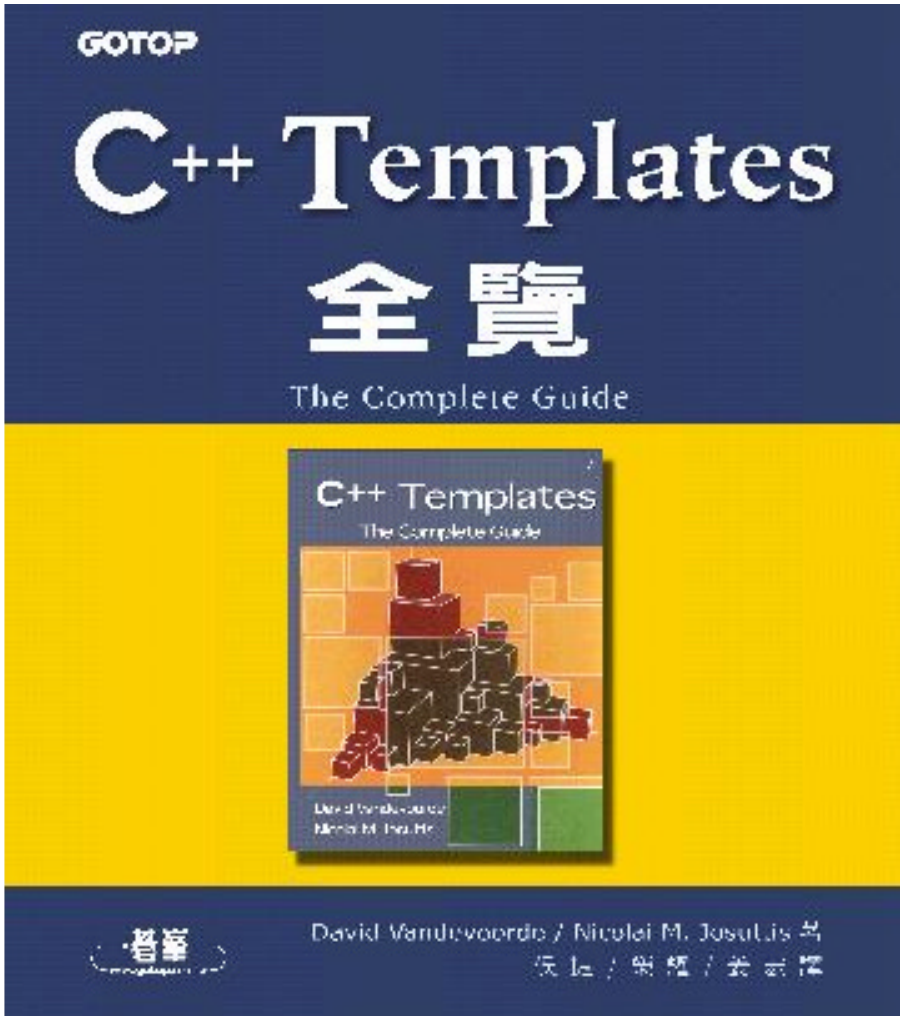
Reproduced with permission from Pearson Education



David Vandevoorde / Nicolai M. Josuttis 著
侯捷 / 榮耀 / 姜宏 譯

C++ Template 全覽

C++ Templates - The Complete Guide

David Vandevoorde
Nicolai M. Josuttis
著

侯捷 / 榮耀 / 彭宏
譯

譯序 by 侯捷

泛型編程（Generic Programming）是繼物件導向（Object Oriented）技術之後，C++ 領域中最被討論和關注的焦點[1]。這樣的關注在 C++ 社群已經持續了數年之久。

談到 C++ 泛型編程，話題離不開 templates[2]，因爲它正是實現泛型之關鍵性 C++ 構件。很多 C++ 經典語言書如《*The C++ Programming Language*》,《*C++ Primer*》和《*Thinking in C++*》都已經花費大量篇幅介紹 templates。這些書籍幾乎已能滿足以「善用 templates 構件」爲目標的讀者。至於 templates 衍化出來的眾多泛型技術和研發成果，諸如 STL, Loki, Boost，也都有了針對性各異的經典書籍幫助我們學習，如《*Generic Programming and the STL*》,《*Effective STL*》,《*Modern C++ Design*》,《*The C++ Standard Library*》,《*The Annotated STL Sources*》,《*The Boost Graph Library*》…。

那麼，在這整個技術主題中，還缺什麼嗎？

就我有限的想像力，思及語法面、語意面、應用面、專家建言、前衛發展、程式庫源碼剖析與技術分析…，幾乎是不缺什麼了。但是人蹤稀寥的角落裡，似乎還欠缺[3]：

(1) 諸如 Friend Name Injection, Curiously Recurring Template Pattern, Template Template Parameters, Member Templates 之類比較罕見的偏鋒。

(2) 諸如 Tuple, Traits Templates, Expression Templates, Template Metaprogramming, Type Functions 之類比較特殊的設計。

(3) 諸如 Template Argument Deduction, Template Overload Resolution, Looking Up Names in Templates, Templates Instantiation 之類的底層運作描述。

---

[1] 爲什麼這個現象沒有發生在其他語言及其所圈圍的技術領域中呢？因爲其他語言如 Java 和 C# 並不支援如此多采的泛型技術（很主要的一個原因是沒有 operator overloading/運算子重載可供搭配）。這種情況可能將有改變，因爲這些最受注目的高階語言不約而同地往 C++ 形式的泛型編程靠近。

[2] template 通常被譯爲「模板」，其意義是母模、模具，而不是土木建築工地現場用的「板模」。

[3] 此處列出的眾多術語皆採英文。附錄 D 有一份詞彙/術語表，其中有譯詞及意義解釋。

(4) 諸如 One Definition Rule, Empty Base Class Optimization 之類的肌理分析。

《*C++ Templates*》彌補了上述欠缺！此書亦對大多數書籍談到的 templates 相關議題做了完善的整理。可以說，就 templates 上上下下裡裡外外而言，這本書是百科全書。

上述所列都是較爲艱澀的主題。一般只做應用（或略探學理）的程式員是否需要如此深刻或如此角落的知識呢？這是見仁見智的個人選擇問題。「一切以眼前實用爲訴求」畢竟還是一種普遍存在的思維。但對高端技術發展而言，底層運作原理和前衛開拓勇氣是非常重要的。我在《*Modern C++ Design*》譯序中對此有過一些看法。

關切泛型編程技術的讀者，可能不復陌生本文一開始所列的那些經典書籍。我個人認爲本書技術層次比較齊近《*Modern C++ Design*》。當然我們都知道，沒有一本書可以涵蓋全世界，亦不會有哪一項知識獨家出現於一本書中。你可以從《*More Effective C++*》獲得本書第 18 章 Expression Templates 和第 20 章 Smart Pointer 的部分知識，可以從《*Modern C++ Design*》獲得本書第 22 章 Function Objects and Callbacks 的相關知識和第 17 章 Metaprograms 的更多知識，以及第 15 章 Policy Classes 的補充知識。你可以在《*The Annotated STL Source*》中看到本書第 15 章 Traits 和第 22 章 Functors、Binder 實作於 STL 的實際面貌。你也可以從《*Inside the C++ Object Model*》及前述三本 C++ 語言書中看到 template 相關構件的討論。本書帶有大量交叉參考，對讀者的旁徵博引是一個助力。

本書（中文版）由三人合譯。北京姜宏先生負責前半部（一二篇），南京榮耀先生負責後半部（三四篇），新竹侯捷總覽總製全書。本書的高品質十分得力於榮耀、姜宏兩位的技術實力。我們三人在網絡上做了許多溝通、討論、檢閱、覆閱、再覆閱。中文版附加大量譯註，包括多種編譯平台上的實測結果，並將定稿前之所有英文版勘誤修正於紙面。感謝兩位夥伴的實踐精神與熱忱，這是一次愉快而極高品質的合作。

關於行文風格，由於文字及版面工作由我總攬終定，所以沿襲侯捷一貫的用語風格和中英術語並陳的習慣。中英並陳無法全面，亦難在此簡述概貌，甚至並不全書一致（例如某些場合使用"pointer"某些場合使用"指標"，某些場合使用"object"某些場合使用"物件"，視上下語感和前後詞性平衡而定）。請讀者諒解一個事實：本書許多術語並無「中文爲主英文爲輔」的前提；在我所選定的某些術語上，中英並重。惟一的基準是：與 template 相關的術語近乎全部保留英文（亦時而並陳中文）。附錄 D 列有一份語彙/術語表，建議讀者先行瀏覽，不僅得以率先綜覽全書術語，亦可對中文譯名有一個梗概認識。

特別要說明的是，英文版的 ordinary 或 regular（例如 ordinary pointer, regular class, regular function），中文版譯爲「常規的」或「一般的」，技術上意指 non-template。

最後，容我感性表白。持續澆灌大量心血於一系列 C++ Templates, Generic Programming, STL 的學習、研究、寫作、翻譯達 5,6 年之久，此刻我很開心以這本書做爲終結。

侯捷 2004/01/08, 新竹

jjhou@jjhou.com（電子郵件）; http://www.jjhou.com（繁體網站）; http://jjhou.csdn.net（簡體網站）

譯序 by 榮耀

Templates（模板），以及以 templates 爲基礎的 generic programming（泛型編程）和 generic patterns（泛型範式），無疑是當今發展最活躍的 C++ 編程技術。這種欣欣向榮的局面並非源於鼓吹和狂熱，而是因爲 templates 及其相關技術的威力已經因爲形形色色各種成功的 templates 程式庫而得到了證明。

Templates 的第一個革命性應用是 Standard Template Library（STL，標準模板程式庫）。STL 將 templates 技術應用於「泛型容器+演算法」領域並展現得淋漓盡致。但 templates 的威力遠不止於此，其編譯期計算能力（compile-time computation）與設計範式（design patterns）的聯合運用，愈發吸引更多的智力投入，並已成爲 Boost、Loki 等新一代 C++ 程式庫之基石。

您現在看到的這本書，填補了 C++ templates 書籍領域由來已久的空白。此前，上有《*Modern C++ Design*》這樣專注於 templates 高級編程技法和泛型範式（generic patterns）的著作，下有《*The C++ Standard Library*》這樣針對特定模板框架和組件的使用指南。然而，如果對 templates 機制缺乏深入了解，您就很難「上下」自如，十有八九會被「卡」住。

本書內容由淺入深共分四篇。一、二兩篇重點介紹 templates 原理和運作機制，其中內容無論在深度或廣度方面均爲空前。三、四兩篇戮力論述 templates 設計技術和高級應用。這兩篇的內容與《*Modern C++ Design*》相比，立意有別，各有千秋。

即便是本書最基礎的第一篇，我相信大多數 C++ 程式員亦能從中學到不少「新知」。而第二篇對 templates 機制的深入講解，更是專家級 template programmer 之必備知識。本書後半部份（三、四篇）有助於您加深理解前半部份（一、二篇）所學知識，同時也是得以游刃有餘運用 templates 編程技術的有益示範，此外也可幫助您更好地理解和使用 STL、Boost、Loki 這一類 templates 程式庫，甚或激發您的創造力。

本書最顯著的風格是將文字說理和程式範例予以有機結合，使二者相輔相成，相得益彰。

說理乾淨俐落、鞭闢入里；範例短小精悍，錦上添花。全書技術飽滿，文字平和，堪稱 C++ 領域經典之作。我向每一位渴望透徹理解 C++ templates 技術的讀友推薦這本書。

在 C++ 學習方面，「過猶不及」往往成了「不求甚解」的藉口。我們在熟稔物件導向（Object Oriented）編程的同時，不要忘了，物件導向絕非 C++ 的全部，Templates 和 Generic Programming（泛型編程）亦佔半壁江山。作爲嚴肅認真的 C++ 程式員的您，面對一項早經驗證的成功技術，還要猶豫什麼？

感謝侯捷先生。先生言傳身教，榮耀受益終生。感謝姜宏先生，在我最繁忙之際，您的鼎力相助使本書初譯工作得以如期完成。感謝內人朱艷，您給予的理解、支持和無微不至的照料永遠是我前進的動力。

榮耀 2004/01/08, 南京
http://www.royaloo.com

譯序 by 姜宏

這是一本值得一讀再讀的好書。

出於對 STL 的興趣，我在 VC 5.0 的時代就開始研習 templates 技術。興趣雖然不低、使用雖然不少，然而直到我拿起這本書，才發現對 templates 的所知其實不多。本書講述了太多我從來未曾留意的細節。某些問題並非不曾碰過，但因工作所限無法深究，多數時間只是匆匆改兩行程式碼，把編譯器哄好便了事。直到今年三月收到侯捷先生的邀請協譯此書，我才真正端正態度，慎重審視起這個尖帽子朋友來。

本書涵括 templates 理念的三個方面：其本、其道、其用。本立道生，道以爲用。本書以 templates 的基本概念和原理開始，輔以 templates 的高級概念，並使用一半篇幅講述 templates 在設計和實際編程技法中的應用。論述明晰，講理清楚，大量實例更有助於讀者消化這樣一帖大劑量補藥。

本書語言頗具特色，由於兩位作者都是 C++ 委員會的活躍核心人物，敘述中極盡嚴密之能事。這固然顯得不那麼文采飛揚，但細細消化後，相信你會品味出作者的良苦用心。

Templates 是泛型思維（Generic Paradigm）的基礎。Templates 固然使你得以立即享用「標準容器」這一道好吃不貴的快餐，它真正的革命性質其實還在於以精練優雅的表現方式，體現設計範式（design patterns）的強大威力。Templates 帶來的泛型編程技術，固然是目前 C++ 編程的風潮，然而只是簡單運用 template 程式庫（如 STL 或 Boost），並不就意味著你抓住了泛型編程（Generic Programming）的精髓。

透過本書，你可以更容易地把握 templates 在泛型編程的實質功用，從而爲你的專案設計和程式碼注入新的活力。

本書與另外一本泛型設計的名著《*Modern C++ Design*》可謂交相輝映。對《*Modern C++ Design*》意猶未盡的讀者，可在本書找到如 type traits、policies、metaprograms、functors、smart pointers 等共通主題，並得以借本書之助，攀越「泛型編程」之關山，從而大步流星地趕上時代潮流。

我要感謝的首先是侯捷先生，他使我有機會參與本書的翻譯工作，我感到非常榮幸。還要感謝作者之一的 Daveed — 翻譯過程中我向他提了數十個問題，他有問必答，反饋迅速，而且答復嚴謹，不厭其詳，使我深受其益。最後，感謝我的同事 zd、cz、cx 長期以來對我的關心和照顧，以及家人對我終日伏案的寬容。

翻譯本書的過程中，我感覺自己對 C++ 的知識幾乎被完全翻修，本書使我以更寬廣的視角來看待 C++ 語言。願讀者和我有相同的體驗。祝您好胃口。

ㄠㄨㄢˊ 2004/01/08, 北京

目 錄

前言

C++ templates（模板）思想由來已逾十年，1990 年它就已經被載入所謂的 ARM 文獻（參見該書 p.653）之中。此前一些更專門的出版物對之亦有描述。儘管如此，十多年後的今天，我們仍然沒有在市面上看到哪一本書專注於這種迷人、複雜而極具威力之 C++ 特性的基礎概念和高級技巧。我們想改變這種狀況，因此決定撰寫這本關於 C++ templates 的書（也許這話說得不太謙虛）。

不過，我們（David 和 Nico）是帶著不同的背景和意圖來處理這項任務的。作爲一名經驗豐富的編譯器實作者和 C++ 標準委員會核心語言工作小組（C++ Standard Committee Core Language Working Group）成員，David 的興趣在於精確而詳盡地描述 templates 的威力（以及存在的問題）。作爲一名應用程式開發人員和 C++ 標準委員會程式庫工作小組（C++ Standard Committee Library Working Group）成員，Nico 的興趣在於以「能夠運用，並可從中受益」的方式來透徹理解所有 templates 技術。殊途同歸，我倆都想和您以及整個 C++ 社群分享知識，以協助避免更多的誤解、迷惑和（或）懸念。

因此，你即將看到帶有日常示例的概念性介紹，也會看到對 templates 精確行爲的詳盡描述。從 templates 基本原理開始，逐步發展到 templates 編程藝術，你將會探索（或重新探索）static polymorphism（靜態多型）、policy classes（策略類別）、metaprogramming（超編程）和 expression templates（算式模板）之類的技術。你還將獲得對 C++ 標準程式庫更深入的理解 — C++ 標準程式庫中的幾乎所有東西都涉及 templates。

本書寫作過程中，我們學到了不少東西，也得到許多樂趣。我們希望您閱讀時也有同樣的體驗。請盡情享用！

致謝

本書所呈現的想法、概念、解決方案和示例，來源甚廣。我們希望在此向過去數年給予我們幫助和支持的所有個人和公司表示謝意。

首先要感謝對我們的初稿給予意見的所有檢閱者。如果沒有他們的投入，本書絕不可能有如此成績和品質。本書檢閱者包括 Kyle Blaney, Thomas Gschwind, Dennis Mancl, Patrick McKillen 和 Jan Christiaan van Winkel。特別感謝 Dietmar Kuhl，他無比細緻地校閱和編輯了整本書，他的反饋訊息對本書品質有驚人的貢獻。

接下來感謝所有給予我們機會，使我們得以在不同平台上使用不同編譯器測試本書示例程式的所有個人和公司。非常感謝 Edison Design Group，感謝他們提供的卓越編譯器，以及其他支援。在 C++ 標準化以及本書編寫過程中，這些都是極大的助力。我們也要向免費的 GNU 和 egcs 編譯器的所有開發者表示深沉的謝意（特別是 Jason Merrill），並感謝 Microsoft 提供的 Visual C++ 評估版（Jonathan Caves, Herb Sutter 和 Jason Shirk 是我們在那兒的聯繫人）。

許多現有的「C++ 智慧靈光」是由 C++ 線上社群（online community）共同創造的。大多數源於有主持人、有管理制度的 Usenet 群組（moderated Usenet groups），像是 comp.lang.c++.moderated 和 comp.std.c++。我們因此特別感謝這些 Usenet 群組活躍的主持人，他們促使線上的討論往有益的方向走，而且富有建設性。我們也非常感謝過去這些年來花時間向大家描述和解釋想法的每一個人。

Addison Wesley 團隊又做了一件了不起的工作。我們非常感激 Debbie Lafferty（我們的編輯），感謝她爲支持本書而發出的溫和敦促、良好建議，以及艱苦不懈的工作。我們還要感謝 Marina Lang，是他首先在 Addison Wesley 內部發起這本書的製作計劃。Susan Winer 爲我們奉獻出前一輪編輯工作，有助於我們後期工作的成型。

Nico 的致謝

我的個人感謝（伴以許多親吻）首先要送給我的家庭：Ulli, Lucas, Anica 和 Frederic，他們爲這本書付出了極大的耐心、體諒和鞭策。

此外，我要感謝 David，他的專家知識十分驚人，他的耐心甚至更好（有時我提的問題真蠢）。與他共事，其樂無窮。

David 的致謝

我的妻子 Karina 對於這本書告一段落發揮了重要作用。感謝她在我生命中扮演的角色。當很多其他活動搶佔你的日程表時，「在業餘時間寫作」很快就失去了規律。Karina 幫我管理這個日程表，教我說『不』，以便有足夠的時間保持正常的寫作進度。不過，首先要感謝的是她對此一寫作專案所付出的令人驚訝的支持。感謝上帝，每一天她都給予我友好和愛意。

我也非常高興能和 Nico 共事。除了直接可見的文字貢獻，他的經驗和素養亦將我那可憐的信手塗鴉變成組織良好的成品。

「Template 先生」John Spicer 和「Overload 先生」Steve Adamczyk 是很棒的朋友和同事，而且在我看來他們倆還是 C++ 核心語言方面的終極權威。他們澄清（闡明）了本書描述的許多詭譎問題。如果你在這本書中發現對 C++ 語言元素的錯誤描述，幾乎肯定可以歸咎於我沒有與他們協商討論。

最後，我想對那些並沒有直接貢獻於此寫作專案，但卻提供了精神支持的人們，表達我的感激之情（喝彩的力量是不可低估的）。首先是我的父母，他們對我的愛護和鼓勵使得一切全然不同。還有許多朋友經常問我們『書寫得怎麼樣了？』，他們也都是精神鼓勵的來源：Michael Beckmann, Brett 和 Julie Beene, Jarran Carr, Simon Chang, Ho 和 Sarah Cho, Christophe De Dinechin, Peter 和 Ewa Deelman, Neil 和 Tammy Eberle, Sassan Hazeghi, Vikram Kumar, Jim 和 Lindsay Long, Franklin Luk, Richard 和 Marianna Morgan, Ragu Raghavendra, Jim 和 Phuong Sharp, Gregg Vaughn 和 John Wiegley。

1

關於本書
About This Book

作爲 C++ 的一部份，儘管 templates（模板）已經存在二十多年（並且以其他多種面目存在了幾乎同樣長的時間），但它還是會招引誤解、誤用和爭議。在此同時，愈來愈多人覺察 templates 是產生更乾淨、更快速、更精明的軟體的一個強而有力的手段。確實，templates 已經成爲數種新興 C++ 編程思維模型（programming paradigms）的基石。

但是我們發現，大多數現有書籍和文章對於 C++ templates 的理論和應用方面的論述，顯得過於淺薄。有些書籍雖然很好地評述了各種 template-based 技法，卻並沒有精確闡述 C++ 語言本身如何使這些技法得以施行。於是，無論新手或專家，都好像在和 templates 較勁，費盡心思去琢磨爲何他們的程式碼不能按想像的方式執行。

這樣的觀察結果，正是我們兩人寫這本書的主要動機之一。對於本書，我倆心中各有獨立的要旨和不同的寫作方式：

- David 的目標是爲讀者提供一份完整的參考手冊，其中講述 C++ template 語言機制的細節，以及重要的 templates 高級編程技法。他比較關注內容的精確與完備。
- Nico 希望爲自己和其他日常生活中使用 templates 的程式員帶來一本有幫助的書。這意味本書將以一種直觀易懂、結合實踐的方式呈現給讀者。

從某種意義上，你可以把我們看做是科學家和工程師的組合：我們按照相同的原則寫作，但側重點有些不同（當然，也有很多情況是重疊的）。

Addison-Wesley 把我倆的努力結合在一起，於是你擁有了一本我倆心目中兩方面緊密結合的書籍：既是 C++ template 教本（tutorial），也是 C++ template 的詳盡參考手冊（reference）。作爲教本，本書不僅涵蓋語言基本要素的介紹，也致力培養某種得以設計出切實可行之解決方案的直覺。作爲參考手冊，本書既包括 C++ template 的語法和語意，也是各種廣爲人知和鮮爲人知的編程慣用手法（idioms）和編程技術的全面總覽。

1.1 閱讀本書之前你應該知道的事

要想具備學習本書的最佳狀態，你應該先已了解 C++。本書中我們只講述某些語言特性的細節，並不涉及語言基礎知識。你應該熟知類別（classes）、繼承（inheritance）等概念，你應該能夠利用 C++ 標準程式庫所提供的組件（例如 iostreams 和各種容器, containers）編寫 C++ 程式。如有必要，我們會回顧某些微妙問題 — 即使這些問題並不直接與 templates 相關。這可確保本書對於專家和中級水準的程式員皆適用。

本書大部份以 1998 年的 *C++ Standard*（[Standard98]）爲依據，同時兼顧 C++ 標準委員會釋出的第一份技術勘誤（[Standard02]）中的說明。如果你覺得你的 C++ 基礎還不夠好，我們推薦你閱讀 [StroustrupC++PL]、[JosuttisOOP] 和 [JosuttisLib] 等書籍來充實知識。這些書非常好地講述了最新的 C++ 語言及其標準程式庫。你可以在附錄 B.3.5 找到更多參考資料。

1.2 本書組織結構

我們的目標是爲兩個族群提供必要資訊：(1) 準備開始使用 templates 並從中獲益者，(2) 富有經驗並希望緊追最新技術者。基於此種想法，我們決定將本書組織爲以下四篇：

- 第一篇介紹 templates 涉及的基本概念。這部份採用循序漸進的教本（tutorial）方式。
- 第二篇展現語言細節，對 template 相關構件（constructs）來說是一份便利的參考手冊。
- 第三篇講述 C++ templates 的基礎設計技術，從近乎微不足道的構想到精巧複雜的編程技法都有。某些內容在其他出版物中甚至從未被提到。
- 第四篇在前兩篇的基礎上討論 templates 的各種普遍應用。

每一篇都包含數章。此外我們還提供多份附錄，涵蓋內容不限於 templates，例如附錄 B 的「C++ 重載決議機制（overload resolution）」綜覽。

第一篇各章應該循序閱讀，例如第 3 章內容就是是建立在第 2 章內容之上。其他三篇各章之間的聯繫較鬆。例如你可以閱讀 functors 那一章（第 22 章），不必先閱讀 smart pointers 那一章（第 20 章）。

最後，我們提供了一份相當完備的索引，便利讀者以自己的方式，跳脫任何順序來閱讀本書。

1.3 如何閱讀本書

如果你是一位 C++ 程式員，打算學習或鞏固 templates 概念，請仔細閱讀第一篇（基礎篇）。即使你已經對 templates 相當熟稔，快速翻閱第一篇也可以幫助你熟悉本書的寫作風格和我們慣用的術語。該篇內容也可以幫助你有條理地組織你的一些 templates 程式碼。

視個人學習方式的不同，你可以先消化第二篇的眾多 templates 細節，也可以跳到第三篇領會實際程式中的 templates 編程技術，再回頭閱讀第二篇以了解更多的語言細節。後一種方式對於那些每天和 templates 打交道的人可能更有好處。第四篇和第三篇有點類似，但更側重於如何運用 templates 協助完成某個特定問題，而不是以 templates 來輔助設計。鑽研第四篇之前，最好先熟悉第三篇的內容。

本書附錄包括了正文之中反復提及的一些內容。我們儘量使這些內容更有趣。

經驗顯示，學習新知識的最好方法是假以實例。所以全書貫穿了諸多實例。某些實例以寥寥數行程式碼闡明一個抽象概念，某些實例則是與正文內容對應的一個完整範例。後一類實例會在開頭以一行註釋標明其檔名。你可以在本書支援網站 http://www.josuttis.com/tmplbook/ 中找到這些檔案。

1.4 本書編程風格（Programming Style）

每一位 C++ 程式員都有自己的一套編程風格，我倆也不例外。這就引來了各種問題：哪兒應該插入空白符號、怎麼擺放分隔符號（大括號、小括號）…等等。我們儘量保持全書風格一致，當然有時候我們也對特殊問題作出讓步。例如在教本（初階）部份我們鼓勵以空白符號和較具體的命名方式提高程式可讀性，而在高階主題中，較緊湊的風格可能更加適宜。

我們有一個他人不太常用的習慣，用以宣告型別（types）、參數（parameters）和變數（variables），希望你能多加注意。下面數種方式無疑都是合理的：

```
void foo (const int &x);
void foo (const int& x);
void foo (int const &x);
void foo (int const& x);
```

儘管較爲罕見，我們還是決定在表達「固定不變的整數」（constant integer）時使用 `int const` 而不寫成 `const int`。這麼做有兩個原因，第一，這很容易顯現出「什麼是不能變動的（*what is constant*）」。不能變動的量總是 `const` 飾詞之前的那個東西。儘管以下兩式等價：

```
const int N = 100;     //一般人可能的寫法
int const N = 100;     //本書習慣寫法
```

但對以下述句來說就不存在所謂的等價形式了：

```
int* const bookmark;  // 指標 bookmark 不能變動，但指標所指內容（int）可以變動
```

如果你把 const 飾詞放在運算子 * 之前，那就改變了原意。本例之中不能變動的是指標本身，不是指標所指的內容。

第二個原因和語法替換原則（syntactical substitution principle）有關，那是處理 template 程式碼時常會遭遇的問題。考慮下面兩個型別定義[1]：

```
typedef char* CHARS;
typedef CHARS const CPTR; // 一個用以「指向 chars」的 const 指標
```

如果我們做文字上的替換，把 CHARS 替換爲其代表物，上述第二個宣告的原意就得以保留：

```
typedef char* const CPTR; // 一個用以「指向 chars」的 const 指標
```

然而如果我們把 const 寫在被修飾物之前，上述規則便不適用。考慮上述宣告的另一種變化：

```
typedef char* CHARS;
typedef const CHARS CPTR; // 一個用以「指向 chars」的 const 指標
```

現在，對 CHARS 進行文字替換，會導出不同的含義：

```
typedef const char* CPTR; // 一個用以「指向 const chars」的指標
```

面對 volatile 飾詞，也有同樣考量。

關於空白符號，我們決定把它放在 "&" 符號和參數名稱中間：

```
void foo (int const& x);
```

這樣可以更加突出參數的型別和名稱。無可否認，以下宣告方式可能較易引起疑惑：

```
char *a, b;
```

根據從 C 語言繼承下來的規則，a 是個指標而 b 是個一般的 char。爲了避免這種混淆，我們可以一次宣告一個變數，不要集中於同一行宣告式。

本書並不是一本討論 C++ 標準程式庫的書，但我們確實在一些例子中用到了標準程式庫。一般來說，我們使用 C++ 特有的表頭檔（例如 <iostream> 而非 <stdio.h>）。惟一的例外是 <stddef.h>，我們使用它而不使用 <cstddef>，以避免型別 size_t 和 ptrdiff_t 被冠以 std:: 前綴詞。這樣做更具可移植性，而且 std::size_t 並不比 size_t 多出什麼好處。

---

[1] 注意，C++ 的 typedef 所定義的是型別別名（type alias），而不是一個新型別。例如：

```
typedef int Length; // 定義 Length 爲 int 的一個別名（alias）
int i = 42;
Length l = 88;
i = l;            // OK
l = i;            // OK
```

1.5 標準 vs.現實（Standard versus Reality）

C++ Standard 自 1998 年末就已制定完備。但是直到 2002 年，仍然沒有一個編譯器公開宣稱自己「完全符合標準」。各家編譯器對於 *C++ Standard* 的支援程度仍然表現各異。有些編譯器可以順利編譯本書大部份程式碼，但也有一些很流行的編譯器無法編譯本書中相當數量的實例。我們儘量提供另一種實現技術，使那些「只支援 *C++ Standard* 子集」的編譯器得以順利工作。但是某些範例並不存在第二種實現技術。我們希望 *C++ Standard* 支援問題得以解決 — 全世界程式員都迫切需要編譯器廠商提供對 *C++ Standard* 的完整支援。

即便如此，C++ 語言仍然隨著時間的推移而不斷演進。已經有爲數眾多的 C++ 社群專家（無論他們是否爲 C++ 標準委員會成員）討論著 C++ 語言的各種改進方案，其中有一些直接影響到 templates。本書第 13 章討論了一些語言演化趨勢。

1.6 範例程式碼及更多資訊

你可以連接本書支援網站，取得所有範例程式碼以及更多資訊。網站的 URL 爲：

```
http://www.josuttis.com/tmplbook
```

另外，你也可以從 David Vandevoorde 的網站獲得許多 templates 相關資訊：

```
http://www.vandevoorde.com/Templates
```

p.499「參考書目和資源」提供了其他一些資訊。

1.7 反饋（Feedback）

無論讚揚或批評，我們都歡迎。我們竭盡所能希望帶給你一本優秀書籍，但總是必須在某個時間點停止寫作、審閱、調整等工作，從而使這本書得以面世。你可能會發現不同形式的錯誤或矛盾，可能會發現某些地方值得改善，可能會覺得某些地方有所遺漏。您的反饋使我們有機會透過本書支援網站通知所有讀者，並讓我們有機會在後續版本中改進。

聯繫我們的最佳方式是電子郵件（email）：

```
tmplbook@josuttis.com
```

提交任何報告之前，請確認您已檢閱網站上的勘誤表。

非常感謝！

第一章 基本認識

The Basic

本篇介紹 C++ templates 的基本概念和語言特性。首先展示 function templates 和 class templates 的範例，開啓對大體目標和基本概念的討論。隨後講述其他 template 基礎技術，例如 nontype template parameters、關鍵字 `typename`、以及 member templates。最後給出一些 templates 實際應用心得。

《*Object-Oriented Programming in C++*》（by Nicolai M. Josuttis, John Wiley & Sons Ltd., ISBN 0-470-84399-3）和本書共享了一部份 templates 入門相關內容。那本書循序漸進地教你 C++ 語言和 C++ 標準程式庫的所有特性及實際用法。

爲什麼使用 Templates？

C++ 要求我們使用各種特定型別（specific types）來宣告變數、函式和其他各種實物（entities）；然而，很多用以處理「不同型別之資料」的程式碼看起來都差不多。特別是當你實作演算法（像是 quicksort），或實作如 linked-list 或 binary tree 之類的資料結構時，除了所處理的型別不同，程式碼實際上是一樣的。

如果你使用的編程語言並沒有針對這個問題支援某種特殊的語言特性，那麼你有數種退而次之的選擇：

1. 針對每一種型別寫一份程式碼。
2. 使用 common base type（通用基礎型別，例如 `Object` 或 `void*`）來撰碼。
3. 使用特殊的 preprocessors（預處理器。譯註：意指編譯之前預先處理的巨集, macros）。

如果你是從 C、Java 或其他類似語言轉到 C++ 陣營，可能這三種方法你都用過。但是每一種方法都有其缺點：

1. 如果爲每一種型別寫一份程式碼，你就是「不斷做重複的事情」。你會在每一份程式碼中犯下相同的錯誤（如果有的話），而且不敢使用更複雜但更好的演算法，因爲這會帶來更多錯誤。

2. 如果你使用一個 common base class（通用基礎類別），就無法獲益於「型別檢驗」。而且某些 classes 可能必須從其他特殊的 base classes 繼承而來，這就給程式維護工作帶來更多困難。
3. 如果你使用特殊的 preprocessors（預處理器），例如 C/C++ preprocessors，你將失去「格式化源碼」（formatted source code）的好處。預處理機制對於作用域（scope）和型別（types）一無所知，只是進行簡單的文字替換。

> 譯註：以上所謂喪失「格式化源碼（formatted source code）的好處」，意思是如果你使用巨集（macros）並發生編譯錯誤，編譯器給出的行號是使用巨集的那一行，而不是定義巨集的那一行。此一補充得原作者 David Vandevoorde 之授意。

Templates 可以解決你的問題，而又不帶上述提到的缺點。所謂 templates，是爲「尚未確定之型別」所寫的 functions 或 classes。使用 templates 時，你可以顯式（明確）或隱式（隱喻）地將型別當做引數（argument）來傳遞。由於 templates 是一種語言特性，型別檢查（type checking）和作用域（scope）的好處不會喪失。

Templates 如今已被大量運用。就拿 C++ 標準程式庫來說，幾乎所有程式碼都以 templates 寫成。標準程式庫提供各式各樣的功能：對 objects 和 values 排序的各種演算法、管理各種元素的資料結構（容器類別，container classes）、支援各種字元集的 string（字串）。Templates 使我們得以將程式的行爲參數化、對程式碼優化（最佳化），並將各種資訊參數化。這些都會在後續章節中講述。現在，讓我們從最簡單的 templates 開始。

2

Function Templates
函式模板

本章將介紹 function templates。所謂 function templates 是指藉由參數化手段表現一整個族群的 functions（函式）。

2.1 Function Templates 初窺

Function templates 可爲不同型別的資料提供作用行爲。一個 function template 可以表示一族（一整群）functions，其表現和一般的 function 並無二致，只是其中某些元素在編寫時尚未確定。換言之，那些「尚未確定的元素」被「參數化」了。讓我們看一個實例。

2.1.1 定義 Template

下面的 function template 傳回兩個數值中的較大者：

```
// basics/max.hpp

template <typename T>
inline T const& max (T const& a, T const& b)
{
    // 如果 a<b 就傳回 b，否則傳回 a
    return a < b ? b : a;
}
```

這一份 template 定義式代表了一整族 functions，它們的作用都是傳回 a 和 b 兩參數中的較大者。兩個參數的型別尚未確定，我們說它是 "template parameter T"。如你所見，template parameters 必須以如此形式加以宣告：

```
template < 以逗號分隔的參數列 >
```

上述例子中，參數列就是 typename T。請注意，例中的「小於符號」和「大於符號」在這裡被當作角括號（尖括號）使用。關鍵字 typename 引入了一個所謂的 type parameter（型別參數）—這是目前爲止 C++ 程式中最常使用的一種 template parameter，另還存在其他種類的 template parameter（譯註：如 nontype parameter，「非型別參數」），我們將在第 4 章討論。

此處的型別參數是 T，你也可以使用其他任何標識符號（identifier）來表示型別參數，但習慣寫成 T（譯註：代表 Type）。Type parameters 可表示任意型別，在 function template 被呼叫時，經由傳遞具體型別而使 T 得以被具體指定。你可以使用任何型別（包括基本型別和 class 型別等等），只要它支援 T 所要完成的操作。本例中型別 T 必須支援 operator< 以比較兩值大小。

由於歷史因素，你也可以使用關鍵字 **class** 代替關鍵字 **typename** 來定義一個 type parameter。關鍵字 **typename** 是 C++ 發展晚期才引進的，在此之前只能經由關鍵字 **class** 引入 type parameter。關鍵字 **class** 目前依然可用。因此 template max() 也可以被寫成如下等價形式：

```
template <class T>
inline T const& max (T const& a, T const& b)
{
    // 如果 a<b 就傳回 b，否則傳回 a
    return a < b ? b : a;
}
```

就語意而言，前後兩者毫無區別。即便使用關鍵字 **class**，你還是可以把任意型別（包括 non-class 型別）當作實際的 template arguments。但是這麼寫可能帶來一些誤導（讓人誤以爲 T 必須是 class 型別），所以最好還是使用關鍵字 **typename**。請注意，這和 class 的型別宣告並不是同一回事：宣告 type parameters 時我們不能把關鍵字 **typename** 換成關鍵字 **struct**。

2.1.2 使用 Template

以下程式示範如何使用 max() function template：

```
// basics/max.cpp

#include <iostream>
#include <string>
#include "max.hpp"

int main()
{
    int i = 42;
    std::cout << "max(7,i):   " << ::max(7,i) << std::endl;

    double f1 = 3.4;
    double f2 = -6.7;
    std::cout << "max(f1,f2): " << ::max(f1,f2) << std::endl;
```

```
    std::string s1 = "mathematics";
    std::string s2 = "math";
    std::cout << "max(s1,s2): " << ::max(s1,s2) << std::endl;
}
```

程式呼叫了 max() 三次。第一次所給引數是兩個 int，第二次所給引數是兩個 double，最後一次給的是兩個 std::string。每一次 max() 均比較兩值取其大者。程式運行結果爲：

```
max(7,i):   42
max(f1,f2): 3.4
max(s1,s2): mathematics
```

注意程式對 max() 的三次呼叫都加了前綴字 "::"，以便確保被喚起的是我們在全域命名空間（global namespace）中定義的 max()。標準程式庫內也有一個 std::max() template，可能會在某些情況下被喚起，或在呼叫時引發模稜兩可（ambiguity，歧義性）[2]。

一般而言，templates 不會被編譯爲「能夠處理任意型別」的單一實物（entity），而是被編譯爲多個個別實物，每一個處理某一特定型別[3]。因此，針對三個型別，max() 被編譯成三個實物。例如第一次呼叫 max()：

```
int i = 42;
... max(7,i) ...
```

使用的是「以 int 爲 template parameter T」的 function template，語意上等同於呼叫以下函式：

```
inline int const& max (int const& a, int const& b)
{
    // 如果 a<b 就傳回 b，否則傳回 a
    return a < b ? b : a;
}
```

以具體型別替換 template parameters 的過程稱爲「具現化」（*instantiation*，或稱「實體化」）。過程中會產生 template 的一份實體（*instance*）。不巧的是，*instantiation*（具現化、具現化產品）和 *instance*（實體）這兩個術語在 OO（物件導向）編程領域中有其他含義，通常用來表示一個 class 的具體物件（concrete object）。本書專職討論 templates，因此當我們運用這個術語時，除非另有明確指示，表達的是 templates 方面的含義。

注意，只要 function template 被使用，就會自動引發具現化過程。程式員沒有必要個別申請具現化過程。

---

[2] 如果某個引數的型別定義於 namespace std 中（例如 string），根據 C++搜尋規則，::max() 和 std::max() 都會被找到（譯註：那就會引發歧義性）。

[3] 「一份實物，適用所有型別」，理論上成立，實際不可行。畢竟所有語言規則都奠基於「將會產出不同實物」的概念（all language rules are based on the concept that different entities are generated）。

類似情況，另兩次對 max() 的呼叫被具現化爲：

```
const double& max (double const&, double const&);
const std::string& max (std::string const&, std::string const&);
```

如果試圖以某個型別來具現化 function template，而該型別並未支援 function template 中用到的操作，就會導致編譯錯誤。例如：

```
std::complex<float> c1, c2;     // 此型別並不提供 operator<
...
max(c1,c2);                     // 編譯期出錯
```

實際上，templates 會被編譯兩次：

1. 不具現化，只是對 template 程式碼進行語法檢查以發現諸如「缺少分號」等等的語法錯誤。
2. 具現化時，編譯器檢查 template 程式碼中的所有呼叫是否合法，諸如「未獲支援之函式呼叫」便會在這個階段被檢查出來。

這會導致一個嚴重問題：當 function template 被運用而引發具現化過程時，某些時候編譯器需要用到 template 的原始定義。一般情況下，對普通的（non-template）functions 而言，編譯和連結兩步驟是各自獨立的，編譯器只檢查各個 functions 的宣告式是否和呼叫式相符，然而 template 的編譯破壞了這個規則。解決辦法在第 6 章討論。眼下我們可以用最簡單的解法：把 template 程式碼以 inline 形式寫在表頭檔（header）中。

2.2 引數推導（Argument Deduction）

當我們使用某一型別的引數呼叫 max() 時，template parameters 將以該引數型別確定下來。如果我們針對參數型別 T const& 傳遞兩個 ints，編譯器必然能夠推導出 T 是 int。注意這裡並不允許「自動型別轉換」。是的，每個 T 都必須完全匹配其引數。例如：

```
template <typename T>
inline T const& max(T const& a, T const& b);
...
max(4, 7);        // OK，兩個 T 都被推導爲 int
max(4, 4.2);      // 錯誤：第一個 T 被推導爲 int，第二個 T 被推導爲 double
```

有三種方法可以解決上述問題：

1. 把兩個引數轉型爲相同型別：

```
max(static_cast<double>(4), 4.2);        // OK
```

2. 明確指定 T 的型別：

```
max<double>(4, 4.2);                     // OK
```

3. 對各個 template parameters 使用不同的型別（譯註：意思是不要像上面那樣都叫做 T）。

下一節詳細討論這些問題。

2.3 Template Parameters（模板參數）

Function templates 有兩種參數：

1. Template parameters（模板參數），在 function template 名稱前的一對角（尖）括號中宣告：

```
template <typename T>       // T 是個 template parameter
```

2. Call parameters（呼叫參數），在 function template 名稱後的小（圓）括號中宣告：

```
... max (T const& a, T const& b);    // a 和 b 是呼叫參數
```

template parameters 的數量可以任意，但你不能在 function templates 中爲它們指定預設引數值[4]（這一點與 class templates 不同）。例如你可以在 max() template 中定義兩個不同型別的呼叫參數：

```
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b)
{
   return a < b ? b : a;
}
...
max(4, 4.2);      // OK。回傳型別和第一引數型別相同
```

這似乎是一個可以爲 max() template 的參數指定不同型別的好辦法，但它也有不足。問題在於你必須宣告傳回值的型別。如果你使用了其中一個型別，另一個型別可能被轉型爲該型別。C++ 沒有提供一個機制用以選擇「效力更大的型別, *the more powerful type*」（然而你可以藉由某些巧妙的 template 編程手段來提供這種機制，參見 15.2.4 節, p.271）。因此，對於 42 和 66.66 兩個呼叫引數，max() 的傳回值要嘛是 double 66.66，要嘛是 int 66。另一個缺點是，把第二參數轉型爲第一參數的型別，會產生一個區域暫時物件（local temporary object），因而無法以 *by reference* 方式傳回結果[5]。因此在本例之中，回傳型別必須是 T1，不能是 T1 const&。

由於 call parameters 的型別由 template parameters 建立，所以兩者往往互相關聯。我們把這種概念稱爲 function template argument deduction（函式模板引數推導）。它使你可以像呼叫一個常規（意即 non-template）函式一樣來呼叫 function template。

然而正如先前提到的那樣，你也可以「明確指定型別」來具現化一個 template：

---

[4] 這個限制主要是由於 function templates 開發歷史上碰到的小問題導致。對新一代 C++ 編譯器而言，這已經不再是問題了。將來這個特性也許會包含於 C++語言本身。參見 13.3 節, p.207。

[5] 你不能以 *by reference* 方式傳出函式內的 local object，因爲它一旦離開函式作用域，便不復存在。

```
template <typename T>
inline T const& max (T const& a, T const& b);
...
max<double>(4,4.2);  // 以 double 型別具現化 T
```

當 template parameters 和 call parameters 之間沒有明顯聯繫，而且編譯器無法推導出 template parameters 時，你必須明確地在呼叫時指定 template arguments。例如你可以爲 max() 引入第三個 template argument type 作爲回傳型別：

```
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b);
```

然而「引數推導機制」並不對回傳型別進行匹配[6]，而且上述的 RT 也並非函式呼叫參數（call parameters）中的一個；因此編譯器無法推導出 RT。你不得不像這樣明確指出 template arguments：

```
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b);
...
max<int,double,double>(4, 4.2);
    // OK，但是相當冗長（譯註：因爲其實只需明寫第三引數型別，卻連前兩個引數型別都得寫出來）
```

以上我們所看到的是，要嘛所有 function template arguments 都可被推導出來，要嘛一個也推導不出來。另有一種作法是只明確寫出第一引數，剩下的留給編譯器去推導，你要做的只是把所有「無法被自動推導出來的引數型別」寫出來。因此，如果把上述例子中的參數順序改變一下，呼叫時就可以只寫明回傳型別：

```
template <typename RT, typename T1, typename T2>
inline RT max (T1 const& a, T2 const& b);
...
max<double>(4,4.2);                  // OK，返回型別爲 double
```

此例之中，我們呼叫 max() 時，只明確指出回傳型別 RT 爲 double，至於 T1 和 T2 兩個參數型別會被編譯器根據呼叫時的引數推導爲 int 和 double。

注意，這些 max() 修改版本並沒帶來什麼明顯好處。在「單一參數」版本中，如果兩個引數的型別不同，你可以指定參數型別和回返值型別。總之，爲儘量保持程式碼簡單，使用「單一參數」的 max() 是不錯的主意。討論其他 template 相關問題時，我們也會遵守這個原則。

引數推導過程的細節將在第 11 章討論。

---

[6] 推導過程也可以看作是重載決議機制（overload resolution）的一部份，兩者都不倚賴回傳值的型別來區分不同的呼叫。惟一的例外是：轉型運算子成員函式（conversion operator members）倚賴回傳型別來進行重載決議（overload resolution）。（譯註：「轉型運算子」函式名稱形式如下：operator *type*()，其中的 *type* 可爲任意型別；無需另外指出回傳型別，因爲函式名稱已經表現出回傳型別。）

2.4 重載（Overloading）Function Templates

就像常規（意即 non-template）functions 一樣，function templates 也可以被重載（譯註：C++標準程式庫中的許多 STL 演算法都是如此）。這就是說，你可以寫出多個不同的函式定義，並使用相同的函式名稱；當客戶呼叫其中某個函式時，C++編譯器必須判斷應該喚起哪一個函式。即使不牽扯 templates，這個推斷過程也非常複雜。本節討論的是，一旦涉及 templates，重載將是一個怎樣的過程。如果你對 non-templates 情況下的重載機制還不太清楚，可以先參考附錄 B，那裡我們對重載機制做了相當深入的講解。

下面這個小程式展示如何重載一個 function template：

```
// basics/max2.cpp

// 傳回兩個 ints 中的較大者
inline int const& max (int const& a, int const& b)
{
    return a < b ? b : a;
}

// 傳回兩任意型別的數值中的較大者
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// 傳回三個任意型別值中的最大者
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);
}

int main()
{
    ::max(7, 42, 68);           // 喚起「接受三個引數」的函式
    ::max(7.0, 42.0);           // 喚起 max<double>（經由引數推導）
    ::max('a', 'b');            // 喚起 max<char>（經由引數推導）
    ::max(7, 42);               // 喚起「接受兩個 int 引數」的 non-template 函式
    ::max<>(7, 42);             // 喚起 max<int>（經由引數推導）
    ::max<double>(7, 42);       // 喚起 max<double>（無需引數推導）
    ::max('a', 42.7);           // 喚起「接受兩個 int 引數」的 non-template 函式
}
/* 譯註：ICL7.1/g++ 3.2 順利通過本例。VC6 無法把最後一個呼叫匹配到常規的（non-template）函式 max()，造成編譯失敗。VC7.1 可順利編譯，但對倒數第二個呼叫給出警告：雖然它喚起的是 function template max()，但它發現常規函式 max()與這個呼叫更匹配。*/
```

這個例子說明：non-template function 可以和同名的 function template 共存，也可以和其相同型別的具現體共存。當其他要素都相等時，重載決議機制會優先選擇 non-template function，而不選擇由 function template 具現化後的函式實體。上述第四個呼叫便是遵守這條規則：

```
::max(7, 42);       // 兩個引數都是 int，吻合對應的 non-template function
```

但是如果可由 template 產生更佳匹配，則 template 具現體會被編譯器選中。前述的第二和第三個呼叫說明了這一點：

```
::max(7.0, 42.0);           // 喚起 max<double>（經由引數推導）
::max('a', 'b');            // 喚起 max<char>（經由引數推導）
```

呼叫端也可以使用空的 template argument list，這種形式告訴編譯器「只從 template 具現體中挑選適當的呼叫對象」，所有 template parameters 都自 call parameters 推導而得：

```
::max<>(7, 42);             // 喚起 max<int>（經由引數推導）
```

另外，「自動型別轉換」只適用於常規函式，在 templates 中不予考慮，因此前述最後一個呼叫喚起的是 non-template 函式。在該處，'a' 和 42.7 都被轉型爲 int：

```
::max('a', 42.7);       // 本例中只有 non-template 函式才可以接受兩個不同型別的引數
```

下面是一個更有用的例子，爲指標型別和 C-style 字串型別重載了 max() template：

```
// basics/max3.cpp

#include <iostream>
#include <cstring>
#include <string>

// 傳回兩個任意型別值的較大者
template <typename T>
inline T const& max (T const& a, T const& b)
{
   return a < b ? b : a;
}

// 傳回兩個指標的較大者（所謂較大是指「指標所指之物」較大）
template <typename T>
inline T* const& max (T* const& a, T* const& b)
{
   return *a < *b ? b : a;
}
```

```
// 傳回兩個 C-style 字串的較大者（譯註：C-style 字串必須自行定義何謂「較大」）。
inline char const* const& max (char const* const& a,
                               char const* const& b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}

int main()
{
    int a=7;
    int b=42;
    ::max(a,b);     // 喚起「接受兩個 int」的 max()

    std::string s = "hey";
    std::string t = "you";
    ::max(s,t);     // 喚起「接受兩個 std::string」的 max()

    int *p1 = &b;
    int *p2 = &a;
    ::max(p1,p2);   // 喚起「接受兩個指標」的 max()

    char const* s1 = "David";
    char const* s2 = "Nico";
    ::max(s1, s2);  // 喚起「接受兩個 C-style 字串」的 max()
}
```

注意，所有重載函式都使用 *by reference* 方式來傳遞引數。一般說來，不同的重載形式之間最好只存在「絕對必要的差異」。各重載形式之間應該只存在「參數個數的不同」或「參數型別的明確不同」，否則可能引發各種副作用。舉個例子，如果你以一個「*by value* 形式的 `max()`」重載一個「*by reference* 形式的 `max()`」（譯註：兩者之間的差異不夠明顯），就無法使用「三引數」版本的 `max()` 來取得「三個 C-style 字串中的最大者」：

```
// basics/max3a.cpp

#include <iostream>
#include <cstring>
#include <string>

// 傳回兩個任意型別值的較大者（call-by-reference）
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}
```

```
// 傳回兩個 C-style 字串的較大者（call-by-value）
inline char const* max (char const* a, char const* b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}

// 傳回三個任意型別值的最大者（call-by-reference）
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);   // 當 max(a,b)採用 by value 形式時，此行錯誤
}

int main()
{
    ::max(7, 42, 68);       // OK

    const char* s1 = "frederic";
    const char* s2 = "anica";
    const char* s3 = "lucas";
    ::max(s1, s2, s3);      // ERROR
}
```

本例中針對三個 C-style 字串呼叫 `max()`，會出現問題。以下這行述句是錯誤的：

```
return ::max (::max(a,b), c);
```

因爲 C-style 字串的 `max(a,b)`重載函式創建了一個新而暫時的區域值（a new, temporary local value），而該值卻以 *by reference* 方式被傳回（那當然會造成錯誤）。

譯註：以 ICL7.1 編譯上述程式，只產生一個警告：

```
warning #879: returning reference to local variable
  return strcmp(a,b) < 0 ? b : a;
```

運行結果正確。VC6 的 `namespace std` 中不包含 `strcmp`；它給出和 ICL7.1 一樣的警告。VC7.1 和 g++ 3.2 則連警告都沒有。

這只是細微的重載規則所引發的非預期行爲例子之一。當函式呼叫動作發生時，如果不是所有重載形式都在當前範圍內可見，那麼上述錯誤可能發生，也可能不發生。事實上，如果把「三引數」版本的 `max()`寫在接受兩個 `ints` 的 `max()`前面（於是後者對前者而言不可見），那麼在呼叫「三引數」`max()`時，會間接喚起「雙引數」`max()` function template：

```
// basics/max4.cpp

// 傳回兩個任意型別值的較大者
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// 傳回三個任意型別值的最大者
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);
      // 即使引數型別都是 int，這裡也會呼叫 max() template。因爲下面的函式定義來得太遲。
}

// 傳回兩個 ints 的較大者
inline int const& max (int const& a, int const& b)
{
    return a < b ? b : a;
}
```

9.2 節, p.121 詳細討論這個問題。就目前而言，你應該遵循一條準則：總是把所有形式的重載函式寫在它們被呼叫之前。

2.5 摘要

- Function templates 可以針對不同的 template arguments 定義一整族（a family of）函式。
- Function templates 將依照傳遞而來的引數（arguments）的型別而被具現化（instantiated）。
- 你可以明確指出 template parameters。
- Function templates 可以被重載（overloaded）。
- 重載 function templates 時，不同的重載形式之間最好只存在「絕對必要的差異」。
- 請確保所有形式的重載函式都被寫在它們的被呼叫點之前。