣執行 `make && make install`。

在 MinGW 下編譯，不論是使用命令列編譯或是 Autotools，只要是在 MinGW 下編譯，最終結果都是原生 Widnows 執行檔。由於 MinGW 與 *cygwin1.dll* 無關，程式也不會呼叫任何 POSIX 函式，最終的執行檔是個對 Windows 友善的執行檔，沒有人能發現執行檔是在 POSIX 環境建立。

MinGW 真正的問題在於預先編譯完成的函式庫太少[4]，如果想要完全排除 *cygwin1.dll*，就不能使用 Cygwin 內附的 *libglib.dll*，需要由原始碼重新編譯 GLib 為原生 Windows DLL — 但 GLib 又透過 GNU 的 `gettext` 函式庫實作國際化（internationalization），所以需要先編譯 `gettext` 函式庫。現代程式碼都是建構在現代函式庫之上，到最後會發現在這些事上花了許多的時間，但在其他系統上卻只需要一行啟動套件管理工具的指令就夠了。稍後還有其他類似的情況，這些情況讓許多人認為在 C 語言這個有 40 年歷史的老舊語言上需要從頭寫過所有工具。

因此，已經事先警告了，微軟公司拒絕溝通，讓其他人實作後油漬搖滾（post-grunge）時代的 C 語言編譯器與環境。Cygwin 擔起了這項任務，提供完整的套件管理工具與大量函式庫，能夠完成程式開發人員部分或所有的工作，但要求使用 POSIX 型式的寫作方式並依賴 Cygwin 的 DLL，如果這會造成問題，就需要花費許多時間建立撰寫進階程式所需的環境與函式庫。

連結函式庫的方式？

現在有了編譯器、POSIX 工具集（toolchain）以及能夠輕易安裝大量函式庫的套件管理工具，可以進入下個階段：使用工具編譯程式。

先從命令列直接執行編譯器開始，但很快就會覺得十分麻煩，雖然實際上只需要三個（有時是三個半）相當簡單的步驟：

1. 設定變數表示使用的編譯器旗標。
2. 設定變數表示連結的函式庫，半個步驟是指有時只需要設定一個指定編譯期連結函式庫的環境變數，有時則需要指定兩個變數，分別表示編譯期連結以及執行期的函式庫。
3. 設定系統使用上述變數進行編譯。

要使用函式庫，就必須告知編譯器引入函式庫中的函式兩次：一次在編譯過程，另一次在連結過程。對於在標準位置的函式庫，這兩個宣告分別是透過程式碼中的 `#include` 指令以及編譯器命令列的 `-l` 旗標達成。

---

[4] 雖然 MinGW 也有套件管理工具，能夠安裝基本系統，提供一些函式庫（大多數是 MinGW 本身需要的函式庫），這些預先編譯函式庫的數量與一般套件管理工具所提供的函式庫數量完全不能相比。實際上，在筆者的 Linux 主機上的套件管理工具提供了比 MinGW 套件管理工具更多以 MinGW 編譯的函式庫。這還只是在本書完稿時的數據，等到各位讀者拿到本書，已經有其他使用者在 MinGW 儲存庫貢獻了更多的套件。

範例 1-1 是個簡單的程式，執行了一些有趣的數學運算（至少筆者認為有趣，如果讀者覺得統計術語像是外星文字，不用太在意），C99 標準的 **誤差函式**（*error function*），`erf(x)`，與 0 到 *x* 值間，平均值為零，標準差為 √2 的常態分佈函數積分值有密切的關係。程式中使用 `erf` 驗證某個受到統計學家喜愛的區域（標準大 *n* 假設檢定的 95% 信賴區間），就將檔案命名為 *erf.c* 吧。

範例 1-1　使用標準函式庫的單行程式（erf.c）

```
#include <math.h>   //erf, sqrt
#include <stdio.h>  //printf

int main() {
    printf("The integral of a Normal(0, 1) distribution "
           "between -1.96 and 1.96 is: %g\n", erf(1.96*sqrt(1/2.)));
}
```

讀者應該都很熟悉 `#include` ，編譯器會將 *math.h* 與 *stdio.h* 的內容複製到原始檔的相對位置，也就是會複製 `printf`、`erf` 與 `sqrt` 的宣告，*math.h* 檔案中的宣告並沒有說明 `erf` 的行為，只表示這個函式需要一個 `double` 參數並傳回一個 `double` 值。這就足以讓編譯器檢查函數使用方式的正確性，並產生帶有註記的目的檔（object file），告知電腦：「要是遇到這個註記，就要去找 `erf` 函式，用 `erf` 函式的傳回值取代註記。」

連結器的工作就是從磁碟中的函式庫裡找出真正的 `erf`，取代目的檔裡的註記。

math.h 檔案中的數學函式被分離放在個別的函式庫，必須利用 `-lm` 旗標告知連結器，其中 `-l` 是指定連結函式庫的旗標，範例中的函式庫名稱只有一個字元 `m`；因為連結器命令的最後預設包含了一個 `-lc` 旗標，連結到標準 `libc` 函式庫，所以使用 `printf` 函式時不需要額外指定連結函式庫。稍後會透過 `-lglib-2.0` 連結 GLib 2.0 以及 `-lgsl` 連結 GNU Scientific Library 等函式庫。

如果將檔案儲存為 *erf.c*，那 `gcc` 編譯器的完整編譯命令如下（包含幾個稍後會介紹的旗標）：

```
gcc erf.c -o erf -lm -g -Wall -O3 -std=gnu11
```

此就能夠透過程式碼的 `#include` 讓編譯器引入數學函式，並透過命令列的 `-lm` 告訴連結器連結到數學函式庫。

`-o` 旗標則是指定輸出檔的名稱；如果沒有特別指定，預設的執行檔名稱是 *a.out*。