前 言

這不是一本討論物件導向程式設計（object-oriented programming）的書。

你或許會覺得奇怪。畢竟，你可能在書店的 C++ 分類區看到這本書，也可能聽過別人把物件導向與 C++ 當同一件事來講，但是這並非 C++ 語言的唯一用途。基本上 C++ 支援多種不同的設計思維模型（paradigms），其中最新也最鮮爲人知的一項就是泛型程式設計（generic programming）。

正如許多新的觀念一樣，泛型程式設計實際上已有很長一段歷史了。早期的泛型程式設計研究論文大約在 25 年前就已經出現；第一個實驗性質的泛型程式庫並非以 C++ 寫成，而是以 Ada [MS89a, MS89b] 和 Scheme [KMS88] 完成。由於泛型演算法太新，所以還沒有相關教科書。

第一個走出研究圈的範例顯得格外重要，它就是 STL：C++ Standard Template Library。STL 是由 Alexander Stepanov（他後來成爲 Hewlett-Packard 實驗室的一員）與 Meng Lee 共同發展，於 1994 年被納入 C++ 標準程式庫的一部份。可免費取得之「HP STL 實作品」[SL95] 也於這一年發行，它是展示 STL 強大威力的範本。

當 Standard Template Library 開始成爲 C++ 標準規格的一部份，C++ 族群立刻意識到這是一個高品質而且高效率的容器類別程式庫（container classes library）。要在一堆物品中找出熟悉的東西是最容易的了，而每一位 C++ 程式員都熟悉容器類別。每個具相當水準的程式也都需要某種管理物件的方法，甚至每位 C++ 程式員都曾實作過 `strings`、`vectors` 或 `lists`。

C++ 早期就已經有容器類別程式庫的存在。這個語言加入 template classes（亦即可參數化的型別）之後，第一個用途 — 事實上也是引入 template 的主要原因 — 就是將容器類別參數化。很多廠商，包括 Borland、Microsoft、Rogue Wave 與 IBM，都有自己的程式庫，其中包括 `Array<T>` 或其對等物。

容器類別的觀念是如此根深蒂固，以致於 STL 給人的初步印象似乎不比其它容器類別程式庫多了些什麼。這種印象使大家疏忽了 STL 的獨特性。

STL 是一種高效、泛型、可交互操作的軟體組件；巨大，而且可以擴充。它包含計算機科學中的許多基本演算法和資料結構，而且它把演算法和資料結構完全分離開來，互不耦合。STL 不只是一個容器類別程式庫，更精確地說它是一個泛型演算法（generic algorithms）函式庫；容器的存在使這些演算法有東西可以操作。

你可以在程式中使用現有的 STL 演算法，正如你使用現有的容器一樣。舉例來說，當你想要使用 C 標準函式庫中的 `qsort` 函式，你也可以使用泛型的 STL `sort`（而且 `sort` 更簡單、更彈性、更安全也更有效率）。很多書籍，包括 David Musser 與 Atul Saini 合著的 *STL Tutorial and Reference Guide* [MS96]，以及 Mark Nelson 的 *C++ Programmer's Guide to the Standard Template Library* [Nel95]，都有說明如何以這種方式使用 STL。

這實際上是非常有用的。重複運用程式碼總比重新撰寫程式碼來得好，現在你可以在你的程式中重複運用既有的 STL 演算法。然而這還只是以某個角度來使用 STL 而已。STL 的設計具有擴充性；也就是說你可以自行撰寫一些組件，與 STL 組件交互作用，就像不同的 STL 組件之間可以交互作用一樣。有效地運用 STL，意味著對它進行擴充。

泛型程式設計（Generic Programming）

STL 並非只是一些有用組件的集合。它還有鮮爲人知而未被瞭解的一面：它是描述軟體組件抽象需求條件的一個正規而有條理的階層架構（formal hierarchy）。由於所有 STL 組件都是精確符合某些特定條件而寫成，所以 STL 組件可以相互作用，可以擴充，而且你可以在增加新演算法和新容器的同時，對於新舊程式碼之間的協同作用深具信心。

計算機科學的重要進步，許多是由於發掘了新的抽象性質而促成。一個被當代所有電腦語言支援的決定性抽象性質就是副程式 subroutine（又名程序 procedure 或函式 function，不同的語言使用不同的詞彙）。C++ 支援另一種抽象性質：抽象資料型別（abstract data typing，ADT）。是的，在 C++ 中，我們可以定義新的資料型別，以及該型別的基本操作行爲。

程式碼與資料的結合形成了抽象資料型別，它必須透過一個具有明確定義的介面來操作。副程式是一種重要的抽象性質，因爲一旦使用副程式，你就不需要仰賴（甚至不必知道）其實際作法；同樣道理，你可以使用抽象資料型別 — 可以操作甚至產生新值 —而不必在意資料的實際表現方式。唯一重要的是其介面。

C++ 也支援物件導向程式設計（OOP）[Boo94, Mey97]，此技術涉及與繼承相關、由多型（polymorphic）資料型別所構成的階層體系。物件導向程式設計（OOP）擁有比抽象資料型別（ADT）更多的間接性，因而達到更進一步的抽象性。某些情況下你可以取用（參考）某值並操作之，卻不必指明其精確型別。你可以撰寫單一函式，用以操作繼承階層體系中的不同型別。

泛型程式設計（Generic Programming）意味一種新的抽象性質。其中心抽象性比早期如副程式（subroutine）或類別（class）或模組（module）的抽象性更難捉摸。它是資料型別的一組需求條件。這很難讓人領悟，因爲它並非與 C++ 的某個性質繫結在一起。C++（甚或任何當代電腦語言）並沒有任何關鍵字可用來宣告一組抽象需求條件。

對於這種「一開始令人洩氣的含糊情況」，泛型程式設計的回報是前所未有的彈性，以及不會損及效率的抽象性。泛型程式設計和物件導向程式設計不同，它並不要求你透過額外的間接層來呼叫函式；它讓你撰寫完全一般化並可重複運用的演算法，其效率與「針對某特定資料型別而設計」的演算法旗鼓相當。

泛型演算法抽離（抽象化）於特定型別和特定資料結構之外，俾得以接受儘可能一般化的引數型別。這意味一個泛型演算法實際上具有兩部分：(1) 用來描述演算法步驟的實際指令，(2) 正確指定「其引數型別必須滿足之性質」的一組需求條件。

STL 的創新在於認知這些型別條件可以被具體指明並加以系統化。也就是說，我們可以定義一組抽象概念（所謂 ***concept***s），只要某個型別滿足某組條件，我們就說此型別符合某個 concept。這些 concepts 非常重要，因爲演算法對於其引數型別的大部份假設，都可以藉由「符合某些 concepts」以及「不同 concepts 之間的關係」來陳述。此外，這些 concepts 形成一個明確定義的階層架構，這讓人聯想到傳統物件導向程式設計中的繼承（inheritance），只不過它是純然的抽象。

這個 concepts 階層架構是 STL 的一個概念性結構，也是 STL 最重要的部分。由於它，使得重複運用和交互操作變得可能。此一概念性結構作爲軟體組件的正規分類法也十分重要 — 即使沒有具體程式碼。STL 確實包含有具象的資料結構如 pair 和 list，但要有效率地運用這些資料結構，你必須瞭解其所依據的概念結構。

定義抽象的 concepts，並根據抽象的 concepts 來撰寫演算法與資料結構，是泛型程式設計的本質。

如何閱讀本書

本書把 Standard Template Library 當作抽象概念庫（library of abstract concepts）來描述。本書定義出 STL 基本的各個 concepts 與抽象性質，並指出所謂「某個型別模塑出某個 concept」是什麼意思，「以某個 concept 的介面寫成一個演算法」又是什麼意思。本書討論 STL 涵蓋的各個類別和演算法，並闡明如何撰寫屬於你自己並相容於 STL 的類別和演算法。此外本書還包含一份所有 STL concepts、類別、演算法的完整參考手冊。

每個人都應該閱讀第一篇，這一部分介紹 STL 與泛型程式設計的主要概念。說明如何使用以及撰寫一個泛型演算法，並解釋「演算法之所以爲泛型」的意義。所謂泛型（Genericity），具有「在多種資料型別上皆可操作」的含意。

探究泛型演算法，自然而然便導出了 ***concept***s、***modeling*** 與 ***refinement*** 的中心觀念，這些觀念之於泛型程式設計，就像多型與繼承之於物件導向程式設計，同樣地基礎，同樣地重要。泛型演算法作用於一維範圍上，導出 STL 的數個基本概念：***iterator***s、***container***s、***function object***s。

第一篇同時也介紹了本書通用的符號及排版習慣：術語 ***modeling*** 和 ***refinement***、***range***s 的非對稱標示法，以及 concept 名稱的特殊字體。

STL 定義了很多 concepts。某些 concepts 只因技術上的細節而互不相同。第一篇是個概論，概略討論 STL concepts 的全貌。第二篇是詳細的參考手冊，包含每個 STL concept 的嚴格定義。你可能不會想要遍讀第二篇；當你需要參考某個 concept 時才到第二篇查詢，應該是更好的作法。（每當需要撰寫符合某個 STL concept 的新型別時，你都應該參考第二篇）

第三篇同樣是參考手冊，提供 STL 預先定義的演算法和類別的說明。這一部分仰賴第二篇的 concept 定義甚多。STL 所有的演算法和幾乎所有的具象型別都是 templates，每個 template 的參數都能以某種 concept 的 model 加以描述。第三篇的定義可以和第二篇對應的章節交互參考。

理想狀況下，本書到達第三篇就可以結束了。不幸的是現實需求使我必須寫出更多章節：一份有關可攜性議題的附錄。STL 問世之初並沒有可攜性問題，因爲只有一份實作品存在。這種情況已不復存在。一旦某種語言或程式庫有一個以上的實作品存在，任何關心可攜性的人都必須知道每個實作品之間的差異。

你仍然可以從 anonymous FTP 站台 `butler.hpl.hp.com` 下載早期的 HP 實作品，但此作品已無人維護。Silicon Graphics Computer Systems (SGI) 有一份較新的免費作品，可以從 `http://www.sgi.com/Technology/STL` 下載。至於 SGI STL 在不同編譯器上的移植版本，可以從 `http://www.metabyte.com/~fbp/stl` 取得。此外這個世界還存在有其他商業化 STL 實作品。

如果你正在撰寫真正的程式，光瞭解函式庫的設計理論是不夠的；你還必須知道不同的 STL 實作品以及不同的 C++ 編譯器之間的差別。這些不怎麼吸引人但卻必要的細節，構成了附錄 A 的主題。

誰該閱讀本書

雖然本書談的幾乎都是以 C++ 寫成的演算法，但這不是一本演算法導入型教科書，也不是一本 C++ 語言教本。本書確實對於兩方「不爲人熟知的觀點」有所解釋，更明確地說，由於 STL 使用 templates 的方式迥異於其他 C++ 程式，所以本書討論了某些 templates 高階技術。本書不應該是你的第一本 C++ 書籍，也不應該是你的第一本演算法分析入門。你應該知道如何撰寫基本的 C++，同時也應該知道 ***O(N)*** 表示法的意義。

演算法與資料結構方面，兩本標準的參考書是 Donand Knuth 的 *The Art of Computer Programming* [Knu97, Knu98a, Knu98b] 和 Cormen, Leiserson, and Rivest 的 *Introduction to Algorithm* [CLR90]。兩本最佳的 C++ 入門書籍則是 Bjarne Stroustrup 的 *The C++ Programming Language* [Str97] 和 Stanley Lippman, and Josee Lajoie 的 *C++ Primer* [LL98]。

本書由來

我於 1996 年加入 Silicon Graphics Computer Systems 的編譯器研發團隊。那時候 Alex Stepanov 已經離開 HP 並加入 SGI 數個月了。當時的 SGI C++ 編譯器並不包含 Standard Template Library 實作品。以 HP 的原始作品爲基礎，Alex、Hans Boehm 和我開始撰寫一個新的 STL 版本，用來和 SGI MIPSpro 編譯器 7.1（以及後續版本）搭配出貨。

SGI Standard Template Library [Aus97] 包含很多嶄新的擴充性質，例如高效且「對多緒而言安全（thread-safe）」的記憶體配置能力、雜湊表（hash tables），以及某些演算法的改良。這些增強功能如果留做私用，對 SGI 的客戶而言沒有任何價值，所以 SGI STL 把它們全部免費公開。所有原始碼及文件都放在 `http://www.sgi.com/Technology/STL`。

網路上的這份文件以網頁形式呈現，將 STL 的概念結構視爲核心，描述組成此結構之各個抽象概念（concepts），並以這些 concepts 來說明 STL 的演算法和資料結構。我們收到很多要求，希望能夠擴充此文件，本書就是對這些要求的回應。本書的參考手冊部份，也就是二、三兩篇，是 SGI STL 網頁的分枝。

整個網頁是爲 SGI 而寫，其版權屬於 SGI 擁有。我在我所屬的管理部門的寬容許可下使用這些網頁內容。

致謝

首先，也是最重要的，如果沒有 Alex Stepanov 的努力，這本書不可能誕生。Alex 參與這本書的每個階段：他把我帶進 SGI，我對泛型程式設計的每一個知識幾乎都是他教導的，他參與 SGI STL 及其網頁的開發，並鼓勵我將網頁的內容寫成一本書。我由衷感謝 Alex 的幫助與鼓勵。

我也要感謝 Bjarne Stroustrup 與 Andy Koenig 幫助我瞭解 C++，並感謝 Dave Musser 對於泛型程式設計、STL 與本書的諸多貢獻（其中有些貢獻可自參考書目中找到）。Dave 使用早期的 SGI STL 網頁內容作爲其課程教材，而透過他與他的學生的意見，這個網頁有了很大的改進。

同樣地，本書透過校閱者的意見而有了很大的改進，校閱者包括 Tom Becker、Steve Clamage、Jay Gischer、Brian Kernighan、Andy Koenig、Angelika Langer、Dave Musser、Sibylla Schupp 和 Alex Stepanov，他們閱讀過本書的每個版本。透過他們的協助，這本書更爲清楚，錯誤也減少了許多。剩

下的任何錯誤都是我的責任。

本書第一刷和第二刷中的許多錯誤已經更正，我要感謝 Sam Bradsher、Bruce Eckel、Guy Gascoigne、Ed James-Beckham、Jon Jagger、Nate Lewis、Shawn D. Pautz、John Potter、George Reilly、Manos Renieris、Peter Roth、Andreas Scherer 和 Jürgen Zeller，使我注意到這些錯誤。

我也要感謝 Addison-Wesley 的工作人員，包括 John Fuller、Mike Hendrickson、Marina Lang 與 Genevieve Rajewski，他們在我寫作的過程中指導我。感謝 Karen Tongish 細心的編輯。

最後，我要感謝我的未婚妻 Janet Lafler 給我的愛與支持，以及對我在很多夜晚和週末寫作的容忍。

我們的貓兒 Randy 與 Oliver，在我的鍵盤上走來走去試著想幫助我，不過最後我刪掉了牠們大部分的貢獻。