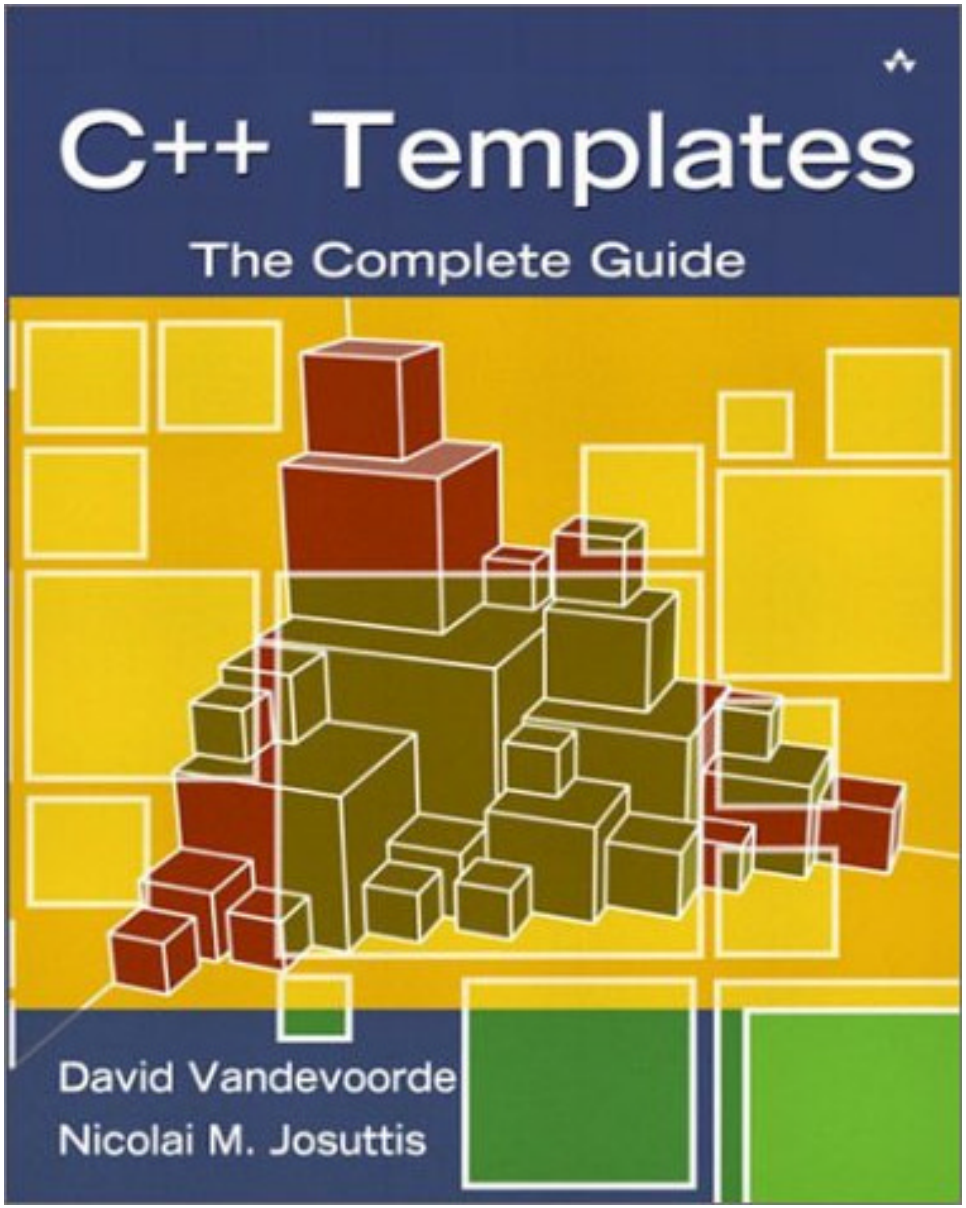


中文版：《C++Templates全覽》 侯捷/榮耀/姜宏 譯

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。

3

Class Templates
類別模板

就像上一章所說的 functions 那樣，classes 也可以針對一或多個型別被參數化。用來管理「各種不同型別的元素」的 container classes（容器類別）就是典型例子。運用 class templates 你可以實作出可包容各種型別的 container class。本章將以一個 stack class 作爲 class templates 實例。

3.1 實作 Class Template `Stack`

和 function templates 一樣，我們在單一表頭檔（header）中宣告和定義 class `Stack<>` 如下（6.3 節, p.68 將討論宣告和定義分離的模型）：

```
// basics/stack1.hpp

#include <vector>
#include <stdexcept>

template <typename T>
class Stack {
private:
    std::vector<T> elems;       // 元素
public:
    void push(T const&);        // push 元素
    void pop();                 // pop 元素
    T top() const;              // 傳回最頂端元素
    bool empty() const {        // stack 是否爲空
        return elems.empty();
    }
};
```

```
template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem);  // 追加（附於尾）
}

template <typename T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop: empty stack");
    }
    elems.pop_back();       // 移除最後一個元素
}

template <typename T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top: empty stack");
    }
    return elems.back();    // 傳回最後一個元素的拷貝
}
```

正如你所見，這個 class template 是以標準程式庫的 class template `vector<>` 爲基礎實作出來的，這樣我們就可以不考慮記憶體管理、*copy* 建構式、*assignment* 運算子等等，從而得以把注意力放在這個 class template 的介面上。

3.1.1 Class Templates 的宣告

宣告 class templates 的動作和宣告 function templates 類似：在宣告式之前加一條述句，以任意標識符號宣告型別參數（type parameters）。我們還是以 `T` 作爲標識符號：

```
template <typename T>
class Stack {
    ...
};
```

相同的規則再次適用：關鍵字 **class** 可以代替關鍵字 **typename**：

```
template <class T>
class Stack {
    ...
};
```

在 class template 內部，T 就像其他任意型別一樣，可用來宣告成員變數（member variables）和成員函式（member functions）。本例之中 T 用來宣告 vector<> 所容納的元素型別、宣告一個「接受 T const&」的 push() 函式、以及宣告一個「回傳型別爲 T」的 top() 函式：

```
template <typename T>
class Stack {
private:
    std::vector<T> elems;       // 元素
public:
    Stack();                    // 建構式
    void push(T const&);        // push 元素
    void pop();                 // pop 元素
    T top() const;              // 傳回最頂端的元素
};
```

這個 class 的型別爲 Stack<T>，T 是一個 template parameter。現在，無論何時你以這個 class 宣告變數或函式時，都應該寫成 Stack<T>。例如，假設你要宣告自己的 *copy* 建構式和 *assignment* 運算子，可以寫爲[7]：

```
template <typename T>
class Stack {
    ...
    Stack (Stack<T> const&);                    // copy 建構式
    Stack<T>& operator= (Stack<T> const&);      // assignment 運算子
    ...
};
```

然而如果只是需要 class 名稱而不是 class 型別時，只需寫 Stack 即可。建構式和解構式的宣告就屬於這種情況。

[7] 根據 C++ *Standard*，這條規則有一些例外，參見 9.2.3 節, p.126。爲了使你的程式碼更穩固，當有必要提供型別時，你應該寫出型別的完整形式。

3.1.2 成員函式（Member Functions）的實作

爲了定義 class template 的成員函式，你必須指出它是個 function template，而且你必須使用 class template 的全稱。因此 Stack<T> 的成員函式 push() 看起來便像這樣：

```
template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem);  // 將傳入的元素 elem 附加於尾
}
```

這裡呼叫了 vector 的成員函式 push_back()，把元素 elem 追加到 elems 尾端。

注意，對一個 vector 進行 pop_back()，只是把最後一個元素移除，並不傳回該元素。這種行爲乃是基於異常安全性（exception safety）考量。實現一個「移除最後元素並傳回，而且完全顧及異常安全性」的 pop() 是不可能的（這個問題最早由 Tom Cargill 在 [CargillExceptionSafety] 中討論過），[SutterExceptional] 條款 10 也有討論）。然而，如果拋開可能的危險，我們可以實作出一個「移除最後元素並傳回」的 pop()，但必須宣告一個型別爲 T 的區域變數：

```
template <typename T>
T Stack<T>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop: empty stack");
    }
    T elem = elems.back();       // 保存最後元素的拷貝
    elems.pop_back();            // 移除最後一個元素
    return elem;                 // 傳回先前保存的最後元素
}
```

當 vector 爲空時，對它進行 back() 或 pop_back() 操作會導致未定義行爲。所以我們必須在操作前先檢查 stack 是否爲空。如果 stack 爲空，就拋出一個 std::out_of_range 異常。top() 之中也需進行相同檢查；該函式傳回 stack 的最後一個元素，但不移除之：

```
template <typename T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top: empty stack");
    }
    return elems.back();    // 傳回最後元素的拷貝
}
```

當然，你也可以把任何 class templates 的成員函式實作碼寫在 class 宣告式中，形成一個 inline 函式。例如：

```
template <typename T>
class Stack {
    ...
    void push(T const& elem) {
        elems.push_back(elem);  // 將傳入的元素 elem 附加於尾
    }
    ...
};
```

3.2 使用 Class Template Stack

爲了使用 class template object，你必須明確指出其 template arguments。下面例子說明如何使用 Stack<> class template：

```
// basics/stack1test.cpp

#include <iostream>
#include <string>
#include <cstdlib>
#include "stack1.hpp"

int main()
{
    try {
        Stack<int>         intStack;        // stack of ints
        Stack<std::string>  stringStack;    // stack of strings

        // 操控 int stack
        intStack.push(7);
        std::cout << intStack.top() << std::endl;

        // 操控 string stack
        stringStack.push("hello");
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }

    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE;    // 傳回錯誤狀態碼
    }
}
```

上例宣告了 Stack<int>這樣一個型別，表示在該 class template 中 T 被替換爲 int。於是 intStack 成爲這樣一個 object：內部使用「可容納 int 資料做爲元素」的 vector，並將被呼叫之任何成員函式都「以 int 型別進行具現化」。同樣道理，Stack<std::string> 表示：stringStack 被創建爲這樣一個 object：內部使用「可容納 string 資料做爲元素」的 vector，並將被呼叫之任何成員函式都「以 std::string 型別進行具現化」。

注意，惟有被呼叫到的成員函式，才會被具現化（instantiated）。對 class templates 而言，只有當某個成員函式被使用時，才會進行具現化。無疑地這麼做可以節省時間和空間。另一個好處是，你甚至可以具現化一個 class template，而具現型別並不需要完整支援「class template 內與該型別有關的所有操作」 — 前提是那些操作並未真正被叫用。舉個例子，考慮某個 class，其某些成員函式使用 operator< 對內部元素排序；只要避免呼叫這些函式，就可以以一個「並不支援 operator<」的型別來具現化這個 class template。

本例中的 *default* 建構式、push()函式和 top()函式都同時被 int 和 string 型別加以實現（具現化）。然而 pop()只被 string 具現化（譯註：因爲 pop()只被 stringStack 呼叫）。如果 class template 擁有 static 成員，這些 static 成員會針對每一種被使用的型別完成具現化。

你可以像面對任何基本型別一樣地使用一個具現化後的 class template 型別，前提是你的使用方式合法：

```
void foo (Stack<int> const& s)  // 參數 s 是一個 int stack
{
    Stack<int> istack[10];      // istack 是一個「含有 10 個 int stacks」的 array
    ...
}
```

運用 typedef，你可以更方便地使用 class templates：

```
typedef Stack<int> IntStack;

void foo (IntStack const& s) // 參數 s 是一個 int stack
{
    IntStack istack[10];     // istack 是一個「含有 10 個 int stacks」的 array
    ...
}
```

注意，在 C++ 中，typedef 並不產生新型別，只是爲既有型別產生一個別名（type alias）。因此在以下述句之後：

```
typedef Stack<int> IntStack;
```

IntStack 和 Stack<int> 擁有（代表）相同型別，可以互換使用，可以相互賦值（*assigned*）。

Template arguments 可以是任意型別，例如可以是「指向 float」的指標，或甚至是個 int stack：

```
Stack<float*>       floatPtrStack;    // stack of float pointers
Stack<Stack<int> >  intStackStack;    // stack of stack of ints
```

惟一的條件是：該型別必須支援所有「實際被呼叫到的操作」。

注意，你必須在相鄰兩個右角括號之間插入一些空白符號（像上面那樣），否則就等於使用了 operator>>，那會導致語法錯誤：

```
Stack<Stack<int>> intStackStack;    // 錯誤：此處不允許使用 >>
```

3.3 Class Templates 的特化（Specializations）

你可以針對某些特殊的 template arguments，對一個 class template 進行「特化」。class templates 的特化與 function template 的重載（p.15）類似，使你得以針對某些特定型別進行程式碼優化，或修正某個特定型別在 class template 具現體（instantiation）中的錯誤行爲。然而如果你對一個 class template 進行特化，就必須特化其所有成員函式。雖然你可以特化某個單獨的成員函式，但一旦這麼做，也就不再是特化整個 class template。

欲特化某個 class template，必須以 template<> 開頭宣告此一 class，後面跟著你希望的特化結果。特化型別（specialized type）將作爲 template arguments 並在 class 名稱之後直接寫明：

```
template<>
class Stack<std::string> {
    ...
};
```

對特化體（specializations）而言，每個成員函式都必須像常規的（一般的）成員函式那樣定義，每一個 T 出現處都必須更換爲特化型別（specialized type）：

```
void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem);  // 將傳入的 elem 附加於尾
}
```

下面是一個針對 std::string 型別而特化的 Stack<> 的完整範例：

```
// basics/stack2.hpp

#include <deque>
```

```
#include <string>
#include <stdexcept>
#include "stack1.hpp"

template<>
class Stack<std::string> {
private:
    std::deque<std::string> elems;  // 元素

public:
    void push(std::string const&);   // push 元素
    void pop();                      // pop 元素
    std::string top() const;         // 傳回 stack 最頂端元素
    bool empty() const {             // stack 是否爲空
        return elems.empty();
    }
};

void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem);         // 追加元素
}

void Stack<std::string>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<std::string>::pop(): empty stack");
    }
    elems.pop_back();              // 移除最後一個元素
}

std::string Stack<std::string>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<std::string>::top(): empty stack");
    }
    return elems.back();           // 傳回最後一個元素的拷貝
}
```

此例之中，我們在 stack 內部改用 deque 代替 vector 來管理元素。這麼做並沒有特別的好處，但它示範「一個特化實作碼可以和其 primary template（主模板。譯註：最原始的那一份定義）有相當程度的差異」[8]。

3.4 偏特化（Partial Specialization）

Class templates 可以被偏特化（partial specialized，或稱部份特化、局部特化）。這使你得以在特定情形下使用特殊實作碼，但仍然留給你（使用者）選擇 template parameters 的能力。例如對於下面的 class template：

```
template <typename T1, typename T2>
class MyClass {
    ...
};
```

以下數種形式的偏特化都是合理的：

```
// 偏特化：兩個 template parameter 相同
template <typename T>
class MyClass<T,T> {
    ...
};

// 偏特化：第二個型別爲 int
template <typename T>
class MyClass<T,int> {
    ...
};

// 偏特化：兩個 template parameter 均爲指標型別
template <typename T1, typename T2>
class MyClass<T1*, T2*> {
    ...
};
// 譯註：VC6 並不支援偏特化，未能通過上例。VC7.1/ICL7.1/g++ 3.2 可順利編譯。
```

[8] 事實上，使用 deque 比使用 vector 帶來一個好處：deque 會在元素被移除時釋放記憶體，並在重新申請記憶體時保持原有元素的位址不變。但這對於 strings 來說並沒有特別好處。基於這個原因，最好在原始的 stack class template 中就使用 deque，C++標準程式庫的 `std::stack<>` class 便是如此。

以下例子示範，下列各種宣告式將使用上述哪一個 class template：

```
MyClass<int,float> mif;         // 使用 MyClass<T1,T2>
MyClass<float,float> mff;       // 使用 MyClass<T,T>
MyClass<float,int> mfi;         // 使用 MyClass<T,int>
MyClass<int*,float*> mp;        // 使用 MyClass<T1*,T2*>
```

如果某個宣告式與兩個（或更多）偏特化版本產生同等的匹配程度，這個宣告式便被視爲模稜兩可（歧義）：

```
MyClass<int,int> m;         // 錯誤：同時匹配 MyClass<T,T> 和 MyClass<T,int>
MyClass<int*,int*> m;       // 錯誤：同時匹配 MyClass<T,T> 和 MyClass<T1*,T2*>
```

爲解除上述第二宣告的歧義性，你可以針對「指向相同型別」的指標，提供另一個偏特化版本：

```
template <typename T>
class MyClass<T*,T*> {
    ...
};
```

偏特化的更詳細討論請見 12.4 節, p.200。

3.5 預設模板引數（Default Template Arguments）

你可以針對 class templates 定義其 template parameters 的預設值，這稱爲 default template arguments（預設模板引數）。預設引數值甚至可以引用前一步宣告的 template parameters。例如在 class Stack<> 中，你可以把「用來容納元素」的容器型別定義爲第二個 template parameter，並使用 std::vector<> 作爲其預設值：

```
// basics/stack3.hpp

#include <vector>
#include <stdexcept>

template <typename T, typename CONT = std::vector<T> >
class Stack {
private:
    CONT elems;             // 元素

public:
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;          // 傳回 stack 的頂端元素
```

```
    bool empty() const {    // stack 是否爲空
        return elems.empty();
    }
};

template <typename T, typename CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem);  // 追加元素
}

template <typename T, typename CONT>
void Stack<T,CONT>::pop()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back();       // 移除最後一個元素
}

template <typename T, typename CONT>
T Stack<T,CONT>::top() const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems.back();    // 傳回最後一個元素的拷貝
}
```

注意上述的 template 如今有兩個參數，所以每一個成員函式的定義式中都必須包含這兩個參數：

```
template <typename T, typename CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem);  // 追加元素
}
```

你可以像面對「單一 template parameter」版本那樣地使用這個新版 stack。這時如果你只傳遞一個引數表示元素型別，stack 會使用預設的 vector 來管理其內部元素：

```
template <typename T, typename CONT = std::vector<T> >
class Stack {
private:
    CONT elems; // 元素
...
};
```

當你在程式中宣告一個 Stack object 時，也可以明確指定元素容器的型別：

```
// basics/stack3test.cpp

#include <iostream>
#include <deque>
#include <cstdlib>
#include "stack3.hpp"

int main()
{
    try {
        // stack of ints
        Stack<int> intStack;

        // stack of doubles，其內部使用 std::deque<> 來管理元素
        Stack<double,std::deque<double> > dblStack;
            //譯註：千萬不要宣告爲 Stack<double, std::deque<int> >，
            //      這是自己砸自己的腳，編譯器無法爲你做些什麼。

        // 操控 int stack
        intStack.push(7);
        std::cout << intStack.top() << std::endl;
        intStack.pop();

        // 操控 double stack
        dblStack.push(42.42);
        std::cout << dblStack.top() << std::endl;
        dblStack.pop();
        dblStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE;    // 以錯誤狀態碼離開程式
    }
}
```

那麼，只要像下面這樣做：

```
Stack<double,std::deque<double> >
```

你就宣告了一個 double stack，其內部以 std::deque<> 來管理元素。

3.6 摘要

- 所謂 class template 是「包含一個或多個尚未確定之型別」的 class。
- 你必須將具體型別當作 template arguments 傳入，才能使用 class template。於是該 class template 便以你所指定的那些型別，由編譯器加以具現化並編譯。
- Class templates 之中，只有實際被呼叫的成員函式，才會被具現化。
- 你可以針對某些特定型別，對 class templates 進行特化（specialize）。
- 你可以針對某些特定型別，對 class templates 進行偏特化（partially specialize）。
- 你可以爲 template parameters 定義預設值（稱爲 default template arguments），該預設引數值可以引用前一步定義的 template parameters。

4

Nontype Template Parameters

非型別模板參數

對 function templates 和 class templates 而言，template parameters 並不一定非要是型別（types）不可，它們也可以是常規的（一般的）數值。當你以型別（types）作爲 template parameters 時，程式碼中尚未決定的是型別；當你以一般數值（non-types）作爲 template parameter 時，程式碼中待定的內容便是某些數值。使用這種 template 時必須明確指定數值，程式碼才得以具現化。本章將利用這個特性實作一個新版本的 stack class template。此外我將舉一個例子，展示如何在 function templates 中使用 nontype template paramaters，並討論此技術的一些侷限。

4.1 Nontype Class Template Parameters（非型別類別模板參數）

上一章實作了一個「元素個數可變」的 stack class。與之對比，你也可以實作另一種 stack，透過一個固定大小（fixed-size）的 array 來容納元素。這樣做的好處是不必考慮諸如記憶體管理之類的問題。然而 array 大小的決定是一件比較困難的事：array 愈小則 stack 愈容易滿溢，array 愈大則愈容易造成空間浪費。一個可行的解決辦法是讓使用者指定 array 大小，這個大小也就是 stack 的最大元素個數。

爲了完成以上想法，我們應該把大小值當作一個 template parameter：

```
// basics/stack4.hpp

#include <stdexcept>

template <typename T, int MAXSIZE>
class Stack {
private:
    T elems[MAXSIZE];       // 元素
    int numElems;           // 當前的元素個數

public:
    Stack();                // 建構式
    void push(T const&);    // push 元素
    void pop();             // pop 元素
```

```
    T top() const;                // 傳回 stack 頂端元素
    bool empty() const {          // stack 是否爲空
        return numElems == 0;
    }
    bool full() const {           // stack 是否已滿
        return numElems == MAXSIZE;
    }
};

// 建構式
template <typename T, int MAXSIZE>
Stack<T,MAXSIZE>::Stack ()
    : numElems(0)          // 一開始並無任何元素
{
    // 不做任何事
}

template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::push (T const& elem)
{
    if (numElems == MAXSIZE) {
        throw std::out_of_range("Stack<>::push(): stack is full.");
    }
    elems[numElems] = elem;      // 追加
    ++numElems;                  // 元素總數加 1
}

template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::pop ()
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::pop(): empty stack.");
    }
    --numElems;                  // 元素總數減 1
}
```

```
template <typename T, int MAXSIZE>
T Stack<T,MAXSIZE>::top () const
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::top(): empty stack.");
    }
    return elems[numElems - 1];      // 傳回最後一個元素
}
```

新加入的第二個 template parameter `MAXSIZE` 隸屬 `int` 型別，用來指定「容納 stack 元素」的那個底部 array 的大小：

```
template <typename T, int MAXSIZE>
class Stack {
private:
    T elems[MAXSIZE];      // 元素
    ...
};
```

`push()` 便是使用 `MAXSIZE` 來檢查 stack 是否已滿：

```
template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::push (T const& elem)
{
    if (numElems == MAXSIZE) {
        throw std::out_of_range("Stack<>::push(): stack is full.");
    }
    elems[numElems] = elem;     // 追加
    ++numElems;                 // 元素總數加 1
}
```

使用上述 class template 時，必須同時指定 (1) 元素型別和 (2) stack 元素的最大數量：

```
// basics/stack4test.cpp

#include <iostream>
#include <string>
#include <cstdlib>
#include "stack4.hpp"

int main()
{
```

```
    try {
        Stack<int,20>          int20Stack;      // 最多容納 20 個 int 元素
        Stack<int,40>          int40Stack;      // 最多容納 40 個 int 元素
        Stack<std::string,40>  stringStack;     // 最多容納 40 個 string 元素

        // 操控「最多容納 20 個 int 元素」的那個 stack
        int20Stack.push(7);
        std::cout << int20Stack.top() << std::endl;
        int20Stack.pop();

        // 操控「最多容納 40 個 string 元素」的那個 stack
        stringStack.push("hello");
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE;    // 傳回一個錯誤狀態碼
    }
}
```

注意，每一個被具現化（instantiated）的 class template 都有各自的型別。（譯註：常見的誤會是：上述三個 stacks 隸屬同一型別。這是錯誤觀念。）因此 `int20Stack` 和 `int40Stack` 是兩個不同型別，不能互相進行隱式或顯式轉換，兩者不能換用（彼此取代），也不能互相賦值。

你可以指定 non-type template parameters 的預設值：

```
template <typename T = int, int MAXSIZE = 100>
class Stack {
    ...
};
```

然而從設計角度來看，這樣做並不恰當。Template parameters 的預設值應該符合大多數情況下的要求，然而把 `int` 當做預設元素型別，或指定 stack 最多有 100 個元素，並不符合一個「通用型 stack」的需求。更好的作法是讓使用者指定這兩個參數的值，並在文件中說明它們的意義。

4.2 Nontype Function Template Parameters（非型別函式模板參數）

你也可以爲 function template 定義 nontype parameters。例如下面的 function template 定義了一組函式，可以將參數 x 累加一個值（VAL）後傳回：

```
// basics/addval.hpp

template <typename T, int VAL>
T addValue(T const& x)
{
    return x + VAL;
}
```

當我們需要把「函式」或「某種通用操作」作爲參數傳遞時，這一類函式就很有用。例如使用 STL（Standard Template Library，標準模板庫）時，你可以運用上述 function template 的具現體（instantiation），將某值加到元素集內的每一個元素身上：

```
// (1)
std::transform (source.begin(), source.end(),      // 來源端起止位置
                dest.begin(),                      // 目的端起始位置
                addValue<int,5>);                  // 實際操作
```

最後一個引數將 function template addValue()具現化了，使其操作成爲「將 5 加進一個 int 數值中」。演算法 transform()會對 source 中的所有元素呼叫這個具現體（函式），然後把結果傳入 dest 中。

注意上述例子帶來的一個問題：addValue<int,5> 是個 function template 實體（instance），而我們知道，所謂「function templates 實體」被認爲是命名了一組重載函式集，即使該函式集內可能只有一個函式。根據目前標準，編譯器無法借助「重載函式集」來進行 template parameter 的推導。因此你不得不把 function template argument 強制轉型爲精確型別：

```
// (2)
std::transform (source.begin(), source.end(),          // 來源端起止位置
                dest.begin(),                          // 目的端起始位置
                (int(*)(int const*)) addValue<int,5>); // 操作
//譯註：VC6 支援形式(1)，面對(2)反而無法編譯。VC7.1/ICL7.1/g++ 3.2 都同時支援兩種形式。
```

C++ *Standard* 中已有一個提案要求修正這種行爲，使你不必在這種場合強制轉型（請參考[CoreIssue115]）。在尚未獲得修正之前，爲保證程式的可移植性，你還是得像上面那麼做。

4.3 Nontype Template Parameters 的侷限

注意，nontype template parameters 有某些侷限：通常來說它們只能是常數整數（constant integral values），包括 enum，或是「指向外部聯結（external linkage）之物件」的指標。

以浮點數或 class-type objects 作爲 nontype template parameters 是不可以的：

```
template <double VAT>       // 錯誤：浮點值不能作爲 template parameters
double process (double v)
{
    return v * VAT;
}

template <std::string name> // 錯誤：class objects 不能作爲 template parameters
class MyClass {
    ...
};
```

不允許浮點字面常數（floating-point literals）或簡單的常量浮點運算式（constant floating-point expressions）作爲 template arguments，其實只是歷史因素，並非技術原因。由於並沒有什麼實作上的困難，或許將來 C++ 會支援它，請參考 13.4 節, p.210。

由於字串字面常數（string literal）是一種採用內部聯結（internal linkage）的物件，也就是說不同模組（modules）內的兩個同值的字串字面常數，其實是不同的物件，因此它們也不能被拿來作爲 template arguments：

```
template <char const* name>
class MyClass {
    ...
};

MyClass<"hello"> x; // 錯誤：不能使用字串常量"hello"
```

此外，全域性指標也不能被拿來作爲 template arguments：

```
template <char const* name>
Class MyClass {
    ...
};

char const* s = "hello";

MyClass<s> x;   // 錯誤：s 是「指向內部聯結（internal linkage）物件」的指標
```

但是你可以這麼寫：

```
template <char const* name>
Class MyClass {
    ...
};

extern char const s[] = "hello";

MyClass<s> x;   // OK
```

全域的 char array s 被初始化爲 "hello"，因此 s 是一個外部聯結（external linkage）物件。8.3.3 節, p.109 對此問題有詳細討論，13.4 節, p.209 則討論了這個問題未來的可能變化。

4.4 摘要

- Templates parameters 不限只能是型別（types），也可以是數值（values）。
- 你不能把浮點數、class-type 物件、內部聯結（internal linkage）物件（例如字串字面常數）當作 nontype template parameters 的引數。

5

高階基本技術

Tricky Basics

本章涵蓋實際編程之中層次較高的一些 template 基本知識，包括關鍵字 typename 的另一種用途、將 member function（成員函式）和 nested class（嵌套類別）定義爲 templates、奇特的 template template parameters、零值初始化（zero initialization）、以字串字面常數（string literals）作爲 function templates arguments 的細節問題…等等。有時候這些問題可能涉及較多技巧，但每一位程式員都應該至少對它們有大略的了解。

5.1 關鍵字 typename

關鍵字 typename 是 C++ 標準化過程中被引入的，目的在於向編譯器說明 template 內的某個標識符號是個型別（而不是其他什麼東西）。考慮下面的例子：

```
template <typename T>
class MyClass {
    typename T::SubType * ptr;
    ...
};
```

在這裡，第二個 typename 關鍵字的意思是：SubType 是 class T 內部定義的一個型別，從而 ptr 是一個「指向 T::SubType 型別」的指標。

如果上例沒有使用關鍵字 typename，SubType 會被認爲是 class T 的一個 static 成員，於是被編譯器理解爲一個具體變數或一個物件，導致以下式子：

```
T::SubType * ptr
```

所表達的意義變成：class T 的 static 成員 SubType 與 ptr 相乘。

譯註：VC6/ICL7.1/g++ 3.2 在上述的 MyClass 被具現化之前，都假設 SubType 是個成員型別，因而認爲上述例子正確。VC7.1 認爲此例錯誤（VC7.1 對 typename 要求極嚴）。然而當具現化時，如果 SubType 不是個成員型別，四個編譯器都會報錯。

通常如果某個與 template parameter 相關的名稱是個型別（type）時，你就必須加上關鍵字 typename。更詳細的討論見 9.3.2 節, p.130。

typename 的一個典型應用是在 template 程式碼中使用「STL 容器供應的迭代器（iterators）」：

```
// basics/printcoll.hpp
#include <iostream>

// 列印某個 STL 容器內的所有元素
template <typename T>
void printcoll (T const& coll)
{
    typename T::const_iterator pos;                 // 一個迭代器，用於巡訪 coll
    typename T::const_iterator end(coll.end());     // 末尾位置

    for (pos=coll.begin(); pos!=end; ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

在這個 function template 中，coll 是個 STL 容器，其元素型別爲 T。這裡使用了 STL 容器的迭代器型別（iterator type）巡訪 coll 的所有元素。迭代器型別爲 const_iterator，每一個 STL 容器都宣告有這個型別：

```
class stlcontainer {
    ...
    typedef ... iterator;       // 可讀可寫的迭代器
    typedef ... const_iterator; // 惟讀迭代器
    ...
};
```

使用 template type T 的 const_iterator 時，你必須寫出全名，並在最前面加上關鍵字 typename：

```
typename T::const_iterator pos;
```

.template 構件（construct）

引入關鍵字 typename 之後，人們又發現了一個類似問題。考慮以下程式碼，其中使用標準的 bitset 型別：

```
template<int N>
void printBitset (std::bitset<N> const& bs)
{
    std::cout << bs.template to_string<char,char_traits<char>,allocator<char> >();
}
/* 譯註：你也可以充份運用 bitset 的 member typedef，像這樣：
    std::cout << bs.template to_string<std::string::value_type,
                                       std::string::traits_type,
                                       std::string::allocator_type>(); */
```

此例中的 .template 看起來有些古怪，但是如果沒有它，編譯器無法得知緊跟其後的 "<" 代表的是個 template argument list 的起始，而非一個「小於」符號。注意，只有當位於 "." 之前的構件（construct）取決於某個 template parameter 時，這個問題才會發生。以上例子中，參數 bs 便是取決（受控）於 template parameter N。

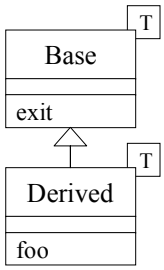
結論是 ".template" 或 "->template" 記號只在 templates 之內才能被使用，而且它們必須緊跟著「與 template parameters 相關」的某物體。細節請見 9.3.3 節, p.132。

5.2 使用 this->

如果 class templates 擁有 base classes，那麼其內出現的成員名稱 x 並非總是等價於 this->x，即使 x 係繼承而來。例如：

```
template <typename T>
class Base {
public:
    void exit();
};

template <typename T>
class Derived : public Base<T> {
public:
    void foo() {
        exit(); // 要不就呼叫外部 exit()，要不就發生編譯錯誤。
    }
};
```



本例在 foo() 內決議（*resolving*）"exit" 符號時，定義於 Base 的 exit() 會被編譯器忽略。因此，你要嘛獲得一個編譯錯誤，要嘛就是喚起一個外部的 exit()。

我們將在 9.4.2 節, p.136 詳細討論這個問題。目前可視爲一個準則：使用與 template 相關的符號時，建議總是以 this-> 或 Base<T>:: 進行修飾。爲避免任何不確定性，可考慮在 templates 內對所有成員存取動作（member accesses）進行以上修飾。

5.3 Member Templates（成員模板）

Class 的成員也可以是 templates：既可以是 nested class templates，也可以是 member function templates。讓我再次使用 Stack<> class templates 來展示這項技術的優點，並示範如何運用這種技術。通常只有當兩個 stacks 型別相同，也就是當兩個 stacks 擁有相同型別的元素時，你才能

對它們相互賦值（*assign*），也就是將某個 stack 整體賦值給另一個。你不能把某種型別的 stack 賦值給另一種型別的 stack，即使這兩種型別之間可以隱式轉型：

```
Stack<int>      intStack1, intStack2;      // stacks for ints
Stack<float>    floatStack;                // stack for floats
...
intStack1 = intStack2;      // OK：兩個 stacks 擁有相同型別
floatStack = intStack1;     // ERROR：兩個 stacks 型別不同
```

default assignment 運算子要求左右兩邊擁有相同型別，而以上情況中，擁有不同型別元素的兩個 stacks，其型別並不相同。

然而，只要把 ***assignment*** 運算子定義爲一個 template，你就可以讓兩個「型別不同，但其元素可隱式轉型」的 stacks 互相賦值。爲完成此事，`Stack<>` 需要這樣的宣告：

```
// basics/stack5decl.cpp

template <typename T>
class Stack {
private:
    std::deque<T> elems;    // 元素
public:
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;          // 傳回 stack 頂端元素
    bool empty() const {    // stack 是否爲空
        return elems.empty();
    }

    // 以「元素型別爲 T2」的 stack 做爲賦值運算的右手端。
    template <typename T2>
    Stack<T>& operator= (Stack<T2> const&);
};
```

對比原先的 `Stack`，這個版本有如下改動：

1. 增加一個 ***assignment*** 運算子，使 `Stack` 可被賦予一個「擁有不同元素型別 `T2`」的 stack。
2. 這個 stack 如今使用 deque 作爲內部容器。這個改動是上述改動的連帶影響（譯註：見下頁說明）。

新增加的 ***assignment*** 運算子實作如下：

```
// basics/stack5assign.hpp

template <typename T>
 template <typename T2>
Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
{
    if ((void*)this == (void*)&op2) {      // 判斷是否賦值給自己
        return *this;
    }

    Stack<T2> tmp(op2);                    // 建立 op2 的一份拷貝
    elems.clear();                         // 移除所有現有元素
    while (!tmp.empty()) {                 // 複製所有元素
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
//譯註：VC6 無法編譯上例中的 noninline operator=。VC7.1/ICL7.1/g++ 3.2 無此問題。
//英文版勘誤：上述的「自我賦值判斷」其實無用，因爲 template member assignment operator
//絕不會被用來在兩個相同型別的 objects 之間賦值。而如果 rhs==this，左右端的兩個 objects
//一定是相同型別。同理，p49,p54 之相同判斷亦屬無用。
```

我們先看看什麼樣的語法可以定義一個 member template。在擁有 template parameter `T` 的 template 中定義一個內層的（inner）template parameter `T2`：

```
template <typename T>
 template <typename T2>
...
```

實作這個運算子（成員函式）時，你可能希望取得「賦值符號右側之 `op2` stack」的所有必要資料，但是這個 stack 的型別和目前（此身）型別不同（是的，如果你以兩個不同的型別具現化同一個 class template，你會得到兩個不同的型別），所以只能通過 public 介面來得到那些資料。因此惟一能夠取得 `op2` 資料的辦法就是呼叫其 `top()`函式。你必須經由 `top()`取得 `op2` 的所有資料，而這必須借助 `op2` 的一份拷貝來實現：每取得一筆資料，就運用 `pop()`把該資料從 `op2` 的拷貝中移除。由於 `top()`傳回的是 stack 之中最後（最晚）被推入的元素，所以我們需要反方向把元素安插回去。基於這種需求，這裡使用了 deque，它提供 `push_front()`操作，可以把一個元素安插到容器最前面。

有了這個 member template，你就可以把一個 `int` stack 賦值（*assign*）給一個 `float` stack：

```
Stack<int>     intStack1, intStack2;      //stack for ints
Stack<float>   floatStack;                //stack for floats
...
floatStack = intStack1;    // OK：兩個 stacks 型別不同，但 int 可轉型爲 float。
```

當然，這個賦值動作並不會改動 stack 和其元素的型別。賦值完成後，floatStack 的元素型別還是 float，而 top() 仍然傳回 float 值。

也許你會認爲，這麼做會使得型別檢查失效，因爲你甚至可以把任意型別的元素賦值給另一個 stack。然而事實並非如此。必要的型別檢查會在「來源端 stack」的元素（拷貝）被安插到「目的端 stack」時進行：

```
elems.push_front(tmp.top());
```

如果你將一個 string stack 賦值給一個 float stack，以上述句編譯時就會發生錯誤：「elems.push_front()無法接受 tmp.top()的回傳型別」。具體的錯誤訊息因編譯器而異，但含義類似。

```
Stack<std::string> stringStack;      // stack of strings
Stack<float>       floatStack;       // stack of floats
...
floatStack = stringStack;            // 錯誤：std::string 無法轉型爲 float
```

注意，前述的 template *assignment* 運算子並不取代 *default assignment* 運算子。如果你在相同型別的 stack 之間賦值，編譯器還是會採用 *default assignment* 運算子。

和先前一樣，你可以把內部容器的型別也參數化：

```
// basics/stack6decl.hpp

template <typename T, typename CONT = std::deque<T> >
class Stack {
private:
    CONT elems;             // 元素

public:
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;          // 傳回 stack 的頂端元素
    bool empty() const {    // stack 是否爲空
        return elems.empty();
    }

    // 以元素型別爲 T2 的 stack 賦值
    template <typename T2, typename CONT2>
    Stack<T,CONT>& operator= (Stack<T2,CONT2> const&);
};
```

此時的 template *assignment* 運算子可實作如下：

```
// basics/stack6assign.hpp

template <typename T, typename CONT>
 template <typename T2, typename CONT2>
Stack<T,CONT>&
Stack<T,CONT>::operator= (Stack<T2,CONT2> const& op2)
{
    if ((void*)this == (void*)&op2) {      // 判斷是否賦值給自己
        return *this;                      // 譯註：請見 p47 之「英文版勘誤」
    }

    Stack<T2,CONT2> tmp(op2);              // 創建 op2 的一份拷貝
    elems.clear();                         // 移除所有現有元素
    while (!tmp.empty()) {                 // 複製所有元素
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
//譯註：VC6 無法編譯上例的 noninline operator=。VC7.1/ICL7.1/g++ 3.2 無此問題。
```

記住，對 class templates 而言，只有「實際被呼叫的成員函式」才會被具現化。因此如果你不至於令不同（元素）型別的 stacks 彼此賦值，那麼甚至可以拿 vector 當作內部元素的容器（譯註：而先前的程式碼完全不必改動）：

```
// stack for ints，使用 vector 作爲內部容器
Stack<int,std::vector<int> > vStack;
...
vStack.push(42);
vStack.push(7);
std::cout << vStack.top() << std::endl;
```

由於 template *assignment* 運算子並未被用到，編譯器不會產生任何錯誤訊息抱怨說「內部容器無法支援 push_front()操作」。

以上例子的完整實作全部包含於以 stack6 開頭的檔案中，位於子目錄 basics 之下[9]。

[9] 如果你的編譯器無法編譯這些例子，請不要太驚訝。在這些例子中我們幾乎用上了 template 的所有重要特性。你最好選一個比較符合 *C++ Standard* 的編譯器來編譯這些例子。

5.4 Template Template Parameters（雙重模板參數）

譯註：VC6 不支援 template template parameter。VC7.1/ICL7.1/g++ 3.2 支援此一特性。

一個 template parameter 本身也可以是個 class template，這一點非常有用。我們將再次以 stack class template 說明這種用法。

爲了使用其他型別的元素容器，stack class 使用者必須兩次指定元素型別：一次是元素型別本身，另一次是容器型別：

```
Stack<int,std::vector<int> > vStack;    // int stack，以 vector 爲容器
```

如果使用 template template parameter，就可以只指明元素型別，無須再指定容器型別：

```
Stack<int,std::vector> vStack;          // int stack，以 vector 爲容器
```

爲了實現這種特性，你必須把第二個 template parameter 宣告爲 template template parameter。原則上程式碼可以寫爲[10]：

```
// basics/stack7decl.cpp

template <typename T,
          template <typename ELEM> class CONT = std::deque >
class Stack {
private:
    CONT<T> elems;          // 元素
public:
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;          // 傳回 stack 頂端元素
    bool empty() const {    // stack 是否爲空
        return elems.empty();
    }
};
```

與先前的 stack 差別在於，第二個 template parameter 被宣告爲一個 class template：

```
template <typename ELEM> class CONT
```

其預設值則由 `std::deque<T>` 變更爲 `std::deque`。這個參數必須是個 class template，並以第一參數的型別完成具現化：

```
CONT<T> elems;
```

[10] 這個版本有個問題，我們會在稍後討論。該問題只會影響預設值的指定（`= std::deque`）。目前只是暫以這個例子來說明 template template parameter 的特性。

本例「以第一個 template parameter 對第二個 template parameter 進行具現化」只是基於例子本身的需要。實際運用時你可以使用 class template 內的任何型別來具現化一個 template template parameter。

和往常一樣，你也可以改用關鍵字 class 而不使用關鍵字 typename 來宣告一個 template parameter；但 CONT 定義的是一個 class 型別，因此你必須使用關鍵字 class 來宣告它。所以，下面的程式碼是正確的：

```
template <typename T,
          template <typename ELEM> class CONT = std::deque >    //OK
class Stack {
    ...
};
```

下面的程式碼則是錯誤的：

```
template <typename T,
          template <typename ELEM> typename CONT = std::deque > //ERROR
class Stack {
    ...
};
```

由於 template template parameter 中的 template parameter 實際並未用到，因此你可以省略其名稱：

```
template <typename T,
          template <typename> class CONT = std::deque >
class Stack {
...
};
```

所有成員函式也必須按此原則修改：必須指定其第二個 template parameter 爲 template template parameter。同樣的規則也適用於成員函式的實作部份。例如成員函式 push() 應該實作如下：

```
template <typename T, template <typename> class CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem);  // 追加元素
}
```

另請注意，function templates 不允許擁有 template template parameters。

Template Template Argument 的比配（matching）

如果你試圖使用上述新版 Stack，編譯器會報告一個錯誤：預設值 std::deque 不符合 template template parameter CONT 的要求。問題出在 template template argument 不但必須是個 template，

而且其參數必須嚴格比配它所替換之 template template parameter 的參數。Template template argument 的預設值不被考慮，因此如果不給出擁有預設值的引數值時，編譯器會認爲匹配失敗。

本例的問題在於：標準程式庫中的 std::deque template 要求不只一個參數。第二參數是個配置器（allocator），它雖有預設值，但當它被用來匹配 CONT 的參數時，其預設值被編譯器強行忽略了。

辦法還是有的。我們可以重寫 class 宣告式，使 CONT 參數要求一個「帶兩個參數」的容器：

```
template <typename T,
          template <typename ELEM,
                    typename ALLOC = std::allocator<ELEM> >
                    class CONT = std::deque>
class Stack {
private:
    CONT<T> elems;         // 元素
    ...
};
```

譯註：g++ 3.2 支援「template template parameters 寬鬆匹配規則」（見 13.5 節, p.211），因此即使不進行以上修改也能通過編譯。

譯註：即使進行以上修改，VC7.1 仍然無法編譯。解決辦法是將 CONT<T> 改爲 CONT<T,std::allocator<T> >。

由於 ALLOC 並未在程式碼中用到，因此你也可以把它省略掉。

譯註：如果略去 ALLOC，VC7.1 會認爲 CONT 與 std::deque 型別不相容。

Stack template 的最終版本如下。此一版本支援對「不同元素型別」之 stacks 的彼此賦值動作：

```
// basics/stack8.hpp

#ifndef STACK_HPP
#define STACK_HPP

#include <deque>
#include <stdexcept>
#include <memory>

template <typename T,
          template <typename ELEM,
                    typename = std::allocator<ELEM> >
                    class CONT = std::deque>
class Stack {
private:
```

```
    CONT<T> elems;          // 元素

public:
    void push(T const&);    // push 元素

    void pop();             // pop 元素
    T top() const;          // 傳回 stack 的頂端元素
    bool empty() const {    // stack 是否爲空
        return elems.empty();
    }

    // 賦予一個「元素型別爲 T2」的 stack
    template<typename T2,
             template<typename ELEM2,
                      typename = std::allocator<ELEM2>
                     > class CONT2>
    Stack<T,CONT>& operator= (Stack<T2,CONT2> const&);
};

template <typename T, template <typename,typename> class CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem);    // 追加元素
}

template<typename T, template <typename,typename> class CONT>
void Stack<T,CONT>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back();         // 移除最後一個元素
}

template <typename T, template <typename,typename> class CONT>
T Stack<T,CONT>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems.back();      // 傳回最後一個元素的拷貝
}
```

```
template <typename T, template <typename,typename> class CONT>
  template <typename T2, template <typename,typename> class CONT2>
Stack<T,CONT>&
Stack<T,CONT>::operator= (Stack<T2,CONT2> const& op2)
{
    if ((void*)this == (void*)&op2) {     // 是否賦值給自己
        return *this;                     // 譯註：請見 p47 之「英文版勘誤」
    }

    Stack<T2,CONT2> tmp(op2);             // 創建 assigned stack 的一份拷貝

    elems.clear();                        // 移除所有元素
    while (!tmp.empty()) {                // 複製所有元素
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}

#endif // STACK_HPP
```

下面程式使用了上述最終版本的 stack：

```
// basics/stack8test.cpp

#include <iostream>
#include <string>
#include <cstdlib>
#include <vector>
#include "stack8.hpp"

int main()
{
    try {
        Stack<int>   intStack;          // stack of ints
        Stack<float> floatStack;        // stack of floats

        // 操控 int stack
        intStack.push(42);
        intStack.push(7);
```

```
        // 操控 float stack
        floatStack.push(7.7);

        // 賦予一個「不同型別」的 stack
        floatStack = intStack;

        // 列印 float stack
        std::cout << floatStack.top() << std::endl;
        floatStack.pop();
        std::cout << floatStack.top() << std::endl;
        floatStack.pop();
        std::cout << floatStack.top() << std::endl;
        floatStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
    }

    // int stack，以 vector 爲其內部容器
    Stack<int,std::vector> vStack;
    // ...
    vStack.push(42);
    vStack.push(7);
    std::cout << vStack.top() << std::endl;
    vStack.pop();
}
```

程式運行的輸出結果爲：

```
7
42
Exception: Stack<>::top(): empty stack
7
```

注意，template template parameter 是極晚近才加入的 C++ 特性，因此上面這個程式可作爲一個極佳工具，用來評估你的編譯器對 template 特性的支援程度。

8.2.3 節, p.102 和 15.1.6 節, p.259 之中對於 template template parameter 有更詳盡的討論。

5.5 零值初始化（Zero Initialization）

對於基本型別如 int、double、pointer type（指標型別）來說，並沒有一個 *default* 建構式將它們初始化爲有意義的值。任何一個未初始化的區域變數（local variable），其值都是未定義的：

```
void foo()
{
    int x;          // x 的值未有定義
    int* ptr;       // ptr 指向某處(而不是哪兒都不指向)
}
```

你可能在 template 程式碼中宣告某個變數，並且想令這個變數被初始化爲其預設值；但是當變數是個內建型別（built-in type）時，你無法確保它被正確初始化：

```
template <typename T>
void foo()
{
    T x;    // 如果 T 是內建型別，則 x 值未有定義
}
```

爲解決這個問題，你可以在宣告內建型別的變數時，明確呼叫其 *default* 建構式，使其值爲零（對 bool 型別而言則是 false）。也就是說 int() 導致 0 值。這樣一來你就可以確保內建型別的變數有正確初值：

```
template <typename T>
void foo()
{
    T x = T();  // 如果 T 是內建型別，則 x 被初始化爲 0 或 false
}
```

Class template 的各個成員，其型別有可能被參數化。爲確保初始化這樣的成員，你必須定義一個建構式，在其「成員初值列」（member initialization list）中對每個成員進行初始化：

```
template <typename T>
class MyClass {
private:
    T x;
public:
    MyClass() : x() {   // 這麼做可以確保：即使 T 爲內建型別，x 也能被初始化。
    }
    ...
};
```

5.6 以字串字面常數（String Literals）作爲 Function Template Arguments

以 *by reference* 傳遞方式將「字串字面常數」（string literals）傳遞給 function template parameters 時，有可能遇上意想不到的錯誤：

```
// basics/max5.cpp

#include <string>

// 注意：使用 reference parameters
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

int main()
{
    std::string s;

    ::max("apple","peach");      // OK：型別相同
    ::max("apple","tomato");     // ERROR：型別不同
    ::max("apple",s);            // ERROR：型別不同
}
```

問題出在這幾個字串字面常數（string literals）的長度不同，因而其底層的 array 型別也不同。換句話說 `"apple"` 和 `"peach"` 的 array 型別都是 `char const[6]`，而 `"tomato"` 的 array 型別是 `char const[7]`。上述呼叫只有第一個合法，因爲兩個參數具有相同型別；然而如果你使用 *by value* 傳遞方式，就可以傳遞不同型別的字串字面常數（string literals），其對應的 array 大小不同：

```
// basics/max6.hpp

#include <string>

// 注意：使用 non-reference parameters
template <typename T>
inline T max(T a, T b)
{
    return a < b ? b : a;
}
```

```
int main()
{
    std::string s;

    ::max("apple","peach");      // OK：型別相同
    ::max("apple","tomato");     // OK：退化爲相同型別
    ::max("apple",s);            // 錯誤：型別不同
}
```

這種方式之所以可行，因爲在引數推導過程中，惟有當參數並不是一個 reference 型別時，「array 轉爲 pointer」的轉型動作（常被稱爲退化, *decay*）才會發生。這個規則可藉以下例子加以說明：

```
// basics/refnonref.cpp

#include <typeinfo>
#include <iostream>

template <typename T>
void ref (T const& x)
{
    std::cout << "x in ref(T const&): "
              << typeid(x).name() << std::endl;
}

template <typename T>
void nonref (T x)
{
    std::cout << "x in nonref(T):     "
              << typeid(x).name() << std::endl;
}

int main()
{
    ref("hello");
    nonref("hello");
}
```

在這個例子中，同一個字串字面常數（string literal）分別被傳遞給兩個 function templates，其一宣告參數爲 reference，其二宣告參數爲 non-reference。兩個函式都使用 `typeid` 運算子列印其具現化後的參數型別。

`typeid` 運算子會傳回一個左值（lvalue），其型別爲 `std::type_info`，其內封裝了「`typeid` 運算子所接收之算式（expression）」的型別表述（representation）。`std::type_info` 的成員函式 `name()` 把這份「型別表述」以易讀的字串形式傳回給呼叫者。*C++ Standard* 並不要求 `name()` 傳回有意義的內容，但是在良好的 C++ 編譯器中，它會傳回一個字串內含「`typeid` 運算子的參數型別」的完好描述。在某些編譯器實作品中，這個字串可能以重排（*mangled*）形式出現（譯註：見 p.162 注釋），但也有些反重排工具（*demangler*）可以把它調整回人類可讀的形式。例如上面程式的輸出可能如下：

```
x in ref(T const&): char [6]
x in nonref(T):     const char *
```

如果你曾經在程式中把「`char` array」和「`char` pointer」混用，你可能被這個令人驚奇的問題搞得頭昏腦脹[11]。不幸的是，沒有一個普遍適用的辦法可以解決這個問題。根據所處情況的不同，你可以：

- 以 *by value* 傳遞方式代替 *by reference* 傳遞方式。然而這會帶來不必要的拷貝。
- 分別對 *by value* 傳遞方式和 *by reference* 傳遞方式進行重載。然而這會帶來模稜兩可問題（ambiguities, 歧義性），請參考 B.2.2 節, p.492。
- 以具體型別（例如 `std::string`）重載之
- 以 array 型別重載之。例如：

```
template <typename T, int N, int M>
T const* max (T const (&a)[N], T const (&b)[M])
{
   return a < b ? b : a;
}
```

- 強迫使用者進行顯式轉型（explicit conversions）

本例之中，最好的方式是對 `string` 重載（請參考 2.4 節, p.16）。這麼做有其必要。如果不這麼做，對兩個字串字面常數（string literals）呼叫 `max()` 是合法的，但 `max()` 會以 `operator<` 比較兩個指標的大小，而所比較的其實是指標的位址，不是兩個字串的字面值。這也是爲什麼使用 `std::string` 比使用 C-style 字串更好的原因之一。

本書 11.1 節, p.168 對這個問題有詳盡的討論。

---

[11] 事實上，這就是爲什麼最初的 C++ 標準程式庫不能建立「由兩個字串字面常數構成」的 pair（參見 [Standard98]）：

```
std::make_pair("key","value");  // 根據[Standard98]，此句錯誤。
```

此問題在 *C++ Standard* 第一次技術勘誤中得到修正。現在（新的）`make_pair` 使用 *by value* 形式，而不使用 *by refernece* 形式。

5.7 摘要

- 當你要操作一個取決於（受控於）template parameter 的型別名稱時，應該在其前面冠以關鍵字 `typename`。
- 嵌套類別（nested classes）和成員函式（member functions）也可以是 templates。應用之一是，你可以對「不同型別但彼此可隱式轉型」的兩個 template classes 互相操作，而型別檢驗（type checking）仍然起作用。
- *assignment*（賦值）運算子的 template 版本並不會取代 *default assignment* 運算子。
- 你可以把 class templates 作爲 template parameters 使用，稱爲 template template parameters。
- Template template arguments 必須完全匹配其對應參數。預設的 template arguments 會被編譯器忽略，要特別小心。
- 當你具現化（*instantiated*）一個隸屬內建型別（built-in type）的變數時，如果打算爲它設定初值，可明確呼叫其 *default* 建構式。
- 只有當你以 *by value* 方式使用字串字面常數（string literals）時，字串底部的 array 才會被轉型（退化）爲一個字元指標（也就是發生 "array-to-pointer" 轉換）。

6

實際運用 Templates

Using Templates in Practice

和一般程式碼相比，template 程式碼有些不同。某種程度而言，templates 處於 macros（巨集）和一般（non-template）宣告之間。雖然這麼說恐怕是過於簡化了，但這種說法頗適用於我們撰寫演算法和資料結構時所使用的 templates，也適用於我們日常後勤處理時用以表達和分析程式所使用的 templates 身上。

本章將探討實際編程可能碰到的部份問題，然而並不試圖探尋這些問題背後的技術細節。技術細節將在第 10 章討論。爲了讓討論更簡單些，這裡假設 C++ 編譯系統由傳統的「編譯器＋聯結器」組成（事實上此種結構以外的 C++ 編譯系統也相當少見）。

6.1 置入式模型（Inclusion Model）

有很多種方法可以組織你的 template 程式碼。本節介紹截至本書完稿爲止最常見的一種方法：置入式模型。

6.1.1 聯結錯誤（Linker Errors）

大多數 C/C++ 程式員大致上都按照以下方式來組織他們的 non-template 程式碼：

- Classes 和其他型別被全體放置於表頭檔（header files）。通常表頭檔的後綴名稱（副檔名）爲 .hpp（或 .H, .h, .hh, .hxx 等等）。
- 全域變數和 non-inline 函式只在表頭檔中置入宣告式，定義式則置於 .C 檔。這裡的 .C 檔案是個統稱，通常其後綴名稱（副檔名）爲 .cpp（或 .C, .c, .cc, .cxx 等等）。

這種方式運作良好，程式能夠輕易找到各個型別的定義，並避免一個變數或函式被多次重複定義，於是聯結器（linker）可以正常工作。

如果牢記上述規則，很多 template 初學者就會犯一個常見錯誤。讓我以下面這個例子加以說明。就像組織一般 non-template 程式碼一樣，我們把 template 宣告於某個表頭檔：

```
// basics/myfirst.hpp

#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// template 宣告式
template <typename T>
void print_typeof (T const&);

#endif // MYFIRST_HPP
```

`print_typeof()`是一個簡單的輔助函式，它列印參數的型別資訊（type information）。函式的實作碼被置於一個 .C 檔案中：

```
// basics/myfirst.cpp

#include <iostream>
#include <typeinfo>
#include "myfirst.hpp"

// template 的實作碼/定義式
template <typename T>
void print_typeof (T const& x)
{
    std::cout << typeid(x).name() << std::endl;
}
```

這個例子使用 `typeid` 運算子，把參數的型別以字串形式列印出來（見 5.6 節, p.58）。

此後，我們又在另外一個 .C 檔案使用這個 template，並將 template 宣告式（的所在檔案）以 `#include` 包含進來：

```
// basics/myfirstmain.cpp

#include "myfirst.hpp"

// 使用含入之 template
int main()
{
```

```
    double ice = 3.0;
    print_typeof(ice); // 以 double 值呼叫 function template
}
```

大多數 C++ 編譯器都可以正確編譯上述程式碼，然而大多數聯結器會報告一個錯誤，表示無法找到 `print_typeof()`函式的定義。

錯誤的原因在於，function template `print_typeof()`的定義並沒有被具現化。爲了具現化一個 template，編譯器必須知道「以哪一份定義式」以及「以哪些 template arguments」對它具現化。不幸的是先前這個例子中，這兩項資訊被分置於兩個分開編譯的檔案。因此當編譯器看到對 `print_typeof()`的呼叫時，看不到其定義，無法以 `double` 型別來具現化 `print_typeof()`。於是編譯器假設這個 template 的定義位於其他某處，因而只生成一個對該定義的 reference，並將這個 reference 所指的定義式留給聯結器去決議（*resolve*）。另一方面，當編譯器處理 `myfirst.cpp` 時，它又察覺不到該以哪個引數型別來具現化其定義式。

6.1.2 把 Templates 放進表頭檔（Header Files）

就像處理 macro（巨集）和 inline 函式一樣，解決上面問題的常見辦法是：把 template 定義式放到其宣告式所在的表頭檔中。面對前例，我們可以在 `myfirst.hpp` 最後一行加入：

```
#include "myfirst.cpp"
```

也可以在用到該 template 的每一個 .C 檔案中 `#include "myfirst.cpp"`。第三種作法是完全丟開 `myfirst.cpp`，把宣告和定義全部放進 `myfirst.hpp`：

```
// basics/myfirst2.hpp

#ifndef MYFIRST_HPP
#define MYFIRST_HPP

#include <iostream>
#include <typeinfo>

// template 的宣告式
template <typename T>
void print_typeof (T const&);

// template 的實作碼/定義式
template <typename T>
void print_typeof (T const& x)
```

```
{
    std::cout << typeid(x).name() << std::endl;
}

#endif // MYFIRST_HPP
```

這種 templates 組織法稱爲「置入式模型」（inclusion model）。現在你會發現，上述程式可被正確編譯、聯結、執行。

有一些需要注意的地方。最該引起注意的是：「含入 `myfirst.hpp`」的代價相當大。此處所謂代價，不僅是指 template 定義式的體積增加，也包括含入 `myfirst.hpp` 時一併含入的其他（由 `myfirst.hpp` 含入的）檔案，本例是 `<iostream>` 和 `<typeinfo>`。你會發現，這可能導致成千上萬行程式碼被含進來，因爲諸如 `<iostream>` 這樣的檔案也是以此種方式來定義 templates。

這是一個很實際的問題。這顯然會在編譯大型程式時顯著增加編譯時間。稍後數節將試圖尋找解決此問題的一些辦法。很多現實世界中的程式會吃掉你數小時的編譯+聯結時間（關於這個我們有多次經驗，編譯和聯結一個程式，竟用掉悠悠數天工夫）。

儘管存在這個問題，我們仍然強烈推薦：只要可能，你應該使用置入式模型（inclusion model）來組織你的 template 程式碼。雖然稍後還將審查另兩種辦法，但那些辦法所帶來工程缺陷看起來遠比時間佔用問題更嚴重得多（不過它們也可能帶來軟體工程之外的其他好處）。

另一個關於置入式模型的考量比較微妙。與 inline 函式及 macros（巨集）明顯不同的是，non-inline function templates 並不在呼叫端被展開。每當它們被具現化一次，編譯器便從頭創建一份函式拷貝。由於這個過程完全自動化，編譯器可能最終在兩個不同的檔案中創建出同一個 templates 具現體的兩份拷貝，而某些聯結器會因爲看到同一函式的兩份定義而報錯。理論上這不該我們操心，這應該是編譯系統的事。實際工作中大多數時候都能夠運作良好，無需我們操心。然而對於「可能自行建立程式庫」的大型專案而言，問題偶爾也可能冒出來。第 10 章會討論具現化的細節，你的 C++ 編譯器文件往往也會對此有所說明，這些都有助於解決問題。

最後要指出的是，本節例中對於 function templates 所談的內容，也適用於 class templates 的成員函式和 static 成員變數。當然，也同樣適用於 member function templates。

6.2 顯式具現化（Explicit Instantiation）

置入式模型（inclusion model）保證所有用到的 templates 都能被具現化。之所以有這層保障，是因爲 C++ 編譯器會在 templates 被使用時自動具現它。*C++ Standard* 另提供一種方式，使你得以手動將 templates 具現化。這種方式稱爲「顯式（明確）具現化指令」（explicit instantiation directive）。

6.2.1 顯式具現化（Explicit Instantiation）示例

爲說明「顯式具現化」概念，讓我們回頭看看那個導致聯結錯誤的例子（p.62）。爲避免聯結錯誤，我們可以爲程式添加如下的一個檔案：

```
// basics/myfirstinst.cpp

#include "myfirst.cpp"

// 明確以 double 型別將 print_typeof() 具現化
template void print_typeof<double>(double const&);
```

這個「顯式具現化指令」由兩部份組成：先是關鍵字 `template`，然後是一個「template parameters 已被完全替換」後的函式宣告。本例是對函式做顯式（明確）具現化動作，我們也可以運用相同手法對一個成員函式或一個 static 成員變數做顯式（明確）具現化動作。例如：

```
// 明確地以 int 型別對 MyClass<> 的建構式進行具現化動作
template MyClass<int>::MyClass();

// 明確地以 int 型別對 function template max() 進行具現化動作
template int const& max (int const&, int const&); //譯註：可參考 p.9
```

你也可以顯式（明確）具現化一個 class template。這是一種簡便做法，相當於要求編譯器具現化該 class template 的所有「可被具現化」成員，但不包括先前已被特化（specialized）或已被具現化的成員：

```
// 明確地以 int 型別對 Stack<> class 進行具現化動作
template class Stack<int>;

// 明確地以 string 型別對 Stack<> 的一部份成員進行具現化動作
template Stack<std::string>::Stack();
template void Stack<std::string>::push(std::string const&);
template std::string Stack<std::string>::top() const;

// 錯誤：不能對「已被顯式（明確）具現化的 class」的某個成員再次具現化
// 譯註：Stack<int> 已在本例第一行被顯式（明確）具現化。
template Stack<int>::Stack();
```

在程式之中，每一個不同物體（distinct entity）最多只能有一份顯式具現體（explicit instantiation）。換句話說你可以明確具現出 `print_typeof<int>` 和 `print_typeof<double>`，但程式中每一條顯式具現化指令（explicit instantiation directive）只能出現一次。如果不遵守這個規則，往往引發聯結錯誤，聯結器會告訴你：具現體（instantiated entities）被重複定義了。

手動進行具現化，有一個明顯缺點：我們必須非常仔細地搞清楚應該具現化哪些物體。在大型專案中，這對程式員來說是一個極重的負擔；基於這個原因，我們並不推薦這種用法。我們曾經在一些實際專案開始時低估了這個負擔的嚴重性。專案日趨複雜後，我們都爲當初的錯誤決定懊悔不已。

然而「顯式（明確、手動）具現化」也有一些好處，畢竟具現化動作因而得以根據程式的實際需求進行最佳調整。很明顯，這麼做得以避免含入巨大的表頭檔。Template 定義式的源碼得以隱藏，不過也因此客戶端無法產生更多具現體。最後要注意的一點是，控制 template 被具現於某個精確位置（我是指目的檔, object file）有時很有用，而「自動具現化」不可能做到這一點（細節請見第 10 章）。

圖 6.1 template 的宣告式與定義式分離

6.2.2 結合置入式模型（Includsion model）和顯式具現化（Explicit Instantiation）

爲了進一步討論究竟該使用「置入式模型」或使用「顯式具現化」，我們將 templates 的宣告式和定義式分別置於不同的兩個檔案。通常我們會令這兩個檔案的副檔名看起來像表頭檔（意欲被含入），這種作法是聰明的。於是我們把 `myfirst.cpp` 易名爲 `myfirstdef.hpp`，並在檔案首尾加入「防止重複含入」的防衛式預處理指令（所謂 guard preprocessor）。圖 6.1 展示這樣一個 `Stack<>` class template。

現在，如果我們想要使用置入式模型，可以簡單地將定義式所在的表頭檔 `stackdef.hpp` 包含進來。如果我們想要使用顯式（明確）具現化，可以將宣告式所在的表頭檔 `stack.hpp` 包含進來，並提供一個 .C 檔，其中有必要的顯式具現化指令（見圖 6.2）。

stacktest1.cpp:

```
#include "stack.hpp"
#include <iostream>
#include <string>

int main()
{
  Stack<int> intStack;
  intStack.push(42);
  std::cout << intStack.top() << std::endl;
  intStack.pop();

  Stack<std::string>  stringStack;
  stringStack.push("hello");
  std::cout << stringStack.top() << std::endl;
}
```

stack_inst.cpp:

```
#include "stackdef.hpp"
#include <string>

// instantiate class Stack<> for int
template class Stack<int>;

// instantiate some member functions of Stack<> for strings
template Stack<std::string>::Stack();
template void Stack<std::string>::push(std::string const&);
template std::string Stack<std::string>::top() const;
```

圖 6.2　搭配兩個 template 表頭檔，完成顯式（明確）具現化

6.3 分離式模型（Separation Model）

前面講述的兩種作法都可以有效解決問題，而且完全符合 *C++ Standard*。但是 *C++ Standard* 還提供了另一種機制，可以匯出（*export*）一個 template。這種機制稱爲 C++ template 的分離式模型（separation model）。

6.3.1 關鍵字 export

原則上，export 的使用相當容易：將 template 定義於某檔案中，並將其定義式及其所有「非定義宣告」（nondefinition declarations）加上關鍵字 export。以先前出現的例子而言，這會導致如下的 function template 宣告：

```
// basics/myfirst3.hpp

#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// template 宣告
export
template <typename T>
void print_typeof (T const&);

#endif // MYFIRST_HPP
```

exported templates 可被直接拿來使用，不需現場看到 templates 定義檔。換言之，template 的使用和定義可分隔於兩個不同的編譯單元。上面的例子中，檔案 myfirst.hpp 如今只含入 class template 的成員函式宣告，這樣就足以使用這些成員函式。原先的程式碼會引發聯結錯誤，新版本可以順利聯結。對比兩個版本，我們只是添加了關鍵字 export。

在一個預處理檔（preprocessed file）內（亦即在一個編譯單元內），只需將 template 的第一個宣告加上關鍵字 export；此後再次宣告或定義，都會自動加入相同屬性，也就是說均會被匯出（*exported*）。這就是 myfirst.cpp 不需修改的原因 — 因爲 template 在表頭檔中被宣告爲 export，因此 myfirst.cpp 內的 template 定義都會被匯出（*exported*）。不過，從另一方面來說，如果你願意在 template 定義式中加入（其實多餘的）關鍵字 export 也非常好，可提高程式碼可讀性。

關鍵字 export 適用於 function templates、class templates 成員函式、member function templates、class templates 的 static 成員變數。關鍵字 export 也可以被用在 class template 宣告式中，其意義是「匯出所有可被匯出的成員」，但 class template 本身並不會被匯出（其定義仍然只在表頭檔中有效）。你也可以一如往常地隱式或顯式定義一些 inline 成員函式，但它們不能被匯出：

```
export template <typename T>
class MyClass {
public:
    void memfun1();     // 會被匯出
    void memfun2() {    // 不會被匯出，因爲它暗自成爲 inline
        ...
    }
    void memfun3();     // 不會被匯出，因爲它明確爲 inline（見下方定義式）
    ...
};

template <typename T>
inline void MyClass<T>::memfun3 ()
{
    ...
}
```

請注意，關鍵字 export 不能和關鍵字 inline 合用，而且關鍵字 export 應該總是寫在關鍵字 template 之前。以下程式碼是不合法的：

```
template <typename T>
class Invalid {
public:
    export void wrong(T);            // 錯誤：export 應該在 template 之前
};

export template <typename T>         // 錯誤：export 與 inline 合用
inline void Invalid<T>::wrong<T>
{
}

export template <typename T>
inline T const& max (T const& a, T const& b)
{                                    // 錯誤：export 與 inline 合用
    return a < b ? b : a;
}
```

6.3.2 分離式模型（Separation Model）的侷限

讀到這裡，你可能會奇怪：既然 export 提供了這麼多好處，爲什麼我們仍然提倡使用「置入式模型」呢？事實是，使用 export 會帶來一些問題。

首先，即使在 C++ *Standard* 已經制定四年的今天（譯註：原書寫於 2002 年），仍然只有一家公司實作出 export[12]。因此 export 並沒有像其他 C++ 特性那樣地被廣爲運用，這也意味在目前的時間點上，所有關於 export 的知識或經驗都頗爲有限；而我們關於此特性的討論也都十分謹慎和保守。也許將來某一天，我們的疑惑會被解開（稍後將告訴你如何未雨綢繆）。

其次，儘管 export 看起來近乎魔術，然而它確確實實存在。具現化過程最終必須處理兩個問題：何時具現 templates，以及 templates 被定義於何處。因此儘管這兩個問題從源碼角度來看並不互相影響（neatly decoupled），但在幕後編譯系統卻爲它們建立了一種不可見的耦合關係（invisible coupled）。這個關係或許可以這樣解釋：當包含 template 定義式的檔案發生變化時，這個檔案，以及所有「具現化該 template」的檔案，都不得不重新編譯。這和「置入式模型」看起來並無本質上的區別，但是從源碼角度，這種關係卻沒有那麼明顯。這樣的後果是，許多以「源碼級（source-base）技術」來管理依存關係（dependency）的工具，例如十分普及的 make 和 nmake 工具程式，如果像對待傳統 non-template 程式碼一樣地對待 exported template 程式碼，將無法正確運作。這也意味編譯器不得不花很多功夫去逐一牢記（bookkeeping）這些依存關係。最終結果是：使用 exported template 並不比使用「置入式模型」節省多少編譯時間。

最後一點：exported templates 有可能導致令人大吃一驚的語意問題。第 10 章會談到此一問題的細節。

很多人對 export 有一個誤解：他們覺得這麼一來就有可能在不提供源碼的情況下發售 template 程式庫（就像發售 non-template 程式庫那樣）[13]。這是一個錯誤觀念，因爲「隱藏程式碼」並非語言層面上的議題：提供一個機制用以隱藏 exported template 定義式，差不多相當於提供一個機制用以隱藏 included template 定義式。雖然這麼做或許可行（目前編譯器並不支援這種作法），但不幸的是，這又帶來新的困難：當編譯器使用這種 exported template 並發生錯誤時，如何在錯誤資訊中引用被隱藏的程式源碼？

---

[12] 就我們目前所知，惟一實作出 export 的公司是 Edison Design Group, Inc.（請參考 [EDG]）。然而它的技術也被其他數家編譯器廠商採用。

[13] 並非每一個人都認爲源碼不公開（closed-source）是個優點。

6.3.3 為分離式模型（Separation Model）預做準備

爲了可以隨時在「置入式模型」和「分離式模型」之間靈活切換，一個實際可行的辦法是：使用預處理指令（preprocessor directives）。具體實作如下：

```
// basics/myfirst4.cpp

#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// 如果定義了 USE_EXPORT，就使用關鍵字 export
#if defined(USE_EXPORT)
#define EXPORT export
#else
#define EXPORT
#endif

// template 的宣告式
EXPORT
template <typename T>
void print_typeof (T const&);

// 如果未定義 USE_EXPORT，就將 template 定義式含入
#if !defined(USE_EXPORT)
#include "myfirst.cpp"
#endif

#endif // MYFIRST_HPP
```

藉由定義或取消 `USE_EXPORT` 預處理符號，我們便可以在兩種模型之間切換。如果程式在含入 `myfirst.hpp` 之前定義有 `USE_EXPORT`，就使用「分離式模型」：

```
// 使用分離式模型：
#define USE_EXPORT
#include "myfirst.hpp"
...
```

如果程式未定義 `USE_EXPORT`，則使用「置入式模型」，因爲 `myfirst.hpp` 會自動含入 `myfirst.cpp` 內的定義式：

```
// 使用置入式模型：
#include "myfirst.hpp"
...
```

儘管有這樣的靈活性，我們還是重申，除了明顯的邏輯差異外，「置入式模型」和「分離式模型」之間還有一些微妙的語意差異。

請注意，我們也可以顯式具現化 exported templates。如此一來 templates 便可以在不同的檔案中定義。爲了能夠在「置入式模型」、「分離式模型」和「顯式具現化」之間切換，我們可以使用 USE_EXPORT 來控制程式結構，並輔以 6.2.2 節, p.67 所說的辦法。

6.4 Templates 與關鍵字 inline

將短小的函式宣告爲 inline，是提高程式執行速度的一個慣用手法。關鍵字 inline 用來告訴編譯器，最好將「函式呼叫」替換爲「被呼叫函式之實作碼」。然而編譯器並不一定進行此一替換工作，換句話說它握有決定權。

藉由「將定義式置於表頭檔，並將該表頭檔含入多個 .C 檔案內」的手法，function templates 和 inline functions 都可以被定義於多個編譯單元內。

這會產生一種假象：function templates 預設情況下就是 inline；其實它們並不是。如果你希望編寫一個 inline function template，應該明確使用關鍵字 inline（除非它本來就是 inline，例如它被寫於一個 class 定義式內）。

所以，對於沒有被寫在 class 定義式內的小型 template functions，你應該將它們宣告爲 inline[14]。

6.5 預編譯表頭檔（Precompiled Headers）

即使沒有 templates，C++ 表頭檔也可能很大，編譯它們需要很長時間。Templates 可能使得編譯時間更長。許多耐不住煎熬的程式員在很多場合中對此大聲疾呼，促使編譯器廠商實作出一個名爲「預編譯表頭檔」（precompiled headers）的機制。這個方案超越 *C++ Standard* 涵蓋範圍，其細節因編譯器廠商而異。雖然我們把「如何產生和使用預編譯表頭檔」的細節留給支援這一特性的編譯系統所帶文件，但在這裡簡短地說明其工作方式，還是非常有益的。

編譯器編譯某個檔案時，會從檔案起始處掃描至檔案尾端。當編譯器處理檔案中的每一個語彙單元（token；可能來自被 #include 的檔案），它便對其內部資料與狀態進行調整，包括將一些條目（entries）添加到符號表（symbols table）中，以便這些符號稍後可被搜尋到。處理過程中，編譯器可能會在目的檔（object file）中產生一些碼。

「預編譯表頭檔」便是基於這樣一個事實：程式中往往有很多檔案一開頭都含入了相同的程式碼。爲了更好地討論這個問題，假設每個檔案的前 N 行程式碼相同。我們可以把這 N 行程式碼編譯出來，並將編譯器此時的內部資料和狀態完整保存於一個所謂的「預編譯表頭檔」中。之後編譯程式的其他每一個檔案時，編譯器便可將此「預編譯表頭檔」載入，然後從第 N+1 行開

[14] 我們也可以不理會這條規則，因爲它可能會轉移當前主題的焦點。

始繼續編譯。值得注意的是，編譯器載入「預編譯表頭檔」的速度，比實際編譯最前面 N 行程式碼，要快上幾個數量級，但第一次把編譯器內部資料與狀態儲存到「預編譯表頭檔」時，通常比編譯 N 行程式碼的代價大得多，往往慢上 20% 到 200%。

有效利用「預編譯表頭檔」的辦法是，保證各個檔案最前面的「相同程式碼」儘可能多。實際情況中，這意味各檔案最前面的「相同的 #include 指令」儘可能多，因爲正是這些 #include 指令佔用了極長的編譯時間。此外最好能注意表頭檔被含入的次序。例如下面兩段程式碼：

```
#include <iostream>
#include <vector>
#include <list>
```

和

```
#include <list>
#include <vector>
```

「預編譯表頭檔機制」在這裡不起作用，因爲這兩段程式碼的次序並不相同。

有些程式員決定多含入一些或許用不到的表頭檔，他們認爲這至少比無法使用「預編譯表頭檔」要好。這不失爲一個可以大幅簡化表頭檔含入問題的良策。例如我們可以寫一個 std.hpp 表頭檔，把所有標準程式庫的表頭檔都包含進來[15]：

```
#include <iostream>
#include <string>
#include <vector>
#include <deque>
#include <list>
...
```

這個表頭檔可以被預編譯。每一個需要用到標準程式庫的程式，只需在開頭簡單加上這麼一行：

```
#include "std.hpp"
...
```

注意，這會花費相當長一段編譯時間，但如果你的系統有足夠的記憶體，「預編譯表頭檔」機制會顯著減少編譯時間 — 尤其是和缺乏「預編譯表頭檔機制」相比。以這種方式來組織標準

[15] 理論上，C++ 標準表頭檔並不需要對應至實際檔案。但實際上它們確實對應了那麼多表頭檔，而且那些檔案非常巨大。

程式庫的表頭檔非常方便，因爲這些檔案很少發生變化。這麼一來 std.hpp 的「預編譯表頭檔」只被生成一次[16]。否則「預編譯表頭檔」應該作爲依存關係（dependency）的一部份被加入專案組態（project confiuration）中。例如當所含入的檔案發生變化時，make 之類的工具就可以發覺並令編譯器重新編譯它。

一個非常不錯的「預編譯表頭檔」管理辦法是，將「預編譯表頭檔」分層：從「使用最廣泛且最不易發生變化者（例如上述的 std.hpp）」到「較不易發生變化且預編譯仍有價值者」。然而如果表頭檔正處於密集開發期而頻繁有著變化，爲它們生成「預編譯表頭檔」所用的時間，會比「預編譯表頭檔」所節省的時間還長。這個方案的關鍵在於，重複使用較穩定層（a more stable layer）的預編譯表頭檔，可以改善較不穩定層（a less stable layer）的預編譯時間。舉個例子，除提供一個 std.hpp 表頭檔（可被預編譯）之外，我們還定義一個 core.hpp 表頭檔，它又含入專案中的數個額外檔案，這麼做是爲了導入一個穩定層：

```
#include "std.hpp"
#include "core_data.hpp"
#include "core_algos.hpp"
...
```

由於這個檔案以 #include "std.hpp" 開始，編譯器就可以將相關的「預編譯表頭檔」載入，然後從下一行繼續編譯，從而避免重新編譯所有的標準程式庫表頭檔。當整個檔案被處理完畢，編譯器會生成一個新的「預編譯表頭檔」。使用者可經由 core.hpp 快速存取其所含入的大量功能函式 — 因爲編譯器會載入那個新的「預編譯表頭檔」。

6.6 Templates 的除錯（Debugging）

對 templates 除錯，可能遭遇兩類難題。第一類對 templates 撰寫者頗爲麻煩：如何保證我們所撰寫的 templates 對於「符合文件要求」的任何 template arguments 都能夠正常運作？另一類問題正好相對：作爲使用者，當 templates 的行爲未與文件一致時，如何得知我們的程式碼違反了哪一個 template parameter 的條件？

深入探討這個問題之前，我們應該先靜下心來細想：template parameters 可能帶來哪些種類的約束條件（constraints）。本節大部份內容都是討論那些「如果違反就會引發編譯錯誤」的約束條件，我們把這些條件稱爲語法約束（syntactic constraints），包括：必須存在特定某種建構式、對某些特殊函式的呼叫會造成模稜兩可（歧義性）等等。其他種類的約束條件稱爲語意約束（semantic constraints）。

[16] 有些 C++ 標準委員會成員發現，std.hpp 相當好用；他們並且已經建議把它視爲一個標準表頭檔加入 C++。這樣我們就可以寫出 #include <std> 含入所有標準程式庫表頭檔。甚至有人建議這個檔案應該被隱式（自動）含入，這樣我們就可以使用標準程式庫的所有機能，而無需撰寫任何的 #include 指令。

這些約束很難以機械化方式查驗出來，通常那是不實際的。例如我們可以要求 template type parameter 必須定義一個 operator<（這是語法約束），但通常我們也會要求這個 operator< 能夠在它所接受的範圍內比較兩值大小（這是語意約束）。

concepts（概念）這一術語經常被用到，表示一個「約束集」（constraints set），在 template 程式庫中會被反復提起。C++ 標準程式庫倚賴這樣一些 concepts：「隨機存取迭代器」（*random access iterator*）和「可被 *default* 建構式加以建構」（*default constructible*）。各個 concept 之間可形成繼承體系，也就是說一個 concept 可以是另一個 concept 的強化（精鍊, refinement）。所謂「更強概念」（more refined concepts）通常不僅包括原概念的約束條件，還可能加上更多約束。例如「隨機存取迭代器」就是「雙向迭代器」（*bi-directional iterator*）的強化。在這些術語的基礎上，我們可以這麼說：對 templates 除錯（debugging），就是找出在 templates 的實作和運用中違反了哪些 concepts（概念）。

6.6.1 解讀式煩錯誤訊息（Decoding the Error Novel）

一般而言，編譯錯誤訊息通常簡潔並直指問題所在。例如，當編譯器給出這樣的錯誤訊息：class X has no member 'fun'，對我們而言找到程式碼中的問題並非難事（也許我們錯把 run 寫成了 fun）。但面對 templates 情況便不相同。考慮下面這個簡單的程式片段，它使用 C++ 標準程式庫並犯了一個錯誤。對 list<string> 進行搜尋時，我們使用 greater<int> 仿函式（function object），而正確的做法應該使用 greater<string>：

```
std::list<std::string> coll;
...
// 搜尋第一個大於 "A" 的元素
std::list<std::string>::iterator pos;
pos = std::find_if(coll.begin(),coll.end(),                    // 範圍
                   std::bind2nd(std::greater<int>(),"A"));  // 搜尋準則
```

這類錯誤通常是由於剪貼/複製程式碼而卻了修改。

GNU C++ 是一個流傳普遍的編譯器，它的某一版本對以上程式碼報出如下錯誤訊息：

```
/local/include/stl/_algo.h: In function `struct _STL::_List_iterator<_STL::basic
_string<char,_STL::char_traits<char>,_STL::allocator<char> >,_STL::_Nonconst_tra
its<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> > > >
_STL::find_if<_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<cha
r>,_STL::allocator<char> >,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::
char_traits<char>,_STL::allocator<char> > > >, _STL::binder2nd<_STL::greater<int
> > >(_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL:
:allocator<char> >,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_tra
its<char>,_STL::allocator<char> > > >, _STL::_List_iterator<_STL::basic_string<c
har,_STL::char_traits<char>,_STL::allocator<char> >,_STL::_Nonconst_traits<_STL:
:basic_string<char,_STL::char_traits<char>,_STL::allocator<char> > > >, _STL::bi
nder2nd<_STL::greater<int> >, _STL::input_iterator_tag)':
```

```
/local/include/stl/_algo.h:115:   instantiated from `_STL::find_if<_STL::_List_
iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> >
,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_traits<char>,_STL::
allocator<char> > > >, _STL::binder2nd<_STL::greater<int> > >(_STL::_List_itera
tor<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> >,_STL
::_Nonconst_traits<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocat
or<char> > > >, _STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<c
har>,_STL::allocator<char> >,_STL::_Nonconst_traits<_STL::basic_string<char,_STL
::char_traits<char>,_STL::allocator<char> > > >, _STL::binder2nd<_STL::greater<i
nt> >)'
testprog.cpp:18:   instantiated from here
/local/include/stl/_algo.h:78: no match for call to `(_STL::binder2nd<_STL::grea
ter<int> >) (_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<cha
r> > &)'
/local/include/stl/_function.h:261: candidates are: bool _STL::binder2nd<_STL::g
reater<int> >::operator ()(const int &) const
```

這樣的錯誤訊息不像是診斷資訊，倒像天書一般。這會嚇倒 template 初級用戶。然而，如果你有一些實踐經驗的話，這類錯誤訊息倒也不是無法讀懂，找出錯誤所在也沒有那麼困難。

這條錯誤訊息的第一部份意思是，在某個 function template 具現體（其名字特長）中存在一個錯誤，對應的表頭檔深藏於 `/local/include/stl/_algo.h` 中。接下來是這個特定具現體得以具現的原因。在這個例子中，`testprog.cpp`（上例檔名）第 18 行引發了 `find_if` template 的具現動作，該 template 位於 `_algo.h` 表頭檔第 115 行。編譯器把這一切都報告給我們。我們並不關心所有這些具現化動作，但這使我們得以確定整個具現化過程的來龍去脈。

上述例子中，我們相信所有這些 templates（譯註）的確需要被具現化，我們想知道爲什麼具現化過程沒有成功。答案在錯誤訊息的最後一部份。"no match for call" 這句話告訴我們，由於參數和引數的型別不匹配，所以無法決議（*resolved*）函式呼叫式。不僅如此，錯誤訊息還告訴我們，型別 `int`（參數型別爲 `const int&`）可能是引發錯誤的原由。回頭看看程式第 18 行（`std::bind2nd(std::greater<int>(),"A")`），我們確實在那兒使用了一個 `int`，它和我們要搜尋的 `list` 的型別並不相容。把 `<int>` 換成 `<std::string>` 問題便解決了。

譯註：讀者可能對「所有這些 templates」的含義有所疑惑，這裡做個簡單解釋：上面的錯誤訊息本質上等同於「在 template1 中有一個錯誤，template1 的具現化是由於 template2 被具現化，而 template2 的具現化是由於 template3 被具現化」...等等等。這裡的 template1, template2, template3 就是作者所說的「所有 templates」。

毫無疑問，錯誤訊息的結構性還可以更好。有些問題可能出現在具現化開始之前。此外，比起使用「完全展開」的 template 具現體名稱（例如 `MyTemplate<YourTemplate<int> >`），將它加以分解可縮短過長的名稱（例如可以寫成 `MyTemplate<T> with T=YourTemplate<int>`）。

然而某些情況下得到完整的錯誤訊息很有用，因此其他編譯器也給出類似的（冗長）錯誤訊息就不令人驚奇了（儘管有些編譯器使用了上面所說的訊息結構）。

補充：Leor Zolman 撰寫了一個實用的程式 STLFilt，可解讀多種編譯器輸出的 STL 錯誤訊息。請參考 http://www.bdsoft.com/tools/stlfilt.html。

6.6.2 淺具現化（Shallow Instantiation）

如果具現化過程鏈（chain of instantiation）很長，編譯器也可能給出前面那種長篇診斷訊息。讓我以下面這個人爲捏造的例子來說明問題：

```
template <typename T>
void clear (T const& p)
{
    *p = 0;  // 假設 T 是一個 pointer-like 型別
}

template <typename T>
void core (T const& p)
{
    clear(p);
}

template <typename T>
void middle (typename T::Index p)
{
    core(p);
}

template <typename T>
void shell (T const& env)
{
    typename T::Index i;
    middle<T>(i);
}

class Client {
public:
    typedef int Index;
};

Client main_client;

int main()
{
    shell(main_client);
}
```

這個例子展現軟體開發中的一種典型分層方式：高層的 function templates（例如 shell()）倚賴其他部份（例如 middle()）完成，它們都用到了提供基本設施的函式（例如 core()）。當我們具現化 shell()時，它的所有下層函式也都會被具現化。本例之中，最深層的 core()以 int 型別具現化（這是由於 middle()以 Client::Index 型別具現化的緣故），卻試圖從該型別中提領（*dereference*）數值，這麼做是錯誤的。良好的一般性診斷訊息會包含「引發錯誤」的每一層呼叫軌跡，但我們的觀察是，這麼多訊息反而沒有多大用處。

你可以在 [StroustrupDnE] 找到一段非常好的論述，圍繞著此問題的核心思想。Bjarne Stroustrup 在討論中給出兩種辦法，使能儘早判定 template arguments 是否滿足某一組約束條件：(1) 經由某種語言擴展性質；(2) 儘早使用 template parameters。我們將在 13.11 節 p.218 簡單討論前一種辦法。後一種辦法是藉由淺具現化（*shallow instantiation*）來曝露問題：安插一些用不到的程式碼，它們不幹別的，只爲了在「template arguments 無法滿足更深層 templates 的需求」時得以引爆錯誤。

我們可以在 shell()中添加一段程式碼，試圖提領一個隸屬於 T::Index 型別的數值。例如：

```
template <typename T>
inline void ignore(T const&)
{
}

template <typename T>
void shell (T const& env)
{
    class ShallowChecks {
        void deref(T::Index ptr) {
            ignore(*ptr);
        }
    };
    typename T::Index i;
    middle(i);
}
```

如果 T 型別使得 T::Index 無法被提領（*dereferenced*），那麼錯誤訊息就首先會針對 shell()內的 ShallowChecks local class。注意，由於這個 local class 實際上並未被使用，因此它不會對 shell()的執行速度造成影響。不幸的是許多編譯器會給出一個警告，說 ShallowChecks class 和其成員沒有被用到。這裡我們使用了一個小技巧，藉由一個什麼都不做的 ignore()來抑制這個警告，但是這樣做會增加程式碼的複雜度。

很明顯，寫出本例中的這種啞碼（dummy code）可能和寫出實作碼一樣複雜。爲了控制複雜度不讓它蔓生，把這一類小程式碼集合在某種程式庫中是很自然的作法。這一類程式庫可能包含

一些巨集（macros），它們可以被展開爲程式碼，當 template parameter 的替換物違反了 parameter 的 concept（概念）時，這些巨集就能引發相應的錯誤訊息。這一類程式庫中最流行的稱爲 Concept Check Library，是 Boost 程式庫的一部份（請參考 [BCCL]）。

不幸的是這種技術的可移植性不高：不同編譯器之間的錯誤分析方式差異極大，有時在較高層面上很難捕捉到問題。

6.6.3 式符號（Long Symbols）

6.6.1 節, p.75 所分析的錯誤訊息，展現了 templates 的另一個問題：被具現化的 templates 程式碼有可能導致很長的符號。例如前面用到的 `std::string` 可能被展開爲：

```
STL::basic_string<char,_STL::char_traits<char>,
                  _STL::allocator<char> >
```

某些運用 C++ 標準程式庫的程式，可能產生長度超過 10,000 個字元的符號。這麼長的符號可能會使編譯器、聯結器或除錯器發出錯誤或警告。現代編譯器運用壓縮技術來降低這個問題，但在錯誤訊息中的效果並不明顯。

6.6.4 追蹤器（Tracers）

目前爲止我們已經討論了編譯或聯結 template 程式碼時可能遭遇的各種問題。然而，即使編譯聯結成功，接踵而來的難題是：如何保證程式運行正常。templates 有時候會加深這個問題，因爲每一個以 templates 表述的泛型程式碼，其行爲都取決於 templates 的客戶端（譯註：也就是取決於「具現化型別」）。這種情況比起在一般（non-templates）classes 和 functions 中要嚴重得多。追蹤器（tracer）是一種軟體裝置，有助於在開發初期檢查 template 定義式的問題，以減輕日後除錯工作的負擔。

追蹤器是一個用戶自定的 class，可作爲「待測試之 template」的 template argument。通常它恰恰符合待測 template 的要求，此外不具更多功能。然而更重要的是，追蹤器可以報告出「在它身上進行的操作的具體軌跡」（a trace of the operations that are invoked on it）。這使我們得以更身臨其境地檢測某個演算法的效能（efficiency），並給出演算過程中所有操作的完整序列（sequence of operations）。

下面是一個追蹤器示例，用來測試一個排序演算法（sorting algorithm）：

```
// basics/tracer.hpp

#include <iostream>
```

```
class SortTracer {
private:
    int value;                  // 用來排序的整數
    int generation;             // 此追蹤器的生成個數
    static long n_created;      // 建構式被呼叫的次數
    static long n_destroyed;    // 解構式被呼叫的次數
    static long n_assigned;     // 賦值次數
    static long n_compared;     // 比較次數
    static long n_max_live;     // 同一時間最多存在幾個 objects

    // 重新計算「同一時間最多存在幾個 objects」
    static void update_max_live() {
        if (n_created-n_destroyed > n_max_live) {
            n_max_live = n_created-n_destroyed;
        }
    }

public:
    static long creations() {
        return n_created;
    }
    static long destructions() {
        return n_destroyed;
    }
    static long assignments() {
        return n_assigned;
    }
    static long comparisons() {
        return n_compared;
    }
    static long max_live() {
        return n_max_live;
    }

public:
    // 建構式（constructor）
    SortTracer (int v = 0) : value(v), generation(1) {
        ++n_created;
        update_max_live();
        std::cerr << "SortTracer #" << n_created
                  << ", created generation " << generation
```

```
              << " (total: " << n_created - n_destroyed
              << ')' << std::endl;
}

// copy 建構式（copy constructor）
SortTracer (SortTracer const& b)
 : value(b.value), generation(b.generation+1) {
   ++n_created;
   update_max_live();
   std::cerr << "SortTracer #" << n_created
             << ", copied as generation " << generation
             << " (total: " << n_created - n_destroyed
             << ')' << std::endl;
}

// 解構式（destructor）
~SortTracer() {
   ++n_destroyed;
   update_max_live();
   std::cerr << "SortTracer generation " << generation
             << " destroyed (total: "
             << n_created - n_destroyed << ')' << std::endl;
}

// 賦值（assignment）
SortTracer& operator= (SortTracer const& b) {
   ++n_assigned;
   std::cerr << "SortTracer assignment #" << n_assigned
             << " (generation " << generation
             << " = " << b.generation
             << ')' << std::endl;
   value = b.value;
   return *this;
}

// 比較（comparison）
friend bool operator < (SortTracer const& a,
                        SortTracer const& b) {
   ++n_compared;
```

```
        std::cerr << "SortTracer comparison #" << n_compared
                  << " (generation " << a.generation
                  << " < " << b.generation
                  << ')' << std::endl;
        return a.value < b.value;
    }

    int val() const {
        return value;
    }
};
```

除了追蹤待排序值 `value` 外，這個追蹤器還提供若干成員，用來追蹤實際排序的過程：`generation` 追蹤每個物件被拷貝的次數，其他 static 成員則追蹤建構式和解構式的被呼叫次數、賦值和比較次數，以及「同一時間最多存在幾個物件」。

static 成員被定義在一個分離的 .C 檔案中：

```
// basics/tracer.cpp

#include "tracer.hpp"

long SortTracer::n_created = 0;
long SortTracer::n_destroyed = 0;
long SortTracer::n_max_live = 0;
long SortTracer::n_assigned = 0;
long SortTracer::n_compared = 0;
```

這個特定的追蹤器負責追蹤「在一個 template 內，建構、解構、賦值、比較等操作究竟以怎樣的方式進行」。下面這個測試程式使用上述追蹤器來追蹤 C++ 標準程式庫的 `std::sort` 演算法：

```
// basics/tracertest.cpp

#include <iostream>
#include <algorithm>
#include "tracer.hpp"

int main()
{
    // 準備樣本資料源
    SortTracer input[] = { 7, 3, 5, 6, 4, 2, 0, 1, 9, 8 };
```

```
    // 列印起始值
    for (int i=0; i<10; ++i) {
        std::cerr << input[i].val() << ' ';
    }
    std::cerr << std::endl;

    // 記錄起始值
    long created_at_start = SortTracer::creations();
    long max_live_at_start = SortTracer::max_live();
    long assigned_at_start = SortTracer::assignments();
    long compared_at_start = SortTracer::comparisons();

    // 執行演算法
    std::cerr << "---[ Start std::sort() ]---------------------" << std::endl;
    std::sort<>(&input[0], &input[9]+1);
    std::cerr << "---[ End std::sort() ]-----------------------" << std::endl;

    // 檢驗結果
    for (int i=0; i<10; ++i) {
        std::cerr << input[i].val() << ' ';
    }
    std::cerr << std::endl << std::endl;

    // 最終報告
    std::cerr << "std::sort() of 10 SortTracer's"
              << " was performed by: " << std::endl
              << SortTracer::creations() - created_at_start
              << " temporary tracers" << std::endl << ' '
              << "up to "
              << SortTracer::max_live()
              << " tracers at the same time ("
              << max_live_at_start << " before)" << std::endl << ' '
              << SortTracer::assignments() - assigned_at_start
              << " assignments" << std::endl << ' '
              << SortTracer::comparisons() - compared_at_start
              << " comparisons" << std::endl << std::endl;
}
```

這個程式一執行，會產生相當數量的輸出，但我們可以由最終結果獲得很多有用資訊。對於某個 std::sort()函式實作版本，我們獲得以下結果：

```
std::sort() of 10 SortTracer's was performed by:
 15 temporary tracers
 up to 12 tracers at the same time (10 before)
 33 assignments
 27 comparisons
```

我們看到了，程式執行期間產生 15 個臨時追蹤器，任一時間點最多有兩個附加追蹤器。

譯註：此一執行結果視 STL 實作版本而異。上述是原作者在 g++ with SGI STL 中得到的結果。改用 VC7.1 with P.J.Plauger STL，執行結果爲：

```
std::sort() of 10 SortTracer's was performed by:
 8 temporary tracers
 up to 11 tracers at the same time (10 before)
 32 assignments
 27 comparisons
```

這個追蹤器成功扮演了兩個角色：它證明我們的追蹤器所提供的功能已經完全滿足標準 sort() 演算法的需要（例如 sort()並不需要 operator== 和 operator>），此外它使我們對 sort 演算法的執行代價有了一些感受。然而它對於檢測 std::sort template 的正確性並沒有太大幫助。

6.6.5 Oracles（銘碼）

追蹤器既簡單又有效，但它只能讓我們針對特定資料或特定行爲，追蹤 templates 的執行。我們也許還想知道，sorting 演算法中的比較運算子（comparison operator）有些什麼要求，然而從上面的例子中我們只能測試出，這個比較運算子的行爲相當於一個「整數小於」功能。

有一個被某些人稱爲 "oracles"（或 run-time analysis oracles）的「追蹤器增強版」。它也是追蹤器，但與一個所謂的「推論引擎」（inference engine）程式相連，該程式可從各種因果關係中得出某些結論。這種系統的某一實作版本已被 C++ 標準程式庫採用，稱爲 MELAS，在 [MusserWangDynaVeri] 中有一些討論[17]。

Oracles 使我們能夠在某些情況下動態檢驗 template 演算法，而無需指定 template arguments 的型別（它實際上被替換爲 oracles 本身）或指定演算法輸入資料（當 oracles 的推理引擎受阻時，可能會要求某種假設性輸入）。然而即便是 oracles 也只能處理複雜度不高的演算法（此乃受限於推理引擎）。基於這些原因，我們並不深入研究 oracles，有興趣的讀者可以參考 [MusserWangDynaVeri]，或其中所引用的其他相關書籍。

[17] 該文作者之一 David Musser，是 C++ 標準程式庫開發過程中的關鍵人物，設計並實作了第一個關聯式容器（associative containers）。

6.6.6 原型/模本（Archetypes）

前面提到過，追蹤器通常提供一個介面，使能滿足「受追蹤之 template 的最小要求」。當這樣一個追蹤器不在執行期產生任何輸出時，我們把它叫做一個「原型/模本」（archetype）。這種東西使我們能夠檢驗「一個 template 實作品除了計劃內的約束外，沒有其他更多語法約束」。很多 template 實作者都會針對 template library 中的每一個 concepts（概念）開發出一個原型/模本（archetype）。

6.7 後記

根據「單一定義規則」（one-definition rule, ODR），人們在實踐中採用「將源碼分置於表頭檔和 .C 檔」的組織形式。附錄 A 對這條規則有深入的討論。

置入式模型（Inclusion Model）和分離式模型（Separation Model）的爭論由來已久。C++ 編譯器實作品顯示，置入式模型相當實用。然而第一個 C++ 編譯器並非如此，它對 template 定義式的含入是隱寓的（implicitly），這往往使人誤以爲它使用了分離式模型。原始模型請參考第 10 章。

[StroustrupDnE] 極好地描述了 Stroustrup 對於 template 程式碼組織方式的一個設想，同時指出它所帶來的實作困難。很明顯，這種組織方式並非「置入式模型」。C++ 標準化過程中的某一段時間，人們認爲置入式模型是惟一可行的組織方式。然而經過激烈論戰後，人們對於可降低「程式碼耦合」的新模型漸漸有了充份的認識，這個模型便是「分離式模型」。和置入式模型不同的是，它並非基於任何既有實作品，而是一個理論上的模型。從被提出到最終實作出來（2002 年 5 月），共耗用了 5 年以上的時間。

如果能夠將「預編譯表頭檔」的概念加以擴充，使得在一次編譯時能夠含入多個表頭檔，將十分具有誘惑力。原則上它會對預編譯帶來更好的粒度（a finer grained approach）。這裡主要的障礙是預處理器（preprocessor）：某個檔案內的巨集（macros）到了另一個檔案中可能有完全不同的作用。然而一旦檔案被預編譯，巨集的處理過程便告完成。實際經驗中很難在處理另一個表頭檔時，對先前預編譯好的表頭檔修修補補。

Jeremy Siek 的 Concept Check Library（請參考 [BCCL]）相當系統化地透過「在高層 templates 添加部份啞碼（dummy code）」來改善 C++ 編譯器的診斷訊息。它是 Boost 程式庫的一部份（請參考 [Boost]）。

6.8 摘要

- Templates 對於古典的「編譯器＋聯結器」模型提出新挑戰。人們提出了 template 程式碼的多種不同組織法：置入式模型（Inclusion Model）、顯式具現化（Explicit Instantiation），以及

分離式模型（Separation Model）。

- 通常情況下應該使用置入式模型（也就是把所有 template 程式碼寫在表頭檔中）。
- 如果把 template 程式碼的宣告式和定義式放置於不同的表頭檔中，可以更加容易地在「置入式模型」和「顯式具現化模型」之間切換。
- *C++ Standard* 爲 templates 定義了一個「分離式模型」（使用關鍵字 `export`）。然而很多編譯器並不支援這個特性。
- 對 template 程式碼偵錯，可能相當困難。
- Templates 具現體（instantiation）的名稱可能很長。
- 爲了從「預編譯表頭檔」機制中獲取最大好處，你應該確保各檔案中的`#include` 指令的次序相同。

7

Template 基本術語

Basic Template Terminology

目前爲止我們已經介紹了 C++ templates 的基本概念。進一步講述細節之前，讓我們先關注一下本書各概念所使用的術語。這非常有必要，因爲在 C++ 社群（甚至 C++ 標準委員會）中，對概念和術語的運用不見得精準。

7.1 是 Class Template 還是 Template Class？

在 C++ 中，structs、classes 和 unions 統稱爲 class types。如果沒有附加額外飾詞，一般字體的 "class" 表示的是關鍵字 `class` 和 `struct` 所代表的型別[18]。特別請注意，"class type" 包括 unions，而 "class" 不包括 unions。

面對一個「本身是 template」的 class，人們的稱謂相當混亂：

- 術語 class template 表示此 class 是個 template。這就是說它是「一整族 classes」的參數化描述（parameterized description）。
- 術語 template class 有以下用法（其中第二和第三種用法的差異甚微，在本書其他部份並無重要用途）：
  - 作爲 class template 的同義語。
  - 表示由 templates 產生的 classes。
  - 表示一個「以 template-id 爲名稱」的 classes。（譯註：template-id 在 p.90 介紹）

鑒於 template class 的含義不夠精確，本書避免使用這個術語，統一使用 class template。

[18] 在 C++ 中，`class` 和 `struct` 的惟一差別是：`class` 的預設存取層級（default access level）是 `private`，而 `struct` 的預設存取層級是 `public`。涉及 C++ 新特性時，我們較多使用關鍵字 `class`；如果用來表現所謂 "Plain Old Data（POD）" 的一般 C 資料結構，我們才使用關鍵字 `struct`。

類似道理，我們使用術語 function template 和 member function template，避免使用 template function 和 template member function。

7.2 具現化（Instantiation）與特化（Specialization）

在 template 中，「以實際值（actual values）做爲 template arguments，從而產生常規的（*regular*）class、function 或 member function」，這個過程稱爲「template 具現化」（template instantiation）。這些隨之產生的物體（*entity*；包括 class, function 或 member function）通稱爲特化體（*specialization*）。（譯註：我所閱讀的眾多泛型編程書籍中，很少將這些物體稱爲特化體，較常稱呼的是具現體, *instantiation*）

在 C++ 中，具現化並非產生特化體的惟一方法。另一種機制可使程式員明白地以某些特定的 template parameters 宣告某些物體。我們在 3.3 節, p.27 中已經介紹過這種方法：透過 `template<>` 進行特化：

```
template <typename T1, typename T2>       // primary class template
class MyClass {
    ...
};

template<>                                // 顯式（明確）特化
class MyClass<std::string, float> {
    ...
};
```

嚴格地說，這應該稱作「顯式特化」（explicit specialization），以別於因具現化或其他方式而產生的特化。

3.4 節, p.29 我們提到，「仍帶有 template parameters」的特化稱爲偏特化（partial specializations）：

```
template <typename T>                     // 偏特化
class MyClass<T,T> {
    ...
};

template <typename T>                     // 偏特化
class MyClass<bool,T> {
    ...
};
```

當我們討論顯式特化或偏特化時，常把「最泛化（*general*）的那個 template」稱爲 primary template（主模板/原始模板）。

7.3 宣告（Declarations）vs. 定義（Definitions）

截至目前，本書只在爲數不多的場合用上了「宣告（式）」和「定義（式）」兩個詞。在 C++ 中這兩個詞都有各自的精確含義，本書的用法與之完全符合。

所謂「宣告」是這樣一種 C++ 構件（construct）：將一個名稱引入（或重新引入）至某個 C++ 作用域（scope）中。這個「被引入體」總是包括該名稱的某部份資訊，不過一個合法宣告並不需要過多細節。例如：

```
class C;          // 宣告：C 是一個 class
void f(int p);    // 宣告：f() 是一個以 p 爲具名參數的函式
extern int v;     // 宣告：v 是一個變數
```

注意，雖然巨集定義（macro definitions）和 goto 標籤（goto labels）也有名稱，但 C++ 並不把它們當作一種宣告。

兩種情況下「宣告」會變成「定義」：(1) 當它們的結構細節被寫明白，(2) 當編譯器必須爲變數配置儲存空間。對 class type 和 function 而言，這意味你必須爲它們的定義提供「以大括號封起來的程式碼」。對 variable（變數）而言，這意味一個初始化動作或一個不以關鍵字 extern 爲前導的宣告。以下補足上例中的每一個宣告式所欠缺的定義：

```
class C {};             // class C 的定義（及宣告）

void f(int p) {         // 函式 f() 的定義（及宣告）
   std::cout << p << std::endl;
}

extern int v = 1;       // 一個初始化動作（這使它成爲變數 v 的一個定義）

int w;                  // 全域變數而且不帶 extern 飾詞（所以既是宣告也是定義）
```

另外，如果 class templates 或 function templates 的宣告（式）帶有實作碼，我們也稱它們爲定義（式）：

```
template <typename T>
void func (T);
```

以上是宣告而不是定義。然而以下是一份定義：

```
template <typename T>
class S {};
```

7.4 單一定義規則（One-Definition Rule, ODR）

C++ 語言對於各種物體（entities）的再宣告（redeclaration）做出了一些限制。這些限制總稱爲「單一定義規則」（One-Definition Rule, ODR）。這條規則的細節頗爲複雜，並且適用於相當大的語言跨度上。本書後續章節會闡述這條規則在各種適用情形下的各方面細節。附錄 A 對此規則亦有完整描述。目前只需記住 ODR 的數條基本守則就夠了：

- Non-inline 函式和成員函式，以及全域變數和 static 成員變數，在整個程式中只能定義一次。
- Class 型別（包括 structs 和 unions）及 inline 函式，在每個編譯單元中最多只能定義一次。如果跨越不同的編譯單元，則其定義必須完全相同。

所謂「編譯單元」是指一個源碼檔案所涉及的所有檔案；也就是說包括 `#include` 指令所含入的所有檔案。

在本書剩餘章節中，所謂「可聯結物」（linkable entity）可以是下列任何一個東西：non-inline 函式或 non-inline 成員函式、全域變數或 static 成員變數，以及所有由 template 程式碼產生的上述四種物體。

7.5 Template Arguments（模板引數）vs. Template Parameters（模板參數）

比較下面兩個 classes，第一個是 class template，第二個是與之類似的常規（non-template）class：

```
template <typename T, int N>
class ArrayInClass {
public:
    T array[N];
};

class DoubleArrayInClass {
public:
    double array[10];
};
```

如果我們分別把 `T` 換成 `double`，把 `N` 換成 `10`，那麼兩個 class 本質上等價。C++ 的這種替換係透過下面這種寫法完成：

```
ArrayInClass<double,10>
```

注意上一行的 template 名稱之後緊跟著以角括號括起來的 template arguments（模板引數）。

無論這些 template arguments 本身是否倚賴（受控於、取決於；*dependent on*）template parameters，我們把「template name + 大括號括起的 template arguments」組合體稱爲 template-id。

這樣一個 template-id 可以像其對應的 non-template 實體一樣地被使用，例如：

```
int main()
{
    ArrayInClass<double,10> ad;        //這是一個 template-id
    ad.array[0] = 1.0;
}
```

區分 template parameters 和 template arguments 是非常重要的。你可以說「把引數傳爲參數」（*pass arguments to become parameters*）[19]，或者更準確地說：

- Template parameters 是在 template 宣告式或定義式中位於關鍵字 `template` 之後的那些名稱（前例中的 `T` 和 `N`）。
- Template arguments 是用以替換 template parameters 的東西（前例中的 `double` 和 `10`）。和 template parameters 不同的是，template arguments 並不限於「名稱」（譯註：意思是也可以爲數據）。

當我們使用 template-id 時，用以替換 template parameters 的那些 template arguments 是被顯式（明確）指定的，但也存在隱式替換的情況，例如 template parameters 可被其預設值替換。

一個基本原則是：任何 template arguments 都必須是編譯期可確定的數量（quantity）或實值（value）。這個要求大大提高了 template 物體的執行期效率，本書後繼章節還會詳細說明這一點。由於 template parameters 最終會被替換爲編譯期實值，因此你也可以用它們構成編譯期運算式（compile-time expressions）。`ArrayInClass` template 就是利用這一點來設定其成員 `array` 的大小 — 那必須是個常數運算式，而 template parameter `N` 符合這一要求。

我們可以把這個論據更推進一步：由於 template parameters 是編譯期物體（compile-time entities），因此它們也可以被用以產生合法的 template arguments。下面是個例子：

```
template <typename T>
class Dozen {
public:
    ArrayInClass<T,12> contents;
};
```

注意，本例之中 `T` 既是 template parameter 也是 template argument。因此我們可以藉由這樣的機制，以簡單的 templates 搭建複雜的 templates。當然，這和「組合型別（types）和函式，以拼裝出更複雜的物體」並沒有本質上的區別。

[19] 在學界，引數（arguments）有時被稱爲「實際參數, actual parameters」，參數（parameters）有時被稱爲「形式參數, formal parameters」。

第二篇 深入模板

Templates in Depth

本書第一篇介紹了 C++ templates 的大部份語言概念，足夠解答日常編程可能遭遇的大部份問題。本書第二篇帶有參考價值：當你進行複雜軟體開發時，會碰到一些不尋常的問題，你可以在本篇找到答案。你也可以跳過這一篇先閱讀後續章節，那些章節（或書後索引）有可能涉及本篇內容，那個時候你可以再回過頭來閱讀本篇。

講解清晰、內容完備，並保持簡明扼要，是我們的寫作目標。基於此，書中例子往往短小並只針對特定問題。這樣做可以保證我們不會偏離主題太遠。

我們還討論了 C++ 語言的 template 特性的未來可能變化。本篇主題包括：

- Template 宣告式的根本問題
- Templates 中的名稱含義
- C++ template 的具現化機制（instantiation mechanisms）
- Template argument deduction（模板引數推導）規則
- 特化（specialization）和重載（overloading）
- 未來可能的發展

8

基礎技術之深入
Fundamentals in Depth

本章將回顧並深入講述第一篇介紹的一些根本知識：templates 的宣告、template parameters 的侷限（restrictions）、template arguments 的約束條件（constraints）…等等。

8.1 參數化宣告（Parameterized Declarations）

C++ 目前支援兩大 templates 基本類型：class templates 和 function templates（這方面的未來可能變化請參考 13.6 節, p.212）。這個分類還包括 member templates。Templates 的宣告與一般 class 和 functions 的宣告頗爲類似，差別在於 templates 宣告式有一個參數化子句（parameterization clause）：

```
template<...parameters here...>
```

也可能長像如下：

```
export template<...parameters here...>
```

（關鍵字 export 的詳細討論出現於 6.3 節, p.68，和 10.3.3 節, p.149）。

稍後我們會回頭討論 template parameter 的宣告。以下實例展示兩種 templates，一種是在 class 之內（譯註：亦即 member templates），一種是在 class 之外且 namespace scope 之內（譯註：global scope 也被視爲一種 namespace scope）：

```
template <typename T>
class List {                          // 一個 namespace scope class template
public:
    template <typename T2>            // 一個 member function template
    List (List<T2> const&);           // (這是個建構式)
    ...
};
```

```
template <typename T>
 template <typename T2>
List<T>::List (List<T2> const& b)   // 一個定義於 class 外的 member function template
{
    ...
}

template <typename T>
int length (List<T> const&);    // 一個 namespace scope function template

class Collection {
    template <typename T>       // 一個定義於 class 內的 member class template
    class Node {
        ...
    };

    template <typename T>       // 又一個 member class template，無定義
    class Handle;

    template <typename T>       // 一個定義於 class 內的 member function template
    T* alloc() {                // 隱寓爲 inline
       ...
    }
    ...
};

template <typename T>           // 一個定義於 class 外的 member class template
class Collection::Handle {
    ...
};
```

注意，定義於 class 外的 member templates 可有多重 `template<...>` 參數化子句，其中一個代表 template 本身，其餘各個子句代表外圍的每一層 class template。這些子句必須從最外層 class templates 開始寫起。

你也可以撰寫 union templates（被視爲 class template 的一種）：

```
template <typename T>
union AllocChunk {
    T object;
    unsigned char bytes[sizeof(T)];
};
```

Function template 可以有預設的呼叫引數（default call arguments），和一般 function 一樣：

```
template <typename T>
void report_top (Stack<T> const&, int number = 10);

template <typename T>
void fill (Array<T>*, T const& = T());  // 若 T 爲內建型別，T() 爲 0 或 false, 見 p.56)
```

後一個宣告式示範如何讓一個 default call arguments 取決於某個 template parameter。當 `fill()` 被呼叫時，如果呼叫者提供了第二引數值，預設引數便不會被具現化。這可確保如果預設引數無法被某個特定型別 `T` 具現化，不會引發編譯錯誤。舉個例子：

```
class Value {
public:
    Value(int);          // 無 default 建構式
};

void init (Array<Value>* array)
{
    Value zero(0);

    fill(array, zero);  // OK：前述宣告中的 T() 未用上。
    fill(array);        // ERROR：前述宣告中的 T() 被用上，然而 T=Value 卻不合法。
      // 譯註：讓我進一步解釋：前述宣告中的 T 被推導爲 Value; fill() 第二引數預設爲 T()，
      //       也就成爲 Value()。然而根據 class Value 的定義，並不存在 Value()。所以失敗。
}
```

除了兩種 template 基本類型，另有三種宣告也可以被參數化，它們都使用類似寫法。三者均相當於 class template 的成員定義[20]：

1. class templates 的成員函式定義
2. class templates 的 nested class members（嵌套類別成員）定義
3. class templates 的 static 成員變數定義

雖然它們也可以被參數化，但它們並不是第一級（first-class）templates。它們的參數完全由它們所隸屬的 template 決定。下面是個例子：

```
template <int I>
class CupBoard {
    void open();                //譯註：隸屬於 CupBoard class template
    class Shelf;                //譯註：隸屬於 CupBoard class template
    static double total_weight; //譯註：隸屬於 CupBoard class template
    ...
};
```

[20] 它們非常類似一般的 class members，然而有時候會被人誤稱爲 member templates。

```
template <int I>
void CupBoard<I>::open()
{
    ...
}

template <int I>
class CupBoard<I>::Shelf {
    ...
};

template <int I>
double CupBoard<I>::total_weight = 0.0;
```

雖說這種參數化定義通常也被稱爲 templates，但很多場合中這種稱謂並不合適。

8.1.1 虛擬成員函式（Virtual Member Functions）

Member function templates 不能宣告成 `virtual`。這個限制有其原因。通常虛擬函式呼叫機制使用一個大小固定不變動的表格，其中每一筆條目（entry）記錄一個虛擬函式入口（entry point）。然而直到整個程式編譯完成，編譯器才能知道有多少個 member function templates 需要被具現化。因此，如果要支援 virtual member function templates，C++ 編譯器和聯結器需要一個全新機制。

然而一般的 class template members 可以是虛擬函式，因爲當 class 被具現化時，那些 members 的數量是固定的：

```
template <typename T>
class Dynamic {
public:
    // 譯註：下面是 class template 的 member function，可爲 virtual。
    virtual ~Dynamic();      // OK：每個 Dynamic<T> 具現體都有一個解構式

    // 譯註：下面是 member (function) template，不可爲 virtual。
    template <typename T2>
    virtual void copy (T2 const&);
        // ERROR：編譯器此時並不知道在一個 Dynamic<T> 具現體中要產生多少個 copy() 具現體。
        // 譯註：因此編譯器無法備妥虛擬表格（virtual table, vtbl）。
};
```

8.1.2 Template 的聯結（Linkage）

每個 template 在其作用域（scope）內必須有一個獨一無二的名稱，除非是被重載的（overloaded）function templates（見 12 章）。特別請注意，class template 不能和其他不同種類的物體（entities）共用同一個名稱，這點與一般（non-template）class 不同。

```
int C;

class C;    // OK：class 名稱和 nonclass 名稱處於不同的空間(space)內

int X;

template <typename T>
class X;    // ERROR：名稱與上述變數 X 衝突

struct S;

template <typename T>
class S;    // ERROR：名稱與上述 struct S 衝突
```

Template 名稱需要聯結（linkage），但不能夠使用 C 聯結方式（C linkage）。非標準的聯結方式可能會有「與實作相依」（implementation-dependent）的意義（但我們無法知道是否某個編譯系統支援非標準的 templates 聯結方式）：

```
extern "C++" template <typename T>
void normal();
    // 這是預設方式。其聯結規格（linkage specification，亦即 extern "C++"）可省略不寫。

extern "C" template <typename T>
void invalid();
    // 這是無效（不合法）的：templates 不能使用 C 聯結方式。

extern "Xroma" template <typename T>
void xroma_link();
    // 這是非標準的，但可能某些編譯器會在某一天支援相容於 Xroma 語言的聯結方式。
```

Templates 通常使用外部聯結（external linkage）。惟一例外是 static namespace scope function templates：

```
template <typename T>
void external();
          // 指涉（refer to）另一檔案中同名且同作用域（scope）的相同物體（entity）。
```

```
template <typename T>
static void internal();   // 與另一檔案中的同名 template 無關
```

注意，函式內不能再宣告 templates。

8.1.3 Primary Templates（主模板/原始模板）

正規、標準的 template 宣告式宣告的是 primary templates。這一類宣告式在 template 名稱之後並不添加由角括號括起的 template argument list：

```
template<typename T> class Box;                 // OK：primary template

template<typename T> class Box<T>;              // ERROR

template<typename T> void translate(T*);        // OK：primary template

template<typename T> void translate<T>(T*);     // ERROR
```

一旦我們宣告一個偏特化 templates，就是產生一個 non-primary class templates，第 12 章將討論這種情況。Function templates 必須是 primary templates。（C++ 語言在這一方面可能將有所變動，請參考 13.7 節, p.213。）

8.2 Template Parameter（模板參數）

Template parameters 有三種類型：

1. Type parameters（型別參數）；這種參數最常用。
2. Nontype parameters（非型別參數）
3. Template template parameters（雙重模板參數）

所謂 template parameters 是在 template 宣告式的參數化子句（parameterization clause）中宣告的變數。Template parameters 不一定得具名：

```
template <typename, int>   // 譯註：兩個 template parameters 都沒有名稱
class X;
```

但是當 template 程式碼中需要用到某個 template parameter 時，後者就必須具名。請注意，後宣告的 template parameters 可以用到先宣告的 template parameters 的名稱，反之不然：

```
template <typename T,              // 第一個參數 T，被用於第二和第三參數的宣告之中。
          T* Root,
          template<T*> class Buf>    // 譯註：Buf 就是個 template template parameter
class Structure;
```

8.2.1 Type Parameters（型別參數）

Type parameters 可以採用關鍵字 `typename` 或關鍵字 `class` 導入，兩者完全等價[21]。其宣告形式是：關鍵字 `typename` 或 `class` 後面跟一個簡單標識符號（做爲參數名稱），該符號後面可跟一個逗號以便區隔下一參數，也可以使用一個閉鎖角括號（`>`）結束參數子句，或跟隨一個等號（`=`）標示出 default template argument（預設模板引數）。

在 template 宣告式中，type parameter 的作用非常類似 typedef 的名稱。例如，你不能使用如 `class T` 這樣的完整名稱，即使 `T` 確實表示一個 class type：

```
template <typename Allocator>
class List {
    class Allocator* allocator;      // ERROR
    friend class Allocator;          // ERROR
    ...
};
```

C++ 將來有可能接受這種型式的 friend 宣告。

8.2.2 Nontype Parameters（非型別參數）

Nontype template parameter 是指那些可在編譯期或聯結期確定其值的常數[22]。此種參數的型別必須是以下三者之一：

- 整數（integral）或列舉（enumeration）型別
- pointer 型別；包括指向常規 objects、指向 functions 和指向 members。
- reference 型別；包括指向（指涉、代表）objects 和指向 functions。

目前不允許使用其他型別（將來有可能允許使用浮點數型別，見 13.4 節, p.210）。

也許你會感到驚訝，nontype parameter 的宣告在某種情況下也以關鍵字 `typename` 爲前綴詞：

```
template<typename T,                          // 一個 type parameter
         typename T::Allocator* Allocator>    // 一個 nontype parameter
class List;
```

---

[21] 關鍵字 `class` 並不意味傳入的引數必須是 class type，該引數事實上可隸屬任何可用的 type。不過，無論 template parameters 是以關鍵字 `typename` 還是以關鍵字 `class` 宣告，函式內定義的 class type（亦即 local classes）都不能被用做 template arguments。

[22] Template template parameters 也不用來表示型別；然而我們討論 nontype parameters 時並不考慮 template template parameters。

這兩種情況很容易區分：頭一種情況中的 typename 後面總是緊跟一個簡單標識符號，第二種情況中的 typename 後面跟一個帶飾詞的名稱（qualified name），亦即一個包含雙冒號（::）的名稱。本書 5.1 節, p.43 和 9.3.2 節, p.130 告訴你何時需要在 nontype parameter 之前方使用關鍵字 typename。

Nontype parameters 也可以是 function 型別或 array 型別，但它們都會退化（*decay*）爲對應的 pointer 型別：

```
template<int buf[5]> class Lexer;       // buf 實際被當作 int*
template<int* buf> class Lexer;         // OK：這是一個再宣告（redeclaration）
```

Nontype template parameters 的宣告方式非常類似變數宣告，但你不能加上諸如 static、mutable 之類的修飾。你可以加上 const 或 volatile，但如果這些飾詞出現在參數型別的最外層，編譯器會忽略它們：

```
template<int const length> class Buffer;      // const 被忽略
template<int length> class Buffer;            // 與上一行宣告式等價
```

最後一點，nontype parameters 總是右值（rvalues）：它們不能被取址，也不能被賦值。

8.2.3 Template Template Parameters（雙重模板參數）

Template template parameters 是一種「class templates 佔位符號（placeholder）」，其宣告方式和 class templates 類似，只是不能使用關鍵字 struct 和 union：

```
template <template<typename X> class C>   // OK
void f(C<int>* p);

template <template<typename X> struct C>  // ERROR：不能使用關鍵字 struct
void f(C<int>* p);

template <template<typename X> union C>   // ERROR：不能使用關鍵字 union
void f(C<int>* p);
```

在它們的作用域內，你可以像使用 class templates 那樣地使用 template template parameters。

Template template parameters 的參數也可以有 default template arguments（預設模板引數）。如果客戶端沒有爲相應的參數指定引數，編譯器就會使用這些預設引數：

```
template <template<typename T,
                   typename A = MyAllocator> class Container>
class Adaptation {
    Container<int> storage;  // 暗自（隱寓）等價於 Container<int, MyAllocator>
    ...
};
```

在 template template parameters 中，template parameter 的名稱只能被用於 template template parameter 的其他參數宣告中。這一點可以下面例子來解釋：

```
template <template<typename T, T*> class Buf>
class Lexer {
    static char storage[5];
    Buf<char, &Lexer<Buf>::storage[0]> buf;
    ...
};

template <template<typename T> class List>
class Node {
    static T* storage;  // ERROR：這裡不能使用 template template parameters 的參數 T
    ...
};
```

通常 template template parameter 中 template parameters 名稱並不會在其他地方被用到，因此，未被用到的 template parameters 可以不具名。例如上頁的 Adaptation template 可被宣告爲：

```
// 譯註：原先（上頁最下）的兩個參數名稱 T 和 A 都被省略了。
template <template <typename,
                    typename = MyAllocator> class Container>
class Adaptation
{
    Container<int> storage;  // 暗自（隱隱）等價於 Container<int, MyAllocator>
    ...
};
```

8.2.4 Default Template Arguments（預設的模板引數）

目前，只有 class template 的宣告式可以存在（擁有）default template arguments（將來 C++ 語言可能會對此做出修改，見 13.3 節, p.207）。無論何種 template parameters 都可以有其預設引數，當然它必須匹配（吻合）對應參數。很明顯，預設引數不能相依於其自身參數，但可以相依於在它之前宣告的參數：

```
template <typename T, typename Allocator = allocator<T> >
class List;
```

和函式的預設引數一樣，某個參數（譯註：不論是 call parameters 或 template parameters）帶有預設引數的條件是：其後續所有參數也都有預設引數。

後續參數的預設引數通常寫在同一個 template 宣告式中，但也可以寫在該 template 更早的某個宣告式中。例如：

```
template <typename T1, typename T2, typename T3,
          typename T4 = char, typename T5 = char>
class Quintuple;  // OK

template <typename T1, typename T2, typename T3 = char,
          typename T4, typename T5>
class Quintuple;  // OK：T4 和 T5 先前已有預設值

template <typename T1 = char, typename T2, typename T3,
         typename T4, typename T5>
class Quintuple;  // 錯誤：T1 不能擁有預設值，因爲 T2 沒有預設值
//譯註：VC6 不支援上述的「組合式」引數預設方式。VC7.1/ICL7.1/g++ 3.2 沒有問題。
```

此外我們也不能重複指定 default template arguments：

```
template<typename T = void>
class Value;

template<typename T = void>  // ERROR：預設引數被重複定義了
class Value;
```

8.3 Template Arguments（模板引數）

所謂 templates arguments 是當編譯器具現化一個 template 時，用來替換 template parameters 的值。編譯器以數種不同的機制來決定以何值替換 template parameters：

- explicit template arguments（明白指定之模板引數）：可在 template 名稱之後跟著一個或多個明確的 template arguments，並以角括號括起。這個完整名稱被稱爲 template-id（p.90）。
- injected class name（內植式類別名稱）：在帶有參數 P1,P2…的 class template X 作用域中，template X 的名稱與 template-id X<P1, P2,...> 等價。細節請見 9.2.3 節, p.126。
- default template arguments：如果存在可用的 default template arguments，我們便可在 class templates 具現體中省略不寫 explicit template arguments。然而即使每一個參數都有預設值，你也必須把開閉兩個角括號寫上（即使括號內什麼都沒有）。
- argument deduction（引數推導）：編譯器可根據 function call arguments 推導出 function template arguments。具體推導細節將在第 11 章講述。其他某些情況下，編譯器也可能動用推導機制。如果所有 template arguments 都可以推導得出，你就無需在 function template 名稱後面加寫角括號。

8.3.1 Function Template Arguments（函式模板引數）

你既可以明白指定 function template 的 template arguments，也可以讓它們被編譯器推導出來。例如：

```
// details/max.cpp

template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

int main()
{
    max<double>(1.0, -3.0);      // explicit template arguments
    max(1.0, -3.0);              // template arguments 被隱式推導爲 double
    max<int>(1.0, 3.0);          // 明確指定<int> 以抑制推導，從而令引數型別爲 int
}
```

某些 template arguments 無法被推導獲得（見 11 章）。你最好把這一類參數放在 template parameter list 的最前面，這樣客戶端（呼叫者）只需明白指定編譯器無法推導的那些引數即可，其餘引數仍可被自動推導而得。例如：

```
// details/implicit.cpp

template <typename DstT, typename SrcT>
inline DstT implicit_cast (SrcT const& x)  // SrcT 可被推導，但 DstT 無法推導。
{                                          // 譯註：因爲 DstT 不出現在引數列，無法進行引數推導。
    return x;
}

int main()
{
    double value = implicit_cast<double>(-1);
}
```

如果不這麼寫，改而把 template parameters 的順序換一下（寫成 `template<typename SrcT, typename DstT>`），我們就不得不在呼叫 `implicit_cast` 時把兩個 template arguments 都明確寫出來。

由於 function templates 可被重載（overloaded），因此即使明確寫出一個 function template 的所有引數，可能也不足以使編譯器確定該呼叫哪一份具體函式，因爲某些情況下符合要求的函式可能有「一群」。下面的例子說明這種情況：

```
template <typename Func, typename T>
void apply (Func func_ptr, T x)
{
    func_ptr(x);
}

template <typename T> void single(T);

template <typename T> void multi(T);
template <typename T> void multi(T*);

int main()
{
    apply(&single<int>, 3);  // OK
    apply(&multi<int>, 7);   // ERROR：符合 multi<int>形式的函式不只一個
}
```

在這個例子中，對 apply()的第一個呼叫動作是合法的，因爲運算式 &single<int> 沒有任何歧義，因此 template parameter Func 的值可被編譯器輕易推導出來。然而在第二個呼叫動作中，&multi<int> 可以是兩個不同型別中的任何一個，這種情況下編譯器無法推導出 Func。

不僅如此，明確指定 template arguments 還可能導致建構出不合法的 C++ 型別。考慮下例的重載函式，其中 RT1 和 RT2 是未定型別：

```
template<typename T> RT1 test(typename T::X const*);    // (1)
template<typename T> RT2 test(...);                     // (2)
```

算式 test<int>對於第一個 function template 而言是無意義的，因爲 int 型別並沒有 member type X。然而第二個 function template 無此問題，因此算式 &test<int> 可確定出惟一一個函式位址。於是儘管第一個 template 以 int 替換失敗，卻沒有造成 &test<int> 隨之不合法。

正是「替換失敗並非錯誤」（substitution-failure-is-not-an-error; SFINAE）這一原則，使得 function templates 的重載實際可行。而且我們可以借助這個原則，創造出很多極出色的編譯期技術（compile-time techniques）。例如，假設型別 RT1 和 RT2 分別定義爲：

```
typedef char RT1;
typedef struct { char a[2]; } RT2;
```

我們可以在編譯期（也就是經由一個常數算式, constant-expression）判斷某給定型別 T 是否有 member type X：

```
#define type_has_member_type_X(T) \
        (sizeof(test<T>(0)) == 1)
```

爲理解上述巨集中的運算式，我們最好由外而內地分析它。首先，如果上頁的第一個 test template 被編譯器選用，sizeof 算式值等於 1（回返型別爲 char，大小爲 1）。如果第二個 template 被選用，則 sizeof 算式值至少爲 2（回返型別是個內含兩個 char 的 array）。換言之此巨集的功用是判別編譯器具現化 test<T>(0)時選中第一個 function template 或第二個。很明顯，如果給定的型別 T 不存在 member type X，第一個 function template 不會被選用（譯註：這時 sizeof 的值不爲 1）。然而如果給定的型別 T 存在 member type X，根據重載決議規則（見附錄 B），第一個 function template 會被優先選用（譯註：這時 sizeof 結果爲 1）；這是因爲重載決議規則會優先考慮「把 0 值轉換爲一個 null 指標」，而不考慮「把 0 值當作省略號（ellipsis）參數」（「省略號參數」是重載決議規則中最後才考慮是否匹配的參數形式）。第 15 章大量用到了類似技法。

上述的 SFINAE 原則只是協助你避免創造出非法型別，並不能防止非法算式（invalid expressions）被編譯器求值（*evaluated*）。因此下面例子是不合法的：

```
template<int I> void f(int (&)[24/(4-I)]);
template<int I> void f(int (&)[24/(4+I)]);

int main()
{
    &f<4>;   // ERROR：除數爲 0（不適用 SFINAE 原則）
}
//譯註：VC7.1 在此例中表現奇怪，它對編譯期運算式 24/(4-4)的求值結果居然是 1，這大概是某種
//古怪的錯誤防護機制作祟(除數爲 0 時認定商爲 1)。VC6/ICL 7.1/g++ 3.2 則給出合理的錯誤資訊。
```

例中的 &f<4>將導致編譯錯誤 — 即使第二個 template 被選用時不會出現「除數爲 0」的情況。這一類錯誤並不出現在「編譯器把算式值綁定（*bind*）至 template parameter」的時候，而是出現在「算式求值過程」中。下面例子是合法的：

```
template<int N>  int g() { return N;  }
template<int* P> int g() { return *P; }

int main()
{
    return g<1>();  // 1 無法適用於 int* 參數，然而 SFINAE 原則在這裡起了作用。
}
//譯註：VC7.1 和 VC6 均無法編譯此例。VC7.1 總是把 template argument 1 匹配至錯誤的
//重載函式上，VC6 則是無法區分兩個 g()，認爲兩者相同。g++ 3.2/ICL 7.1 無此問題。
```

關於 SFINAE 原則的進一步應用，請看 15.2.2 節, p.266 和 19.3 節, p.353。

8.3.2 Type Arguments（型別引數）

Template type arguments 是針對 template type parameters 而指定的「值」。我們慣用的大多數 types 都可以作爲 template arguments 使用，但有兩個例外：

1. local classes 和 local enum types（亦即宣告於函式定義內部的 classes 和 enums）不能做爲 template type arguments 使用。
2. 如果某個 type 涉及無名的（unnamed）class types 或無名的（unnamed）enum types，這樣的 type 不能做爲 template type arguments 使用；但如果運用 `typedef` 使其具名，便可用做 template type arguments。

下面例子展示了上述兩種例外情況：

```
template <typename T> class List {
    ...
};

typedef struct {
    double x, y, z;
} Point;          // typedef 使某個 unnamed type 有了名稱

typedef enum { red, green, blue } *ColorPtr;

int main()
{
    struct Association           //譯註：這是一個定義於函式內部的所謂 local type
    {
        int* p;
        int* q;
    };
    List<Association*> error1;   // ERROR：template argument 不能是個 local type
    List<ColorPtr> error2;       // ERROR：template argument 不能是個 unnamed type
    List<Point> ok;              // OK：原本無名的 type 因 typedef 而有了名稱
}
```

儘管通常各式各樣的 types 可被當做 template arguments 使用，但以那些 types 替換 template parameters 之後，其結果必須是個合法的 C++ 構件（constructs）：

```
template <typename T>
void clear (T p)
{
    *p = 0;      // unary operator* 必須可施行於 T 身上
}
```

```
int main()
{
    int a;
    clear(a);   // ERROR：int 並不支援 unary operator*
}
```

8.3.3 Nontype Arguments（非型別引數）

Nontype template arguments 是針對 nontype template parameters 而指定的「值」，它必須符合下列條件之一：

- 是另一個具有正確型別的 nontype template parameter。
- 是一個編譯期整數型（integer）或列舉型（enum）常數，但必須與對應的參數型別匹配，或者可以被隱式轉型爲該型別（例如 `int` 參數可以使用 `char` 值）。
- 以內建一元取址運算子（unary addressof operator&）爲前導的外部變數或函式。面對函式或 array 變數，可省略不寫 '&'。這一類 template arguments 匹配 *pointer* type nontype parameter。
- 如上所述但無前導 '&'。匹配的是 *reference* type nontype parameter。
- 一個 pointer-to-member 常數；亦即形如 `&C::m` 的運算式，其中 `C` 是個 class type，`m` 是個 non-static 成員（函式或變數）。它只匹配 *pointer-to-member* type nontype parameters。

當以一個 template argument 匹配一個 pointer type 或 reference type 的 template parameter 時，(1) 使用者自訂轉換（包括單引數建構式及轉型函式）以及(2) derived-to-base 轉換都不被編譯器考慮，即使它們在其他情境中是合法有效的隱式轉換。至於爲引數添加 `const` 飾詞或 `volatile` 飾詞是可以的。

下面例子中的 nontype template arguments 都合法：

```
template <typename T, T nontype_param>
class C;

C<int, 33>* c1;     // 整數型（integer type）

int a;
C<int*, &a>* c2;    // 外部變數的位址

void f();
void f(int);
C<void (*)(int), f>* c3;
                 // 一個函式名稱。重載決議規則在此選用 f(int)。& 會被隱寓加入。
```

```
class X {
public:
    int n;
    static bool b;
};

C<bool&, X::b>* c4; // static class members 都可接受

C<int X::*, &X::n>* c5;
                    // pointer-to-member 常數亦可接受

template<typename T>
void templ_func();

C<void (), &templ_func<double> >* c6;
                    // function template 具現體也是一種函式
```

譯註：VC7.1 無法通過此例中的 `c6` 宣告。它認爲`&templ_func<double>`在編譯期不可求值。g++ 3.2 認爲 `void()()`不是編譯期常數。只有 ICL7.1 能順利編譯此例。把 `void()`改爲 `void(*)()` 後，VC7.1 編譯通過，VC6 失敗依然。`c6` 的宣告語法應是正確的，見 *C++ standard* 14.1/8。

template arguments 的一個一般性約束條件（general constraint）是：必須能夠在編譯期或聯結期求值。「只在執行期才能求值」的運算式（例如某區域變數的地址）不能作爲 nontype template arguments 使用。

即使如此，可能令你感到驚奇的是，至少你目前還不能使用以下各種常數值：

- null pointer 常數
- 浮點數（floating-point numbers）
- 字串字面常數（string literals）

不能以字串字面常數作爲 nontype template arguments 的一個技術難題在於：兩個內容完全相同的字串字面常數可能存在兩個不同的位址上。一種稍顯笨拙的解法是引入一個字串 array：

```
template <char const* str>
class Message;

extern char const hello[] = "Hello World!";

Message<hello>* hello_msg;
```

注意這裡必須使用關鍵字 `extern`，因爲 `const` array 採用內部聯結（internal linkage）。

4.3 節, p.40 有這個問題的另一個例子。13.4 節, p.209 討論了 C++ 語言在這個問題上的未來可能變化。

這裡還有一些比較不那麼令人驚訝的非法實例：

```
template<typename T, T nontype_param>
class C;

class Base {
public:
    int i;
} base;

class Derived : public Base {
} derived_obj;

C<Base*, &derived_obj>* err1;   // ERROR：「derived-to-base 轉換」不被考慮

C<int&, base.i>* err2;          // ERROR：成員變數不被考慮

int a[10];
C<int*, &a[0]>* err3;           // ERROR：不能使用 array 內某個元素的位址
```

8.3.4 Template Template Arguments（雙重模板引數）

Template template argument 必須是這樣一個 class template：其參數完全匹配待替換之 template template parameter 的參數。Template template argument 的 default template argument（預設模板引數值）會被編譯器忽略，除非對應的 template template parameter 有預設引數。下面示範非法情況：

```
#include <list>
    // 宣告：
    // namespace std {
    //     template <typename T,
    //               typename Allocator = allocator<T> >
    //     class list;
    // }
```

```
template<typename T1,
         typename T2,
         template<typename> class Container>
                          // Container 要求只帶一個參數
class Relation {
public:
    ...
private:
    Container<T1> dom1;
    Container<T2> dom2;
};

int main()
{
    Relation<int, double, std::list> rel;
        // ERROR：std::list 擁有不止一個 template parameters
    ...
}
```

問題出在標準程式庫的 `std::list` template 擁有不止一個參數。第二參數（是個 allocator, 配置器）有預設值，但編譯器把 `std::list` 匹配至 `Container` 時，該預設值被忽略了。

有些時候這種問題可以繞道解決：只需要爲 template template parameter 加上一個帶有預設值的參數即可。對上述例子來說，我們可以把 template `Relation` 重寫爲：

```
#include <memory>

template<typename T1,
         typename T2,
         template<typename T,
                  typename = std::allocator<T> > class Container>
           // Container 現在可接受標準程式庫的 container templates 了
class Relation {
public:
    ...
private:
    Container<T1> dom1;
    Container<T2> dom2;
};
```

很明顯，這麼做並不完全令人滿意，但畢竟使我們得以運用標準程式庫的容器（container templates）。13.5 節, p.211 討論了 C++ 語言對這個問題的未來可能變化。

雖然從語法上說，你只能以關鍵字 `class` 來宣告一個 template template parameter，但這並不意味 template template argument 必須是 class type。事實上以關鍵字 `struct` 和 `union` 宣告的 templates 也都可以當作 template template parameters 的合法引數。這和我們先前講過的「任意型別都可作爲由關鍵字 `class` 宣告之 template type parameters 的合法引數」類似。

8.3.5 等價（Equivalence）

當兩組 template arguments 的元素一一對等時，我們稱這兩組引數等價。對於 type arguments，`typedef` 的名稱並不影響對比過程：最終被比較的是 `typedef` 所指代的 type。對於整數型 nontype arguments，比較的是引數值，與引數表達方式無關。下面例子闡釋了這個概念：

```
template <typename T, int I>
class Mix;

typedef int Int;

Mix<int, 3*3>* p1;
Mix<Int, 4+5>* p2;  // p2 和 p1 具有相同型別
```

一個由 function template 產生的函式，和一個常規函式，無論如何不會被編譯器視爲等價，即使它們的型別和名稱完全相同。這對 class members 造成兩個重要結果：

1. 由 member function template 產生的函式不會覆蓋虛擬（virtual）函式。
2. 由 constructor template 產生的建構式不會被當做 *default copy* 建構式。（同樣道理，由 assignment template 產生的 *assignment* 運算子不會被當做一個 *copy-assignment* 運算子。這個問題較少出現，因爲 *assignment* 運算子不像 *copy* 建構式那樣會被隱式呼叫。）

8.4 Friends

Friend 宣告式的基本概念很簡單：指定某些 classes 或 functions，讓它們可以對 friend 宣告式所在的 class 進行特權存取。但是下面兩個事實使這個概念複雜化了：

1. friend 宣告式可能是某一物體（entity）的惟一宣告（譯註：意思是 friend 僅在 class 內宣告，別無其他兄弟）。
2. friend 函式宣告可以就是其定義。

friend class 宣告式不能成爲一個定義式，這就大大降低了問題的發生。涉及 templates 時，惟一需要考慮的新增情況是：你可以把某個 class template 的特定具現體（實體）宣告爲 friend：

```
template <typename T>
class Node;

template <typename T>
class Tree {
    friend class Node<T>;
    ...
};
```

注意，在 class template 的某一實體（如上述 Node<T>）成爲其他 class 或 class template（如上述 Tree）的 friend 之前，class template Node 必須已被宣告並可見。但對常規 class 來說沒有這個限制。

```
template <typename T>
class Tree {
    friend class Factory;   // OK，即使這是 Factory 的首次宣告
    friend class Node<T>;   // ERROR(如果此前並未宣告 Node)
};
```

9.2.2 節, p.125 對此有更多討論。

8.4.1 Friend Functions

Function template 具現體（實體）可以成爲別人的一個 friend function，前提是該 function template 名稱之後必須緊跟著以角括號括起的引數列 — 如果編譯器可推導出所有引數，引數列可以爲空：

```
template <typename T1, typename T2>
void combine(T1, T2);

class Mixer {
    friend void combine<>(int&, int&);
                    // OK：T1 = int&, T2 = int&
    friend void combine<int, int>(int, int);
                    // OK：T1 = int, T2 = int
    friend void combine<char>(char, int);
                    // OK：T1 = char T2 = int
    friend void combine<char>(char&, int);
                    // ERROR：與 combine() template 不匹配
    friend void combine<>(long, long) { ... }
                    // ERROR：不能在此處定義函式
};
```

注意，我們無法定義一個 template 實體（*instance*），最多只能定義一個特化體（*specialization*）。因此一個「令某實體獲得名稱」的 friend 宣告式，不能夠是個定義式。

譯註：作者的意思是，template 實體（*instances*）只能由編譯器產生，不能由程式員定義；程式員所定義的，稱爲特化體（*specializations*）。見 p.88。

如果 friend 名稱之後沒有跟著角括號，有兩種可能：

1. 如果這不是個資格修飾名稱（亦即不含::），就絕不會引用某個 template 實體。如果編譯器無法在 friend 宣告處匹配一個 non-template function，則這個 friend 宣告便被當作這個函式的首次宣告。這個宣告式也可以是個定義式。
2. 如果這是一個資格修飾名稱（亦即含有::），它就必定引用一個先前已定義的 function 或 function template。編譯器會優先匹配常規(non-template)函式，然後才匹配 function templates。這個 friend 宣告式不能是個定義式。

以下例子有助於理解各種可能情況：

```
void multiply (void*);      // 常規函式（ordinary function）

template <typename T>
void multiply(T);           // function template

class Comrades {
   friend void multiply(int) {}
                    // 以上定義了一個新函式 ::multiply(int)

   friend void ::multiply(void*);
                   // 以上引用先前定義的常規函式，而非 multiply<void*> 實體

   friend void ::multiply(int);
                   // 以上引用 template 的一個實體

   friend void ::multiply<double*>(double*);
                   // 資格修飾名稱也可以帶角括號，但此時編譯器必須見到該 template

   friend void ::error() {}
                   // ERROR：帶資格修飾的 friend，不能是個定義
};
```

先前數個例子中，我們將 friend function 宣告在常規 class 內。如果把 friend 宣告於 class templates 中，先前所說的規則也全部適用，而且 template parameter 可參與到 friend function 之內：

```
template <typename T>
class Node {
    Node<T>* allocate();
    ...
};

template <typename T>
class List {
    friend Node<T>* Node<T>::allocate();
    ...
};
```

然而把 friend function 定義於 class template 中可能會引發一個有趣的錯誤，因爲任何只在 template 中被宣告的 object，都是直到 template 被具現化後才能成爲具體實物（concrete entity）。考慮下面例子：

```
template <typename T>
class Creator {
    friend void appear()
                  // 定義一個新函式::appear()，但是只有當 Creator 被具現化它才存在
        ...
    }
};

Creator<void> miracle;  // ::appear()此時被生成
Creator<double> oops;   // ERROR：試圖再次生成::appear()
```

在這個例子中，兩個不同的具現體產生出兩個完全相同的定義，這直接違反了 ODR 原則（見附錄 A）。

因此，我們必須確保 class template 的 template parameters 出現在「定義於該 template 之內的所有 friend function」的型別之中（除非我們想要阻止這個 class template 在一個檔案中被多次具現化，但沒有什麼人會這樣做）。現在讓我們對先前的例子進行一些改動：

```
template <typename T>
class Creator {
    friend void feed(Creator<T>*){    // 不同的 T 會產生不同的::feed 函式定義
        ...
    }
};
Creator<void> one;          // 產生::feed(Creator<void>*)
Creator<double> two;        // 產生::feed(Creator<double>*)
```

在這個例子中，每一個 Creator 具現體會產生一個不同的函式。注意，雖然這些函式是在 template 具現化過程中產生的，但它們仍是常規函式，不是某個 templates 的實體。

另外請注意，由於這些函式被定義於 class 定義式內，因此它們暗自成爲 inline。而且如果你在兩個不同的編譯單元內產生同一個函式，編譯器不會認爲錯誤。9.2.2 節, p.125 和 11.7 節, p.174 對此問題有更多論述。

8.4.2 Friend Templates

通常，當我們定義一個 friend 而它是個 function 或是個 class template 時，我們可以明確指定以哪一個物體（entity）做爲 friend。但有時候把一個 template 的所有實體都指定爲某個 class 的 friend 也相當有用。這是經由一種所謂的 friend template 機制實現的。例如：

```
class Manager {
    template<typename T>
        friend class Task;
    template<typename T>
        friend void Schedule<T>::dispatch(Task<T>*);
    template<typename T>
        friend int ticket() {
            return ++Manager::counter;
        }
    static int counter;
};
```

和常規的 friend 宣告式一樣，只有當 friend template 產生一個無修飾函式名（unqualified function name），而且該名稱之後不緊跟著角括號，這個 friend template 才可以是一個定義式。

Friend template 只能宣告 primary templates 及 primary templates 的成員。任何與該 primary template 相關的偏特化體（partial specializations）和明確特化體（explicit specializations）都將自動被編譯器視爲 friends。

8.5 後記

C++ templates 的一般概念和語法，自從出現於 1980 年代晚期就基本固定了。Class templates 和 function templates、type parameters 和 nontype parameters 等概念都是 template 最初機制的一部份。

然而由於 C++ 標準程式庫的需求所帶來的策動，template 原始設計發生了一些重大增建。Member templates 是這些重大增建中最基礎的部份。有趣的是，只有 member function templates 被正式票選決議爲 *C++ Standard* 的一部份，但由於一個文字上的疏漏，member class templates 也成爲 *C++ Standard* 的一部份。

譯註：C++ 標準委員會只投票通過加入 member function templates，C++ 標準檔案也隨之做出增補。然而這部份文字寫得並不嚴謹，使得人們誤以爲 member class templates 也被加入 C++ *Standard*。之後不久，某些標準程式庫也在實作中使用了 member class templates。這使得這個誤會更鑿實，人們再也無法把它從語言標準規格中剔除。

Friend templates、default template arguments 和 template template parameters 是晚近才加入 C++ 語言陣營的。「宣告 template template parameters」的能力有時稱爲更高級泛型（higher-order genericity），它們最初引入 C++ 是爲了支援某個配置器模型（allocator model），但之後這項任務就被另一個並不依賴 template template parameters 特性的模型所取代。由於缺乏一個完備規格，這一特性在標準化工作後期幾乎要被從標準規格中去掉。最終是標準委員會的多數成員投票通過，留下了這些特性，並完備其規格。

9

Templates 裏的名稱

Names in Templates

名稱是絕大多數編程語言最基礎的概念。程式員可以通過名稱來引用先前建構的物體（entities）。當 C++ 編譯器碰到一個名稱，它必須搜尋這個名稱引用的是哪個物體。從編譯器實作者的角度來看，C++ 名稱體系非常複雜。試著考慮述句 `x*y;`，如果 `x` 和 `y` 是變數名稱，則這個述句表示一個乘法；但如果 `x` 是個型別名稱，則這個述句宣告一個指標 `y`，指向「型別爲 `x` 的物體」。

這個小例子說明 C++（以及 C）是一種所謂「前後脈絡敏感」（context-sensitive）的語言：如果沒有前後脈絡，就不可能知道某一構件（construct）的具體含義。這和 templates 有何關係呢？唔，templates 是一種「涉及前後脈絡更多更廣」的構件，這些前後脈絡包括：(1) templates 出現的位置和方式；(2) templates 被具現化的位置和方式；(3)「用以具現化 templates」的那些 template arguments 的前後脈絡。因此，如果我說在 C++ 中必須小心處理名稱，你應該不會感到驚訝。

9.1 名稱分類學（Name Taxonomy）

C++ 以各式各樣的方法來對名稱分類。爲幫助你理解如此多的術語，這裡提供表 9.1 和表 9.2，描述了這些分類。幸運的是，如果你能夠熟悉兩個最主要的命名概念，就能夠洞察大多數 C++ template 問題：

1. 當一個名稱被稱爲「受飾名稱」（qualified name）時，其含義是：以 *scope resolution* 運算子（`::`）或 *member access* 運算子（`.` 或 `->`）指明該名稱所屬作用域。例如 `this->count` 是一個受飾名稱，`count` 不是（即使兩者等價）。
2. 當一個名稱被稱爲「受控名稱」（dependent name）時，其含義是：它在某些方面受控（倚賴）於某個 template parameter。如果 `T` 是個 template parameter，`std::vector<T>::iterator` 就是一個受控名稱；但如果 `T` 是個已知型別（例如 `int`），`std::vector<T>::iterator` 是一個非受控名稱（nondependent name）。

分類	解釋和注意
Identifier 標識符號	一個由不間斷的字母、底線和數字組成的序列，不能以數字開頭。某些標識符號保留給編譯器使用，應用程式中無法引用它們（通則之一是：避免以底線或雙底線作爲標識符號的前導）。字母概念較廣，包括特殊的 universal character names（UCNs），亦即「非字母語言」的字型編碼集。
Operator-function-id 運算子函式 標識符號	關鍵字 `operator` 後跟一個「表示某項操作」的符號。例如 `operator new` 和 `operator []`。很多運算子有另一種表示法。例如 `operator &` 和 `operator bitand` 等價（即使做爲一元取址（*address-of*）運算子也如此）。
Conversion-function-id 轉型函式 標識符號	用來表示使用者自定之隱式轉型函式。例如 `operator int&`，它有另一種易招困惑的寫法：`operator int bitand`。
Template-id 模板 標識符號	Template 名稱之後跟著以角括號括起的 template argument list。例如 `List<T,int,0>`。嚴格說來 *C++ Standard* 只允許 template-id 的 template 名稱使用簡單標識符號，這大概是個疏漏。實際上應該也允許使用 Operator-function-id，例如 `operator+< X<int> >`。
Unqualified-id 非受飾 標識符號	這是標識符號的泛化。可以是上述名稱中的任意一個，或是一個「解構式名稱」（例如`~Data` 或 `~List<T,T,N>`）。
Qualified-id 受飾 標識符號	當一個 unqualified-id 被 class 名稱或 namespace 名稱或 global scope operator 修飾後，便成爲 qualified-id。注意，class 名稱或 namespace 名稱本身也可以帶修飾。下面是一些例子： `::X`, `S::x`, `Array<T>::y`, `::N::A<T>::z`。
Qualified name 受飾 名稱	這個術語在 *C++ Standard* 中並無定義，但我們用它表示那些經歷了所謂「受飾查詢」（*qualified lookup*）的名稱。具體而言這是指一個在 member access operator（`.`或`->`）之後的 qualified-id 或 unqualified-id。例如 `S::x`, `this->f` 和 `p->A::m`。然而即使 `class_mem` 在上下脈絡（context）中隱式等價於 `this->class_mem`，它也不是一個受飾名稱，因爲 member access operator 必須明確出現。
非受飾 名稱 Unqualified name	一個並非 qualified name 的 unqualified-id。這並不是一個標準術語。我們用它來表示那些經歷了所謂「非受飾查詢」（*unqualified lookup*）」的名稱。

表 9.1 名稱分類表（第一部份）

分類	解釋和注意
Name 名稱	既可以是個受飾名稱, qualified name，也可以是個非受飾名稱, unqualified name。
Dependent name 受控名稱	它在某些方面受控（倚賴）於某個 template parameter。顯然，任何包含 template parameter 的受飾名稱或非受飾名稱都是一種受控名稱。一個以 member access operator（.或->）修飾的受飾名稱，如果左側內容倚賴某個 template parameter，便可視爲受控名稱。具體而言，如果 template 程式碼中出現 `this->b`，那麼 `b` 是一個受控名稱。最後，出現於呼叫型式 `ident(x,y,z)` 中的標識符號 `ident` 是個受控名稱 — 若且惟若任一引數型別倚賴某個 template parameter。
Nondependent name 非受控名稱	只要不符合上述受控名稱（dependent name）所描述的，便是此類。

表 9.2 名稱分類表（第二部份）

仔細閱讀上述表格內容，熟悉其中所列術語，會對你帶來幫助。這些術語偶而會出現在 C++ template 相關主題之中。但是沒有太大必要去牢記每個術語的精確含義。當你需要了解某個術語的意義時，你可以從索引中輕易查到它。

9.2 名稱查詢（Looking Up Names）

C++ 語言的名稱搜尋機制涉及極多細節，我們只集中在數個主要概念上。眾多細節只是爲了保證：(1) 通常情況下這些查詢機制符合人們的直覺；(2) 特別複雜的情況可以在 *C++ Standard* 中找到解答。

受飾名稱（qualified names）的查詢範圍是在其「修飾構件所意味的作用域」（the scope implied by the qualifying construct）內。如果該作用域是個 class，則其 base class 也將被查詢。然而編譯器查詢受飾名稱時並不考慮其「圈封作用域」（enclosing scopes）。下面的例子闡示這個基本法則：

```
int x;

class B {
public:
    int i;
};

class D : public B { };
```

```
void f(D* pd)
{
    pd->i = 3;  // 編譯器找到 B::i。（譯註：pd->i 是一個 qualified name）
    D::x = 2;   // ERROR：在 D 作用域（包括 B 作用域）中找不到 x
}
```

與之對比的是，編譯器通常會依次在「更外層的圈封作用域」（more enclosing scopes）中查詢非受飾名稱（unqualified names）— 儘管在成員函式定義式內編譯器會先查詢 class 作用域和 base classes 作用域，然後才是其他圈封作用域。這便是所謂的 ***ordinary lookup***（常規查詢）。下面例子展示 *ordinary lookup* 的基本概念：

```
extern int count;               // (1)

int lookup_example(int count)   // (2)
{
    if (count < 0) {
        int count = 1;          // (3)
        lookup_example(count);  // 非受飾的 count 代表的是(3)
    }
    return count + ::count;     // 第一個 count(非受飾) 代表(2)，
                                // 第二個::count(受飾) 代表(1)。
}
```

在 *ordinary lookup*（常規查詢）之外還有一種作法用來查詢非受飾名稱（unqualified names）。這種機制有時稱爲 *argument-dependent lookup*（ADL；「相依於引數」的查詢）[23]。深入 ADL 細節之前，我先介紹一個引發此機制的例子：

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}
```

現在假設我們需要把這個 template 應用到定義於另一個 namespace 內的型別身上：

```
namespace BigMath {
    class BigNumber {
        ...
    };
    bool operator< (BigNumber const&, BigNumber const&);
    ...
}
```

[23] 這種機制也稱爲 *Koenig lookup*（或 extended Koenig lookup），以 Andrew Koenig 命名。他首先提出了這個機制的一個變體。

```
using BigMath::BigNumber;

void g (BigNumber const& a, BigNumber const& b)
{
    ...
    BigNumber x = max(a,b);
    ...
}
```

這裡的問題在於，`max()` template 對 `BigMath` namespace 一無所知，而 *ordinary lookup*（常規查詢）機制無法找到一個 `operator<` 可施行於 `BigNumber` 型別。如果沒有某種特殊規則，這個問題就大大縮減了 templates 在 C++ namespaces 情形下的應用性。C++ 對這個問題的解決辦法就是 ADL。

9.2.1「相依於引數」的查詢（Argument-Dependent Lookup, ADL）

ADL 只適用於這樣的非受飾名稱（unqualified names）：在函式呼叫動作中用到的一個非成員函式名稱。如果 *ordinary lookup* 可找到一個成員函式名稱或一個型別名稱，編譯器就不啓動 ADL。如果被呼叫函式的名稱被寫進一對小（圓）括號內，ADL 也不起作用。

否則，如果名稱之後跟著的是一個以小（圓）括號括起的引數算式列（argument expressions list），ADL 就會在「與所有 call arguments 型別相關聯」的每一個 namespace 和 class 中查詢該名稱。所謂「相關聯的 namespaces 和 classes」的精確定義將在稍後給出，但你可以直觀認爲，它們與給定之型別有十分直接的關係。假設給定的型別是個指標，指向 class `X`，編譯器便認爲 `X` 以及 `X` 隸屬的所有 classes 和 namespaces 都與該指標型別相關聯（*associated*）。

下面是「與 call arguments 型別相關聯之 namespaces 集和 classes 集」的精確定義：

- 對內建型別而言：空集合。（譯註：此條款在實際運用時有些例外情況，請參考 10 章, p.147）
- 對 pointer 型別和 array 型別而言：基礎構成型別（underlying type；亦即 pointer 所指型別或 array 元素型別）。
- 對 enum 型別而言：宣告所在之 namespace。如果其成員是 classes，那麼圈封類別（enclosing class）就是其關聯類別（associated class）。
- 對 class（含 union）型別而言：「相關聯之 class 集」包括(1) class 或 union 自身、(2) 其圈封類別（enclosing class）、以及 (3) 任何直接或間接的 base classes。「相關聯之 namespace 集」則包括所有「相關聯之 classes」宣告所在的 namespaces。如果我們所討論的這個 class 是個 class template 具現體，那麼所謂「相關聯之 class 集」和「相關聯之 namespace 集」就還包括 (1) 每一個 template type argument 型別，以及 (2) 每一個 template template arguments 宣告所在的 classes 和 namespaces。

- 對 function 型別而言：所有「與參數型別和回返值型別相關聯」的 classes 和 namespaces。
- 對 pointer-to-member-of-class-X 型別而言：所有「與 X 相關聯」以及「與被指向之成員相關聯」的 classes 和 namespaces。如果我們所討論的是個 pointer-to-member-function 型別，那麼就還包括所有「與參數和回返值相關聯」的 classes 和 namespaces。

ADL 會在所有「相關聯的 namespaces」中查詢名稱，就像該名稱被所有這些 namespaces 修飾過一樣（只不過略去 `using` 指令罷了）。下面例子展示了 ADL 過程：

```
// details/adl.cpp

#include <iostream>

namespace X {
    template<typename T> void f(T);
}

namespace N {
    using namespace X;
    enum E { e1 };
    void f(E) {
        std::cout << "N::f(N::E) called" << std::endl;
    }
}

void f(int)
{
    std::cout << "::f(int) called" << std::endl;
}

int main()
{
    ::f(N::e1);  // 受飾（qualified）函式名稱：不使用 ADL。
    f(N::e1);    // ordinary lookup 找到::f，而 ADL 找到 N::f()，編譯器會優先考慮後者
}
```

注意這個例子，當編譯器動用 ADL 機制時，namespace `N` 中的 `using` 指令被忽略。因此 `main()` 中的 `f()` 不會被編譯器認爲是對 `X::f()` 呼叫。

9.2.2 Friend 名稱植入（Friend Name Injection）

friend function 宣告式可以是該函式的第一份宣告式。這種情況下，該函式會被視爲宣告於封住 class X（內含該 friend 宣告）的「最內層 namespace（有可能是 global namespace）」作用域中。這個宣告在「接受其植入」的作用域內是否可見，是人們經常爭論的問題。這個問題很大程度是 templates 帶來的。考慮下面例子：

```
template<typename T>
class C {
    ...
    friend void f();
    friend void f(C<T> const&);
    ...
};

void g (C<int>* p)
{
    f();    // 此處是否可見 f()？
    f(*p);  // 此處是否可見 f(C<int> const&)？
}
```

問題在於，如果 friend 宣告式在其圈封之命名空間（enclosing namespace）中可見，那麼當我們具現化一個 class template 時會造成常規函式（ordinary function）的宣告也可見。有些程式員會對此感到驚訝，因此 *C++ Standard* 規定 friend 宣告式不得造成其常規函式名稱在圈封作用域（enclosing namespace）中可見。

然而，有一個有趣的編程技術，依靠「只在 friend 宣告式中宣告或定義函式」來實現（見 11.7 節, p.174）。因此 *C++ Standard* 又規定，當函式的 friend class 是符合 ADL 規則之「相關聯 classes」中的一個時，該 friend function 可見。

讓我們重新考慮上面的例子。由於 `f()` 呼叫中沒有引數，因此它沒有相關聯的 classes 或 namespaces，所以上例中的它是個非法呼叫。然而 `f(*p)` 確實與 class `C<int>` 相關聯（因爲後者是`*p` 的型別），也與 global namespace 相關聯（因爲`*p` 的型別宣告於 global namespace），於是由於 class `C<int>` 在此呼叫動作發生之前就已被具現化，因此第二個 friend function 宣告式是可見的。爲確保這一點，一個會造成「在相關聯之 classes 內查詢 friends」的函式呼叫動作，會引發「相關聯之 classes」被具現化（如果當時它還未被具現化的話）[24]。

[24] *C++ Standard* 實際上保證這一點，但是並沒有明確寫出。

9.2.3 植入 Class 名稱（Injected Class Names）

Class 名稱會被「植入」class 自身作用域中，因此你可以在該作用域中透過未受飾名稱（unqualified name）的形式使用該名稱。你不能透過受飾名稱（qualified name）來存取它，因爲受飾名稱在語法上被用來表示該 class 的建構式。舉個例子：

```
// details/inject.cpp

#include <iostream>

int C;

class C {
private:
    int i[2];
public:
    static int f() {
        return sizeof(C);
    }
};

int f()
{
    return sizeof(C);
}

int main()
{
   std::cout << "C::f() = " << C::f() << ","
             << " ::f() = " << ::f() << std::endl;
}
```

成員函式 `C::f()` 傳回型別 `C` 的大小，`::f()` 傳回變數 `C` 的大小，兩者都表示一個 `int` object 的大小。

Class templates 也會內植 class 名稱，但和常規的「內植 class 名稱」比起來有點特別：它們的後面可以跟著 template arguments（此時稱爲「內植 class template 名稱」）。如果後面不跟著 template arguments，代表的是「以參數做爲引數（對 class template 偏特化來說則是做爲特化用引數）」的 class。下面的例子可以解說這些規則：

```
template<template<typename> class TT> class X {
};

template<typename T> class C {
  C* a;          // OK：與 C<T>* a; 相同
  C<void> b;     // OK
  X<C> c;        // ERROR：不帶 template argument list 的 C 並不象徵一個 template
  X<::C> d;      // ERROR：<: 是 [ 的另一種寫法
  X< ::C> e;     // OK：有必要在 < 和 :: 之間加入空格
};
```

注意這個例子中的非受飾名稱（unqualifited name）代表的是「被內植名稱」（injected name）。如果它後面不跟著 template argument list，編譯器不認爲它是 template 名稱。爲了彌補這個問題，我們可以使用 scope qualifier `::`（作用域修飾符號）修飾之，強迫 template 名稱被找到。這麼做確實可行，但我們必須小心不要寫出「雙字元語彙單元」（digraph token）`<:`，因爲它會被編譯器解釋爲一個 `'['`。雖然你可能很少使用這種寫法，但萬一出現這種錯誤，恐怕很難診斷。

9.3 解析（Parsing）Templates

大多數編程語言編譯器會進行兩項基礎任務：(1) ***tokenization***（語彙單元化），又稱掃描（scanning）或律法分析（lexing）；(2) ***parsing***（詞法解析）。*tokenization* 會把源碼讀入一個字元序列（characters sequence），再由其中產生一個語彙單元序列（tokens sequence）。舉個例子，當編譯器看到字元序列 `int* p = 0;`，tokenizer（語彙單元建立器）會產生如下的語彙單元：一個關鍵字 `int`，一個運算子（或符號）`*`，一個標識符號（identifier）`p`，一個運算子（或符號）`=`，一個整數字面常數 `0`，和一個運算子（或符號）`;`。

之後，*parser*（詞法解析器）會在語彙單元序列（tokens sequence）中搜尋已知的 patterns（樣式），作法是遞迴縮減 token，或是把先前搜尋到的 patterns 組成更高級別的構件（high level constructs）。例如「token `0`」是個合法算式，「`*` 後跟一個標識符號 `p`」是合法的宣告符號，其後再跟一個 `'='`，而後再跟算式 `'0'`，仍然是個合法宣告符號。最後，關鍵字 `int` 是個已知型別，當它後跟一個 `*p=0` 宣告符號時，就得到了 `p` 的初始化宣告式。

9.3.1 Nontemplates 的前後脈絡敏感性（Context Sensitivity）

正如你所知道或所猜想的那樣，*tokenizing* 要比 *parsing* 容易。幸運的是 *parsing* 理論基礎已經相當穩固，很多語言都可以根據這個理論不困難地進行 *parsing*。這個理論對於「前後脈絡無關（context-free）」的語言最有效，然而我們先前已經說過，C++ 是一個前後脈絡敏感的語言。爲了進行 *parsing*，C++ 編譯器把一個符號表（symbol table）結合於 *tokenizer* 和 *parser* 身上：當某個宣告被成功解析（*parsed*）後，便進入符號表中。當 *tokenizer* 找到一個標識符號（identifier），便查詢符號表，如果在其中找到對應（同名）型別，就將 *resulting token* 標註出來。

舉個例子。如果 C++ 編譯器看到

```
x*
```

於是 *tokenizer* 查詢 x。如果在符號表中找到型別 x，*parser* 會看到

```
identifier, type, x
symbol, *
```

於是推斷這是一個宣告的開始。然而如果 x 不是型別，那麼 *parser* 會從 *tokenizer* 獲得：

```
identifier, nontype, x
symbol, *
```

於是這一構件（construct）就只能被合法解析爲一個乘法操作。這些原則的細節取決於具體實作策略，但要旨不變。

下面這個例子說明「前後脈絡敏感性」（context sensitivity）。考慮以下運算式：

```
X<1>(0)
```

如果 x 是個 class template 名稱，這個運算式便是將整數 0 轉型爲 x<1> 所指涉的型別。但如果 x 不是個 template，這個運算式等價於：

```
(X<1)>0
```

換句話說 x 和 1 的比較結果（true 或 false）被隱式轉型爲 1 或 0，然後再和 0 比較。雖然這種寫法很少見，但它確實是合法的 C++ 程式碼（也是合法的 C 程式碼）。只有當 C++ *parser* 看到一個 template 名稱時，它才會認爲其後跟隨的< 是個角括號，否則會認爲那是個「小於」符號。

這種「前後脈絡敏感性」，是當初以角括號作爲 template argument list 界定符號的不幸產物。下面是另一個例子：

```
template<bool B>
class Invert {
public:
    static bool const result = !B;
};

void g()
{
    bool test = Invert<(1>0)>::result;  // 小（圓）括號必須存在！
}
//譯註：VC6 對 nontype template parameters 支援不完善，
//      它把 = !B（此例等價於 = 0）理解爲 pure specifier。
```

如果 `Invert<(1>0)>` 中的小括號省略，第一個「大於」符號會被編譯器誤認爲是 template argument list 的結束符號，於是編譯器認爲這是個非法算式（等價於 `((Invert<1>))0>::result`）[25]。

tokenizer 也會碰到角括號的問題。前面我們已經提醒過（見 3.2 節, p.27），在嵌套的（nested）template-ids 之間應該加入空格：

```
List<List<int> > a;
             //^-- 這裡的空格是必要的
```

兩個右角括號之間的空格是必要的：如果沒有空格，兩個緊鄰的 `>` 會合成一個右移運算子 `>>`，不會被當作兩個獨立的 `>`。這便是所謂 *maximum munch tokenization principle*（語彙單元化之最大吞入原則）的結果：C++ 編譯器必須使一個 token（語彙單元）所含的連續字元儘可能地多。

這個問題使得很多 template 初學者相當困惑。因此有些 C++ 編譯器實作品做了修改，俾能夠在特定情況下將 `>>` 當成兩個獨立的 `>`（同時給出「不符合 *C++ Standard*」的警告）。C++ 標準委員會也正在考慮把此一行爲加入 *C++ Standard* 的某個修訂版本中（見 13.1 節, p.205）。

「最大吞入原則」的另一個例子較少引人注意：*scope resolution* 運算子如果和角括號連用，也必須小心：

```
class X {
    ...
};

List<::X> many_X;    // 語法錯誤！
```

問題在於，字元序列 `<:` 是個所謂的「雙字元語彙單元」（*digraph*）[26]，是 `[` 的另一種寫法。編譯器實際處理的將是 `List[:X> many_X;`，而這不具合法意義。這個問題同樣可以透過空格來解決：

```
List< ::X> many_X;
   //^-- 這裡的空格是必要的
```

[25] 注意，這裡使用了雙重括號以避免 *parser* 認爲 `(Invert<1>)0` 是個轉型操作（這是另一個可能的語法歧義）。

[26] 由於國際間某些語種鍵盤無法輸入某些字元如 `#`, `[`, `]` 等等，因此 C++ 加入了「雙字元語彙單元」（digraphs）。

9.3.2 型別的受控名稱（Dependent Names）

Templates 內的名稱，其問題在於：它們往往不具備足夠的確定性。更明確地說，一個 template 不能窺見另一個 template 內部，因爲後者的內容可能因爲明確特化（explicit specialization）而變得不合法（細節見第 12 章）。下面的例子可以說明這一點：

```
template<typename T>
class Trap {
public:
    enum { x };         // (1) 在這裡，x 並不是型別
};

template<typename T>
class Victim {
public:
    int y;
    void poof() {
        Trap<T>::x*y;   // (2) 這是一個宣告還是一個乘法運算？
    }
};

template<>
class Trap<void> {      // 「惡意」特化
public:
    typedef int x;      // (3) 在這裡，x 是個型別
};

void boom(Victim<void>& bomb)
{
    bomb.poof();
}
```

當編譯器對(2)進行 *parsing* 時，它必須確定(2)是個宣告還是個乘法，而這取決於「受控受飾名稱」（dependent qualified name）Trap<T>::x 是不是一個型別名稱。你可能會想去 template Trap 裡頭查看，於是從(1)得知 Trap<T>::x 不是型別，這就使我們認爲(2)是個乘法。然而稍後的程式碼打破了這種想法：令 T 爲 void，將泛化的 Trap<T> 加以特化，於是 Trap<T>::x 事實上成了 int 型別。

C++ 語言對此問題做出了明確的規定：通常一個「受控受飾名稱」（dependent qualified name）並不指涉某個型別，除非該名稱以關鍵字 typename 爲前導。如果在 template arguments 替換過程之後，該名稱最終並不是個型別名稱，則編譯器認爲程式不合法，並在具現期（instantiation time）報錯。注意此處的 typename 關鍵字並非用來指涉一個 template type parameter 參數，因此它和 type parameter 不同，你不能把 typename 換以 class。下列情況的名稱必需加上 typename 前導：

1. 該名稱在 template 中出現。
2. 該名稱是個受飾名稱（qualified name）。
3. 該名稱不被用於 base class list，也不被用於建構式的成員初值列（member initializers）。

> 譯註：上述說的 base class list 是指這種情況：
>
> ```
> template <typename T>
> class Derived : Base1<typename T::x>, Base2<typename T::y> {
>     // Base1 和 Base2 之前不能加 typename，
>     // 但是當 x 和 y 都是型別時，T::x 和 T::y 之前必須加 typename
>     ...
> };
> ```

4. 該名稱受控（依賴）於某個 template parameter。

除非前三種情況都成立，否則你不能使用 typename 前導詞。考慮下面例子[27]：

```
template<typename1 T>
struct S: typename2 X<T>::Base {
    S(): typename3 X<T>::Base(typename4 X<T>::Base(0)) {}
    typename5 X<T> f() {
        typename6 X<T>::C * p;          // 宣告指標 p
        X<T>::D * q;                    // 乘法
    }
    typename7 X<int>::C * s;
};

struct U {
    typename8 X<int>::C * pc;
};
```

每一處 typename 不論正確與錯誤，都加了下標以便區分。第 1 個 typename 表示 T 是個 template parameter，先前的規則不適用於這裡。第 2 和第 3 個 typename 錯誤，因爲根據前述第三條規則，在 base classes list 或 member initialization list 中不能使用 typename。第 4 個 typename 是必要的，因爲這裡的 base class 名稱並無「被初始化」或「被繼承」的含義，而是用來以引數 0 建構一個暫時的 X<T>::Base（你也可以認爲這是一種轉型）。第 5 個 typename 錯誤，因爲 X<T> 不是個受飾名稱（qualified name）。第 6 個 typename 所在述句如果是個指標宣告，那麼 typename 是必需的；其下一行沒有使用關鍵字 typename，編譯器因而認爲那是個乘法。第 7 個 typename 可有可無，因爲它同時滿足了前三條規則。第 8 個 typename 錯誤，因爲它沒有被用在 template 內。

[27] 這個例子來自 [VandevoordeSolutions]，證明 C++ 的確助長程式碼的復用（code reuse）。

9.3.3 Templates 的受控名稱（Dependent Names）

當 template 之中有個受控名稱（*dependent name*）時，你可能會碰到和前面類似的問題。通常如果 < 緊跟於一個 template 名稱之後，C++ 編譯器會認爲 < 代表一個 template argument list 的開始；其他情況下編譯器會認爲 < 是個「小於」符號。這就好像面對型別名稱時，編譯器必須假設受控名稱（*dependent name*）並不表示某個 template，除非你以關鍵字 template 提供額外資訊：

```
template<typename T>
class Shell {
public:
    template<int N>
    class In {
    public:
        template<int M>
        class Deep {
        public:
            virtual void f();
        };
    };
};

template<typename T, int N>
class Weird {
public:
    void case1(typename Shell<T>::template In<N>::template Deep<N>* p) {
        p->template Deep<N>::f();  // 抑制 virtual call
    }
    void case2(typename Shell<T>::template In<N>::template Deep<N>& p) {
        p.template Deep<N>::f();   // 抑制 virtual call
    }
};
```

這個例子有些複雜，它闡示在「可用以修飾（qualify）名稱」的所有運算子（::, ->, .）之後，可能需要加上關鍵字 template。當修飾運算子（qualifying operator）之前的「名稱或算式的型別」受控於某個 template parameter，而修飾運算子（qualifying operator）之後的名稱是個 template-id（亦即 template 名稱再加上以角括號括起來的 template argument list）時，就必須這麼辦。例如以下算式：

```
p.template Deep<N>::f()
```

p 的型別受控（取決）於 template parameter T。C++ 編譯器無法知道 Deep 是否是個 template，我們必須透過 template 前導詞才能指明 Deep 是個 template 名稱，否則 p.Deep<N>::f() 會被

解析（*parsed*）爲`((p.Deep)<N)>f()`。另請注意，對一個受飾名稱（qualified name），有時你可能需要添加多個 template 前導詞，因爲飾詞（qualifier）本身也可能被一個受控飾詞（dependent qualifier）修飾（前例的 `case1` 和 `case2` 兩宣告就屬於這種情況）。

如果在這些情況下不寫 `template` 關鍵字，左角括號和右角括號就被爲編譯器視爲「小於」和「大於」符號。然而如果某處並非絕對需要這個關鍵字，那就是不需要[28]。你不能到處噴灑（濫用）`template` 飾詞。

9.3.4 `using` 宣告式的受控名稱（Dependent Names）

`using` 宣告式可以把 namespaces 和 classes 內的名稱帶入當前作用域。這裡不考慮 namespaces 的情況，因爲並不存在所謂 namespace templates。當你使用 `using` 宣告式帶入 classes 內的名稱時，可以把 derived class 內的名稱帶入 base class。你可以把這一類 `using` 宣告式看成「爲 derived class 建立一個通往 base class 的符號連結（symbolic links）或捷徑（shortcuts）」，如此一來就允許 derived class 的成員可以存取 base class 成員，就好像那些 base class 成員宣告於 derived class 一樣。下面這個 non-template 的小例子可以很好地說明這個問題：

```
class BX {
public:
    void f(int);
    void f(char const*);
    void g();
};

class DX : private BX {
public:
    using BX::f;
};
```

上例中的 `using` 宣告式把 base class `BX` 內的名稱 `f` 帶入 derived class `DX`。此時的名稱 `f` 與兩個宣告相關聯，這正強調了這種機制帶入的是一組名稱，而不是一組符合該名稱的個別宣告。注意這種 `using` 宣告式可以使 derived class 存取它原本無權存取的 base class 成員。本例的 base class `BX`（及其成員）對 class `DX` 來說是 private，但 `BX:f` 函式被引入成爲 `DX` 的 public 介面，於是 class `DX` 使用者也能取用 `BX:f` 函式。由於 `using` 機制具備這個機能，先前的「存取宣告」（access declaration）在 C++ 中便作廢了（未來 C++ 修訂版可能不會再支援這種機制）：

[28] C++ *Standard* 文本對此問題並沒有明確規定，但是文本撰寫者似乎認同這一點。

```
class DX : private BX {
public:
    BX::f;  // 存取宣告（access declaration）語法被廢除，應以 using BX::f 代替
};
```

現在你或許察覺到了，使用 using 宣告式將一個受控（dependent）class 內的名稱帶入當前作用域，會造成什麼問題。儘管我們知道某個名稱的存在，但我們不知道它是個 type 還是個 template，或是別的什麼東西：

```
template<typename T>
class BXT {
public:
    typedef T Mystery;
    template<typename U>
    struct Magic;
};

template<typename T>
class DXTT : private BXT<T> {
public:
    using typename BXT<T>::Mystery;
    Mystery* p;  // 如果上一行無 typename，此處語法有誤。
};
//譯註：g++ 3.2 無法識別此例中的 using 宣告。VC6/VC7.1/ICL7.1 沒有這個問題。
```

如果我們希望以一個 using 宣告式將一個受控名稱（dependent name）帶入當前作用域，並以它指涉（代表）一個型別，就必須使用關鍵字 typename 明確告知編譯器。奇怪的是 C++ 並沒有提供類似機制來指示「受控名稱（dependent name）是個 template」。下面例子說明這個問題：

```
template<typename T>
class DXTM : private BXT<T> {
public:
    using BXT<T>::template Magic;     // 錯誤：無此用法
    Magic<T>* plink;                  // 語法錯誤：Magic 不是個確知的 template
};
```

這大概是 *C++ Standard* 的疏忽。未來修訂版可能會考慮加入這種用法，從而使上述例子合法。

9.3.5 ADL 和 Explicit Template Arguments（明確模板引數）

考慮下面例子：

```
namespace N {
    class X {
        ...
    };

    template<int I> void select(X*);
}

void g (N::X* xp)
{
    select<3>(xp);  // 錯誤：ADL 不適用
}
```

此例之中，我們可能希望編譯器看到 `select<3>(xp)` 時透過 ADL 找到 template `select()`。但情況並非如此，只有在確認 `<3>` 是個 template argument list 時，編譯器才能確認 `xp` 是一個 function call argument。反過來說，只有當編譯器發現 `select()` 是個 template，它才能確認 `<3>` 是個 template argument list。這是個「先有雞還是先有蛋」的問題；這個算式只能被解析（*parsed*）爲 `(select<3)>(xp)`，而這毫無意義。

9.4 衍生（Derivation）與 Class Templates

Class templates 可以繼承其他 classes，也可以被其他 classes 繼承。大多數情況下 class templates 和 non-template classes 在這方面並沒有什麼重大區別。但是有一個微妙而重要的問題，出現在「由一個受控名稱（dependent name）所指涉的 base class，衍生出一個 class template」時。我們先從較簡單的 non-dependent base classes 說起。

9.4.1 非受控的（Non-dependent）Base Classes

在 class template 中，所謂 non-dependent base class 是指一個無需知道任何 template arguments 就可完全確定的型別。換句話說，「用來指涉該 base class」的名稱，是個非受控名稱（non-dependent name）。例如：

```
template<typename X>
class Base {
public:
    int basefield;
```

```
    typedef int T;
};

class D1: public Base<Base<void> > {      // 並不真正是個 template
public:
    void f() { basefield = 3; }           // 以通常方式存取繼承而來的成員
};

template<typename T>
class D2 : public Base<double> {          // non-dependent base class
public:
    void f() { basefield = 7; }           // 以通常方式存取繼承而來的成員
    T strange;                       // T 是 Base<double>::T，不是 template parameter
};
```

non-dependent base templates 的行爲和常規的 non-template base classes 非常相似，但是有個令人驚奇的差異：當編譯器在 derived class template 中查詢一個未受飾名稱（unqualified name）時，會優先查詢 non-dependent base templates，然後才查詢 template parameters。這意味上述例子中，class template D2 的成員 strange 將擁有 Base<double>::T 型別（本例爲 int）。因此以下函式非法（續上）：

```
void g (D2<int*>& d2, int* p)
{
    d2.strange = p;  // 錯誤：型別不匹配
}
```

這與直覺相悖，derived template 編寫者因此必須特別留心其 non-dependent bases 中的名稱 — 甚至即使是間接繼承，或名稱都是 private。也許較好的作法是把 template parameters 放在被它們模板化的物體（the entity they templatized）的作用域內。

9.4.2 受控的（Dependent）Base Classes

上面的例子中，base class 是完全確定的。它不取決於某個 template parameter。也就是說 C++ 編譯器只要見到了 template 的定義，就可以在 base classes 中查詢非受控名稱（non-dependent names）。另一種作法（不被 C++ 接受）是將這一類名稱的查詢過程推遲，直到 template 被具現化後才開始。這種作法的缺點是，由於查詢過程被推遲至具現化之後，也會把查詢過程中可能產生的「找不到符號」錯誤資訊推遲。因此 C++ *Standard* 規定，編譯器只要見到了一個非受控名稱（non-dependent name），就立即開始查詢（looked up）。牢記這一點之後，考慮下面的例子（譯註：其中 class Base 的定義請見 9.4.1 節）：

```
template<typename T>
class DD : public Base<T> {          // dependent base
public:
    void f() { basefield = 0; }      // (1) 有問題…
};

template<>                           // 明確特化（explicit specialization）
class Base<bool> {
public:
    enum { basefield = 42 };         // (2) 小花招
};

void g (DD<bool>& d)
{
    d.f();                           // (3) 喔哦?
}
```

在(1)處我們發現，這裡使用了一個非受控名稱 basefield：編譯器一定會立即查詢它。假設我們在 template Base 中查詢它，並將它繫結爲一個（我們所找到的）int 成員。然而在這之後我們立刻對這個泛型定義進行特化，並在(2)處將 basefield 改變爲我們委任的定義。於是當(3)處具現化 DD:f 的定義時，會發現(1)處對 basefield 的型別解釋並不準確。(2)處特化的 DD<bool> 之中並沒有可變化的 basefield，因此編譯器會發出一個錯誤訊息。

爲了解決這個問題，C++ *Standard* 規定非受控名稱（non-dependent names）不在受控的 base class 中查詢[29]（但是見到這種名稱仍會立刻展開查詢）。因此符合標準的 C++ 編譯器會在(1)處報錯。欲修正上述程式碼，我們可以令 basefield 成爲一個受控名稱（dependent names），而受控名稱的查詢過程會發生在它被具現化之後，此時 basefield 的特化型別就可以被編譯器發現。例如在(3)處，編譯器會知道 DD<bool> 的 base class 是 Base<bool>，而 Base<bool> 已被具現化。據此，我們可以將程式碼修改爲：

```
// 修改 1
template<typename T>
class DD1 : public Base<T> {
public:
    void f() { this->basefield = 0; }  // 查詢過程被推遲
};
```

[29] 這是所謂「二段式查詢」（two-phase lookup）規則的一部份。第一階段發生在編譯器首次看到 template 定義時，第二階段發生在 template 被具現化時（見 10.3.1 節, p.146）。

另一種作法是使用一個受飾名稱（qualified name）導入相依性（受控性, dependency）：

```
// 修改 2
template<typename T>
class DD2 : public Base<T> {
public:
    void f() { Base<T>::basefield = 0; }
};
```

你必須小心使用這種解法。如果在 virtual function call 中使用非受飾、非受控名稱（unqualified nondependent name），修飾符號會使 virtual call 機制失效，程式碼意義將因此改變。然而也存在只能用第二種方法，不能用第一種方法的情況：

```
template<typename T>
class B {
public:
   enum E { e1 = 6, e2 = 28, e3 = 496 };
   virtual void zero(E e = e1);
   virtual void one(E&);
};

template<typename T>
class D : public B<T> {
public:
    void f() {
        typename D<T>::E e; // this->E 語法有錯
        this->zero();       // D<T>::zero()會抑制 virtual call
        one(e);             // one 是受控名稱，因爲其引數是受控的
   }
};
```

注意，one(e)中的 one 是個受控名稱，受到 template parameter 的影響。這是因爲該呼叫中的一個明確引數的型別是受控型別（譯註：e 隸屬受控型別 D<T>::E，受到 template parameter T 的影響）。對於被隱式使用的預設引數來說，如果它會受到 template parameter 的影響，就會被編譯器忽略。這是因爲編譯器決定查詢（lookup）它時，卻無法驗證（查清, verify）它，這又是一個雞和蛋的問題。爲了避免此種微妙，我們推薦你在所有情形中使用 this-> 前綴詞 — 即使在書寫 non-template 程式碼時亦然。

如果你覺得這麼多修飾符號（飾詞, qualifier）會弄亂程式碼，可透過 using 宣告式將某個名稱從受控的 base class 中帶到當前作用域，於是你就可以肆意使用它。

```
// 修改 3
template<typename T>
class DD3 : public Base<T> {
public:
```

```
    using Base<T>::basefield;     // (1) 現在，basefield 在當前作用域中爲受控名稱
    void f() { basefield = 0; }   // (2) 正確
};
```

(2)處進行的查詢過程將會成功，它會找到(1)處的 using 宣告（亦即找到 basefield 名稱）。然而直到具現期（instantiation time）之前編譯器並不會驗證 using 宣告式，因此我們就達到了目的。這個方案也有一些微妙限制，例如繼承自多個 base classes 時，程式員必須明確指定需要哪一個 base class 的成員。

9.5 後記

可對 template 定義式進行詞法解析（*parse*）的第一個編譯器是 Taligent 公司於 1990s 年代中期開發的。在此之前（及之後），大多數編譯器都把 templates 當作一個 *tokens sequence*（語彙單元序列），並在具現期（instantiation time）將它交給 *parser*。因此除了小量情況足以標示「template 定義結束位置」，大部份時候 *parser* 都無法正常工作。Bill Gibbons 是 Taligent 公司在 C++ 標準委員會的代表，他是使 templates 可被精確進行詞法解析的主要貢獻者。Taligent 的成果一直都未公開，後來 Hewlett-Packard（HP）獲得了它並加以完善，並把它變爲 aC++ 編譯器。除了極富競爭力的各種優點外，其診斷資訊也被公認爲有著極高品質。由於它不把診斷資訊推遲至 template 具現化之後，因此獲得人們更高的評價。

在 templates 技術發展前期，Tom Pennello（一位公認的 *parsing* 專家，任職於 Metaware）注意到角括號帶來的一些問題。Stroustrup 在 [StroustrupDnE] 中對此作出評論，並認爲相對於小（圓）括號而言，人們更喜歡使用角括號。然而也有人偏愛其他符號作爲 template parameter list 的界定符號，例如 Pennello 在 1991 年於 Dallas（達拉斯）舉行的 C++ 標準委員會議上[30]，就提議使用大（花）括號（像是 List{::X}）。當時這個問題不像如今這麼凸顯，因爲當時還不允許把一個 template 嵌套於另一個 template 之中（成爲 member templates），因此 9.3.3 節, p.132 的討論大部份在當時都沒有意義。最終結果是：委員會否決了以大（花）括號替代角（尖）括號。

C++ *Standard* 於 1993 年引入非受控名稱（non-dependent names）和受控 base classes 的名稱查詢規則（name lookup rule，見本書 9.4.2 節, p.136）。Bjarne Stroustrup 於 1994 年初在 [StroustrupDnE] 中向公眾做了相關描述。但是這個規則的第一個可用實作品直到 1997 年才出現：HP 在它的 aC++ 編譯器中引入了這個特性。此後有大量程式碼從受控的 base classes 中繼承出 class templates。當 HP 工程師開始進行測試時，他們發現大部分對 template 運用頗深的程式碼再也無法成功編譯[31]。

---

[30] 大（花）括號也無法解決所有問題。特別是 class templates 的特化語法需要做不小的改動。

[31] 幸運的是，標準委員會在發佈這個新特性之前，發現了問題。

特別是所有 STL 實作品都違犯了這個規則，違犯點成百上千[32]。爲了緩解其顧客在過渡期間所遇到的難題，HP 放寬了這個規則：當根據 C++ 標準無法找到一個 class template 作用域中的非受控名稱時，aC++ 會在其受控 base classes 中繼續尋找。如果還是找不到這個名稱，編譯器會報錯，宣告失敗。然而如果在受控 base classes 中找到了該名稱，編譯器會給出一個警告，同時該名稱會被標示爲受控（dependent），這麼一來當該名稱被具現化時，編譯器會再次查詢它。

根據查詢規則（lookup rule），非受控 base class 中的名稱可能會導致編譯器無法見到一個與之同名的 template parameter（見 9.4.1 節, p.135）。這是一個疏漏，未來 *C++ Standard* 修訂版可能會解決它。目前你最好避免這種重名情況。

Andrew Koenig 首先提議在運算子函式中使用 ADL（這就是爲什麼 ADL 有時被稱爲 *Koenig lookup* 的原因）。這其實是個審美觀問題：明顯地將運算子名稱飾以「圈封之 namespace」的名稱，委實有些笨拙（例如我們不能寫 `a+b`，必須寫成 `N::operator+(a, b)`）；而且在每一個運算子前面使用 `using` 宣告式，也使程式碼難看。因此人們決定讓編譯器在查詢運算子時，同時查詢其引數之相關聯 namespaces。後來 ADL 又被擴展，適用於查詢常規函式名稱，以容許一些有限的「friend 名稱植入」（friend name injection），並支援 templates 的兩段式查詢（two-phase lookup model）模型和具現化。這種更爲泛化的 ADL 規則又稱爲 *extended Koenig lookup*。

[32] 諷刺的是，第一個 STL 實作品正是 HP 自己開發的。

10

具現化/實體化

Instantiation

Template 具現化（instantiation）是「由泛化的 template 定義式產生出實際型別和函式」的過程[33]。C++ templates 具現化是一個基礎而複雜的概念。說它複雜的一個原因是，由 template 所產生的物體（entities）的定義不限於源碼的某一單獨位置上。template 所在位置、template 被使用位置、以及 template arguments 定義位置，都在「構成物體意義」一事上扮演各自的角色。

本章講述如何組織我們的程式碼才能以正確方式運用 templates。我們還將講述爲解決具現化問題，大多數流行的 C++ 編譯器所使用的方法。儘管這些方法應該是語意等價的（semantically equivalent），但了解你的編譯器的具現化策略，會對你帶來幫助。構築真實世界中的軟體時，每種機制都有不足，然而 C++ 標準規格正是在這些機制基礎上才得以成形。

10.1 隨需具現化（On-Demand Instantiation）

當 C++ 編譯器見到程式中使用一個 template 特化體（specialization），便將 template parameters 替換爲所需的 argument，以創建出這個特化體[34]。這個過程自動完成，無需借助任何 client 程式碼（或 template 定義）的指示。正是這種「隨需具現化」（有需要時便具現化）的特性使得 C++ template 機制有別於其他語言的類似機制。這種機制有時也稱爲隱式（*implicit*）或自動（*automatic*）具現化。

---

[33] 術語「具現化」（instantiation）在某些時候也用來表示「創建某種型別的 object」。然而在本書中它指的是 template 具現化。

[34] 術語「特化體」（specialization）用於一般意義中的「某個 template 的特定具現形式」（請參考第 7 章），而不代表第 12 章描述的「明確特化」（explicit specialization）機制。

隨需具現化（on-deman instantiation）意味著當程式用到 template 時，編譯器需要得知該 template 和其某些成員的完整定義（而不僅僅是 template 的宣告）。考慮下面例子：

```
template<typename T> class C;   // (1) 只有宣告

C<int>* p = 0;                  // (2) 沒問題，不需 C<int> 的定義

template<typename T>
class C {
public:
    void f();                   // (3) 成員宣告
};                              // (4) class template 定義完備

void g (C<int>& c)              // (5) 只用到 class template 的宣告
{
   c.f();                     // (6) 用到了 class template 的定義；需要 C::f()的完整定義
}
```

在(1)處，編譯器只能見到 template 的宣告，見不到其定義（這種宣告又稱爲前置宣告，*forward declaration*）。和常規 classes 的規則一樣，不需 class template 的定義你就可以宣告該型別的 pointers 或 reference（如(2)）。例如函式 g()的參數型別 C<int>& 並不需要 template C 的完整定義。然而一旦編譯器需要知道某個 template 特化體的大小，或程式碼中取用了該特化體的某個成員，編譯器就必須見到 template 的定義。這說明爲什麼在(6)處，class template 的定義必須可見；如果見不到這個定義，編譯器就無法確認這個成員是否存在，或程式碼是否有權取用它（必須不爲 private 或 protected，才能被取用）。

這裡有另外一個運算式，需要 class template 具現體，因爲此處編譯器必須知道 C<void>的大小：

```
C<void>* p = new C<void>;
```

這裡必須完成具現化，這麼一來編譯器才知道 C<void>的大小。你可能會注意到，無論把 template C 的參數 T 以什麼樣的型別 X 替換，其 template 具現體的大小都不會受到影響：任何情況下 C<X> 都是個 empty class。但你不能強求編譯器知道這一點。不僅如此，在這個例子中，編譯器也需要以具現化過程來得知 C<void> 是否有一個可被取用的 ***default*** 建構式，並確保 C<void> 沒有定義 private operator new 和 private operator delete。

有時候，從程式碼本身並不能看出某個 class template 的成員會被取用。例如 C++ 的重載決議機制（overload resolution）便需要得知各候選函式的參數的 class types：

```
template<typename T>
class C {
public:
    C(int);     // 單引數建構式（one-argument ctor）有可能被用於「隱式轉型」。
};

void candidate(C<double> const&);    // (1)
void candidate(int) {}               // (2)

int main()
{
    candidate(42);  // 上面宣告的兩個函式都可被呼叫
}
```

編譯器會將 `candidate(42)` 決議（resolve）爲 `(2)` 的宣告。然而 `(1)` 處的宣告也可能被具現化，用來確定它是否也是上述呼叫的一個合法候選函式（本例之中具現化可能發生，因爲 `42` 可隱寓通過「單引數建構式」轉型爲 `C<double>` 型別的 rvalue）。注意，即使編譯器可以不透過具現化來決議（resolve）某個重載呼叫，*C++ Standard* 仍然允許（但不要求）編譯器進行具現化（放到本例之中，具現化就非必要了，因爲已有一個完全匹配的函式 `candidate(int)`，這使得編譯器不會選用其他需要隱式轉型的候選函式）。注意，將 `C<double>` 具現化可能會觸發一個令你驚訝的錯誤。

10.2 緩式具現化（Lazy Instantiation）

目前爲止，我們所舉的例子中，還沒有什麼東西與 non-template classes 有本質上的區別。很多情況下編譯器需要一個「完整的」class 型別。運用 template 時，編譯器會從 class template 定義式中產生這個「完整的」定義。

這就引發一個相關問題：template 之中有多少內容需要被具現化？一個模糊的回答是：只有「真正被用到」的那些才需要被具現化。換句話說編譯器應該儘量推遲 templates 具現化的進行，這便是所謂的 lazy instantiation。我們來看看 lazy 的具體是什麼含義。

當 class template 被隱式具現化，其中的每一個成員宣告也都同時被具現化，然而並非所有對應的定義也都被具現化，其中有些例外。首先，如果 class template 內含一個不具名的 union，這個 union 的成員也都會被具現化[35]。另一個例外與 virtual 成員函式有關。當 class template 被具現化時，virtual 成員函式的定義可能被（也可能不被）具現化。事實上很多編譯器都會具現化這些定義，因爲 virtual call 機制的內部結構要求：virtual 函式必須作爲可聯結物（linkable entities），存在於某個結構（譯註：vtbl/vptr）中。

[35] 不具名的（anonymous）unions 總是比較特殊：其內成員可被視爲外層 class 的成員。一個不具名的 union 實際上表現出這樣一種構件：它的所有成員共用同一段儲存空間。

當 templates 被具現化時，*default* call arguments 通常被個別考慮。除非有些呼叫真正用上了 *default* arguments，否則那些 *default* arguments 不會被具現化。如果呼叫函式時明確指定了 arguments，那麼 *default* arguments 也不會被具現化。

下面的例子涵蓋了上述內容：

```
// details/lazy.cpp

template <typename T>
class Safe {
};

template <int N>
class Danger {
public:
    typedef char Block[N];  // 如果 N<=0，會報錯
};

template <typename T, int N>
class Tricky {
public:
    virtual ~Tricky() {
    }
    void no_body_here(Safe<T> = 3);
    void inclass() {
        Danger<N> no_boom_yet;
    }
    // void error() { Danger<0> boom; }
    // void unsafe(T (*p)[N]);
    T operator->();
    // virtual Safe<T> suspect();
    struct Nested {
        Danger<N> pfew;
    };
    union {               // 不具名的 union
      int align;
      Safe<T> anonymous;
    };
};
```

```
int main()
{
   Tricky<int, 0> ok;
}
//譯註：VC6 未能通過此例，因爲它不遵循「假設最好情況」（assume the best）規則（見下文）。
//VC 7.1/ICL 7.1/g++ 3.2 均無問題。
```

首先考慮 `main()` 不存在的情況。符合標準的 C++ 編譯器通常會編譯 template 定義式以進行語法檢查，並進行一般的語意約束（semantic constraints）檢查。然而在檢查 template parameters 的約束條件時，它會「假設最好情況」。例如 `Block` 內的 `typedef` 的參數 `N` 有可能是 0 或負值（這是非法的），但編譯器假設這種情況不會發生。類似情況，class template `Tricky` 的成員函式 `no_body_here` 的宣告式中，預設引數（`=3`）頗爲可疑，因爲 template `Safe` 無法以一個整數值初始化，但編譯器會假定 `Safe<T>` 的泛化定義並不需要這個預設引數。另外，如果不把成員函式 `error()` 註釋掉，它會引發一個錯誤，因爲它會用到 `Danger<0>` 特化體，而這個特化體試圖以 `typedef` 定義「0 個元素的 array」。即使成員函式 `error()` 未被呼叫（因而也未被具現化），這個錯誤也會在編譯期對整個泛型 template 進行處理的時候觸發。但成員函式 `unsafe(T (*p)[N])` 並沒有這個問題，因爲此時的 `N` 仍是個泛型參數，template parameters 的替換並未發生。

現在讓我們分析 `main()` 存在時的情況。它會將 template `Tricky` 的參數 `T` 替換爲 `int`，參數 `N` 替換爲 `0`。編譯器並不需要所有成員的定義，但 *default* 建構式（本例爲隱寓宣告）和解構式一定會被叫用，因此編譯器必須見到它們的定義。virtual 成員的定義也必須可見，否則會聯結錯誤。如果拿掉 virtual 函式 `suspect()` 的註釋符號，此例可能會引發聯結錯誤，因爲我們沒有給出 `suspect()` 定義式。`Danger<0>` 的完整型別（前面說過這會產生一個非法的 typedef）也是需要的，因爲它在成員函式 `inclass()` 和 `struct Nested` 中出現，但由於這些定義沒有被用到，所以它們不會被生成，從而不會引發錯誤。然而，編譯器會生成所有成員的宣告式，並可能因爲參數/引數的替換而在這些宣告式中存在非法型別。例如，如果我們拿掉 `unsafe(T (*p)[N])` 宣告式的註釋符號，會再次創建一個「0 個元素的 array」，這會引發錯誤。同樣道理，如果成員 `anonymous` 並非宣告爲 `Safe<T>` 型別而是宣告爲 `Danger<N>` 型別，由於 `Danger<0>` 型別並不合法，會引發一個錯誤。

最後請注意 `operator->`。通常這個運算子必須傳回一個 pointer 型別，或一個可被 `operator->` 施行於其上的型別。於是你可能認爲 `Tricky<int,0>` 會引發錯誤，因爲 `operator->` 的回返型別是 `int`。然而由於某些自然的（natural）class template 觸發了這種定義[36]，因此 C++ 語言在此較爲靈活。使用者自定的 `operator->` 只需在「被重載決議機制選中」時傳回一個「可施行另一個 `operator->`（例如內建的那個）」的型別即可。這在不涉及 template 的情況時也適用（雖然那種情況下就沒什麼太大用處）。因此這裡的宣告不會引發錯誤，即使它傳回的是個 `int`。

[36] 典型的例子是所謂的 *smart pointer* templates（例如標準程式庫提供的 `std::auto_ptr<T>`）。請參考第 20 章。

10.3 C++具現化模型（C++ Instantiation Model）

所謂 template 具現化，是「適當地替換 template parameters，以便從 template 獲得常規 class 或常規 function」的過程。聽來平淡無奇，但實際上這個過程涉及極多細節。

10.3.1 兩段式查詢（Two-Phase Lookup）

在第 9 章中我們知道，當編譯器對 templates 進行 *parsing*（詞法解析）時，它無法解析受控名稱（dependent names）。這些受控名稱將在具現點被再次查詢（lookup）。然而非受控名稱（non-dependent names）會較早被查詢，如此一來當 template 首次被編譯器看到時，就可以較多地診斷出錯誤。這就是「兩段式查詢」概念[37]：第一階段發生在 template *parsing*（詞法解析）時刻，第二階段發生在具現化時刻。

在第一階段，編譯器會查詢非受控名稱（non-dependent names），這個過程會用到常規查詢（*ordinary lookup*）規則；如果情況適用也會動用 ADL（*argument-dependent lookup*）規則。非受飾的受控名稱（unqualified dependent names；由於它們在一個帶有 dependent arguments 的函式呼叫中看起來像個函式名稱，所以是受控的, dependent）也會以此方式被查詢，然而其結果並不完備，因此會在 template 具現化時再次被查詢。

在第二階段，也就是在一個所謂「具現點」（*point of instantiation*, **POI**）處，編譯器會查詢受控受飾名稱（dependent qualified names；將特定具現體的 template parameters 代換爲 template arguments），也會對非受飾受控名稱（unqualified dependent names）額外加以 ADL 查詢。

10.3.2 具現點（Points of Instantiation）

我們已經說過，template 程式碼中有這樣一些位置點；C++ 編譯器在這些點上必須能夠取得某個 template 物體的宣告或定義。一旦程式碼需要用到 template 特化體，而編譯器需要見到該 template 的定義以創造出該特化體時，就會在這個具現點（POI）上進行具現過程。具現點（POI）就是替換後之 template 可插入的源碼位置點。例如：

```
class MyInt {
public:
  MyInt(int i);
};

MyInt operator - (MyInt const&);
bool operator > (MyInt const&, MyInt const&);
```

[37] 「二段式查詢」的英文術語是：*two-phase lookup* 或 *two-stage lookup* 或 *two-phase name lookup*。

```
typedef MyInt Int;

template<typename T>
void f(T i)
{
    if (i>0) {
        g(-i);
    }
}
// (1)
void g(Int)
{
    // (2)
    f<Int>(42);  // 呼叫點
    // (3)
}
// (4)
```

當編譯器看到呼叫動作 `f<Int>(42)`，它知道必須將 `T` 替換爲 `MyInt` 來具現化 template `f`。這於是就產生了一個 POI（具現點）。`(2)`和`(3)`距離呼叫點很近，但它們不能作爲 POI，因爲 C++ 語法不允許我們在這兩處插入`::f<Int>(Int)`的定義。`(1)`和`(4)`的本質差異在於，函式 `g(Int)` 在`(4)`處可見，於是 template dependent call `g(-i)` 可被決議（*resolved*）出來。如果把`(1)`當做 POI，`g(-i)`就無法被成功決議出來，因爲編譯器在該處看不到 `g(Int)`。幸運的是，針對 "a referendce to a nonclass specialization", C++ 設計的 POI 是在「最內層之 namespace 作用域（惟需內含該 reference）」的宣告向或定義式的緊臨後方。本例中這個點是`(4)`。

你可能會奇怪爲什麼本例使用 `MyInt` 而不是簡單地使用 `int`。答案是在 POI 處進行的第二階段查詢只動用 ADL（「相依於引數的查詢」）。由於 `int` 並無相應的 namespace，不會發生 POI 查詢，從而編譯器無法找到函式 `g()`。如果你把 `Int` 的 `typedef` 換成：

```
typedef int Int;
```

上述例子就無法通過編譯[38]。

譯註：不過事實上 VC6/VC7.1/ICL7.1/g++3.2 都可以在 `typedef int Int;` 情形下順利編譯上例。

對於 class 特化體，POI 的位置有所不同。見下一個例子：

[38] 直到 2002 年，C++ 標準委員會仍在研究其他方法，使後一種 typedef 也可順利通過編譯。

```
template<typename T>
class S {
  public:
    T m;
};
// (5)
unsigned long h()
{
    // (6)
    return (unsigned long)sizeof(S<int>);
    // (7)
}
// (8)
```

函式作用域內的(6)和(7)不能作爲 POI，因爲 class S<int>（它構成一個 namespace 作用域）的定義式不能出現在這兩個地方（template 不能在函式作用域內出現）。如果我們按照 nonclass 的方式思考，POI 將是(8)，但這樣就造成算式 S<int> 不合法，因爲在(8)之前 S<int> 的大小無從得知。因此對於 "a reference to a template-generated class instance"，其 POI 位置被定義爲：「最內層之 namespace 作用域（惟需內含該 class instance）」的宣告式或定義式的緊臨前方。本例中這個點是(5)。

當 template 被實際具現化，有可能引發其他具現化動作。考慮下面例子：

```
template<typename T>
class S {
public:
    typedef int I;
};

// (1)
template<typename T>
void f()
{
    S<char>::I var1 = 41;
    typename S<T>::I var2 = 42;
}

int main()
{
    f<double>();
}
// (2):(2a),(2b)
```

根據先前討論，f<double> 的 POI 是(2)。function template f()也用到了 S<char> 這個 class 特化體，其 POI 在(1)。還用到了 S<T>，但在(1)處 S<T> 仍然受控（dependent），因此在(1)還不能進行具現化。然而如果我們在(2)具現化 f<double>，我們同時必須具現化 S<double>。這種次級 POI（*secondary* POI，或稱 *transitive* POI）的定義稍有不同。針對 nonclass 物體，次級 POI 和主要 POI（*primary* POI）完全相同，但針對 class 物體，次級 POI 緊臨於其基本 POI 之前。上例 f<double>的 POI 在(2b)，其前先的(2a)是 S<double>的二級 POI。請注意這與 S<char> 的 POI 有所不同。

在一個編譯單元中，同一個具現體往往有多個 POI。針對 class template 實體，編譯器只保留頭一個 POI，忽略所有後續 POI（編譯器並不真正認爲它們是 POI）。針對 nonclass 實體，所有 POI 都被保留。不論哪一種情況，ODR（單一定義原則）都要求：被編譯器保留的所有 POI 彼此必須等價，但編譯器不需要檢驗是否有違例情況。這就使得編譯器可以一個 nonclass POI 進行真正的具現化動作，而不需憂慮其他 POI 可能導致其他具現體。

事實上，大多數編譯器都會把對 noninline function templates 的真正具現化過程推遲至編譯單元尾端。這也就是把相應之 template 特化體的 POI 移到了編譯單元尾端。C++ 編譯器實作者認爲這是一種合法的實作技術，但 *C++ Standard* 對此並無明確態度。

10.3.3 置入式（Inclusion）和分離式（Separation）模型

無論何時遇到一個 POI，編譯器必須能夠取得對應之 template 的定義。對於 class 特化體而言，這意味在當前編譯單元中，class template 的定義式必須出現於 POI 之前。對 nonclass POI 的態度雖然也如此，但典型情況是 nonclass template 的定義式被放在表頭檔中，由當前編譯單元以 #include 將它包含進來。這種 template 定義式的源碼組織方式稱爲置入式模型（inclusion model），也是截至本書完稿時最被大眾採用的模型。

針對 nonclass POI 另還存在一種方式：nonclass template 可被宣告爲 export，並定義於另一個編譯單元。這種方式稱爲分離式模型（separation model）。下面例子展示這種情況，仍然用的是我們的老朋友 max()：

```
// 編譯單元 1
#include <iostream>

export template<typename T>
T const& max (T const&, T const&);

int main()
{
    std::cout << max(7, 42) << std::endl;   // (1)
}
```

```
// 編譯單元 2
export template<typename T>
T const& max (T const& a, T const& b)
{
   return a < b ? b : a;                    // (2)
}
```

當第一個檔案被編譯時，編譯器認爲(1)是 POI，此時 T 被替換爲 int。編譯系統必須確保編譯單元 2 中的 max()定義式被具現化，從而滿足 POI 的需求。

10.3.4 跨越編譯單元尋找 POI

假設上述的編譯單元 1 被重寫爲：

```
// 編譯單元 1
#include <iostream>

export template<typename T>
T const& max(T const&, T const&);

namespace N {
  class I {
  public:
     I(int i): v(i) {}
     int v;
  };

  bool operator < (I const& a, I const& b) {
     return a.v < b.v;
  }
}

int main()
{
   std::cout << max(N::I(7), N::I(42)).v << std::endl;  // (3)
}
```

POI 誕生於(3)，而且編譯器需要編譯單元 2 中的 max()定義。然而 max()使用重載的 operator<，而後者卻是在編譯單元 1 中被宣告，不可見於編譯單元 2。爲了正確處理這種情況，編譯器顯然必須在具現化過程中參考兩個不同的「宣告脈絡」（declaration contexts）[39]：一是 template 定義於何處，二是型別 I 宣告於何處。爲涉入這兩個脈絡（contexts），編譯器使用兩段式查詢（見 10.3.1 節, p.146）來對付 template 中的名稱。

[39] 所謂「宣告脈絡」（declaration context）是指：給定地點可存取之所有宣告式的集合。

第一階段發生在 templates 被 *parsing*（詞法解析）時，換句話說在 C++ 編譯器首次見到 template 定義式時。在此階段，編譯器會使用 *ordinary lookup* 規則和 *ADL* 規則來查詢非受控名稱（non-dependent names）。另外編譯器還使用 *ordinary lookup* 規則來查詢受控函式（dependent functions；肇因於其函式引數受控）的非受飾名稱（unqualified names），但它會記住查詢結果而不企圖進行重載決議（overload resolution）。重載決議工作是在第二階段後進行的。

第二階段發生在 POI（具現點）。在 POI 處，編譯器會使用 *ordinary lookup* 規則和 *ADL* 規則來查詢受控受飾名稱（dependent qualified names）。至於受控非受飾名稱（dependent unqualified names；已於第一階段以 *ordinary lookup* 規則查詢過），編譯器只運用 *ADL* 規則去查詢，再把查詢結果結合第一階段的結果。這兩個結果構成的集合將被編譯器用來完成重載函式的決議過程（overload function resolution）。

儘管兩段式查詢機制本質上使分離式模型（separation model）的實現成爲可能，但這種機制也用於置入式模型（inclusion model）。然而在很多早期的內置式模型實作品中，所有查詢過程都被推遲至 POI[40]。

10.3.5 舉例

拿出一些實例來，可以較有效地解釋我們先前所講述的內容。第一個例子是置入式模型（inclusion model）的簡單情況：

```
template<typename T>
void f1(T x)
{
    g1(x);  //(1)
}

void g1(int)
{
}

int main()
{
    f1(7);    // ERROR：找不到 g1()
}             // (2)f1<int>(int)的 POI
```

//譯註：以上雖說有 ERROR，事實上 VC6/VC7.1/ICL7.1/g++3.2 都可順利編譯此例。請參考 p.147。

[40] 這種行爲很接近於 macros（巨集）展開機制。

呼叫 f1(7)會製造出一個 POI，位於 main()的緊臨外側（(2)處）。這個具現化過程的關鍵在於函式 g1()的查詢。當 template f1 的定義式首次出現，編譯器會注意到非受飾名稱 g1 是受控的（dependent），因爲它是「帶有受控引數（dependent arguments；引數 x 的型別取決於 template parameter T）之函式呼叫」的函式名稱。因此編譯器在(1)處使用 *ordinary lookup* 規則來查詢 g1，然而此刻 g1 不可見。在(2)處，亦即 POI，編譯器再次於相應的 namespaces 和 classes 中查詢 g1，但因 g1()的惟一引數是 int 型別，而 int 型別沒有相應的 namespaces 和 classes，所以編譯器最終沒有找到 g1 的完整型別 — 即使發生於 POI 的 *ordinary lookup* 確實找到了 g1 的名稱。

第二個例子示範：跨編譯單元時，分離式模型（inclustion model）可以導致重載歧義（overload ambiguities）。此例由三個檔案組成，其中一個是表頭檔：

```
// common.hpp 檔案
export template<typename T>
void f(T);

class A {
};
class B {
};

class X {
public:
    operator A() { return A(); }
    operator B() { return B(); }
};

// a.cpp 檔案
#include "common.hpp"

void g(A)
{
}

int main()
{
    f<X>(X());
}
```

```
// b.cpp 檔案
#include "common.hpp"

void g(B)
{
}

export template<typename T>
void f(T x)
{
   g(x);
}
```

在檔案 a.cpp 中，main()呼叫 f<X>(X())。f 是個 exported template，定義於檔案 b.cpp，其內的呼叫動作 g(x)因而被引數型別 X 具現化。g()被查詢兩次：第一次使用 *oridnary lookup* 規則在檔案 b.cpp 中查詢（當 template 被 *parsing* 時），第二次使用 *ADL* 規則在檔案 a.cpp 中查詢（那是 template 具現化動作所在）。第一次查詢找到 g(B)，第二次查詢找到 g(A)。兩個結果都合理（因爲 class X 內有對應的兩個使用者自定轉型運算子），因此這個呼叫帶有歧義性。

注意，檔案 b.cpp 中的 g(x)呼叫動作看起來一點問題也沒有。完全是因爲兩段式查詢機制才帶來額外的候選函式。因此，撰寫 export templates 程式碼和其說明文件時，一定要極度小心。

10.4 實作方案（Implementation Schemes）

譯註：以下以 obj 表示目的檔（object files），lib 表示程式庫（libraries），exe 表示可執行檔（executable files）。

本節之中我們將回顧普及的 C++ 編譯器產品實現置入式模型（inclustion model）的方法。所有這些產品都倚賴兩個典型組件：編譯器和聯結器。編譯器用來將源碼轉譯爲 obj 檔，其內包含帶符號標注（symbolic annotations；用以交叉引用其他 obj 檔和 lib 檔）的機器碼。聯結器用來合併 obj 檔、決議（*resolving*）obj 檔所含之符號式交叉引用（symbolic cross-references），最終生成 exe 檔或 lib 檔。接下來的內容中，我們假設你的編譯系統使用這種模型，當然以其他方式實作 C++ 編譯系統也是完全可能的（但不普及）。例如你可以想像一個 C++ 直譯器（interpreter）。

當一份 class template 特化體被用於多個編譯單元中，編譯器會在處理每個編譯單元時重複具現化。這不會引發什麼問題，因爲編譯器並不直接根據 class 定義式產生機器碼。C++ 編譯器只是在其內部用到這些具現體，以便檢驗或解釋其他各種算式和宣告。從這個角度說，class 定義式的多次具現化，與 class 定義式在多個編譯單元被多次含入（典型情況是透過含入檔）並無本質上的區別。

然而，如果你具現化一個 noninline function template，情況可能有所不同。如果你提供了一個常規的 noninline function 的多份定義，你將違反 ODR（單一定義原則）。假設你要編譯並聯結一個程式，而它由下面兩個檔案組成：

```
// a.cpp 檔案
int main()
{
}

// b.cpp 檔案
int main()
{
}
```

C++ 編譯器會順利編譯兩個檔案，不會遭遇任何問題，因爲它們確實都是合法的 C++ 編譯單元。然而如果你試圖聯結兩個 obj 檔，聯結器幾乎肯定要發出抗議，因爲你在同一個程式中重複定義了 `main()`，這是不允許的。

對比地，考慮 template 的情況：

```
// t.hpp 檔案
// 普通的表頭檔（置入式模型）
template<typename T>
class S {
public:
    void f();
};

template<typename T>
void S::f()     // 成員定義
{
}

void helper(S<int>*);

// a.cpp 檔案
#include "t.hpp"

void helper(S<int>* s)
{
    s->f();    // (1) S::f 的第一個 POI（具現點）
}

// b.cpp 檔案
#include "t.hpp"

int main()
{
    S<int> s;
    helper(&s);
    s.f();     //(2) S::f 的第二個 POI
}
```

如果聯結器以「對待常規函式或常規成員函式的方式」來處理 templates 的具現化成員，那麼編譯器便需確保在兩個 POI 處只生成一份程式碼：不是(1)處就是(2)處，但不能兩處都生成程式碼。爲了達到這個目標，編譯器不得不把某個編譯單元的資訊帶到其他單元，而這在 templates 概念被引入之前，編譯器是絕對無需這麼做的。接下來的內容將討論三種主要解決辦法，它們在各種 C++ 編譯器實作品中運用極廣。

注意，同樣問題也會出現在 template 具現化過程所產生的所有可聯結物（linkable entities）上：具現化的 function templates 和 member function templates，以及具現化的 static 成員變數。

10.4.1 貪婪式具現化（Greedy Instantiation）

頭一個廣泛運用「貪婪式具現化」機制的編譯器，出自於 Borland 公司。後來這種機制逐漸成爲各種 C++ 編譯系統中最常用的技術，特別是在 Microsoft Windows 開發環境中，它幾乎成爲所有編譯系統使用的機制。

「貪婪式具現化」機制假設，聯結器知道某些物體（特別是可聯結之 template 具現體）可能在各個 obj 檔和 lib 檔中存在多份。編譯器通常會以特殊方法在這些物體上作出標記。當聯結器發現存在多個重複具現體時，它只保留一個，並將其他具現體全部丟棄。就是這麼簡單。

「貪婪式具現化」在理論上有一些嚴重缺點：

- 編譯器可能生成並優化 N 個 template 具現體，最後卻只保留一個。這會嚴重浪費時間。
- 通常聯結器並不檢查兩個具現體是否完全相同（identical），因爲一個 template 特化體的多個實體之間往往存在一些不重要的差異，這是合法的。這些小差異不會導致聯結失敗（只是可能在具現化時造成編譯器的內部資料有所不同）。然而這也經常導致聯結器注意不到更大的差異，例如某個具現體可能在編譯期針對最大效能進行優化，另一個具現體在編譯期含入更多除錯資訊。
- 和其他機制對比，「貪婪式具現化」機制可能會產生總長度大得多的 obj 檔，因爲相同的程式碼可能重複出現多次。

這些缺點實際並不是什麼大問題。這可能是因爲「貪婪式具現化」機制在一個重要方面明顯優於其他機制：傳統的「源碼-目的檔」（source-object）依存關係得以保留，特別是：一個編譯單元只生成一個 obj 檔，每個 obj 檔都包含對應源碼檔案中的所有可聯結定義（其中包括具現化後的定義）。

最後一個值得注意的問題是，如果聯結機制「允許多個可聯結物（linkable entities）的定義存在」，這個機制通常可被用來處理重複出現、到處散落的 inline 函式[41]和虛擬函式分派表（virtual function dispatch tables）[42]。當然，如果不支援這種機制，也可能使用其他機制以內部聯結方式（internal linkage）產生這些東西，代價是產生出來的 obj 檔比較大。

10.4.2 查詢式具現化（Queried Instantiation）

此類機制最普及的實作品來自 Sun Microsystems 公司，第一個支援此機制的是其編譯器 4.0 版。「查詢式具現化」概念上極其簡單優雅，如果按發生時間排列，它也是我們回顧的這些具現化機制中最晚出現的。在此機制中，編譯器維護一個「由程式各編譯單元共享」的資料庫（database）。這個資料庫可用來追蹤「哪些特化體在哪個相關源碼檔案中被具現化」。生成的所有具現體也會連同這些資訊被儲存於資料庫中。當編譯器遇到一個可聯結物（linkable entity）的 POI（具現點）時，可能會發生以下三種情況之一：

1. 找不到其特化體：這種情況下，編譯器會進行具現化過程，產生的特化體被存入資料庫。
2. 找到了其特化體，但已過期（out of date），也就是特化體被生成後源碼又被改動過。和 1. 相同，編譯器於是進行具現化過程，然後把新生成的特化體存入資料庫中。
3. 在資料庫中找到了最新（up-to-date）特化體。編譯器於是什麼都不用做。

雖然概念上看來簡單，但實作時這種設計會帶來一些難題：

- 編譯器需要正確地根據源碼狀態來維護資料庫中的各種依存關係，而這並非垂手可得。雖然「把第三種情況誤當作第二種情況來處理」並不算錯誤，但這會增加編譯器的工作量，也就是增加整個編譯時間。
- 並行（concurrently）編譯多份源碼如今已是頗爲平常的事了。那麼，編譯系統的實作者必須提供一個工業強度的資料庫，恰當地控制各種並行情況。

儘管存在這些困難，這個機制還是可以實作出極高效率。而且也不存在什麼情況會使這種解決辦法無法處理大規模程式。相反地，如果採用「貪婪式具現化」機制，處理大規模程式時會產生許多工效（工時）上的浪費。

不幸的是，資料庫的使用會帶給程式員一些問題。多數問題的根源在於，繼承自大多數 C 編譯器的傳統編譯模型，此處不再適用，因爲單一編譯單元不再產生單一的獨立 obj 檔。舉個例子，

---

[41] 當編譯器無法「inline 化」你標記爲 inline 的所有函式呼叫時，它會在目的檔中產生該函式的一份拷貝。這一問題可能出現於多個 obj 檔中。

[42] 虛擬函式呼叫（virtual function calls）通常被實作爲「透過一個函式指標表」的間接呼叫方式。如果你想深入了解 C++實作品中的這方面細節，請參考 [LippmanObjMod]。

假設你想要聯結你的最終版本的程式，聯結操作中不僅需要編譯單元相應的各個 obj 檔，也需要儲存於資料庫中的 obj 檔。同樣道理，如果你建立一個二進制 lib 檔，你必須確保生成該檔案的工具程式（通常是 linker 或 archiver）能夠感知資料庫的內容。更常見的情況是，所有 obj 檔操作工具都需要感知資料庫的內容。許多這些問題都可以透過「不要將具現體保存於資料庫」而得以緩和，換之以另一種方式：吐露出 obj 檔中「首先引發具現化」的 obj 碼。

lib 檔是另一個大難題。很多編譯器生成的特化體可能會被打包到一個 lib 檔中。當另外一個專案使用這個 lib 時，該專案的資料庫必須感知到這些既有的特化體。如果無法感知，編譯器就會爲此專案產生出它自己的 POI（具現點）。然而實際上這些特化體已經存在於 lib 檔中，於是造成重複具現化。一個可能解決這個問題的策略是：使用類似「貪婪式具現化」的聯結技術，讓聯結器可以感知已經生成的特化體，並可清除所有重複特化體（相對於「貪婪式具現化」策略來說，這種情況較少出現）。使用其他精妙的安排方式來組織源碼檔、obj 檔或 lib 檔，還是可能引發令人沮喪的問題，例如由於「含有所需具現體之 obj 檔未被聯結至最終 exe 檔」，導致聯結器抱怨找不到某些具現體。諸如此類的問題不該被看作是「查詢式具現化」機制的缺點，實在是由於程式開發環境過於複雜和微妙。

10.4.3 迭代式具現化（Iterated Instantiation）

第一個支援 C++ templates 的編譯器是 Cfront 3.0，這是 Bjarne Stroustrup 寫來發展 C++ 語言的編譯器的直系後裔[43]。Cfront 的一個硬性條件是：必須具備高度移植性。這意味(1)它在所有目標平台上都以 C 語言爲共通表述；(2)它使用平台提供的聯結器。更明確地說這意味聯結器對 template 一無所知。事實上 Cfront 將 template 具現體生成爲常規 C 函式，因此它必須避免產出重複的具現體。儘管 Cfront 的源碼模型（source model）不同於標準的置入式模型（inclusion model）和分離式模型（separation model），但是它的具現化策略可加以改編適應置入式模型。也因此它被人們譽爲「迭代式具現化」機制的第一個化身。Cfront 的「迭代式具現化」機制描述如下：

1. 編譯源碼，但不具現化任何所需的可聯結特化體（linkable specialization）。
2. 使用一個「預聯結器」（prelinker）聯結 obj 檔。
3. 預聯結器喚起聯結器，並解析其錯誤資訊以便得知是否遺漏任何具現體。如果真有遺漏，預聯結器會喚起編譯器，編譯那些「內含所需 template 定義」的檔案。喚起時帶一個選項，讓編譯器產出先前遺漏的具現體。
4. 如果產出任何具現體定義，則重複 3.。

之所以需要迭代（反覆）步驟 3，因爲一個可聯結物（linkable entity）的具現化過程可能引發另

[43] 別因爲這個描述而誤以爲 Cfront 只是個抽象原型：它被用於工業環境中，並且是很多 C++ 商業化編譯器的基礎。Release 3.0 於 1991 年發佈，但 bugs 實在太多。不久發佈的 3.0.1 版使 templates 成爲可用的 C++語言特性。

一個物體被具現化。最終，迭代過程會使所有具現化需求都獲得滿足，從而讓聯結器成功生成一個完整的可執行檔。

原始的 Cfront 方案有一些嚴重缺點：

- 聯結時間大大膨脹了，不僅因爲預聯結器的額外開銷，每次必要的再編譯和再聯結也需要高額的時間代價。以 Cfront 爲基礎之編譯系統的用戶們說，如果使用其他具現化機制用掉大約一個小時，那麼使用「迭代式具現化」機制會花掉好幾天。
- 診斷資訊（錯誤資訊和警告）被推遲至聯結期。漫長的聯結時間對程式員而言非常痛苦，他們可能不得不等上數小時之久而最後發現只是 template 定義式有個筆誤。
- 編譯器必須特別記住內含特定定義的源碼位於何處（前述步驟 1）。明確地說，Cfront 使用了一個集中式容器，它必須實現出類似「查詢式具現化」機制中的資料庫機能。而且，最初的 Cfront 實作品並不支援並行編譯（concurrent compilations）。

儘管有這些不足，仍有兩個 C++ 編譯系統使用改良後的「迭代式具現化」機制，並開拓了更高級的 C++ template 特性[44]，它們是 Edison Design Group（EDG）的版本和 HP 的 aC++ 版本[45]。接下來我將介紹 EDG 開發的技術，展示其 C++ 前端（front-end）技術[46]。

EDG 的迭代方式是在預聯結器和各個編譯步驟之間建立雙向通信：預聯結器可將「特定編譯單元所需的具現化」透過一個「具現化申請檔」（instantiation request file）轉給編譯器處理；藉由「在 obj 檔中嵌入某些資訊」或「產生個別的 template 資訊檔」，編譯器有能力告訴預聯結器可能的 POI 位置。「具現化申請檔」及「template 資訊檔」與對應之待編譯檔同名，只不過分別帶有尾詞 `.ii` 和 `.ti`。迭代機制按以下步驟工作：

1. 編譯某個編譯單元的源碼時，EDG 編譯器讀取對應的 `.ii` 檔（如果存在的話），並產生需要的具現體。同時它將「原本可兌現的 POI」（譯註：兌現意指實現 `.ii` 檔案中的具現化請求）寫入當前編譯產生的 obj 檔，或寫入到一個獨立的 `.ti` 檔案中。同時也將這個檔案如何編譯的資訊寫入。

---

[44] 我們不敢說自己不偏不倚，然而諸如 member templates、偏特化、templates 內的名稱查詢新法、template 分離式模型等高級特性，的確都是這些公司首先實作出來的。

[45] HP 的 aC++是在 Taligent 公司（後被 IBM 收購）的技術上發展出來的。HP 並在 aC++中加入了「貪婪具現化」機制，並把它當作預設的具現化機制。

[46] EDG 並不向終端用戶出售 C++ 編譯器，而是向其他編譯器廠商提供 C++編譯器實作品的基礎可移植組件。其他廠商可將 EDG 的技術整合到自己的編譯器中，形成一個完備的、特定平台的解決方案。某些 EDG 客戶選擇保留可移植的「迭代式具現化」機制，但可容易將它改編以適應「貪婪具現化」機制（後者不具可移植性，因爲它倚賴某些特殊的聯結器特性）。（譯註：關於前端 front-end 的具體描述，可參考 http://www.edg.com/cpp.html）

2. 聯結步驟被預聯結器攔截。預聯結器檢查參與聯結的 obj 檔和對應的 .ti 檔，如果發現有「尚未產生的具現體」，就把請求寫入可兌現此一請求的編譯單元所對應的 .ii 檔中。
3. 如果預聯結器察覺到任何 .ii 檔被更改過，就重新呼叫編譯器（步驟 1）來處理對應的源碼檔，然後迭代（反覆）預聯結步驟。
4. 當所有具現體都被產出完畢（迭代完成），聯結器便會進行實際的聯結工作。

這個機制解決了並行建造（concurrent build）問題，它把所有用到的資訊分置於各編譯單元中。使用這種機制時，聯結時間仍然可能遠遠超過使用「貪婪式具現化」或「查詢式具現化」機制所用掉的時間，但由於聯結工作實際並未進行（只是「預聯結」而已），時間的增長並不如想像中那般可怕。更重要的是，由於聯結器在各個 .ii 檔中保留了相關資訊，下次編譯、聯結程式時，這些檔案仍然可用。特別是在對源碼進行一些修改後，程式員會重新編譯、聯結這些修改後的檔案，這時所引發的編譯動作就可根據上次編譯留存的 .ii 檔，立刻進行具現化工作。如果運氣好，聯結期需要重新編譯的可能性並不大。

現實中 EDG 方案的表現非常好。雖然一個徹頭徹尾的編譯聯結過程仍然比其他機制耗用更多時間，但後續的編譯聯結速度很有競爭力。

10.5 明確具現化（Explicit Instantiation）

明確指定（產生）某 template 特化體的 POI（具現點）也是可能的，稱爲「明確具現化」指令。語法上規定，「明確具現化」指令由「關鍵字 template」及「待具現化之特化體的宣告」構成。例如：

```
template<typename T>
void f(T) throw(T)     //譯註：最後面那個 throw(T) 就是"exception spec."（異常規格）
{
}

// 四個合法的「明確具現化」指令：
template void f<int>(int) throw(int);
template void f<>(float) throw(float);
template void f(long) throw(long);
template void f(char);
```

注意，上述所有具現化指令都合法。template arguments 可被編譯器推導而出（見第 11 章），異常規格（exception specifications）可省略。如果它們沒被省略，就必須和 template 的那一個匹配。

> 譯註："exception specification" 就意義而言應該譯爲「異常明細」（spec.有「明細」之義）。但大多數人恐怕不易接受這一譯詞，總直覺把 spec.想爲「規格」。所以還是譯爲「異常規格」）

class templates 的成員也可以這麼地明確具現化：

```
template<typename T>
class S {
public:
    void f() {
    }
};

template void S<int>::f();
template class S<void>;
```

不僅如此，明確具現化一個 class template 特化體，會使其所有成員都被明確具現化。

許多早期 C++ 編譯系統剛加入對 template 的支援能力時，並不具備自動具現化的能力。某些系統要求：一個程式用到的 function template 特化體必須在某個獨一無二的位置上由程式員手動進行具現化。手動式具現化通常以編譯器實作品專屬之#pragma 指令完成。

爲了這樣的原因，*C++ Standard* 以清晰的語法「法定化」這種具現指令。*C++ Standard* 並指出，在一個程式中，每個 template 特化體最多只能有一次明確具現化。而且如果一個 template 特化體被明確具現化（explicit instantiated），就不能被明確特化（explicit specialized）；反之亦然。

在手動式具現化的原始含義下，上述限制似乎並不會帶給人們任何煩惱，然而實際上它們可能讓我們吃到苦頭。

首先考慮一個程式庫實作者發佈了某個 function template 的第一版：

```
// toast.hpp 檔案
template<typename T>
void toast(T const& x)
{
    ...
}
```

客戶端程式碼可以含入這個表頭檔，並明確具現化這個 template：

```
// 客戶端程式碼
#include "toast.hpp"

template void toast(float);
```

不幸的是，如果程式庫作者也將 toast<float> 明確具現化了，便會使客戶端程式碼非法。如果這個程式庫是由不同廠家實作出來的標準程式庫，更有可能發生問題。有些實作品明確具現化了一些標準 templates，另一些實作品沒有這麼做（抑或它們明確具現化的是另一些標準 templates），於是客戶端程式碼無法以一種可移植性的方式來對程式庫組件進行明確具現化。

本書撰寫之際（2002），C++ 標準委員會似乎傾向於：如果一個明確具現化指令（explicit instantiation directive）接在一個針對相同物體的明確特化（explicit specialization）之後，那麼該指令不具任何效果（此議尚未定論，而且有可能被否決，因爲技術上似乎不可行）。

Templates 明確具現化的當前限制的第二個挑戰是：如何提高編譯速度。很多 C++ 程式員都發覺，template 自動具現化機制對編譯時間有不小的負面影響。一種改善編譯速度的技術是：在某些位置上手動具現化某些特定的 template 特化體，而在其他所有編譯單元中不做這些具現化工作。惟一一個具有可移植性並能阻止「在其他單元發生具現化」的方法是：不要提供 template 定義式，除非是在它被明確具現化的編譯單元內。例如：

```
// 編譯單元 1
template<typename T> void f();  // 無定義：避免在本單元中被具現化

void g()
{
    f<int>();
}

// 編譯單元 2
template<typename T> void f()
{
}

template void f<int>();         // 手動具現化

void g();

int main()
{
    g();
}
```

這個解法運作良好，但程式員必須擁有「提供 template 介面」之源碼檔案的控制權。通常程式員無法擁有這種控制權。「提供 template 介面」的源碼檔案無法被修改，而且它們通常會提供 templates 定義式。

一個有時候被用到的技巧是，在所有編譯單元中宣告某 template 特化體（這可禁止該特化體被自動具現化），惟有在「該特化體被明確具現化」的編譯單元中例外。爲示範這種作法，讓我們修改前一個例子，用以含入一個 template 定義式：

```
// 編譯單元 1
template<typename T> void f()
{
}

template<> void f<int>();       // 宣告但不定義

void g() {
    f<int>();
}

// 編譯單元 2
template<typename T> void f()
{
}

template void f<int>();         // 手動具現化

void g();

int main()
{
    g();
}
```

不幸的是這裡假設「對一個明確特化之特化體的呼叫」等價於「對一個相匹配之泛型特化體的呼叫」。這個假設並不正確。很多 C++ 編譯器爲這兩個物體生成了不同的重整名稱（mangled names）[47]。如果使用這些編譯器，obj 碼就無法被聯結成一個完整的 exe 檔。

譯註：下面是 VC6 生成的重整名稱（mangled names）：

```
?_Copy@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@AAEXI@Z
```

其函式原形是：

```
void std::basic_string<char,struct std::char_traits<char>,
                       class std::allocator<char> >::_Copy(unsigned int)
```

某些編譯器提供了一種擴展語法，用來向編譯器指示：一個 template 特化體不應該在其編譯單元內被具現化。一個普及（但不符合標準）的語法形式是：在一個「明確具現化指令」之前添加關鍵字 extern。如果編譯器支援這種語法，前例中的第一個源碼檔可改變如下：

[47] 聯結器看到的所有名稱都是重整後的名稱(mangled names)，由函式名、參數特徵、template arguments 以及其他可能屬性組合而成的一個全域性獨一無二的名稱，和任何合法的重載函式名都不衝突。

```
// 編譯單元 1
template<typename T> void f()
{
}

extern template void f<int>();  // 宣告但未定義

void g()
{
    f<int>();
}
```

10.6 後記

本章討論了兩個相關但有區別的問題：C++ template 編譯模型和各種 C++ template 具現化機制。

編譯模型（compilation model）用來在程式的各個編譯階段決定一個 template 的意義。更明確地說，它決定了「template 被具現化」時其內各個構件的意義。名稱查詢（name lookup）當然是編譯模型中最基本的部份。當我們談論置入式（inclusion）模型和分離式（separation）模型時，我們就是在談論編譯模型。這些模型是語言定義的一部份。

具現化機制是定義於 C++ 語言之外的機制，它使 C++ 編譯器可以正確產生具現體。這些機制可能受制於聯結器或其他開發工具。

然而最早的（Cfront）templates 編譯器實作品超越了這兩個概念。它爲 template 具現體產生新的編譯單元，並使用特殊的源碼檔案組織方式。這些新編譯單元使用「本質上爲置入式模型」的方式編譯（雖然其「名稱查詢規則」與置入式模型所使用者大有不同）。因此儘管 Cfront 沒有實作出 templates 分離式模型，但它使用隱式置入式模型，產生分離式模型的幻覺。很多後來的實作品（例如 Sun Microsystems）預設使用「隱式置入式模型」機制，或把這種機制作爲可選方式（例如 HP、EDG），給那些使用 Cfront 開發的程式碼提供一定的相容性。

下面例子展示 Cfront 實作方案的細部：

```
// template.hpp 檔案
template<class T>      // Cfront 不支援關鍵字 typename
void f(T);
```

```
//File template.cpp:
template<class T>     // Cfront 不支援關鍵字 typename
void f(T)
{
}

// app.hpp 檔案
class App {
    ...
};

// main.cpp 檔案
#include "app.hpp"
#include "template.hpp"

int main()
{
    App a;
    f(a);
}
```

在聯結期，Cfront 的「迭代式具現化」機制會產生一個新編譯單元，其中含入一些檔案，它期望那些檔案含有它「在表頭檔中發現的 templates」的實作碼（定義式）。Cfront 的習慣是將表頭檔副檔名.h（或類似者）替換爲.c（或.C 或.cpp）。這時它生成的編譯單元變成：

```
// main.cpp 檔案
#include "template.hpp"
#include "template.cpp"
#include "app.hpp"

static void _dummy_(App a1)
{
    f(a1);
}
```

這個編譯單元會在編譯期獲得一個特殊參數，禁止爲「定義於 included file 中的任何物體」生成 obj code。這將使得「含入 template.cpp」這一事實不會造成「template.cpp 之中（可能）含有的任何可聯結物（linkable entities）產生重複定義」。

函式_dummy_()被用來建立一個 reference，指向「必須被具現化」的那個特化體。請注意表頭檔的順序重新排過了：Cfront 使用「表頭檔分析程式」（header analysis code）去除新編譯單元中無用的表頭檔。不幸的是，由於可能存在「跨表頭檔作用域」的巨集，導致這個技術並不穩固。

與之對比的是，標準「C++ 分離式模型」將兩個（或多個）編譯單元分別編譯，並使用 ADL 跨越編譯單元，產生一個「可使用各編譯單元內之物體」的具現體。由於這種機制並不倚賴檔案含入關係（*not based on inclusion*），也就不倚賴特殊的表頭檔組織（排列）方式，而編譯單元中的巨集也不會「污染」其他編譯單元。然而就像本章先前所說，巨集並不是 C++ 中惟一帶來驚奇的東西，「匯出模型」（export model）是另一種形式的「污染」。

11

Template 引數推導
Template Argument Deduction

如果你在每一個 function template 呼叫式中明確指定 template arguments（例如 `concat<std::string, int>(s, 3)`），程式碼會顯得笨拙又難看。幸運的是 C++ 編譯器通常可以自動判定你所要的 template arguments 的型別，這是透過一個名爲 template argument deduction（模板引數推導）的強大機制完成的。

本章詳細解說 template argument deduction 細節。正如其他 C++ 特性一樣，很多推導規則所產生的結果都符合人們的直觀認知。本章帶來的扎實基礎可以讓我們避免對更多情況感到驚訝。

11.1 推導過程（Deduction Process）

編譯器會比對「呼叫式內某引數的型別」和「function template 內對應的參數化型別」，並試圖總結出「被推導的一或多個參數」的正確替換物。每一對「引數-參數」獨立分析。如果推導結束時結果有異，則推導失敗。考慮下面這個例子：

```
template<typename T>
T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

int g = max(1, 1.0);
```

在這裡，第一個 call argument 的型別爲 `int`，因此 `max()` 的 template parameter `T` 被暫時推導爲 `int`。然而第二個 call argument 的型別爲 `double`，因此 `T` 應該爲 `double`：這與第一個推導結果衝突。注意，我們說的是「推導失敗」而不是「程式非法」；推導程序還是有可能成功找到另

一個名爲 max 的 template（因爲 function templates 可以重載，就像常規 functions 那樣；見 2.4 節, p.15 和第 12 章）。

即使所有的 template parameters 推導結果前後一致，但如果以引數替換進去後會導致函式宣告的其餘部份變爲非法，推導過程還是有可能失敗。例如：

```
template<typename T>
typename T::ElementT at (T const& a, int i)
{
    return a[i];
}

void f (int* p)
{
    int x = at(p, 7);
}
```

在這裡 T 被推斷爲 **int***（T 只在一個參數型別中出現，因此顯然不會發生衝突）。然而，將回返型別 T::ElementT 中的 T 替換爲 **int*** 顯然是非法的，因此推導失敗[48]。錯誤訊息大約會說『找不到與呼叫式 at()匹配的東西』。但如果你明確指定所有 template arguments，推導程序就沒有機會成功匹配另一個（可能存在的）template，這時的錯誤訊息會是『at()的 template argument 非法』。你可以使用你喜歡的編譯器，把下面例子的錯誤診斷訊息和前一例做個對比：

```
void f (int* p)
{
    int x = at<int*>(p, 7);
}
```

我們還需要探究「引數-參數」匹配是如何進行的。今後的描述將出現符號 **A** 和 **P**，意義如下：將型別 **A**（由 argument type 導出）匹配至一個參數化型別 **P**（由 template parameter 的宣告導出）；如果參數被宣告爲以 *by reference* 方式傳遞，則 **P** 表示被指涉（*referenced*）型別，**A** 爲引數型別；否則 **P** 表示宣告之參數型別，**A** 是個「由 array 或 function 退化而成[49]」並「去除外圍之 const 和 volatile 飾詞」的 pointer 型別。例如：

[48] 在這種情況下，推導失敗會引發編譯錯誤。然而這遵循 SFINAE 法則（請參考第 8.3.1 節, p.106）：如果存在另一個函式可使推導成功，則程式碼仍然合法。

[49] 退化（decay）的意義是：「function 型別和 array 型別轉換至 pointer 型別」的隱式轉型動作。

```
template<typename T> void f(T);      // P 爲 T

template<typename T> void g(T&);     // P 仍爲 T

double x[20];

int const seven = 7;

f(x);         // nonreference parameter：T 爲 double*
g(x);         // reference parameter：T 爲 double[20]
f(seven);     // nonreference parameter：T 爲 int
g(seven);     // reference parameter：T 爲 int const
f(7);         // nonreference parameter：T 爲 int
g(7);         // reference parameter：T 爲 int => 錯誤：不能把 7 當做 int&傳遞
```

對於呼叫式 `f(x)`，`x` 由 array 型別退化爲 `double*`型別，`T` 被推導爲該型別。呼叫式 `f(seven)` 中，`const` 飾詞被卸除，從而 `T` 被推導爲 `int`。呼叫式 `g(x)`中，`T` 被推導爲 `double[20]`（並無發生型別退化）。類似情況，`g(seven)`的引數是一個 `int const` 型別的左值（lvalue），此外由於 `const` 和 `volatile` 飾詞在匹配 reference parameters 時不被卸除，因此 `T` 被推導爲 `int const`。然而請注意呼叫式 `g(7)`中 `T` 被推導爲 `int`（因爲 nonclass rvalue 運算式不能含有帶 `const` 或 `volatile` 飾詞的型別），因此呼叫失敗，因爲引數 `7` 不能傳給一個 `int&` 型別的參數。

當字串字面常數（string literals）型別的引數被繫結至 reference parameters 時，不會發生退化，這可能使你驚訝。重新考慮先前的 `max()` template：

```
template<typename T>
T const& max(T const& a, T const& b);
```

很多人希望在 `max("Apple", "Pear")`運算式中，`T` 被推導爲 `char const*`，這合乎常情。然而 `"Apple"` 的型別卻是 `char const[6]`，`"Pear"` 的型別是 `char const[5]`。array 型別至 pointer 型別的退化過程並未發生（因爲 `max` 的參數使用 *by reference* 傳遞方式），因此如果推導成功，`T` 必須既是 `char[6]`又是 `char[5]`，那當然不可能。請參考 5.6 節, p.57，那裡有更多相關討論。

11.2 推導之前後脈絡（Deduced Contexts）

比 `T` 更複雜的參數化型別也可以被匹配到一個給定的引數型別上。下面是一些基本例子：

```
template<typename T>
void f1(T*);

template<typename E, int N>
void f2(E(&)[N]);
```

```
template<typename T1, typename T2, typename T3>
void f3(T1 (T2::*)(T3*));   // 譯註：灰色部分即是 pointer-to-member type
     // 譯註：意思是：f3 的參數是個 T2 成員函式，該函式的回返型別爲 T1，接受一個 T3* 參數。

class S {
public:
   void f(double*);
};

void g (int*** ppp)
{
   bool b[42];
   f1(ppp);       // T 被推導爲 int**
   f2(b);         // E 被推導爲 bool，N 被推導爲 42
   f3(&S::f);     // 推導結果：T1=void，T2=S，T3=double
}
```

複雜的型別宣告是由基本構件（pointer、reference、array、function 宣告符號、pointer-to-member 宣告符號、template-id 等等）搭建而成的，匹配過程從上層構件開始，在各組成元素中遞迴進行。公正地說，大多數「型別宣告構件」（type declaration constructs）都可以用這種方式進行匹配，這些構件被稱爲 deduced contexts（推導之前後脈絡）。有些構件並不是 deduced contexts：

- 受飾型別（qualified type）名稱。例如 `Q<T>::X` 不會被用以推導 template parameter `T`。
- 「非單純只是個非型別參數（nontype parameter）」的一個非型別算式（nontype expression）。例如型別名稱 `S<I+1>` 不被用於 `I` 的推導；推導機制也不認爲 `T` 能夠與 `int(&)[sizeof(S<T>)]` 的參數匹配。

譯註：pointer-to-member 型別包括兩個組成：(1) member class 和(2) member type。例如在 `int* C::*` 這一 pointer-to-member 型別中，(1) 是 `C` 而 (2) 是 `int*`。兩者都是 deducible contexts。

不必對這些限制太過驚訝，因爲通常說來推導結論並不惟一（甚至不是有限個），雖然受飾型別（qualified）的名稱有時很容易被忽視。一個 nondeduced context 並不意味程式有錯，甚至不意味正被分析的參數不能用於型別推導。考慮下面這個更複雜的例子：

```
// details/fppm.cpp

template <int N>
class X {
public:
   typedef int I;
   void f(int) {
   }
};
```

```
template<int N>
void fppm(void (X<N>::*p)(typename X<N>::I));
     // 譯註：意思是：fppm 的參數是個「X<N> 成員函式」，
     //       該函式的回返型別爲 void，接受一個 X<N>::I 參數。

int main()
{
    fppm(&X<33>::f);  // 正確：N 被推導爲 33
}
```

譯註：g++ 3.2、ICL7.1 和 VC6 均編譯成功，唯獨 VC7.1 出錯。即使將 `void f(int)` 改爲 `void f(I)`，仍出現以下錯誤訊息：

```
2.cpp(16) : error C2784: 'void fppm(void (__thiscall X<N>::* )(X<N>::I))' : could
not deduce template argument for 'void (__thiscall X<N>::* )(X<N>::I)'
from 'void (__thiscall X<N>::* )(X<N>::I)' with [ N=33 ]
2.cpp(12) : see declaration of 'fppm'
```

不知如何解決。

在 function template `fppm()` 中，子構件 `X<N>::I` 是個 nondeduced context。然而 pointer-to-member 型別中的 `X<N>` 卻是個 deducible context，而且當由此推導出的參數 `N` 被塞入 nondeduced context 時，我們將獲得一個與實際引數`&X<33>::f` 相容的型別。於是「引數-參數」推導成功。

對一個「完全以 deduced contexts 構建而成」的參數型別進行推導，反而可能導致矛盾結果。以下例子假設 class templates `X` 和 `Y` 都已正確宣告：

```
template<typename T>
void f(X<Y<T>, Y<T> >);

void g()
{
    f(X<Y<int>, Y<int> >());   // OK
    f(X<Y<int>, Y<char> >());  // ERROR：推導失敗
}
```

第二個 function template `f()` 呼叫式的問題在於，根據兩個引數推導出不同的 template parameter `T`，這是不合法的。上述兩種情況中，call arguments 都是「由 class template `X` 的 *default* 建構式建立」的一個暫時物件（temporary object）。

11.3 特殊推導情境（Special Deduction Situations）

有兩種情形使得 `(A,P)` 並非由 function template 的呼叫引數和 function template 的參數獲得。第一種情況發生在對 function template 取址時。這時 `P` 是 function template 宣告式中的參數化型別，`A` 是被初始化或被賦值之指標的「檯面下（underlying）函式型別」。例如：

```
template<typename T>
void f(T, T);

void (*pf)(char, char) = &f;     //譯註：取 function template 的位址
```

這個例子中，**P** 是 `void(T, T)`，**A** 是 `void(char, char)`，推導成功，並將 `T` 替換爲 `char`，`pf` 被初始化爲「`f<char>` 特化體的位址」。

另一種特殊情形發生在 conversion operator templates（轉型運算子模板）。例如：

```
class S {
public:
    template<typename T, int N> operator T&();
};
```

這時**(P,A)**的獲得猶如涉及某引數（其型別等於「吾人企圖轉換過去之目標型別」）及某參數型別（也就是「轉型運算子之回返型別」）。下面例子展示了上述情況的一個變異：

```
void f(int (&)[20]);

void g(S s)
{
    f(s);
}
```

這裡我們試圖把 `S` 轉型爲 `int(&)[20]`。因此型別 **A** 爲 `int[20]`，型別 **P** 爲 `T`。推導成功，`T` 被替換爲 `int[20]`。

11.4 可接受的引數轉型（Allowable Argument Conversions）

通常，template 推導機制會試圖尋找 function template parameters 的替換品，使參數化型別 **P** 與型別 **A** 完全相同。然而當它無法達到這個目標時，推導機制容許以下差異：

- 如果原始參數使用 *by reference* 傳遞方式，則被替換型別 **P** 比型別 **A** 可以多帶上 `const` 飾詞或 `volatile` 飾詞。
- 如果型別 **A** 是一個 pointer 型別或 pointer-to-member 型別，**A** 可以透過（添加）一個 `const` 飾詞或 `volatile` 飾詞，轉型爲被替換型別 **P**。
- 除非推導程序因 conversion operator template 而發生，否則被替換型別 **P** 可以是「型別 **A** 的 base class 型別」；或者當 **A** 是個 pointer-to-X 型別時，**P** 可以是個 pointer-to-base-class-of-X。例如：

```
template<typename T>
class B {
};
```

```
template<typename T>
class D : public B<T> {
};

template<typename T> void f(B<T>*);

void g(D<long> dl)
{
    f(&dl);  // 推導成功：T 被替換爲 long
}
```

當推導機制無法進行完全匹配時，才會考慮這種寬鬆的匹配條件。即使如此，惟有當推導機制找到惟一一個「使型別 **A** 符合被替換型別 **P**」的替換方式時，推導才會成功。

11.5 Class Template Parameters（類別模板參數）

Template argument deduction 專只用於 function templates 和 member function templates。更明確地說，class template 的引數不從外界對其某一建構式的 call arguments 中推導出來。例如：

```
template<typename T>
class S {
public:
    S(T b) : a(b) {
    }
private:
    T a;
};

S x(12);   // 錯誤：class template parameter T 不從建構式的 call argument 12 中推導出來
```

11.6 預設的呼叫引數（Default Call Arguments）

在 function templates 中你也可以指定預設的函式呼叫引數，就像在常規函式中一樣：

```
template<typename T>
void init (T* loc, T const& val = T())
{
    *loc = val;
}
```

事實上，正如這個例子所展示，預設的 funcion call arguments 也可以倚賴某個 template parameter。這種受控預設引數（dependent default argument）只在呼叫端沒有使用明確引數時才會被具現化。正是這個法則使得下面的例子合法：

```
class S {
public:
    S(int, int);
};

S s(0, 0);

int main()
{
   init(&s, S(7, 42));
       // 當 T=S 時 T()不合法，但由於給定了一個明確引數，因此不需將預設引數 T()具現化
}
```

即使預設的 funcion call arguments 並非受控的（dependent），它也無法被用來推導 template arguments。這意味下面這個例子不合法：

```
template<typename T>
void f (T x = 42)
{
}

int main()
{
    f<int>();      // OK：T = int
    f();           // ERROR：無法從預設的 call argument 推導 T
}
```

11.7 Barton-Nackman Trick

1994 年 John J. Barton 和 Lee R. Nackman 展現一種 template 技術，他們稱之爲 restricted template expansion（受限的模板擴展技術）。此技術的動機源於一個事實：當時的 function template 不能被重載[50]，並且大多數編譯器不支援 namespaces。

[50] 如果想了解 function template 的重載機制在現代化 C++ 編譯器中是如何工作的，第 12.2 節, p.183 值得一讀。

爲闡示這種技術，假設我們有一個 class template Array，並爲它定義一個 *equality* 運算子 operator==。一種可能是將這個運算子定義爲 class template 的成員，但這不是個好辦法，因爲第一引數（繫結於 this 指標）的型別如果和第二引數不同，兩者的轉型規則也不同。由於 operator== 應該符合交換律，把它定義爲 namespace 作用域下的函式會更好些。一個很自然的實作法可能看起來這個樣子：

```
template<typename T>
class Array {
public:
    ...
};

template<typename T>
bool operator == (Array<T> const& a, Array<T> const& b)
{
    ...
}
```

如果 function templates 不能被重載，這就引發一個問題：在此作用域中不能定義其他的 operator== template，然而其他 class templates 很可能也需要定義這樣一個 operator==。Barton 和 Nackman 解決這個問題的辦法是：把 operator==定義爲 class 內的一個普通的 friend 函式：

```
template<typename T>
class Array {
public:
    ...
    friend bool operator == (Array<T> const& a,
                             Array<T> const& b) {
        return ArraysAreEqual(a, b);
    }
};
```

假設這個 Array 被具現化爲 float 型別，導致 friend operator function 被宣告，但是注意，這個函式本身並非是某個 function template 的具現體。它只是個普通的 non-template function，因具現化過程的影響而被插入 global 作用域中。由於它是個 non-template function，可以和其他重載形式的 operator== 共存，即使當時 C++ 語言尚未能夠支援 function template 重載。Barton 和 Nackman 把這種技術稱爲 restricted template expansion（受限的模板擴展），因爲它可以避免使用一個適用於所有型別的 operator==(T,T)（也就是一個 unrestricted expansion, 毫不受限的擴展）。

由於 operator== (Array<T> const&, Array<T> const&)定義在 class 定義式內，也就隱隱成爲一個 inline 函式，因此我們將這個函式的實作委託（*delegate*）給一個 function template

`ArraysAreEqual()`，後者不必須爲 inline，不致於和同名的其他 templates 起衝突。

Barton-Nackman trick 的最初目的已經消失了，但研究它仍然很有趣，因爲它允許我們在生成 class template 具現體的同時，生成 non-template functions。由於後者並非由 function templates 產生，所以無需 template argument 的推導，但是仍然符合正常的重載決議規則（見附錄 B）。理論上這意味當把一個 friend 函式匹配到一個特定呼叫場所（call site）時，匹配過程會考慮進行隱式轉型。然而這並沒帶來什麼明顯好處，因爲在標準 C++ 中（這和 Barton 及 Nackman 發明該手法時的 C++ 已是今非昔比），被植入到 global 作用域中的 friend 函式並非無條件地在其外圍作用域中可見：只有 ADL 適用時它們才可見。這意味呼叫式中的引數必須將「含有該 friend 函式之 class」做爲相關 class。如果函式引數是個不相關的 class 型別，但可被轉型爲「含有該 friend 函式之 class」，這個 friend 函式也是不可見的。例如：

```
class S {
};

template<typename T>
class Wrapper {
private:
    T object;
public:
    Wrapper(T obj) : object(obj) {   // 可從 T 隱式轉型到 Wrapper<T>
    }
    friend void f(Wrapper<T> const& a) {
    }
};

int main()
{
    S s;
    Wrapper<S> w(s);
    f(w);     // OK：Wrapper<S>是 w 的相關聯 class
    f(s);     // ERROR：Wrapper<S>與 S 不相關聯
}
/* 譯註：VC6/VC7.1/ICL7.1/g++ 3.2 都能順利編譯此例，然而這未符合標準。Wrapper<S> w(s)
引發 friend 宣告被植入 global namespace 中，這是源自 Cfront 編譯器的行爲；後來的 C++ 標
準明確禁止了這一點。上述編譯器之所以還容許這種做法，大概是爲了相容於極多的既有程式碼。*/
```

在這個例子中，呼叫式 `f(w)` 是合法的，因爲函式 `f()` 是宣告於 `Wrapper<S>` 內的一個 friend 函式，而 `Wrapper<S>` 與引數 `w` 相關聯[51]。然而呼叫 `f(s)` 時，friend 函式宣告 `f(Wrapper<S> const&)`

[51] 注意 `S` 也是一個與 `w` 相關聯的 class，因爲它是 w 的型別的 template argument。

並不可見，因爲 class `Wrapper<S>` 與型別爲 `S` 的引數 `s` 不相關聯。即使存在一個從 `S` 到 `Wrapper<S>` 的隱式轉換（透過 `Wrapper<S>` 建構式），這個轉換也不會被考慮，因爲候選函式 `f` 並沒有先被找到。

結論是，與「簡單定義一個常規的 function template」相較，「在 class template 內定義一個 friend 函式」並沒有多大好處。

11.8 後記

針對 function template 而做的 template argument deduction 是 C++ 原始設計的一部份。事實上直到很多年後，由 explicit template arguments 帶來的另一種引數推導機制才成爲 C++ 的一部份。

「Friend 名稱植入（name injection）」被許多 C++ 語言專家認爲有害，因爲它使「程式合法性」過於受到「具現化的順序」的影響。Bill Gibbons（當時致力編寫 Taligent 編譯器）代表解決這個問題的主要聲音，因爲去除了「具現化順序」的依存關係後，會使得新穎而有趣的 C++ 開發環境成爲可能（聽說 Taligent 正爲此而努力）。然而 Barton-Nackman trick 需要某種形式的「friend 名稱植入（name injection）」，正是因此才使得後者以目前的（弱化後的）形態留存於 C++ 語言至今。

有趣的是，很多人都聽說過 Barton-Nackman trick，但很少有人知道它的實際描述。於是你會發現很多涉及 friends 及 templates 的技巧都被誤稱爲 Barton-Nackman trick（例如 16.5 節, p.299）。

12

特化與重載
Specialization and Overloading

目前爲止，我們已經學習了 C++ 如何使一個泛型定義被展開爲一族系相關的 classes 或 functions。雖然這個機制強勁有力，在許多情形下對某些操作而言，「以一個特定替換物取代泛化的 template paramters」遠遠達不到優化的要求。

就各流行編程語言對泛型編程的支援程度而言，C++ 可說是相當獨特，因爲它提供了一組豐富的特性供程式員透通地將一個泛型定義替換爲一個更特定（更專門）的定義。本章中我們將學習兩種機制，它們將純粹的泛型導向更多實用價值，它們是：(1) template 的特化 (2) function templates 的重載。

12.1 當泛型碼（Generic Code）不合用...

考慮下面例子：

```
template<typename T>
class Array {
private:
    T* data;
    ...
public:
    Array(Array<T> const&);
    Array<T>& operator = (Array<T> const&);

    void exchange_with (Array<T>* b) {
        T* tmp = data;
        data = b->data;
        b->data = tmp;
    }
```

```
    T& operator[] (size_t k) {
        return data[k];
    }
    ...
};

template<typename T> inline
void exchange (T* a, T* b)
{
    T tmp(*a);
    *a = *b;
    *b = tmp;
}
```

對於簡單型別，exchange()的泛型實作碼可以有效運作。然而對於「*copy* 操作代價高昂」的型別而言，這個泛型實作碼可能帶來更爲高昂的代價：既增加 CPU 負擔，也佔用更多記憶體。這遠不如一個量身而製的實作碼來得高效。在這個例子中，泛型實作碼需要一個對 Array<T> *copy* 建構式的呼叫，以及兩個對其 *copy assignment* 運算子的呼叫。對大型資料結構來說，這些 *copy* 過程往往涉及大量記憶體複製。然而 exchange()的功能往往可以利用「交換兩個內部指標」就達成，就像成員函式 exchange_with()所作所爲那樣。

12.1.1 透通訂製（Transparent Customization）

前面的例子中，成員函式 exchange_with()爲泛化的 exchange()函式提供一個有效的特殊實作，但是從數種理由來看，使用另一個函式很不方便：

1. class Array 的使用者不得不記住它有個額外的介面，並且得花心思在必要時使用它。
2. 實作泛型演算法時，你通常無法區分何時該使用這個額外介面，何時不用它。例如：

```
template<typename T>
void generic_algorithm(T* x, T* y)
{
    ...
    exchange(x, y);  // 我們如何選擇正確的演算法？
    ...
}
```

出於這些考量，C++ templates 提供了一些辦法，讓我們得以透通地訂製 function templates 和 class templates。對於 function templates 來說，你可以借助重載機制達到目的。例如我們可以寫出一組重載的 quick_exchange() function templates 如下：

```
template<typename T> inline
void quick_exchange(T* a, T* b)                    // (1)
{
    T tmp(*a);
    *a = *b;
    *b = tmp;
}

template<typename T> inline
void quick_exchange(Array<T>* a, Array<T>* b)      // (2)
{
    a->exchange_with(b);
}

void demo(Array<int>* p1, Array<int>* p2)
{
    int x, y;
    quick_exchange(&x, &y);                        // 使用(1)
    quick_exchange(p1, p2);                        // 使用(2)
}
```

對 quick_exchange()的第一次呼叫有兩個 `int*` 引數，推導機制發現只有第一個 template（宣告於(1)處）適用，於是將 `T` 替換爲 `int`，不存在「究竟該呼叫哪個函式」的疑慮。第二次呼叫匹配兩份 templates：「把第一個 template 中的 `T` 替換爲 `Array<int>`」或是「把第二個 template 中的 `T` 替換爲 `int`」都可行。不僅如此，兩種替換方法引發的函式參數型別都與第二次呼叫中的引數型別匹配。通常這會使我們認爲這個呼叫有歧義性，然而（稍後會有討論）C++ 語言認爲第二個 template 比第一個 template 更特別（more specialized），因此在其他因素平手的情況下，更特別（更特化）的 template 勝出。

12.1.2 語意的透通性（Semantic Transparency）

上一節所說的重載機制可以幫助我們實現具現化過程的透通訂製，但認識到以下這一點也是很重要的：這種「透通性」倚賴諸多實作細節。爲說明這個問題，請考慮先前的 `quick_exchange()` 方案。儘管兩個泛型演算法以及我們爲 `Array<T>` 訂製的演算法最終都能交換兩個指標所指的值，但不同的操作帶來的副作用差異很大。

考慮下面這個有點過於戲劇化的例子。請比較「struct objects 之間的互換」和「Array<T> 之間的互換」：

```
struct S {
    int x;
} s1, s2;

void distinguish (Array<int> a1, Array<int> a2)
{
    int* p = &a1[0];
    int* q = &s1.x;
    a1[0] = s1.x = 1;
    a2[0] = s2.x = 2;
    quick_exchange(&a1, &a2);  // 呼叫後*p 仍然爲 1
    quick_exchange(&s1, &s2);  // 呼叫後*q 爲 2
}
```

這個例子顯示，在呼叫 quick_exchange()之後，「指向第一個 Array」的指標 p 改而「指向第二個 Array」。然而呼叫 quick_exchange()之後，「指向 non-Array s1」的指標仍然指向 s1，只是指標所指的值互換了。這個差別已經足夠迷惑這個 template 的用戶。前導詞 quick_ 倒是頗能吸引人們注意力：有一條捷徑可以實現你所渴求的操作。然而最初的泛化 exchange() template 也可以對 Array<T> 進行優化：

```
template<typename T>
void exchange(Array<T>* a, Array<T>* b)
{
    T* p = &(*a)[0];
    T* q = &(*b)[0];
    for (size_t k = a->size(); k-- != 0; ) {
        exchange(p++, q++);
    }
}
```

這個版本相對泛化版本的優勢在於，不需要一個大型的暫態 Array<T>。exchange() template 被遞迴呼叫，因此即使面對 Array<Array<char> > 它也能有不錯的表現。另外請注意，這個較爲特化的 template 版本並沒有被宣告爲 inline，因爲它內部做了相當數量的工作；而原始的泛化版本是 inline，因爲它只進行少數幾個操作（但每個操作的代價都有可能很高）。

12.2 ㄓㄨㄥˋ載 Function Templates

前一節中我們知道，同名的兩個 function templates 可以共存，即使它們可能在被具現化後擁有完全相同的參數型別。下面是個小例子：

```
// details/funcoverload.hpp

template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}
//譯註：對於呼叫式 f((int*)0)，VC7.1/g++ 3.2 可以正確解析爲
//      呼叫第二個 function template；然而 VC6 認爲此一呼叫模稜兩可（帶有歧義）。
```

當第一個 template 中的 T 被替換爲 int*，如果把第二個 template 中的 T 替換爲 int，我們將得到「參數型別和回返型別完全相同」的兩個函式。不光是這些 templates 可以共存，它們的具現體（即使擁有完全相同的參數和回返型別）也可以共存。

下面這個例子展示：以明確指定的 template argument 語法生成兩個完全相同的函式：

```
// details/funcoverload.cpp

#include <iostream>
#include "funcoverload.hpp"

int main()
{
    std::cout << f<int*>((int*)0) << std::endl;
    std::cout << f<int>((int*)0)  << std::endl;
}
```

輸出結果爲：

```
1
2
```

爲了搞清楚這個例子，讓我們詳細分析呼叫式 `f<int*>((int*)0)`[52]。語法 `f<int*>`表示我們要將 template `f` 的第一個 template parameter 替換爲 `int*`，而不借助於引數推導機制。本例的 template `f` 不止一個，於是編譯器產出一個「內含兩個函式」的重載集合：從第一個 template 中產出 `f<int*>(int*)`，從第二個 template 中產出 `f<int*>(int**)`。call argument `(int*)0` 的型別爲 `int*`，只匹配重載集合中的第一個函式，因此最終喚起的是第一個函式。

你可以使用類似方法來分析第二個呼叫式。

譯註：以下三個小標題：12.2.1 Signatures, 12.2.2 Partial Ordering of Overloaded Function Templates 和 12.2.3 Formal Ordering Rules，均未譯。這些原文名詞在英文書籍、雜誌、網站、新聞組或論壇上被大量使用，硬譯並無好效果。但盼諒解。

12.2.1 Signatures（訊號式）

兩個函式可以共存於一個程式中，只要彼此的 signatures 不同。我們定義函式的 signature 包含以下資訊[53]：

1. 函式的非受飾（*unqualified*）名稱（或生成該函式之 function template 的名稱）
2. 該名稱的 class/namespace 作用域；以及它所宣告於其中的編譯單元（如果該名稱是內部聯結方式）
3. 該函式所帶的 const、volatile 或 const volatile 飾詞（如果它是一個成員函式並帶有上述飾詞）
4. 函式參數的型別（如果該函式是從一個 function template 生成，此處指的是 template paramters 被替換前的型別）
5. 回返型別（如果函式從一個 function template 生成）
6. template parameters 和 template arguments（如果函式從一個 function template 生成）

這意味著下面的 templates 和其具現體原則上可共存於一個程式中：

```
template<typename T1, typename T2>
void f1(T1, T2);

template<typename T1, typename T2>
void f1(T2, T1);
```

[52] 注意，算式 `0` 是個整數，而不是一個 null pointer 常數。在一個特殊隱式轉型後它變爲 null pointer 常數，然而這種轉型在 template 引數推導機制中不被考慮。

[53] 此處給出的定義不同於 *C++ Standard* 中的定義，但兩者等價。

```
template<typename T>
long f2(T);

template<typename T>
char f2(T);
```

然而，當這些函式被宣告於同一個作用域（scope）中，它們無法被使用。因爲只要具現化其中任意兩個，就會造成重載歧義性（overload ambiguity）。例如：

```
#include <iostream>

template<typename T1, typename T2>
void f1(T1, T2)
{
    std::cout << "f1(T1, T2)" << std::endl;
}

template<typename T1, typename T2>
void f1(T2, T1)
{
    std::cout << "f1(T2, T1)" << std::endl;
}

// 目前還好

int main()
{
    f1<char, char>('a', 'b');  // 錯誤：有歧義
}
```

在這裡，函式 `f1<T1=char, T2=char>(T1,T2)` 可以和函式 `f1<T1=char,T2=char>(T2,T1)` 共存，但是重載決議（overload resolution）機制永遠無法得知哪一個更好。如果這些 templates 出現在不同的編譯單元中，那麼兩個具現體可共存於同一個程式中（聯結器不會抱怨有重複定義，因爲這些具現體的 signatures 是可以區分的）：

```
// 編譯單元 1
#include <iostream>

template<typename T1, typename T2>
void f1(T1, T2)
{
    std::cout << "f1(T1, T2)" << std::endl;
}

void g()
{
    f1<char, char>('a', 'b');
}
```

```
// 編譯單元 2
#include <iostream>

template<typename T1, typename T2>
void f1(T2, T1)
{
    std::cout << "f1(T2, T1)" << std::endl;
}

extern void g();  // 定義於編譯單元 1

int main()
{
    f1<char, char>('a', 'b');
    g();
}
```

這個程式合法，其輸出爲：

```
f1(T2, T1)
f1(T1, T2)
```

12.2.2 Partial Ordering of Overloaded Function Templates（重載化函式模板的偏序規則）

重新考慮先前的例子：

```
#include <iostream>

template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}
```

```
int main()
{
   std::cout << f<int*>((int*)0) << std::endl;
   std::cout << f<int>((int*)0)  << std::endl;
}
```

我們發現，在替換了給定之 template argument list（<int*> 和 <int>）後，重載機制最終正確選擇了它要呼叫的函式。然而即使不提供明確的 template arguments，重載機制也會選中某個函式；這種情況下引數推導機制加入戰局。我們對 main() 函式稍做修改來討論這種機制：

```
#include <iostream>

template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}

int main()
{
    std::cout << f(0)       << std::endl;
    std::cout << f((int*)0) << std::endl;
}
```

考慮第一個呼叫式 f(0)：引數型別爲 int，與第一個 template 的參數型別匹配（如果將 T 替換爲 int）。至於第二個 template 的參數型別是個指標，不可能匹配。因此推導程序完成後惟一的候選函式是第一個 template 產生的具現體。在此情況下，重載決議機制的作用不大。

第二個呼叫式 f((int*)0) 比較有趣：引數推導機制成功作用於兩個 templates 身上，並產生 f<int*>(int*) 及 f<int>(int*) 兩個函式。從傳統的重載決議（overload resolution）角度看，兩個函式對於「以 int* 爲引數」的呼叫式匹配程度相同，這也許使你認爲這裡發生歧義（ambiguous）。然而在這種情況下，另一個重載決議準則參與了進來：「更特化」的 template 所產出的函式將被選用。在這裡，第二個 template 被認爲更特化（稍後討論），因此，這個例子的輸出結果仍然是：

```
1
2
```

12.2.3 Formal Ordering Rules（正式規則）

上個例子中你可以直觀看出第二個 template 比第一個更特化，因爲第一個 template 幾乎可以容納任意型別，而第二個 template 只接受指標型別。然而其他情形就未必這麼直觀。接下來將講述在一組重載函式中如何「判斷某個 function template 比另一個更特化」。然而亦請注意所謂的 **partial ordering rules**（偏序規則）：給定的兩個 templates 有可能誰也不比誰更特化。如果重載機制必須在這樣的兩個 templates 之間作出選擇，那它做不了任何決定：程式帶有歧義性錯誤。

假設比較兩個同名的 function templates $\texttt{ft}_1$ 和 $\texttt{ft}_2$，它們都適用於某個給定的呼叫式。被「預設引數」和「省略號參數（ellipsis parameters）」涵蓋的那些參數不在 **ordering rules** 考慮之內。接下來我們合成兩組引數型別（針對轉型運算子則還包括一個回返型別），並逐一以下面方式替換 template parameters：

1. 將每個 type template parameter 替換爲一個獨一無二的「編造型別」（"made up" type）。
2. 將每個 template template parameter 替換爲一個獨一無二的「編造出來的」class template。
3. 將每個 nontype template parameter 替換爲一個獨一無二且在相稱型別之下的「編造值」（"made up" value）。

如果「以第一組合成型別對第二個 template 進行引數推導」產生出一個完全匹配，反之不然，我們便說第一個 template 比第二個更特化。相反地，如果「以第二組合成型別對第一個 template 進行引數推導」產生出一個完全匹配，反之不然，我們便說第二個 template 比第一個更特化。當兩個推導都成功或都失敗時，兩個 templates 之間無順序可言（no ordering）。

讓我們更具體說明上面所講的內容，把它運用到前例的兩個 templates 身上。在那兩個 templates 中我們合成兩組引數型別，並以上述所談方式去替換 template parameters。兩組型別分別爲 `(A1)` 和 `(A2*)`，這裡的 `A1` 和 `A2` 都是獨一無二編造得來的型別。很明顯，將 `T` 替換爲 `A2*` 後，「以第二組引數型別替換第一個 template」的推導是成功的。但我們無法「將第二個 template 的 `T*` 匹配於第一列引數中的非指標型別 `A1`」。因此我們得出結論：第二個 template 比第一個更特化。

最後，考慮一個更複雜的例子，涉及多個函式參數：

```
template<typename T>
void t(T*, T const* = 0, ...);

template<typename T>
void t(T const*, T*, T* = 0);

void example(int* p)
{
    t(p, p);
}
```

首先，因爲實際呼叫式中並未用到第一個 template 的「省略號參數」，而第二個 template 的最後一個參數被其預設引數頂替，因此在 partial ordering 中這些參數都被忽略。注意，第一個 template 的預設引數未被使用，因此對應的參數會參與到 ordering 之中。

合成的引數型別爲 `(A1*,A1 const*)` 和 `(A2 const*,A2*)`。前者對第二個 template 的推導過程成功，以 `A1 const` 替換 `T`，但引發的匹配並不完全，因爲以型別 `(A1*,A1 const*)` 喚起 `t<A1 const>(A1 const*,A1 const*,A1 const*=0)` 時，需要一個 `const` 飾詞。類似情況，以第二組引數型別 `(A2 const*,A2*)` 推導第一個 template 的過程中，也無法找到完全匹配。因此兩個 templates 之間沒有順序關係（no ordering），這個呼叫帶有歧義性。

Formal ordering rules（正序規則）通常可以導致直觀的選擇。然而，曾經有一個例子顯示，這些規則偶爾也會與你的直觀選擇相悖。C++ 標準委員會將來很有可能修訂這些規則，使它們與人們的直覺更相符。

12.2.4 Templates 和 Nontemplates

Function templates 可以被 non-template function 重載。在其他因素相同的情形下，編譯器會優先選擇 non-template function。下面例子展示這一點：

```
// details/nontmpl.cpp
#include <string>
#include <iostream>

template<typename T>
std::string f(T)
{
    return "Template";
}
std::string f(int&)
{
    return "Nontemplate";
}
int main()
{
    int x = 7;
    std::cout << f(x) << std::endl;
}
```

程式輸出：

```
Nontemplate
```

12.3 明確特化（顯式特化；Explicit Specialization）

我們可以重載 function templates，並結合 partial ordering rules 以選擇最匹配的 function templates，這就使我們可以爲一個泛型實作品加入更多的特化 templates，以便透通實現程式碼的優化，得到更高效能。然而 class templates 無法被重載，它的訂製可採用另一種機制：明確（顯式）特化。標準術語「明確特化」也就是某些人所謂的「全特化」（full specialization）語言特性。它提供一種 template 實作方式：將所有 template parameters 替換掉，不留任何 template parameters。class templates 和 function templates 都可以被全特化。定義於 class 定義式之外的 class template 成員（例如成員函式、嵌套類別（nested classes）和 static 成員變數）也可以被全特化。

稍後我們將描述偏特化（partial specialization）。它與全特化類似，但與「替換所有 template parameters」不同的是，允許在 template 的某一份實作品中保留部份參數化特性。全特化和偏特化在程式碼中都是「明確指定」的，這就是爲什麼我們避免使用「明確特化」這個術語的原因。全特化和偏特化都不會引入全新的 template 或 template 具現體。這些構件爲「泛化（非特化）template 之中已被隱寓宣告」的實體提供另一種定義。這是一個相當重要的概念，也是與 overloaded templates 的一個關鍵區別。

12.3.1 Class Template 全特化（Full Specialization）

「全特化」係藉由「三個符號所組成的序列」引入：`template, < 和 >`[54]。緊跟在 class 名稱宣告之後的是 template arguments。下面例子展示這一點：

```
//譯註：以下是泛化（非特化）
template<typename T>
class S {
public:
   void info() {
      std::cout << "generic (S<T>::info())" << std::endl;
   }
};
```

[54] 宣告 function template 全特化體時，也使用相同的序列。早期 C++ 語言設計並不包括這個前導語彙單元，但由於後來增加了 member templates 語言特性，所以需要額外的語法，俾能消除複雜的特化情形所可能引發的歧義（ambiguity）。

```
//譯註：以下全泛化
template<>
class S<void> {
public:
    void msg() {
        std::cout << "fully specialized (S<void>::msg())" << std::endl;
    }
};
```

注意，全特化的實作不必與泛化定義有任何關聯。這使我們得以擁有不同名稱的成員函式（info 和 msg）。兩者（全特化與泛化）之間只靠 class template 名稱相連。

全特化所指定的 template argument list 必須與 template parameter list 對應。如果你對一個 template type parameter 指定一個 nontype 值，是不合法的。然而，對於帶有 default template argument 的那些 template parameters，你可以不予理會：

```
template<typename T>
class Types {
public:
    typedef int I;
};

template<typename T, typename U = typename Types<T>::I>
class S;                                // (1)

template<>
class S<void> {                         // (2)
public:
    void f();
};

template<> class S<char, char>;         // (3)

template<> class S<char, 0>;            // ERROR：不能以 0 替換 U

int main()
{
    S<int>*       pi;       // OK：使用(1)，無需定義
    S<int>        e1;       // ERROR：使用(1)，但找不到其定義
    S<void>*      pv;       // OK：使用(2)
    S<void,int>   sv;       // OK：使用(2)，其定義可用
    S<void,char>  e2;       // ERROR：使用(1)，但找不到其定義
    S<char,char>  e3;       // ERROR：使用(3)，但目前找不到其定義
}

template<>
class S<char, char> {       // (3)的定義
};
```

這個例子展示的是，全特化 template 的宣告式不必須爲定義式。然而當一個全特化體被宣告後，對於給定之 template arguments 集合，泛化定義式就不再會被使用。因此如果沒有提供所需定義，程式會出錯。撰寫 class template 特化碼時，前置宣告（forward declaration）很有用，使你得以宣告彼此相依（mutually dependent）的型別。全特化宣告式與常規 class 宣告式完全相同（也就是說前者並不被視爲一個 template 宣告），惟一區別是宣告的語法，而且該宣告必須與先前的 template 宣告匹配。由於它不是一個 template 宣告，所以 class templates 全特化體可以使用常規的 out-of-class 成員定義語法（但也就是說你不能使用 `template<>` 前導詞）：

```
template<typename T>
class S;

template<> class S<char**> {
public:
    void print() const;
};

// 以下定義式的前端不能出現 template<>
void S<char**>::print() const
{
    std::cout << "pointer to pointer to char" << std::endl;
}
```

下面這個稍複雜一些的例子也許可以強化你的印象：

```
template<typename T>
class Outside {
public:
    template<typename U>
    class Inside {
    };
};

template<>
class Outside<void> {
    // 下面的 nested class 不必與其泛化版 template 的定義式有任何關連
    template<typename U>
    class Inside {
    private:
        static int count;
   };
};

// 以下定義式的前端不能出現 template<>
template<typename U>
int Outside<void>::Inside<U>::count = 1;
```

所謂全特化就是「某確鑿之泛化 template 具現體」，因此同一個程式中不能同時存在一個「template 明確具現體」和一個由前者產生的具現體。如果同時使用兩者，編譯器會抓出你的小辮子：

```
template <typename T>
class Invalid {
};

Invalid<double> x1;     // 引發 Invalid<double> 被具現化

template<>
class Invalid<double>;  // 錯誤：Invalid<double> 已被具現化
```

不幸的是，如果你在不同的編譯單元中這麼寫，編譯器很難找出問題。下面是一個非法的 C++ 例子，由兩個檔案組成。它可以順利通過很多編譯器和聯結器，但它不但非法而且危險：

```
// 編譯單元 1
template<typename T>
class Danger {
public:
    enum { max = 10 };
};

char buffer[Danger<void>::max]; // 使用泛化值

extern void clear(char const*);

int main()
{
    clear(buffer);
}

// 編譯單元 2
template<typename T>
class Danger;

template<>
class Danger<void> {
public:
    enum { max = 100 };
};

void clear(char const* buf)
{
    // array 邊界不匹配（不吻合）
    for (int k = 0; k < Danger<void>::max; ++k) {
        buf[k] = '\0';
    }
}
```

這個例子刻意維持短小，但它說明：你必須小心確保「特化體的宣告對泛化 template 的所有使用者都可見」。現實而言，這意味特化體的宣告通常應該在表頭檔中緊跟著其泛化 template 的宣告。但是當泛化實作源自一個外部源碼檔（因此你無法修改其對應表頭檔），上述說法就不太實際。不過你還是可以建立一個表頭檔，含入泛化 template，然後是其特化體的宣告，這就可以避免這些難以發現的錯誤。我們發現，最好避免特化「外來源碼所含的 templates」，除非對方明確標示其設計目標就是如此。

12.3.2 Function Template 全特化（Full Specialization）

function template（明確）全特化的語法和原則非常類似 class template 全特化，只是你需要附加考慮重載（overloading）和引數推導（argument deduction）。

當被特化之 template 可經由引數推導（亦即將宣告式中的「參數型別」視爲「引數型別」）決定時，全特化宣告式中可以省略不寫明確的 template arguments：

```
template<typename T>
int f(T)                    // (1)
{
    return 1;
}

template<typename T>
int f(T*)                   // (2)
{
    return 2;
}

template<> int f(int)       // OK：(1)的特化
{
    return 3;
}

template<> int f(int*)      // OK：(2)的特化
{
    return 4;
}
```

function template 的全特化體不能含有預設引數值。然而明確特化時，仍然適用 template 的預設引數：

```
template<typename T>
int f(T, T x = 42)
{
    return x;
}

template<> int f(int, int = 35)  // 錯誤！全特化體不能含有預設引數值。
{
    return 0;
}

template<typename T>
int g(T, T x = 42)
{
    return x;
}
```

```
template<> int g(int, int y)
{
    return y/2;
}

int main()
{
    std::cout << g(0) << std::endl;  // 輸出 21
}
```

全特化宣告在很多方面類似於一個普通宣告（或說「再宣告」，redeclaration）。更明確地說，它並不是宣告一個 template，因此程式中的 noninline function template 全特化定義只能出現一份。但我們仍然必須確保全特化宣告式緊跟在 template 之後，以求防範使用該 template 產生的函式。前例中的 template g 因此往往宣告於兩個檔案中。介面檔（表頭檔）可能看來像這樣：

```
#ifndef TEMPLATE_G_HPP
#define TEMPLATE_G_HPP

// template 定義式應該寫在表頭檔中
template<typename T>
int g(T, T x = 42)
{
    return x;
}

// 特化體的宣告將會抑制 template 的具現化
// 不應該將定義寫在這裡，用以避免重複定義
template<> int g(int, int y);

#endif //TEMPLATE_G_HPP
```

對應的實作檔是這樣：

```
#include "template_g.hpp"

template<> int g(int, int y)
{
    return y/2;
}
```

另外，特化體可以宣告爲 inline。在這種情況下，其定義可以（並且應該被）寫在表頭檔中。

12.3.3 Member 全特化（Full Specialization）

不僅是 member templates，連 class templates 內的 static 成員變數/成員函式也可以被全特化。語法規定你必須在每個圈封的（enclosing）class template 前加上 `template<>` 前導詞。如果你特化一個 member template，也必須使用 `template<>` 指出它是被特化的。爲了更佳說明這些規定，假設有以下定義：

```
template<typename T>
class Outer {                              // (1)
public:
    template<typename U>                   // member template
    class Inner {                          // (2)
    private:
        static int count;                  // (3)
    };
    static int code;                       // (4)
    void print() const {                   // (5)
        std::cout << "generic";
    }
};

template<typename T>
int Outer<T>::code = 6;                    // (6)

template<typename T>
template<typename U>
int Outer<T>::Inner<U>::count = 7;         // (7)

template<>
class Outer<bool> {                        // (8)
public:
    template<typename U>
    class Inner {                          // (9)
    private:
        static int count;                  // (10)
    };
    void print() const {                   // (11)
    }
};
```

泛型 template Outer（位於(1)）的常規成員 code（位於(4)）和 print()（位於(5)）只有一個圈封的（enclosing）class template，因此需要以 template<>爲前導，針對一組特定的 template arguments 進行全特化：

```
template<>
int Outer<void>::code = 12;          //譯註：這是一個定義

template<>
void Outer<void>::print() const      //譯註：這是一個定義
{
    std::cout << "Outer<void>";
}
```

這些定義被用於(4)和(5)泛化版本中特定的 Outer<void>。Outer<void> 的其他成員仍將由泛化版本（位於(1)）產生。請注意，提供這些定義之後，你就不能再提供 Outer<void> 的一個明確特化體（explicit instantiation）了。

和 function templates 全特化一樣，我們需要一種方式用以宣告 class template 的常規成員的特化體，但不指定任何定義（以防止重複定義）。雖然 C++ 不允許常規 class 的成員函式和 static 成員變數存在著「非定義之 out-of-class 宣告式」，但是當特化 class templates 成員時它們是合法的。前面的定義可以被宣告爲：

```
template<>                    //譯註：這是一個非定義宣告式（nondefining declaration）
int Outer<void>::code;

template<>                    //譯註：這是一個非定義宣告式（nondefining declaration）
void Outer<void>::print() const;
```

用心的讀者可能會指出，Outer<void>::code 全特化體的「非定義宣告式」（nondefining declaration），和一個「以 *default* 建構式進行初始化」的定義式，語法相同。的確如此，但這類宣告總是會被編譯器理解爲「非定義宣告式」。

因此，我們無法提供一個「只能以 *default* 建構式初始化」的型別，又提供一個 static 成員變數的全特化定義。

```
//譯註：下面是個「只能以 default 建構式初始化」的型別
class DefaultInitOnly {
public:
    DefaultInitOnly() {      //這是個 default ctor
    }
private:
    DefaultInitOnly(DefaultInitOnly const&);
            // 不允許 copying（因爲刻意把 copy ctor 宣告於 private 區）
};
```

```
template<typename T>
class Statics {
private:
    static T sm;
};

// 下面是個宣告；不存在任何 C++語法可爲它提供一份定義
template<>
DefaultInitOnly Statics<DefaultInitOnly>::sm;
```

member template `Outer<T>::Inner` 也可以經由一個給定的 template argument 進行特化，不會影響某特定之 `Outer<T>` 具現體（我們正是在其中特化 member template `Inner`）中的其他成員。由於此處只有一層圈封的 template，我們只需要一個 `template<>` 前導詞。程式碼可被寫爲：

```
template<>
   template<typename X>
   class Outer<wchar_t>::Inner {
   public:
       static long count;    // 成員的型別改變了（譯註：原爲 int，現改爲 long）
   };

template<>
   template<typename X>
   long Outer<wchar_t>::Inner<X>::count;
```

template `Outer<T>::Inner` 也可以被全特化，但只能在某個給定的 `Outer<T>` 實體中對它全特化。我們現在需要兩個 `template<>`前導詞：一個是針對外層 class，另一個是針對內層 template：

```
template<>
    template<>
    class Outer<char>::Inner<wchar_t> {
    public:
        enum { count = 1 };
    };

// 下面程式碼不合法：template<>不能跟在一個 template parameter list 之後
template<typename X>
template<> class Outer<X>::Inner<void>;  // ERROR！
```

你可以把這段程式碼和 Outer<bool>的 member template 特化體進行對照。由於後者已經被完全特化，沒有外層圈封之 template，因此只需要一個 template<> 前導詞：

```
template<>
class Outer<bool>::Inner<wchar_t> {
public:
    enum { count = 2 };
};
```

12.4 Class Template 偏特化（Partial Specialization）

template 全特化非常有用，但如果我們想針對「一整個族系的 template arguments」而不是「特定某一組 template arguments」來特化 class template，也是很自然的想法。例如，假設我們有個 class template，實作出一個 linked list：

```
template<typename T>
class List {       // (1)
public:
   ...
   void append(T const&);
   inline size_t length() const;
   ...
};
```

在使用上述 template 的大型專案中，可能會將其成員具現化爲多種型別。對那些不是 inline 的成員函式（例如 List<T>::append()）而言，這將導致 obj 碼的不小膨脹。然而從底層角度看，List<int*>::append()的 obj 碼和 List<void*>::append()的 obj 碼是一樣的。換句話說我們希望所有「以指標爲元素」的 Lists 共用一份實作碼。儘管無法以 C++ 表達這樣的需求，但我們可以指定所有「以指標爲元素」的 Lists 都以另外一個 template 定義式進行具現化：

```
template<typename T>
class List<T*> {    // (2)
private:
    List<void*> impl;
    ...
public:
    ...
    void append(T* p) {
        impl.append(p);
    }
```

```
    size_t length() const {
       return impl.length();
    }
    ...
};
```

在本例情境中，(1)處的原始 template 稱爲 primary template，後一個定義稱爲 partial specialization（偏特化體；部份特化體），因爲你只指定了這個 template 的一部份 template arguments。偏特化的語法是：template parameter list 的宣告（template<...>）加上 class template 名稱，再加上一組明確指定的 template arguments（本例爲<T*>）。

上述程式碼有個問題：List<void*> 之中包含另一個 List<void*> 成員，構成遞迴宣告（譯註：當 T 是 void 時）。我們可以在偏特化之後加以一個全特化，終結這個遞迴圈：

```
template<>
class List<void*> {   // (3)
    ...
    void append (void* p);
    inline size_t length() const;
    ...
};
```

這種方法之所以可行，乃是因爲全特化會被優先選用，然後才是偏特化。於是所有「以指標爲元素」的 Lists 的成員函式都經由 inline 函式轉呼叫 List<void*> 實作碼。這是一個有效對抗所謂「程式碼膨脹」的辦法。

偏特化宣告式的 parameter list（參數列）和 argument list（引數列）存在諸多限制。其中一些如下：

1. 偏特化 template 的引數必須與其 primary template 的參數種類（type, nontype 或 template）逐一匹配。
2. 偏特化 template 的 parameter list（參數列）不能擁有預設引數；它將使用 primary class template 的預設引數。
3. 偏特化 template 的 nontype arguments（非型別引數）只能是非受控數值（non-dependent value）或簡樸的（plain）nontype template parameters，不能是較復雜的受控運算式（dependent expressions）如 2*N（N 是某個 template parameter）。
4. 偏特化 template 的 template argument list 不能與 primary template 的 template argument list 完全相同（無論 template parameter 是否以其他名稱出現）。

下面的例子展示上述限制：

```
template<typename T, int I = 3>
class S;                    // primary template

template<typename T>
class S<int, T>;            // 錯誤：參數種類不匹配

template<typename T = int>
class S<T, 10>;             // 錯誤：無預設引數

template<int I>
class S<int, I*2>;          // 錯誤：不能是 nontype 運算式

template<typename U, int K>
class S<U, K>;              // 錯誤：與 primary template 無顯著區別
```

所有的偏特化體和全特化體一樣，與其 primary template 相關聯。當你使用一個 template 時，編譯器總會首先查詢 primary template，但隨後各個引數會與其關聯的各個特化體進行匹配，決定到底選用哪一份 template 實作碼。如果有多個匹配結果，最特化（從 function templates 的重載決議機制看來）那個會勝出。如果無法比較哪一份最特化，程式便被視爲模稜兩可（帶有歧義）。

最後我們要指出，class template 的偏特化完全有可能比其 primary template 擁有更多或更少的參數。再次考慮我們的泛化 template `List`（宣告於(1)）。我們已經討論過如何對「以 pointer 爲元素」的 `Lists` 進行優化，但我們也可能想對 pointer-to-member 型別做同樣的事情。下面程式碼針對 pointer-to-member-pointers 達到同樣的優化目的：

```
template<typename C>
class List<void* C::*> {  // (4)
public:
    // 對任何 pointer-to-void* member 的偏特化
    // 任何其他 pointer-to-member-pointer 型別將使用這一份偏特化定義
    typedef void* C::*ElementType;
    ...
    void append(ElementType pm);
    inline size_t length() const;
    ...
};
```

```
template<typename T, typename C>
class List<T* C::*> {     // (5)
private:
    List<void* C::*> impl;
    ...
public:
    // 針對「先前處理過之 pointer-to-void* member 型別」以外
    //    的任何 pointer-to-member-pointer 型別進行特化
    // 這個特化體有兩個 template parameters，而 primary template 只有一個 template parameter
    typedef T* C::*ElementType;
    ...
    void append(ElementType pm) {
        impl.append((void* C::*)pm);
    }
    inline size_t length() const {
        return impl.length();
    }
    ...
};
```

除了關注 template parameters 的個數，我們還要注意，定義於(4)的通用實作版本，是其他版本（如定義於(5)）的轉呼叫（forwarding）目標，本身就是個偏特化（對簡單指標而言則是個全特化）。然而很明顯(4)要比(5)更特化，因此程式並沒有歧義性。

12.5 後記

一開始，template 全特化就是 C++ template 機制的一部份。function template 的重載和 class template 的偏特化則是晚近才加入 C++。HP 的 aC++ 編譯器最早實作出 function template 重載機制，EDG 的 C++ 前端系統最早實作出 class template 偏特化機制。本章描述的 partial ordering rules（偏序原則）源自 Steve Adamczyk 與 John Spicer 的發明（他們都在 EDG 工作）。

人們知道「如何終結一個無限遞迴之 template 定義」（例如之前出現之 List<T*>，見 12.4 節, p.200）已有一段不短的時間。然而 Erwin Unruh 可能是意識到「這可以引領出一種有趣的所謂 template metaprogramming 編程技法」的第一人；這種技法利用 template 具現化機制，在編譯期進行甚具意義的計算。第 17 章專門講述這個主題。

你完全有理由感到奇怪：爲什麼只能偏特化 class templates？這基本上是歷史原因。技術上其實完全有可能爲 function template 定義相同機制（請參考第 13 章）。某種程度上 function templates 的偏特化與 class templates 的偏特化類似，但兩者之間仍有精微的差異。這些差異大多與以下事實有關：當一個 template 被使用時，編譯器只需查詢 primary template；各個特化體是在後來才被考慮，用以決定使用哪一份實作碼。對比的是所有重載的 function templates 都會被帶入一個集合之中，供編譯器查詢；而這些 function templates 可能來自不同的 namespaces 或 classes。這在某種程度上增加了「無意中重載一個 template 名稱」的可能性。

相反地，我們也可以想像出一種重載 class templates 的形式。下面是個例子：

```
// 非法重載 class templates
template<typename T1, typename T2> class Pair;
template<int N1, int N2> class Pair;
```

然而人們對這種機制似乎並沒有迫切的需求。

13

未來發展方向

Future Directions

從 1988 年的最初設計，到 1998 年的標準化（其中技術工作於 1997 年 11 月完成），C++ templates 有長足的發展。此後數年 C++ 的語言定義趨於穩定。但是在那期間，人們對於 C++ templates 又有了各種新需求。其中有些是對語言的一致性（consistency）和正交性（orthogonality）的更高期望。例如，既然允許 class templates 有預設引數，爲什麼不允許 function templates 也有預設引數？其他的期待主要來自日漸複雜的 template 編程手法的要求，這些手法往往要求既有的編譯器擴展自身能力。

後續我們將描述 C++ 語言及編譯器設計者多次暢議的 C++ 語言擴展機能。這些擴展常常源自各種高級 C++ 程式庫（包括 C++ 標準程式庫）設計者的促使與提示。我們並不保證這裡所列的任何一條將成爲 C++ *Standard* 的一部份，不過我們知道已有某些 C++ 編譯器實作品包含了其中一些擴展機能。

譯註：本節討論的各個可能的 C++ 擴展機能，只有 13.5 節「寬鬆的 template template parameter 匹配」已在 g++ 3.2 中實現，其他任何擴展未見於 VC6/VC7.1/ICL7.1/g++ 3.2。

13.1 Angle Bracket Hack（角括號對付法）

一個最使 template 編程初學者感到驚訝的地方是：連續兩個右角括號之間必須加入空格。例如：

```
#include <list>
#include <vector>

typedef std::vector<std::list<int> > LineTable;   // OK

typedef std::vector<std::list<int>> OtherTable;  // 語法錯誤
```

第二個 typedef 是錯誤的，因爲兩個連續、中間無空格的右角括號會形成一個「右移」運算子(>>)，而那對本段程式碼並無意義。

然而「檢測出這個錯誤，並悄悄地將 >> 運算子當成兩個右角括號」（這個特性有時被稱爲 **angle bracket hack**）比起 C++ 源碼詞法解析器（parser）的其他功能來說算是相對簡單的。事實上已有很多編譯器可以辨識這種情形並正確分析程式碼（帶著一個警告訊息）。

因此，未來版本的 C++ 語言很可能認爲前例中的 `OtherTable` 宣告式合法。然而我們應該注意 angle bracket hack 中隱藏著一些隱微問題。確實存在這樣的情形：>> 運算子在 template argument list 中是合法記號。以下展示這一點：

```
template<int N> class Buf;

template<typename T> void strange() {}
template<int N> void strange() {}

int main()
{
    strange<Buf<16>>2> >();  // >>記號並不是錯誤
}
```

另一個某種程度上有所關連的問題是雙字符記號 `<:` 的偶爾誤用。這個記號等價於中括號（方括號）'['（見 9.3.1 節, p.129）。考慮下面程式碼片段：

```
template<typename T> class List;
class Marker;

List<::Marker>* markers; // 錯誤
```

例中最後一行會被當作 `List[:Marker>* markers;`，這將不具任何意義。編譯器完全有理由注意到一個像 `List` 這樣的 template 不可能後面跟一個 '['，這種情況下不應該把 `<:` 當作中括號（方括號）對待。

13.2 寬鬆的 typename 使用規則

某些程式員和語言設計者認爲關鍵字 `typename` 的使用規則（見 5.1 節, p.43 和 9.3.2 節, p.130）過於嚴苛。例如下面程式碼中，`typename Array<T>::ElementT` 裡面的 `typename` 必不可缺，但是卻嚴禁在 `typename Array<int>::ElementT` 裡頭使用 `typename`（否則就報錯）：

```
template <typename T>
class Array {
public:
   typedef T ElementT;
   ...
};

template <typename T>
void clear (typename Array<T>::ElementT& p);      // OK

template<>
void clear (typename Array<int>::ElementT& p);    // ERROR
```

這種例子令人驚訝。對 C++ 編譯器實作品而言，忽略這個額外的 typename 關鍵字毫無困難，因此，語言設計者正考慮允許在「未被 struct/class/union/enum 等關鍵字標注之受飾型別名稱（qualified typename）」前使用關鍵字 typename。如果對此做出定論，也將有益於澄清「何時允許使用.template, ->template 和::template 構件」的問題（見 9.3.3 節, p.132）。

從編譯器實作者的角度來看，忽略這些額外的 typename 和 template 相對簡單。有趣的是也存在這種情形：語言要求某處必須使用這些關鍵字，然而如果程式員沒有照做，某些編譯器實作版本也順利放行。例如前面的 function template clear()中，編譯器知道 Array<T>::ElementT 不能是別的什麼，只能是個型別名稱（此處不允許任何運算式），這種情形下關鍵字 typename 可有可無。C++ 標準委員會也在檢討是否能夠減少「必須使用關鍵字 typename 和 template」的情形。（譯註：VC6 在這方面相對寬鬆一些，VC7.1 對關鍵字 typename 的要求幾近苛刻。）

13.3 Function Template 的預設引數

templates 最初加入 C++ 語言時，明確指定 function template arguments 就不被語言所容許。Function template arguments 總是必須由呼叫式推導得出。這就似乎沒有什麼理由允許 function template 有預設引數，因爲預設值總是會被推導值覆寫（overridden）。

既然如此，我們總該可以爲「無法被推導得出」之引數明確指定 function template arguments 吧。因此，爲那些「無法被推導得出」的 template arguments 指定預設值，也就成了很自然的想法。考慮下面例子：

```
template <typename T1, typename T2 = int>
T2 count (T1 const& x);
```

```
class MyInt {
    ...
};

void test (Container const& c)
{
    int i = count(c);
    MyInt j = count<MyInt>(c);
    assert(j == i);
}
```

此例之中我們遵循一條約束條件：如果某個 template parameter 有預設引數值，則它後面的所有 template parameters 也必須擁有預設引數值。這個限制對 class templates 而言是必需的，否則就無法在一般情況下指定吊尾（trailing）的引數。下面的錯誤程式碼展示了這一點：

```
template <typename T1 = int, typename T2>
class Bad;

Bad<int>* b;  // 這個 int 是用來替換 T1 還是 T2？
```

然而對 function templates 來說，吊尾引數可由推導機制得出。因此不難改寫本例如下：

```
template <typename T1 = int, typename T2>
T1 count (T2 const& x);

void test (Container const& c)
{
    int i = count(c);
    MyInt j = count<MyInt>(c);
    assert(j == i);
}
```

本書撰寫之際，C++ 標準委員會正考慮在這方面對 function templates 進行擴展。

有些程式員事後注意到，不使用明確指定的（explicit）template arguments 也可以達到相同目的。

```
template <typename T = double>
void f(T const& = T());
```

```
int main()
{
    f(1);               // OK：推導出 T 爲 int
    f<long>(2);         // OK：T 爲 long，不需推導
    f<char>();          // OK：與 f<char>('\0')相同
    f();                // 與 f<double>(0.0)相同
}
```

這裡透過 default template argument 也實現了 default call argument，然而不需明確指定（explicit）template arguments。

13.4 以字串字面常數（String Literal）和浮點數（Floating-Point）作爲 Template Arguments

在 nontype template arguments 的限制中，最可能令 template 初學者和高級使用者感到驚訝的是，他們無法使用字串字面常數作爲 template argument。

下面是個看起來十分直觀可行的例子：

```
template <char const* msg>
class Diagnoser {
public:
    void print();
};

int main()
{
    Diagnoser<"Surprise!">().print();
}
```

然而其中有些潛在問題。在標準 C++ 中， `Diagnoser` 的兩個具現體只有當它們擁有相同的 template arguments 時，其型別才相同。本例的 template arguments 是個指標，或說是個位址。然而位於不同源碼檔案中的兩個內容相同的字串字面常數並不一定落在同一位址上。我們可能會因此尷尬地發現，`Diagnoser<"X">` 與 `Diagnoser<"X">` 是兩個不同而且不相容的型別！（注意 "X" 的型別是 `char const[2]`，當它被用作 template argument 時會退化爲 `char const*`）

基於這些考慮（及其相關考慮），*C++ Standard* 禁止使用字串字面常數作爲 template arguments。然而某些編譯器會以擴展形式提供這項機能 — 他們使用字串字面值作爲 template 具現體的內部表述（internal representation）。雖然這顯然是可行的，但有些 C++ 語言評論者認爲，「可被替換爲字串字面常數」之 nontype template parameter 的宣告方式，應該與「可被替換爲位址」之 nontype template parameter 的宣告方式有所不同。本書撰寫之際，並沒有哪一種宣告語法獲得壓倒性的支持。

我們應該注意到這個問題中的一個小小的技術缺陷。考慮下面的 template 宣告式，並假設語言已被擴展爲「允許以字串字面常數作爲 template arguments」：

```
template <char const* str>
class Bracket {
public:
    static char const* address();
    static char const* bytes();
};

template <char const* str>
char const* Bracket<str>::address()
{
    return str;
}

template <char const* str>
char const* Bracket<str>::bytes()
{
    return str;
}
```

上述程式碼中，兩個成員函式除了名稱不同，其他完全相同（這種情形並不少見）。假設編譯器實作品使用類似巨集（macros）展開的方式來具現化 `Bracket<"X">`，那麼如果兩個成員函式被具現化於不同的編譯單元，它們會傳回不同的值。有趣的是，我們測試了一些支援本節擴展的 C++ 編譯器，發現它們也受到這種令人驚訝的行爲的折磨。

一個與之相關的問題是使用浮點字面常數（以及簡單的浮點常數運算式）作爲 template arguments。例如：

```
template <double Ratio>
class Converter {
public:
    static double convert (double val) {
        return val*Ratio;
    }
};

typedef Converter<0.0254> InchToMeter;
```

也有一些 C++ 編譯器實作品提供了這個特性，它並不帶來任何技術上的困難，不像以字串字面常數作爲 template arguments 那樣困難。

13.5 Template Template Parameters 的寬鬆匹配規則

一個 template 如果被用來「替換某個 template template parameter」，兩者的 parameter list 必須匹配。有時這會引發令人驚訝的結果，例如以下面例子所展示：

```
#include <list>
    // 宣告：
    // namespace std {
    //     template <typename T,
    //               typename Allocator = allocator<T> >
    //     class list;
    // }

template<typename T1,
         typename T2,
         template<typename> class Container>
                                  // Container 期望的是僅帶單一參數的 template
class Relation {
public:
    ...
private:
    Container<T1> dom1;
    Container<T2> dom2;
};

int main()
{
    Relation<int, double, std::list> rel;
        // 錯誤：std::list 擁有不止一個 template parameter
    ...
}
```

這個程式並不合法，因爲我們的 template template parameter `Container` 要求的是「僅有一個參數」的 template，然而 `std::list` 除了有一個標示元素型別的參數外，還有個 `allocator` 參數。

然而由於 `std::list` 的 `allocator` 參數有一個預設的 template argument，因此如果說 `Container` 被認爲與 `std::list` 匹配，而且 `Container` 的每一個具現體都將使用 `std::list` 的預設 template argument（見 8.3.4 節, p.112），也是可能的。

贊成「維持現狀」（不企圖匹配）的一個論點是：function 型別的匹配也如此。然而在這個情況中，預設引數並非總是能夠被推導而出，因爲一個 function 指標值必須直到運行期才能確定下來。與之對比的是，並不存在所謂 template 指標，而且推導所需的所有資訊在編譯期都可取得。

某些編譯器以擴展形式提供了寬鬆匹配原則（relaxed matching rule）。這個問題和 typedef templates 有關（下一節討論）。考慮以下面的程式碼替換前例的 main() 定義：

```
template <typename T>
typedef std::list<T> MyList;

int main()
{
    Relation<int, double, MyList> rel;
}
```

typedef template 引入一個新的 template，它與 Container 的 template pamameter list 完全匹配。這種情況究竟是強化還是弱化寬鬆匹配規則，尚有爭議。

這個議題已向 C++ 標準委員會提出，但委員會並不傾向於把寬鬆匹配規則加入 C++ 語言。

13.6 Typedef Templates

Class templates 常以十分精巧的方式組合在一起，以求獲得其他的參數化型別。當這種參數化型別在源碼中反復出現時，我們很自然會想要以一種簡短形式（shortcut）來表示它，就像 typedef 爲一般非參數化型別提供簡短形式一樣。

所以，C++ 語言設計者正在考慮這樣一種構件：

```
template <typename T>
typedef std::vector<std::list<T> > Table;
```

有了這個宣告後，Table 將成爲一個新的 template，可被具現化成爲一個具體型別定義。這種 template 被稱爲 typedef template（相對於 class template 或 function template）。例如：

```
Table<int> t;       // t 的型別爲 vector<list<int> >
```

目前我們可以運用 class templates 的 member typedefs 來暫時取代 typedef templates。對於前面例子我們可以這麼寫：

```
template <typename T>
class Table {
public:
    typedef std::vector<std::list<T> > Type;
};

Table<int>::Type t;  // t 將是一個 vector<list<int> > 型別
```

由於 typedef templates 也是一種具有完全特性的 templates，因此也可以像對待 class templates 一樣地加以特化：

```
// primary typedef template
template<typename T> typedef T Opaque;

// 偏特化
template<typename T> typedef void* Opaque<T*>;

// 全特化
template<> typedef bool Opaque<void>;
```

typedef templates 的肌理並不是那麼直觀。例如人們尚未清楚在推導過程中它將如何作用：

```
void candidate(long);

template<typename T> typedef T DT;

template<typename T> void candidate(DT<T>);

int main()
{
    candidate(42);  // 應該呼叫哪個 candidate()？
}
```

目前還不清楚這種情況下是否應該讓推導成功。可以肯定的是，推導機制不能對任意的 typedef patterns 進行推導。

13.7 Function Templates 偏特化（partial specialization）

第 12 章我們討論過，class templates 可以被偏特化，而 function templates 只能被重載。這兩種機制在某種程度上是不同的。

偏特化並不引入一個全新的 template：它只是既有之 template（所謂 primary template）的擴展。當編譯器查詢一個 class template 時，首先只考慮 primary templates。如果編譯器選中了某個 primary template 之後發現它有一個偏特化體，其 argument list 與具現體的 argument list 匹配，編譯器會具現化這個偏特化體（的定義），而不具現化 primary template。Template 全特化工作方式與此完全相同）。

與之相反的是，重載化的 function templates 是彼此完全獨立的 templates。當編譯器準備選擇某個 template 進行具現化時，所有重載化的 function templates 都被考慮，並由重載決議機制選擇一個最適任者。雖然這似乎也是個不錯的第二選擇，但實際上存在不少限制：

- 你可以特化一個 class 的 member templates，而無需修改這個 class 的定義。然而如果你增加一個重載成員，就必須改動 class 的定義。許多情況下我們之所以無法選用這個方法，因爲我們可能無權修改 class。而且目前的 *C++ Standard* 並不允許我們往 `std` namespace 添加新的 templates，但卻允許我們特化 `std` namespace 中的 templates。
- 若要重載 function templates，各重載函式的參數就必須有實質上的差異。考慮一個 function template `R convert(T const &)`，這裡 `R` 和 `T` 是 template parameters。我們可能非常想針對 `R` 爲 `void` 的情況對它進行特化，然而這無法透過重載來實現（譯註：因爲 `R` 不在參數列中）。
- 程式碼即使對非重載函式合法，有可能在函式被重載後不再合法。更具體地說，給定兩個 function templates `f(T)` 和 `g(T)`（這裡 `T` 是 template parameter），只有當 `f` 未被重載時，運算式 `g(&f<int>)` 才是合法的（否則無法判斷 `f` 的意義）。
- friend 宣告式可引用一個特定的 function template 或一個特定 function template 的具現體。然而 function template 的重載版本不具有 primary template 所可能擁有的特權（privileges）。

把這些加在一起，就構成了「加入 function templates 偏特化支援」的強有力論據。

function templates 偏特化語法是 class template 偏特化語法的泛化：

```
template <typename T>
T const& max (T const&, T const&);              // primary template

template <typename T>
T* const& max <T*>(T* const&, T* const&);       // 偏特化
```

有些語言設計者擔心這種偏特化方式將干擾 function template 的重載機制，例如：

```
template <typename T>
void add (T& x, int i);     // 一個 primary template

template <typename T1, typename T2>
void add (T1 a, T2 b);      // 另一個(重載的)primary template

template <typename T>
void add<T*> (T*&, int);    // 這一行打算特化哪一個 primary template？
```

然而我們希望這種情形被視爲一種錯誤，這才不致於對這個語言特性的效用有太大的影響。

本書撰寫之際，C++ 標準委員會還在考量這一項擴展提議。

13.8 typeof 運算子

撰寫 templates 程式碼時，如果能夠表達「與 template 相依之運算式」的型別，將非常有用。或許最具代表性的例子就是：一個針對數值型 array template 而設計的算術運算子，且其運算元（都是 array 的元素）的型別是混雜的。下面的例子展示這一點：

```
template <typename T1, typename T2>
Array<???> operator+ (Array<T1> const& x, Array<T2> const& y);
```

如你所料，這個運算子產生一個 array，由 array x 和 array y 對應元素之和構成。因此運算成果的型別應該是 x[0]+y[0] 的型別。不幸的是 C++ 並沒有提供什麼方法讓我們以 T1 或 T2 來表達這個成果型別。

某些編譯器以擴展形式提供了一個 typeof 運算子來解決這個問題。它讓人聯想到 sizeof 運算子：它們都接受一個運算式，並從該運算式中產生一個編譯期物體（compile-time entity）。只不過在這裡，和 sizeof 不同的是，這個編譯期物體起到一個「型別名稱」作用。我們可以利用這項擴展機能改寫前面的例子爲：

```
template <typename T1, typename T2>
Array<typeof(T1()+T2())> operator+ (Array<T1> const& x,
                                    Array<T2> const& y);
```

這樣很好，但還不夠理想，因爲這會要求那些給定型別必須可以被其預設值初始化。我們可以用一個輔助的 template 解決這個問題：

```
template <typename T>
T makeT();  // 不需定義

template <typename T1, typename T2>
Array<typeof(makeT<T1>()+makeT<T2>())>
  operator+ (Array<T1> const& x,
             Array<T2> const& y);
```

我們其實更想以 x 和 y 作爲 typeof 的引數，但是不能這樣做，因爲在 typeof 構件處，x 和 y 尚未宣告。解決這個問題的一個激進作法是引入另一個函式宣告語法，把回返型別寫到參數型別之後：

```
// operator function template
template <typename T1, typename T2>
operator+ (Array<T1> const& x, Array<T2> const& y)
  -> Array<typeof(x[0]+y[0])>;
//譯註：以上是作者假設的一種新式宣告語法，實際並不存在。

// regular function template
template <typename T1, typename T2>
function exp(Array<T1> const& x, Array<T2> const& y)
  -> Array<typeof(exp(x[0], y[0]))>
//譯註：以上是作者假設的一種新式宣告語法，實際並不存在。
```

正如本例所示，對於非運算子（non-operator）函式來說，這個新語法需要引入一個新關鍵字（上述的 `function`）。對於運算子函式，關鍵字 `operator` 已經能夠給予詞法解析器（parser）足夠導引。

注意，`typeof` 必須是一個編譯期運算子。更明確地說，`typeof` 將不考慮協變回返型別（covariant return types），如下所示：

```
class Base {
public:
    virtual Base* clone();
};

class Derived : public Base {
public:
    virtual Derived* clone();  // 協變回返型別（covariant return types）
};

void demo (Base* p, Base* q)
{
   typeof(p->clone()) tmp = p->clone();   // tmp 的型別總是爲 Base*
   ...
}
```

請參考 15.2.4 節, p.271，那裡介紹的 promotion traits 可以使目前缺少 `typeof` 運算子的情況獲得部份抒解。

13.9 Named Template Arguments（具名模板引數）

16.1 節, p.285 描述一種技術，可以讓我們爲特定的 template parameter 提供非預設的（nondefault）template argument，而無需爲其他已有預設值的 template parameters 指定 template arguments。儘管這是一項很有趣的技術，但很明顯我們殺雞用了牛刀。因此我們很自然地希望 C++ 能夠提供一個機制讓我們對 template arguments 命名。

在這一點上，我們應該注意，早在 C++ 標準化過程中，Roland Hartinger 就提交過一個類似的擴展機能（請參考 [StroustrupDnE], 6.5.1 節）。雖然技術上沒什麼問題，但是基於數種原因，這個提議最終沒有被接納。目前看來沒什麼理由相信 named template arguments 會被納入 C++ 語言。

然而，出於完備性的考量，以下將簡述數種語法設計中的一種：

```
template<typename T,
         Move: typename M = defaultMove<T>,
         Copy: typename C = defaultCopy<T>,
         Swap: typename S = defaultSwap<T>,
         Init: typename I = defaultInit<T>,
         Kill: typename K = defaultKill<T> >
class Mutator {
   ...
};

void test(MatrixList ml)
{
   mySort (ml, Mutator <Matrix, Swap: matrixSwap>);
}
```

注意引數名稱（在一個冒號之前）是如何與參數名稱區分的。這使我們實作程式碼時可以針對參數使用短名，並因爲引數名稱的存在而使得引數的作用一目瞭然。由於這在某些編碼風格中可能顯得過於累贅，任何都可能想像，當引數名稱與參數名稱相同時，引數名稱可省略不寫：

```
template<typename T,
         : typename Move = defaultMove<T>,
         : typename Copy = defaultCopy<T>,
         : typename Swap = defaultSwap<T>,
         : typename Init = defaultInit<T>,
         : typename Kill = defaultKill<T> >
class Mutator {
   ...
};
```

13.10 靜態屬性（Static Properties）

第 15 章和第 19 章將討論數種在編譯期間歸類（categorize）型別的方法。如果有必要「根據型別的靜態屬性來選擇 templates 的特化方式」，那兩章提到的 type traits（型別特徵萃取機制）非常重要。例如你可以參考 15.3.2 節, p.279 的 `CSMtraits` class，它可以選擇「最優」或「接近最優」的策略（policies），對隸屬其引數型別的元素進行 *copy*、*swap* 或 *move* 操作。

有些語言設計者觀察到，如果這類「選擇特化方式」（specialization selections）的需求非常普遍，C++ 語言就不應該要求使用者提供各種精巧複雜而煞費苦心的「自定程式碼」，用以探尋編譯器內部已知的屬性。C++ 語言實在可以提供一些內建的 type traits。如果有了這項擴展，下面的 C++ 程式就完全合法：

```
#include <iostream>

int main()
{
    std::cout << std::type<int>::is_bit_copyable << std::endl;
    std::cout << std::type<int>::is_union << std::endl;
}
```

儘管我們可以爲這一類構件發展出個別而獨特的語法，但如果讓它配合一個「使用者可自定」的語法，便可獲得一種更平滑的過渡 — 從「當前語言」過渡到「包含這種特性的語言」。但是某些 C++ 編譯器可輕易提供的靜態屬性，卻無法透過 traits 傳統技術獲得（例如判斷一個型別是否爲 union）。這正是「讓這個特性成爲語言元素之一」的主要支撐論據。另一個論據是：如果擁有這種語言元素，當我們編譯「倚賴這些屬性」的程式時，可以顯著降低編譯器所耗用的記憶體及所消耗的機器時間。

13.11 訂製的具現化診斷訊息（Custom Instantiation Diagnostics）

很多 templates 對其參數都有隱式要求。當你以「無法滿足這些要求」的引數來進行 templates 具現化時，要嘛引發一個一般性錯誤，要嘛具現化之後的函式無法正常運作。早期 C++ 編譯器中，具現化 template 所引發的一般性錯誤往往極度晦澀（p.75 有個好例子）。晚近的編譯器中，這些錯誤訊息已經足夠清晰易讀，有一定經驗的程式員可以很快追查到問題所在。但是人們仍希望這種情形能夠再加改善。考慮下面這個刻意製造的例子（展示真實 template 程式庫中所發生的情況）：

```
template <typename T>
void clear (T const& p)
{
    *p = 0;  // 假設 T 是一個「類似指標」的型別
}
```

```
template <typename T>
void core (T const& p)
{
    clear(p);
}

template <typename T>
void middle (typename T::Index p)
{
    core(p);
}

template <typename T>
void shell (T const& env)
{
    typename T::Index i;
    middle<T>(i);
}

class Client {
public:
    typedef int Index;
    ...
};

Client main_client;

int main()
{
    shell(main_client);
}
```

這個例子展示了典型的軟體開發分層策略：高層 function templates 如 `shell()` 倚賴諸如 `middle()` 的組件，後者又使用 `core()` 之類的基礎設施。當我們具現化 `shell()` 時，其每一個下層都需要被具現化。本例的問題出在最底層 `core()`：它被具現化爲 `int` 型別（`middle()` 之中呼叫 `core()` 時的引數型別是 `Client::Index`），而它試圖對該型別取值（*dereference*），這是錯誤的。一個表現良好的通用診斷訊息將會回溯引發問題的各層，但如此大量的訊息通常難以被使用。

另一個經常被提倡的方法是：在最高層安插一個裝置，使得如果來自更底層的需求無法獲得滿足，就抑制底層的具現化。已有各種「運用現有 C++ 構件」而進行的嘗試（例如 [BCCL]）試圖達到這個目標，但它們並不總是已獲得實戰效能。也因此，那麼多人提議增加這項語言擴展，也就不讓人驚訝了。這項擴展顯然可以構築於先前討論的靜態屬性設施基礎上。例如我們可以想像把 `shell()` template 修改如下：

```
template <typename T>
void shell (T const& env)
{
    std::instantiation_error(
          !std::type<T>::has_member_type<"Index">,
          "T must have an Index member type");
    std::instantiation_error(
          !std::type<typename T::Index>::dereferencable,
          "T::Index must be a pointer-like type");
    typename T::Index i;
    middle(i);
}
```

僞函式（pseudo-function）`instantiation_error()`應該會導致編譯器中斷具現化過程（這就避免了由 `middle()`具現化觸發的診斷訊息），並使得編譯器報出給定的診斷訊息。

雖然這是可行的，但這個方法有一些缺陷。例如，描述型別的所有屬性既笨重又不必要。有些人提議使用所謂「啞碼」（dummy code）構件來當作中斷具現化的條件。下面是眾多提議中的一種（不引入新關鍵字）：

```
template <typename T>
void shell (T const& env)
{
    template try {
       typename T::Index p;
       *p = 0;
    } catch "T::Index must be a pointer-like type";
    typename T::Index i;
    middle(i);
}
```

這裡呈現的想法是，`template try` 子句將被暫時具現化，但不真正生成 obj 碼。如果此處發生錯誤，子句之後的診斷訊息會被報出。不幸的是這種機制難以實現，因爲即使可以免除 obj 碼的生成，也會給編譯器的內部資料帶來難以避免的負面作用。換句話說，這個相當小的特性很可能需要對現有編譯技術進行大規模重新規劃。

大多數此類方案都有各種其他限制。例如許多 C++ 編譯器可以使用各種語言（英文、德文、日文等等）報出診斷訊息，但要求在源碼中提供各語種的診斷訊息就有些過份了。不僅如此，如果具現化過程被中斷後，這些預先處理沒有在診斷訊息中被清晰而系統化地說明，程式員或許還不如面對那些一般的診斷訊息（雖然笨拙難用）。

13.12 經過重載的（Overloaded）Class Templates

完全可以想像，class templates 可透過其 template parameters 進行重載。例如我們可以想像這樣一種方式：

```
template <typename T1>
class Tuple {
   // singleton
   ...
};

template <typename T1, typename T2>
class Tuple {
   // pair
   ...
};

template <typename T1, typename T2, typename T3>
class Tuple {
   // three-element tuple
   ...
};
```

下一節將討論這種重載方式的一個應用。

重載方式不必侷限於 template parameters 個數（這樣的重載可使用偏特化來模擬，請參考第 22 章實作的 `FunctionPtr`）。參數種類也可以是多樣的：

```
template <typename T1, typename T2>
class Pair {
   // pair of fields
   ...
};

template <int I1, int I2>
class Pair {
   // pair of constant integer values
   ...
};
```

雖然某些語言設計者曾經非正式地討論過這個想法，但它還未被提交到 C++ 標準委員會。

13.13 List Parameters（- 式列參數）

有時人們需要將一整串型別（a list of types）當作單一 template argument 傳遞。通常這是出於下面兩種目的之一：(1) 宣告一個函式，其參數個數是參數化的；(2) 定義一個型別結構（type structure），其成員個數是參數化的。

舉個例子，我們也許想要定義一個 template，用來計算任意數值列中的最大值。一個可能的宣告語法是：使用省略符號（ellipsis）表示「最後一個 template parameter 可匹配任意數量的引數」：

```
#include <iostream>

template <typename T, ... list>      //譯註：...就是所謂的 list template parameter
T const& max (T const&, T const&, list const&);

int main()
{
    std::cout << ::max(1, 2, 3, 4) << std::endl;
}
```

這樣一個 template 可以有多種實作可能。以下辦法不需要引入新關鍵字，但它必須對 function template 重載機制加入一條新規則：優先選用「不含 list template parameter」的 function template：

```
template <typename T> inline
T const& max (T const& a, T const& b)
{
    // 這是慣用的二元 max()
    return a<b ? b : a;
}

template <typename T, ... list> inline
T const& max (T const& a, T const& b, list const& x)
{
   return ::max (a, ::max(b,x));
}
```

讓我們分析呼叫式 `::max(1,2,3,4)` 的各個執行步驟。由於這個呼叫式有四個引數，二元的 `max()` 函式與之不匹配，但是當 `T=int` 而 `list=int,int` 時，第二個 template 與之匹配。這使得我們喚起二元 function template `max()`，第一個引數爲 1，第二個引數是 `::max(2,3,4)` 的求值結果。面對後者，二元 `max()` 再次不匹配，於是喚起 list parameter 版的 `max()`，並使 `T=int` 而 `list=int`。這一次子運算式 `max(b,x)` 被展開爲 `max(3,4)`，於是選用二元 function template 並結束遞迴。

感謝 function templates 的重載能力，它運作得極好。當然實際情況比我們講的要複雜一些。例如我們必須精確指出 `list const&` 在當前情境（前後脈絡, context）下的意義。

有時人們希望引用參數列中的某個特別元素或某個子集。我們可以使用下標運算子（subscript operator; 方括號）滿足需要。下面例子展示如何建構一個 metaprogram（超程式），並使用這種技術統計 list 中的元素個數：

```
template <typename T>
class ListProps {
public:
    enum { length = 1 };
};

template <... list>
class ListProps {
public:
    enum { length = 1+ListProps<list[1 ...]>::length };
};
```

這個例子說明 list parameters 對於 class templates 也相當有用。它也可以結合前面討論的 class template 重載概念，使我們得以增強各種 template metaprogramming（模板超編程）技術。

list parameter 也可以用來宣告一串欄位（fields, 成員變數）：

```
template <... list>
class Collection {
    list;
};
```

在這種設施基礎上，我們可以發展出數量驚人的基礎工具（dundamental utilities）。有關這方面的更多構想，我們推薦你閱讀《*Modern C++ Design*》（請見 [AlexandrescuDesign]），書中告訴你大量的 templates-based 和 macros-based 超編程（metaprogramming），取代目前缺乏的 list parameters。

13.14 佈局控制（Layout Control）

template 編程中一個頗爲普遍的困難是宣告一個夠大（但不過份大）的 bytes array，用以容納一個「型別未知的 object」（或說某個 template parameter）。一個應用是所謂的 discriminated unions（有分辨能力的 unions；又稱爲 variant types（易變型別）或 tagged unions（帶標記的 unions））：

```
template <... list>
class D_Union {
public:
    enum { n_bytes };
    char bytes[n_bytes];  // 最終將容納 template arguments 所描述之各類型別中的一種
    ...
};
```

常數值 n_bytes 不能強定爲 sizeof(T)，因爲 T 可能比 bytes 有更嚴厲的齊位（alignment）條件。很多這方面的探討都考慮到了齊位方式，但它們通常過於複雜，或是存在某種程度上的武斷假設。

對於此類應用，我們實際需要的是「以常數運算式表述某個型別的齊位方式」的能力，以及反過來能夠「對某個型別或某個欄位（field）或某個變數進行齊位」的能力。很多 C 和 C++ 編譯器提供__alignof__ 運算子，它傳回某給定型別或運算式的齊位方式。這幾乎完全和 sizeof 運算子相同，只是它傳回的是齊位方式（alignment）而不是型別大小。很多編譯器也提供#pragma 指令或類似裝置來設定某個物體（entity）的齊位方式。一個可能的作法是引入關鍵字 alignof，它既可用於在宣告中設定齊位方式，也可用於在運算式中求取齊位方式。

```
template <typename T>
class Alignment {
public:
    enum { max = alignof(T) };
};

template <... list>
class Alignment {
public:
    enum { max = alignof(list[0]) > Alignment<list[1 ...]>::max
                 ? alignof(list[0])
                 : Alignment<list[1 ...]>::max }
};

// 也可運用同樣道理設計一組 Size templates，用以獲得一大串型別中的體積（尺碼）最大者。
```

```
template <... list>
class Variant {
public:
    char buffer[Size<list>::max] alignof(Alignment<list>::max);
    ...
};
```

13.15 初始式的推導（Initializer Deduction）

人們常說程式員懶惰，有時這句話指的是我們總想在程式中少寫些標記。考慮下面的宣告：

```
std::map<std::string, std::list<int> >* dict
  = new std::map<std::string, std::list<int> >;
```

這很囉嗦。現實之中我們往往（也應該）引入該型別的一個 typedef 同義詞。然而這個宣告似乎有些多餘：我們指定了型別 dict，而它應該隱寓成爲其初始式（initializer）中的型別。如果可以寫個等價宣告，其中只需指定一個型別符號，那將多麼優雅。例如：

```
dcl dict = new std::map<std::string, std::list<int> >;
```

上面這個宣告式中，變數型別是從其初始式推導出來的。這裡需要一個關鍵字（上例用的是 dcl，但也有其他提議如 var、let 甚至 auto）使它異於常規賦值操作（ordinary assignment）。

到目前爲止，這不是一個只與 template 相關的議題。事實上一個相當早期的 Cfront 版本就已經出現過這個構件（時值 1982 年，早在 templates 出現之前）。但正由於許多 template-based 型別過於冗長，人們才如此迫切需要這個特性。

人們也想像了局部推導（partial deduction）機制。此機制適用於「只有 template arguments 需要推導」的情況：

```
std::list<> index = create_index();
```

這個機制的另一個變體是由建構式引數推導出 template arguments。例如：

```
template <typename T>
class Complex {
public:
    Complex(T const& re, T const& im);
    ...
};

Complex<> z(1.0, 3.0);  // 推導出 T 爲 double
```

由於建構式乃至建構式模板（constructor templates）都可能存在重載，因此這種推導機制的精確規格描述變得更爲複雜。假設我們的 Complex template 包含一個 constructor template，以及一個常規的 *copy* 建構式：

```
template <typename T>
class Complex {
public:
    Complex(Complex<T> const&);

    template <typename T2> Complex(Complex<T2> const&);
    ...
};

Complex<double> j(0.0, 1.0);
Complex<> z = j;  // 究竟想用哪一個建構式？
```

在第二個初始式宣告中，使用者應該是想呼叫常規的 *copy* 建構式；因此 z 應該與 j 具有相同型別。然而如果制訂一個「忽略 constructor templates」的隱式規則，似乎又過於呆板了。

13.16 Function Expressions（函式運算式）

本書第 22 章說明一個主題：將小型函式（或仿函式, functors）作爲參數傳遞給其他函式，常常能夠帶來方便。第 18 章也提到 expression template 技術可用來搭建小型仿函式（functors），無需承受因明確宣告而可能帶來的額外開銷（見 18.3 節, p.340）。

例如我們可能需要針對標準 vector 的每一個元素，呼叫元素型別所提供的某個特定成員函式，以便進行各個元素的初始化：

```
class BigValue {
public:
    void init();
    ...
};

class Init {
public:
    void operator() (BigValue& v) const {
        v.init();
    }
};
```

```
void compute (std::vector<BigValue>& vec)
{
   std::for_each (vec.begin(), vec.end(),
                  Init());
   ...
}
```

爲了這個目的而定義出一個個別的 class `Init`，似乎有些笨拙。或許我們可以寫出一個（不具名）函式，作爲運算式的一部份：

```
class BigValue {
public:
   void init();
   ...
};

void compute (std::vector<BigValue>& vec)
{
   std::for_each (vec.begin(), vec.end(),
                  $(BigValue&) { $1.init(); } );
   ...
}
```

這裡的構想是引入一個所謂的 function expression（函式運算式），語法如下：一個特殊記號`$`，後跟著以小（圓）括號括起來的參數型別，然後是一個大括號括起來的函式本體。在這樣一個構件中，我們可以透過特殊記號`$n`引用某個函式參數，此處 `n` 是個常數，代表參數序號。

這種形式與其他語言所謂的 lambda expressions（或稱 *lambda functions*）和 *closures* 很有關（譯註：*lambda* 是希臘字母中的第 11 個字母）。其他解法也是可能的，例如我們可以使用不具名內層（annoymous inner）class，像 Java 那樣：

```
class BigValue {
public:
   void init();
   ...
};

void compute (std::vector<BigValue>& vec)
{
   std::for_each (vec.begin(), vec.end(),
                  class {
                    public:
                      void operator() (BigValue& v) const
                                       v.init();
                      }
                  };
                 );
   ...
}
```

雖然這些構件方式經常在語言設計者之間被討論，但很少有人提出具體的提案。這或許是因爲這種擴展的設計難度相當可觀，遠遠超出前面例子所涵蓋的範圍。待解決的問題包括回返型別的詳細規範、如何決定 function expressions（函式運算式）中哪些物體可用…等等。舉個例子，外層函式的 local 變數可以被取用嗎？function expressions 也應該可以是個 templates，因爲其參數型別可由用戶呼叫式中推導而得。這種處理方式可以使前面的例子更加簡練（我們可以完全省略 parameter list），但這又給 template argument 的推導系統帶來了新的挑戰。

目前還不完全清楚 C++ 是否會包含 function expressions 概念。然而由 Jaakko Järvi 和 Gary Powell 實作出來的 Lambda 程式庫（見 [LambdaLib]）在這方面取得了長足進展 — 儘管它會耗費相當多的編譯時間。

13.17 後記

在大多數 C++ 編譯器才剛剛能夠符合 1998 *C++ Standard*（*C++98*）的今天，談論對這個語言的擴展似乎爲時過早。然而正是在努力符合標準的過程中，我們才得以洞察 C++ 真正的侷限（特別是 templates 的侷限）。

爲了迎合 C++ 程式員的新需求，C++ 標準委員會（或名 ISO WG21/ANSI J16 或簡稱 WG21/J16）已經開始了新標準的征途：*C++0x*。2001/04 於哥本哈根召開預備陳述會後，WG21/J16 開始著手審查具體的程式庫擴展提案。

審查目標儘可能限於 C++ 標準程式庫的擴展。然而人們都理解，某些擴展需要語言核心的搭配。預計 C++ 語言核心所進行的許多修改都將與 templates 有關，就像 1990 年代 STL 被引入 C++ 標準程式庫促進了 template 技術的發展。

最後要說的是，*C++0x* 也致力於解決 *C++98* 中的一些困窘。人們希望這麼做可提高 C++ 的可用性。本章也對這個方向上的某些擴展做了一些討論。

第三篇 模板與設計

Templates and Design

一般而言，程式通常使用「與選定之編程語言所提供的機制有良好映射關係」的某種設計來加以建構（constructed）。由於 templates 是一種全新的語言機制，它需要新的設計元素，也就不足爲奇。本篇將探索這些元素。

Templates 不同於傳統語言之處在於，它允許我們將程式碼中的型別和常數參數化（parameterize）。當它結合 (1) 偏特化（partial specialization）、(2) 遞迴具現化（recursive instantiation），會產生讓人目瞪口呆的威力。此點將於接下來的數章中以如下的眾多設計技術加以說明：

- Generic programming（泛型編程）
- Traits（特徵、特徵萃取）
- Policy classes（策略類別）
- Meta-programming（超編程）
- Expression templates（算式模板）

我們的目標並不僅止於列出形形色色的設計元素，還要傳達當初激發這些設計的本源，從而讓大家更有可能創造出新技術。

譯註：關於 meta-program（超程式），在 www.dictionary.com 中有這樣的解釋：

所謂 meta-program 是一個會「修改或生成其他程式」的程式。編譯器就是一個例子：它以某個程式爲輸入，並以另一個（被編譯後）的程式爲輸出。

A program which modifies or generates other programs. A compiler is an example of a metaprogram: it taks a program as input and produces another (compiled) one as output.

14

Templates 的多型威力

The Polymorphism Power of Templates

所謂多型（Polymorphism），是一種「以單一泛化記號（generic notation）表述多種特定行爲」的能力[55]。多型是物件導向編程思維模型（object-oriented programming paradigm）的基石，C++ 主要透過 class 的繼承和虛擬函式（virtual functions）支援多型。由於這些機制（全部或部份）生效於執行期，所以稱爲動態多型（**dynamic polymorphism**）。談論 C++ 多型時，所指的通常便是動態多型。然而 templates 也允許我們以單一泛化記號表述多種特定行爲，只不過一般發生在編譯期，因此稱爲靜態多型（**static polymorphism**）。本章將檢閱上述兩種多型，並討論各自適用的場合。

14.1 動態多型（Dynamic Polymorphism）

從歷史上看，C++ 一開始只是透過「繼承機制與虛擬函式的結合運用」來支援多型[56]。這種背景下的多型設計藝術是：在彼此相關的 object types 之間確認一套共通能力，並將其宣告爲某共通基礎類別（common base class）的一套虛擬函式介面。

這種設計最具代表性的例子就是一個「管理若干幾何形狀」的程式，這些幾何形狀可以某種方式著色（例如在螢幕上著色）。在這樣的程式中，我們可以定義一個所謂的抽象基礎類別（abstract base class, ABC）`GeoObj`，在其中宣告適用於幾何物件的一些共通操作（operations）和共通屬性（properties；譯註：意指資料）。每一個針對特定幾何物件而設計的具象類別（concrete class）都衍生自 `GeoObj`（見圖 14.1）

[55] 「多型」的字面含義是：具有多種形式（forms）或形貌（shapes），源自希臘語 "polumorphos"。

[56] 嚴格地說，巨集（macros）也可被視爲一種早期的「靜態多型」形式，不過我們不考慮它們，因爲它們和其他語言機制的關係不大。

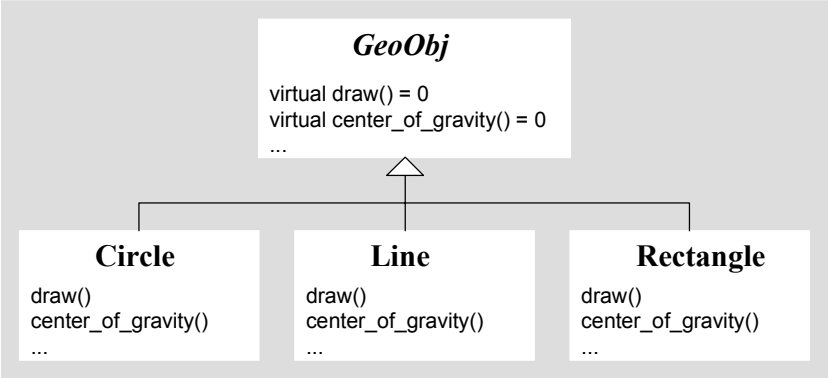


圖 14.1 多型（polymorphism）透過繼承（inheritance）實現出來

```
//poly/dynahier.hpp

#include "coord.hpp"

// 針對幾何物件而設計的共通抽象基礎類別（common abstract base class）GeoObj
class GeoObj {
  public:
    // 繪製幾何物件:
    virtual void draw() const = 0;
    // 傳回幾何物件的重心（center of gravity）:
    virtual Coord center_of_gravity() const = 0;
    //...
};

// 具象幾何類別（concrete geometric class）Circle
// - 衍生自 GeoObj
class Circle : public GeoObj {
  public:
    virtual void draw() const;
    virtual Coord center_of_gravity() const;
    //...
};
```

```
// 具象幾何類別 Line
// - 衍生自 GeoObj
class Line : public GeoObj {
  public:
    virtual void draw() const;
    virtual Coord center_of_gravity() const;
    //...
};
//...
```

建立具象物件（concrete objects）之後，客端程式碼可以透過「指向基礎類別」的 references 或 pointers 來操縱這些物件，這會啓動虛擬函式分派機制（virtual function dispatch mechanism）。當我們透過一個「指向基礎類別之子物件（subobject）」的 references 或 pointers 來呼叫某虛擬函式，會喚起「被指涉（*refered*）的那個特定具象物件」的相應成員函式。

以本例而言，具體程式碼大致描繪如下：

```
//poly/dynapoly.cpp

#include "dynahier.hpp"
#include <vector>

// 繪製任何 GeoObj
void myDraw (GeoObj const& obj)
{
    obj.draw();           // 根據物件的型別呼叫 draw()
}

// 處理兩個 GeoObjs 重心之間的距離
Coord distance (GeoObj const& x1, GeoObj const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs();       // 傳回座標絕對值
}

// 繪出 GeoObjs 異質群集（heterogeneous collection）
void drawElems (std::vector<GeoObj*> const& elems)
{
    for (unsigned i=0; i<elems.size(); ++i) {
        elems[i]->draw();  // 根據物件的型別呼叫 draw()
    }
}
```

```
int main()
{
    Line l;
    Circle c, c1, c2;

    myDraw(l);              // myDraw(GeoObj&) => Line::draw()
    myDraw(c);              // myDraw(GeoObj&) => Circle::draw()

    distance(c1,c2);        // distance(GeoObj&,GeoObj&)
    distance(l,c);          // distance(GeoObj&,GeoObj&)

    std::vector<GeoObj*> coll;   // 異質群集（heterogeneous collection）
    coll.push_back(&l);          // 插入一個 line
    coll.push_back(&c);          // 插入一個 circle
    drawElems(coll);             // 繪出不同種類的 GeoObjs
}
```

上述程式的關鍵性多型介面元素是 `draw()`和 `center_of_gravity()`，兩者都是虛擬函式。程式示範了它們在 `myDraw()`、`distance()`和 `drawElems()`函式內被使用的情況 — 由於這三個函式使用共通基礎類別 `GeoObj` 作爲表達手段，因而無法在編譯期決定使用哪一個版本的 `draw()`或 `center_of_gravity()`。然而在執行期，「喚起虛擬函式」的那個物件的完整動態型別會被取得，以便對呼叫式進行分派（dispatch）。於是，根據幾何物件的實際型別，程式得以完成適當操作：如果對一個 `Line` 物件呼叫 `myDraw()`，函式內的 `obj.draw()`就喚起 `Line::draw()`；對 `Circle` 物件喚起的則是 `Circle::draw()`。同樣道理，對 `distance()`而言，喚起的將是「與引數物件相應」的那個 `center_of_gravity()`。

或許「動態多型」最引人注目的特性就是處理異質物件群集（heterogeneous collections of objects）的能力。`drawElems()`闡示了這樣一個概念：下面的簡單運算式

```
elems[i]->draw()
```

將根據「目前正被處理的元素」型別，喚起不同的成員函式（譯註：都名爲 `draw()`）。

14.2 靜態多型（Static Polymorphism）

Templates 也可以用來實作多型，然而它們並不倚賴「分解及抽取 base classes 共通行爲」。在這裡，共通性是指：應用程式所提供的不同幾何形狀，必須以共通語法支援其操作（也就是說，

相關函式必須同名）。具象類別（concrete classes）之間彼此獨立定義（見圖 14.2）。當 templates 被具象類別「具現化」，便獲得（被賦予）多型的威力。

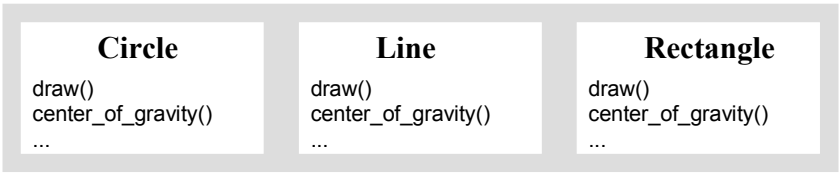


圖 14.2 多型（polymorphism）透過模板（templates）實現出來

例如前一節中的函式 myDraw()：

```
void myDraw (GeoObj const& obj)     //GeoObj 是抽象基礎類別（abstract base class）
{
    obj.draw();
}
```

可設想改寫如下：

```
template <typename GeoObj>
void myDraw (GeoObj const& obj)     //GeoObj 是模板參數（template parameter）
{
    obj.draw();
}
```

比較前後兩份 myDraw()實作碼，我們可以得出一個結論：兩者的主要區別在於 GeoObj 如今是個 template parameter 而非一個 common base class。然而在此表象背後，還有一些更根本的區別。比方說，如果使用動態多型，執行期只會有一個 myDraw()函式，但如果使用 templates，我們會擁有不同的函式如 myDraw<Line>()和 myDraw<Circle>()。

讓我們嘗試使用靜態多型機制改寫前一節的例子。首先不再使用幾何類別階層體系，而是編寫若干個獨立的幾何類別：

```
//poly/statichier.hpp

#include "coord.hpp"

// 具象的幾何類別 Circle
// - 不衍生自任何類別
```

```
class Circle {
  public:
    void draw() const;
    Coord center_of_gravity() const;
    //...
};

// 具象的幾何類別 Line
// - 不衍生自任何類別
class Line {
  public:
    void draw() const;
    Coord center_of_gravity() const;
    //...
};
//...
```

現在，這些 classes 的應用程式看起來像這樣：

```
//poly/staticpoly.cpp

#include "statichier.hpp"
#include <vector>

// 繪出任何給定的 GeoObj
template <typename GeoObj>
void myDraw (GeoObj const& obj)
{
    obj.draw();    // 根據物件的型別呼叫 draw()
}

// 處理兩個 GeoObjs 重心之間的距離
template <typename GeoObj1, typename GeoObj2>
Coord distance (GeoObj1 const& x1, GeoObj2 const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs();    // 傳回座標絕對值
}
```

```
// 繪出 GeoObjs 同質群集（homogeneous collection）
template <typename GeoObj>
void drawElems (std::vector<GeoObj> const& elems)
{
    for (unsigned i=0; i<elems.size(); ++i) {
        elems[i].draw();    // 根據元素的型別呼叫 draw()
    }
}

int main()
{
    Line l;
    Circle c, c1, c2;

    myDraw(l);       // myDraw<Line>(GeoObj&) => Line::draw()
    myDraw(c);       // myDraw<Circle>(GeoObj&) => Circle::draw()

    distance(c1,c2);  // distance<Circle,Circle>(GeoObj1&,GeoObj2&)
    distance(l,c);    // distance<Line,Circle>(GeoObj1&,GeoObj2&)

    // std::vector<GeoObj*> coll;    // ERROR: 不可以是異質群集
    std::vector<Line> coll;          // OK: 同質群集沒問題
    coll.push_back(l);               // 安插一條直線
    drawElems(coll);                 // 畫出所有直線
}
```

`distance()`不可再以 `GeoObj` 爲具體參數型別，`myDraw()`的情況也一樣。我們改而提供兩個 template parameters `GeoObj1` 和 `GeoObj2`。透過兩個不同的 template parameters，一對「幾何型別組」可被拿來計算兩幾何物件間的距離：

```
distance(l,c);   //distance<Line,Circle>(GeoObj1&,GeoObj2&)
```

然而，異質群集（heterogeneous collections）就不再能夠被透通地處理了。這是「靜態多型」的靜態性質所帶來的限制：所有型別都必須在編譯期決定。不過我們可以輕易爲不同的幾何型別引入不同的群集，而且無需再將群集內的元素侷限於 pointers（譯註：如果使用動態多型，異質群集內就一定得放置 pointers），這在執行效率和型別安全方面都有顯著的優勢。

14.3 動態多型 vs.靜態多型

讓我們對動態和靜態兩種多型加以歸類和比較。

術語

動態多型和靜態多型分別支援不同的 C++ 編程手法（idioms）[57]：

- 經由繼承而實現的多型是 bounded（綁定的、已繫結的）和 dynamic（動態的）：
  - **bounded** 意味「參與多型行爲」的型別，其介面係透過 common base class 的設計而預先定妥。此概念的另一些描述術語爲 invasive（侵略性的）或 intrusive（侵入式的）。
  - **dynamic** 意味介面的繫結（綁定; binding）動態完成於執行期。
- 經由 templates 而實現的多型是 unbounded（非綁定的）和 static（靜態的）：
  - **unbounded** 意味「參與多型行爲」的型別，其介面並非預先決定好。此概念的另一些描述術語有 noninvasive（非侵略性的）或 nonintrusive（非侵入式的）。
  - **static** 意味介面的繫結（綁定; binding）靜態完成於編譯期。

因此，嚴格說來（以 C++ 行話來說），動態多型和靜態多型分別是**綁定之動態多型**（bounded dynamic polymorphism）和**綁定之靜態多型**（unbounded static polymorphism）的另一說法。其他語言另有其他結合方式，例如 Smalltalk 提供**綁定之動態多型**（unbounded dynamic polymorphism）。「動態多型」和「靜態多型」兩個簡練術語在 C++ 環境中並不會招致混淆。

優點和缺點

C++ 的動態多型表現出如下優點：

- 可優雅處理異質群集（Heterogeneous collections）。
- 可執行碼的體積（executable code size）可能比較小。因爲只需一個多型函式。靜態多型則必須產生不同的 template 實體以處理不同的型別。
- 程式碼可被完全編譯，因而無需非得發佈實作源碼不可。如果你發行的是 template 程式庫，通常還需釋出其實作源碼。

相較之下，以下的優點屬於 C++ 靜態多型：

- 內建型資料群集（collections of built in types）可被輕鬆實作出來。更一般地說，共通性介面無需透過一個共通基礎類別（common base class）來表達。
- 生成的程式碼可能執行速度較快。因爲無需透過 pointers 進行間接操作，而且非虛擬函式（nonvirtual functions）被 *inlined*（內聯化）的可能性比較大。

---

[57] 關於「多型」術語的詳細討論，也可以參考 [CzarneckiEisenecker-GenProg], 6.5-6.7 節。

- 即使只提供部份介面的具象型別（concrete types）仍然可用 — 如果最終只有這一部份介面被應用程式用上的話。

通常靜態多型被認爲比動態多型更具型別安全性（type safe），因爲所有繫結（binding）都被檢查於編譯期。舉個例子，這種程式不可能存在這樣的危險：將一個型別不匹配的物件插入一個從 template 具現出來的容器內。然而如果容器期望獲得的元素是 pointer to common base class，這個 pointer 有可能無意中指向不同型別的物件身上。

現實之中，如果不同的「語意假設」隱藏於外觀完全一致的介面背後，template 具現體也可能招致某些麻煩。例如當 template 將 `operator+` 針對某型別具現化，但該型別實際上與 + 操作並無關聯，人們可能會大感驚訝。在現實經驗中，此類「語意不匹配」很少發生於以繼承爲基礎的 class 階層體系，這或許是因爲在那種情況下介面的規格有更嚴明的指定。

將兩種形式結合起來

當然了，你可以結合使用靜態多型和動態多型。例如你可以從一個 common base class 衍生出不同種類的幾何物件，以便經營一個異質幾何物件群集，而同時仍可針對某些種幾何物件以 templates 編寫其程式碼。

第 16 章將就繼承機制和 templates 的結合運用作進一步討論。我們將看到成員函式的虛擬性如何被參數化、以繼承爲基礎的 CRTP（curiously recurring template pattern, 奇特遞迴模板範式）又是如何爲靜態多型提供額外的彈性。

14.4 Design Patterns（設計範式）的新形式

「靜態多型」爲 design patterns（設計範式）帶來了新的實作方式。舉個例子，***bridge*** 範式在 C++ 程式中扮演重要角色，其運用目標之一是「在同一介面的不同實作之間進行切換」。根據 [DesignPatternsGoV] 的描述，通常的達成方式是：以一個指標指向實際實作品，並將所有呼叫動作都委託（*delegating*）給該 class（見圖 14.3）。

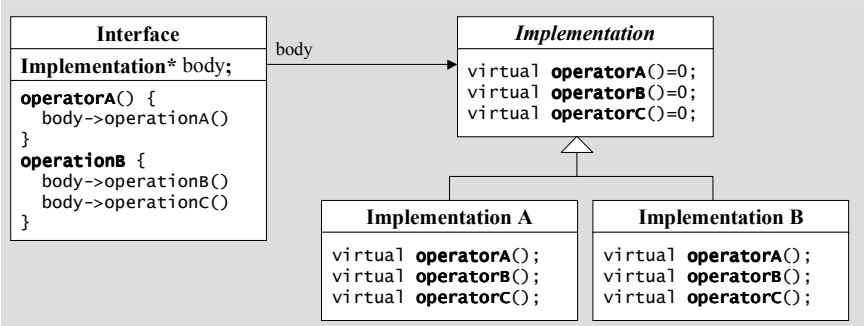


圖 14.3 運用繼承（inheritance）實作出 ***Bridge*** 範式

然而如果編譯期就能夠知道實作品的型別，你就能夠透過 templates 使用這一方案（見圖 14.4）。這會帶來更好的型別安全性（type safety），又可避免運用指標，而且執行時應該會更快速。

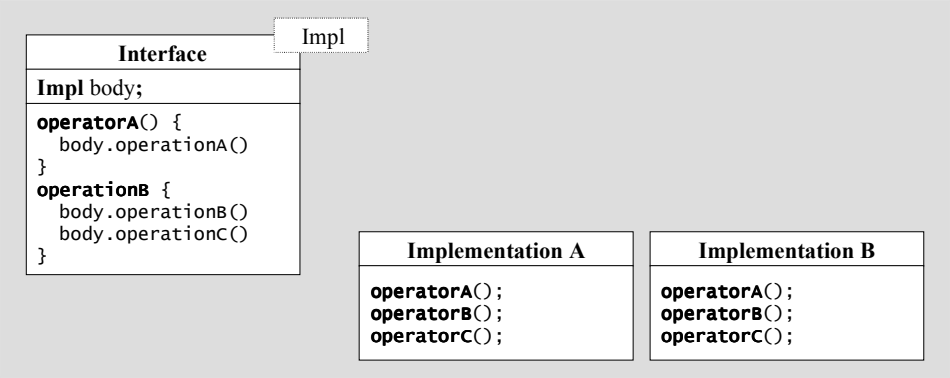


圖 14.4 運用 templates 技術實作出 *Bridge* 範式

14.5 泛型編程（Generic Programming）

靜態多型帶來了「泛型編程」概念。然而世上並不存在一個被普遍認同的泛型編程定義（就像不存在被普遍認同的物件導向編程定義一樣）。按 [CzarneckiEiseneckerGenProg] 的描述，眾多定義的根源是：「使用泛化參數（generic parameters）進行編程，以便找出高效演算法（efficient algorithms）之最抽象表述（most abstract representation）」。那本書最後總結如下：

> 泛型編程是一個計算機科學分支，適合用來找出（發現）高效演算法、資料結構、其他軟體概念…之抽象表述，並用以處理它們的系統化組織方式。泛型編程主要用於表現「領域概念」族系（families of domain concepts）（p.169 和 p.170）。

在 C++ 環境中，泛型編程有時被定義爲「以 templates 進行編程」（而物件導向編程則被認爲是「以虛擬函式（virtual functions）進行編程」）。從這個意義上說，只要使用了 C++ templates，任何編程行爲都可被視爲泛型編程。不過專業人士通常認爲泛型編程還具有其他某些本質要素：爲進行大量有意義的聯合運用，templates 應該被設計於框架（framework）之中。

迄今爲止，這個領域中最耀眼的貢獻是 STL（Standard Template Library，後被編入 C++ 標準程式庫內）。STL 是個框架（framework），提供大量有用操作，稱爲演算法（**algorithms**），以及大量用於處理物件群集（collections of objects）的線性資料結構，稱爲容器（**containers**）。

演算法和容器都是 templates。不過最關鍵的還在於：演算法並非容器的成員函式，而是以某種泛化方式編寫而成，因此可被任何容器（以及由元素組成的任何線性群集）所用。爲了做到這一點，STL 設計者確認了一個抽象概念：迭代器（**iterators**），它可被用於任何種類的線性群集身上。事實上容器的各項操作中，凡與群集相關的部分（譯註：例如取前一元素、取下一元素、取第一元素，取最後元素…），都已被抽取（分解）爲迭代器的機能。

因此，計算「序列內的最大值」這般操作，便可在不了解序列實值（values）儲存細節的情況下完成：

```
template <class Iterator>
Iterator max_element (Iterator beg,   //指向（代表）群集的頭
                      Iterator end)   //指向（代表）群集的尾
{
    //只需使用 Iterator 的某些操作，在群集的每個元素身上來回移動，
    //找出帶有最大值的那個元素，並將其位置包裝爲一個 Iterator 傳回。
    ...
}
```

每個線性容器不再需要提供諸如 `max_element()` 這樣的操作。容器只需提供迭代器型別（iterators type）用以巡訪（遍歷）容器所含的實值序列（sequence of values），並提供成員函式建立這樣的迭代器即可：

```
namespace std {
   template <class T, ...>
   class vector {
     public:
       typedef ... const_iterator;        //與實作相依（implementation-specific）
       ...                                //的 iterator 型別，用於 const vectors.
       const_iterator begin() const;      //傳回一個 iterator，指向群集頭部
       const_iterator end() const;        //傳回一個 iterator，指向群集尾端
       ...
   };

   template <class T, ...>
   class list {
     public:
       typedef ... const_iterator;        //與實作相依（implementation-specific）
       ...                                //的 iterator 型別，用於 const lists.
       const_iterator begin() const;      //傳回一個 iterator，指向群集頭部
       const_iterator end() const;        //傳回一個 iterator，指向群集尾端
       ...
   };
}
```

現在，只要以群集起點和終點元素爲引數，呼叫泛化後的 `max_element()`，就可以找出任何群集的最大值（這兒暫略對空群集的特別處理）：

```
//poly/printmax.cpp

#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include "MyClass.hpp"

template <typename T>
void print_max (T const& coll)
{
    // 宣告群集的區域迭代器（local iterator）
    typename T::const_iterator pos;

    // 計算最大值的位置
    pos = std::max_element(coll.begin(),coll.end());

    // 列印出 coll 的最大元素（如果有的話）：
    if (pos != coll.end()) {
        std::cout << *pos << std::endl;
    }
    else {
        std::cout << "empty" << std::endl;
    }
}

int main()
{
    std::vector<MyClass> c1;
    std::list<MyClass>   c2;
    //...
    print_max (c1);
    print_max (c2);
}
```

由於操作式對迭代器（iterators）實施了參數化，STL 得以避免操作式的數量爆炸。如今我們不再需要爲每個容器實作每一個操作式，只需演算法實作一次便可用於所有容器。迭代器是一種泛化膠水（generic glue），由容器提供，被演算法運用。這種機制之所以有效運作，原因在於迭代器具有某種介面，該介面由容器提供並爲演算法所用。這種介面常被稱爲 **concept**（概念），標記出一套約束條件（constraints）— 每一個 templates 都必須實現它們才能符合 STL 框架要求。

原則上，動態多型也可以實現 STL 這樣的機能，然而實際很少那麼做。因爲和虛擬函式呼叫機制相比，迭代器概念（iterator concept）顯得輕巧許多。如果添加一個虛擬函式介面層，操作速度極有可能被拖慢一個數量級（甚至不止）。

泛型編程之所以實用，正是因爲它倚賴「於編譯期間對介面完成決議（*resolves*）」的靜態多型機制。另一方面，「於編譯期間對介面完成決議」這一需求，也聲聲呼喚著新的設計原理，這些新原理在很多方面都不同於物件導向設計原理。本書剩餘章節將描述許多極其重要的泛型設計原理（generic design principles）。

14.6 後記

容器型別（container types）是將 templates 引入 C++ 編程語言的首要動機。templates 出現前，多型階層體系（polymorphic hierarchies）是處理容器的流行方式。一個廣爲流傳的例子是 NIHCL（National Institutes of Health Class Library），它很大程度「翻譯」了 Smalltalk 的容器類別階層體系，參見圖 14.5。

NIHCL 和 C++ 標準程式庫很像，也支援豐富而多樣的容器和迭代器。然而此一實作品沿襲 Smalltalk 的動態多型 — 其 `Iterators` 使用抽象基礎類別 `Collection` 來操作不同種類的群集：

```
Bag c1;
Set c2;
...
Iterator i1(c1);
Iterator i2(c2);
...
```

遺憾的是，不論從執行時間或記憶體用量來看，這種方式的代價都太高。其執行時間比 C++ 標準程式庫的等價程式碼相差數個數量級，因爲絕大多數操作最終都呼叫虛擬函式（而 C++ 標準程式庫內的許多操作都是 *inlined*，且其迭代器和容器的介面也不涉及虛擬函式）。此外，由於其介面是 bounded（已繫結/綁定）（這和 smalltalk 不同），所以內建型別（built-in types）不得不被包裝爲較大的多型類別（polymorphic classes；NIHCL 就提供了這樣的包裝器; wrapper），從而導致儲存空間的需求呈戲劇性增長。

有些人在巨集（macros）中尋求慰藉，然而即使到了今天這樣的 templates 時代，許多專案只是

做出次佳的多型選擇（譯註：意思可能是說應該選用帶有型別檢測能力的 templates，卻選用了無任何檢測能力的 macros）。毫無疑問許多時候動態多型的確是正確的選擇 — 異質物件迭代（Heterogeneous iterations）就是個明顯例子。然而在此同時，許多編程任務也可以自然而高效地以 templates 解決 — 同質容器（homogeneous containers）就是個例子。

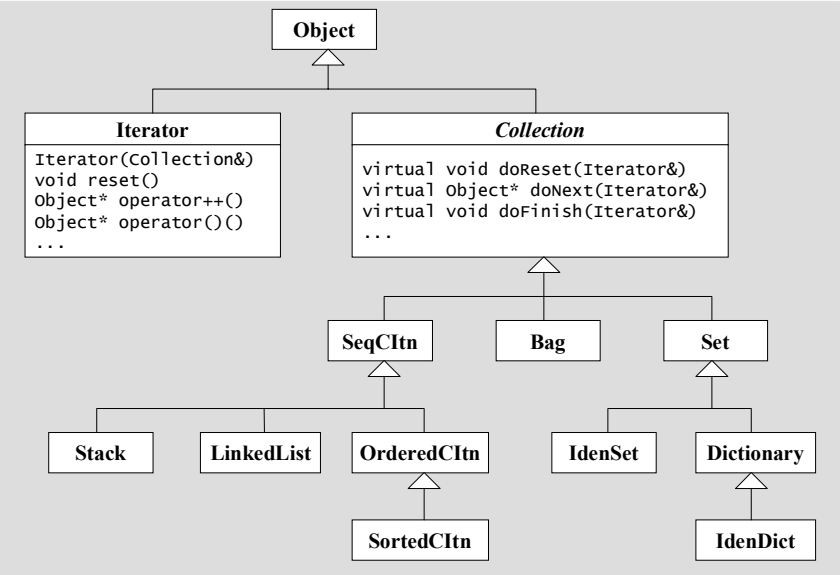


圖 14.5 NIHCL 的類別階層體系（class hierarchy）

「靜態多型」適合編寫十足的基礎計算結構。相形之下，「動態多型」需要選擇一個共通基礎型別（common base type），意味其程式庫往往是領域專屬的（domain-specific）。因此，毫不奇怪，C++ 標準程式庫中的 STL 從未包含「多型容器」（譯註：意指可放置任意類型的元素，類似 Java Collection classes），而是包含了一大套使用靜態多型機制的容器和迭代器（正如 14.5 節, p.241 所示）。

中大型 C++ 程式通常同時需要處理本章討論的兩種多型。某些情況下甚至需要將二者緊密結合。按照我們的討論，許多情況下的最佳設計其實十分明顯。不過，花一點時間考慮可能的長遠演進，也幾乎總是能夠獲得良好的回報。

15

Traits 和 Policy Classes

「特徵萃取」和「策略類別」

譯註：traits：特徵、特性。features：特性、性質。characteristics：特徵。這些名詞的中譯容易讓人混淆。本章在可能混淆之處儘量保留（或加上）英文。traits 可能視情況被譯爲「特徵萃取」。

Templates 使我們能夠以形形色色的型別對 classes 和 functions 進行參數化。或許存在這麼一種誘惑：爲程式引入儘可能多的 templates parameters，俾使能夠訂製（*customize*）型別或演算法的方方面面。然後我們的 templatized 組件就可以被具現化，滿足客端程式碼的確切需求。然而從實踐觀點來看，很少爲了求取最大限度的參數化而引入大量 templates parameters。如果客端程式碼必須一一指定所有相應引數，那可真是無趣至極。

幸運的是，我們引入的大多數附加參數（extra parameters）都可以有合情合理的預設值。某些情況下，附加參數完全由少量主參數（main parameters）決定，而且我們發現，這樣的附加參數完全可以省略。其他某些參數可以倚賴主參數而獲得預設值。預設值往往足以滿足大多數場合的需要，不過有時候（針對特殊應用）它們也必須被覆寫。另有一些附加參數和主參數並不相干，從某種意義上來說它們也是主參數，只是存在著「幾乎總是能夠滿足需求」的預設值。

譯註：把參數分爲「主要」和「附加」兩類，並不是嚴謹或常見的說法。在這兒，作者的意思是，一定得由呼叫端給值（給予引數）者，稱爲「主參數」，可由主參數推導者，稱爲「附加參數」。額外參數應可於函式重新編寫後被去除。

Policy classes 和 **traits**（或 traits templates）是這樣一種 C++ 編程裝置（programming device）：它們可爲「工業強度之 templates 設計」中的附加參數帶來很大方便的管理。本章展示了許多情況，證明它們很有用，並示範形形色色的技術。你可以憑藉這些技術編寫一些強固而威力強大的裝置（devices）。

15.1 示例：序列的累計（Accumulating a Sequence）

計算數值序列（sequence of values）的總和，是十分常見的任務。這個看似簡單的問題爲我們提供了優秀的示例，讓我們得以介紹 policy classes 和 traits 可發揮的不同層面。

15.1.1 Fixed Traits（固定式特徵）

譯註：此處所謂 "fixed"，意指「不經由 template parameters 傳遞」。見 p.259 上端。

首先假設，求和對象（所有數值）都被儲存於一個 array 內，而且我們手上有兩個指標，分別可以存取首元素和尾元素。由於這是一本 templates 相關書籍，所以我們希望編寫出一個適用多種型別的 template。就目前而言，以下程式碼看起來直截了當[58]：

```
//traits/accum1.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

template <typename T>
inline
T accum (T const* beg, T const* end)
{
    T total = T();  // 假設 T() 產生一個零值
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif // ACCUM_HPP
```

這裡惟一有點微妙的地方在於：如何正確地爲某個型別產生一個零值，以便開始求和運算。這兒用的是算式 `T()`，它對 `int` 和 `float` 這樣的內建數值型別而言，可以有效運作（見 5.5 節, p.56）。

爲促成我們的第一個 traits template，看看下面程式如何使用 `accum()`：

```
//traits/accum1.cpp

#include "accum1.hpp"
#include <iostream>

int main()
{
```

[58] 爲求簡單起見，本節大多數例子都使用一般指標（ordinary pointers）。但是很顯然，具備工業強度的介面，更傾向於使用「遵從 C++ 標準程式庫約定」的迭代器（iterators）（見 [JosuttisStdLib]）。稍後我們會重新檢討這些例子。

```
    // 建立一個帶有 5 個整數值的 array
    int num[] = { 1, 2, 3, 4, 5 };

    // 列印平均值
    std::cout << "the average value of the integer values is "
              << accum(&num[0], &num[5]) / 5
              << '\n';

    // 建立一個帶有 char 元素值的 array
    char name[] = "templates";
    int length = sizeof(name)-1;

    // 嘗試列印 char 平均值
    std::cout << "the average value of the characters in \""
              << name << "\" is "
              << accum(&name[0], &name[length]) / length
              << '\n';
}
```

程式的前半部份以 `accum()` 求 5 個整數的和：

```
int num[] = { 1, 2, 3, 4, 5 };
...
accum(&num[0], &num[5])
```

只要將求和結果除以 array 的元素個數，就可以得到平均值。

程式後半段試圖對單字 "templates" 中的所有字母做同樣的事(前提是:在一個實際字元集中，字元 a 到字元 z 構成一個連續序列。這對 ASCII 字元集是成立的，對 EBCDIC 字元集就不然[59])。按照推測，計算結果應該位於 a 和 z 之間。在今天的大多數平台上，實際值由 ASCII 碼決定：a 被編碼爲 97，z 被編碼爲 122，因此我們預期計算結果落在 97 和 122 之間。然而在我們的平台上，程式輸出如下：

```
the average value of the integer values is 3
the average value of the characters in "templates" is -5
```

問題在於我們的 template 乃是針對 char 型別而具現化，這個型別的數值範圍太小，以至於即使面對相當小的數值都不足以完成累計動作。顯然，我們可以引入額外的 templates parameter

[59] EBCDIC 是 "Extended Binary-Coded Decimal Interchange Code" 的縮寫，這是一種 IBM 字元集，廣泛應用於大型 IBM 電腦。

AccT 來描述變數 total 的型別（同時也是回返值型別），解決這個問題。然而這會造成此一 template 所有用戶的一份額外負擔：每次喚起這個 template 都不得不多指定一個型別。我們的例子因而必須改寫爲：

```
accum<int>(&name[0],&name[length])
```

這種約束（或說負擔）雖不過份，畢竟可以避免。

下面是上述「附加參數」解法的一個替代方案：爲呼叫 accum() 時的「累積物型別 T」和容納累計結果的「回傳型別 R」產生一份關聯（association），這份關聯可被視爲型別 T 的一種特徵（characteristic）。因此回傳型別（總量型別）有時稱爲 T 的一個 **trait**（特徵、特性）。這個關聯性可被編寫爲 template specializations：

```
//traits/accumtraits2.hpp

template<typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
  public:
    typedef int AccT;  //譯註：一個 int 通常可以容納兩個 char 相加
};

template<>
class AccumulationTraits<short> {
  public:
    typedef int AccT;  //譯註：一個 int 通常可以容納兩個 short 相加
};

template<>
class AccumulationTraits<int> {
  public:
    typedef long AccT; //譯註：一個 long 通常可以容納兩個 int 相加
};

template<>
class AccumulationTraits<unsigned int> {
  public:
    typedef unsigned long AccT;
                    //譯註：一個 unsigned long 通常可以容納兩個 unsigned int 相加
};

template<>
class AccumulationTraits<float> {
  public:
    typedef double AccT;    //譯註：一個 double 通常可以容納兩個 float 相加
};
```

上述的 template `AccumulationTraits` 即是所謂的 traits template，因爲它持有其參數型別的一個 trait（特徵）。通常會有一個以上的 traits 和一個以上的參數。此例並不爲這個 template 提供一般性定義（譯註：也就是不定義一個定義式通吃各種型別 `T`），因爲如果我們不知道元素型別是什麼，就沒有很好的辦法爲它選取合適的「成果型別」。不過元素型別 `T` 本身通常可以成爲「成果型別」的優秀人選（儘管先前的例子顯然沒有這麼做）。

有了這樣的想法，我們將 `accum()` template 改寫如下：

```
//traits/accum2.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits2.hpp"

template <typename T>
inline
typename AccumulationTraits<T>::AccT accum (T const* beg,
                                            T const* end)
    //譯註：以上灰底程式碼的意思是：把 T 丟進某個「特徵萃取機制」中，取其 Acct 特徵。
{
    // 令回返型別（return type）爲「元素型別的特徵（traits）」
    typedef typename AccumulationTraits<T>::AccT AccT;

    AccT total = AccT();  // 假設 AccT() 真的產生一個零值
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif // ACCUM_HPP
```

於是程式的輸出結果就成了我們所期望的：

```
the average value of the integer values is 3
the average value of the characters in "templates" is 108
```

大體上，由於增加了一個「可協助我們訂製演算法」的極有用機制，因此任何修改也就不怎麼引人注目了。更進一步說，若有任何新型別打算用於 accum()，我們可以將一個適當的 AccT 與之產生關聯 — 只要爲 AccumulationTraits template 宣告另一份顯式特化（explicit specialization）即可。任何型別都可以如法炮製，包括基礎型別（fundamental types）、宣告於其他程式庫中的型別…等等。

15.1.2 Value Traits（數值式特徵）

到目前爲止，我們已經看過以 traits 來表達「與某給定主型別有所關聯」的型別附加資訊。本節將告訴你，這種附加資訊並不僅僅侷限於型別（譯註：上一節的 Acct）。事實上常數和其他種類的數值（譯註：本節將出現的 zero）一樣可以和某個型別產生關聯。

accum() template 最初版本乃是利用「回返型別（return type）之 *default* 建構式」，將用以儲存結果的那個變數初始化爲「類零值」（zero-like value）（譯註：之所以說「類零值」乃是因爲並非全爲零，例如 bool 的初值是 false）：

```
AccT total = AccT();  //假設 AccT() 真的產生一個零值（zero value）
...
return total;
```

但顯然這並不保證能夠產生一個得以開始進行「累計迴圈」的良好值，因爲 AccT 說不定根本就沒有 *default* 建構式。

這時候 traits 可以再次充當救兵。就本例而言，我們可以對 AccumulationTraits 加入一個新的 value trait：

```
//traits/accumtraits3.hpp

template<typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
  public:
    typedef int AccT;
    static AccT const zero = 0;
};

template<>
class AccumulationTraits<short> {
  public:
    typedef int AccT;
    static AccT const zero = 0;
};
```

```
template<>
class AccumulationTraits<int> {
  public:
    typedef long AccT;
    static AccT const zero = 0;
};
//...
```

這種情況下，新的 trait 是一個可於編譯期核定的常數。於是 `accum()` 可改寫爲：

```
//traits/accum3.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits3.hpp"

template <typename T>
inline
typename AccumulationTraits<T>::AccT accum (T const* beg,
                                            T const* end)
{
    // return type is traits of the element type
    typedef typename AccumulationTraits<T>::AccT AccT;

    AccT total = AccumulationTraits<T>::zero; //譯註：使用前述三個 zero 之一
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif // ACCUM_HPP
```

在這段程式碼中，用以儲存累計值的那個變數，其初始化動作還是十分簡單明瞭：

```
AccT total = AccumulationTraits<T>::zero;
```

這種作法的缺點之一在於，面對 class 內的 static const 成員變數，只有當它是整數（integral）或 enum 時，C++ 才允許我們對它進行初始化動作；我們自己定義的 classes 被排除在外，浮點數也被排除在外，不得直接設定初值。因此下面的特化版本是錯誤的：

```
...
template<>
class AccumulationTraits<float> {
public:
    typedef double AccT;
    static double const zero = 0.0;  //錯誤：不是整數型別（integral type）
};
```

一個簡單的替代方案是：不要在 class 內「定義」value trait：

```
...
template<>
class AccumulationTraits<float> {
public:
    typedef double AccT;
    static double const zero;    //譯註：不再定義其值
};
```

這麼一來初值設定器（initializer）就會到某個源碼檔案中尋找類似下面的句子：

```
...
double const AccumulationTraits<float>::zero = 0.0;
```

儘管這樣可以有效運作，但它有個缺點：對編譯器而言更加不透明了。編譯器處理客戶端（client）檔案時往往察覺不出「定義式」位於其他檔案中。這種情況下編譯器無法運用「數值 zero 其實就是 0.0」這一事實。

因此，我們更傾向於實作這樣的 value traits：不強求以 *inlined* 成員函式傳回一個整數值（integral values）[60]。 比方說我們可以將 AccumulationTraits 改寫如下：

```
// traits/accumtraits4.hpp

template<typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
  public:
    typedef int AccT;
    static AccT zero() {
```

[60] 大多數現代 C++ 編譯器可以識破「簡單 inline 函式」的呼叫。

```
    return 0;
  }
};

template<>
class AccumulationTraits<short> {
  public:
    typedef int AccT;
    static AccT zero() {
      return 0;
    }
};

template<>
class AccumulationTraits<int> {
  public:
    typedef long AccT;
    static AccT zero() {
      return 0;
    }
};

template<>
class AccumulationTraits<unsigned int> {
  public:
    typedef unsigned long AccT;
    static AccT zero() {
      return 0;
    }
};

template<>
class AccumulationTraits<float> {
  public:
    typedef double AccT;
    static AccT zero() {
      return 0.0;
    }
};

...
```

對應用程式而言，新作法造成的惟一區別是：改用函式呼叫語法（而不是較簡單的「static 成員變數存取語法」）：

```
AccT total = AccumulationTraits<T>::zero();
```

譯註：上述所謂較簡單的「static 成員變數存取語法」，即是：

```
AccT total = AccumulationTraits<T>::zero;        //見 p.251
```

顯然 traits 的作用不僅僅侷限於「提供額外型別」。在我們的例子中，它可以作爲一種機制，提供 `accum()`需要知道的「元素型別的所有必要資訊」。這正是 traits 概念的關鍵：它提供一種途徑，用以爲具體元素（通常是某些型別）設定某些資訊，供泛型計算時使用。

15.1.3 Parameterized Traits（參數式特徵）

先前小節中的 `accum()`所使用的 traits 被稱爲 **fixed**（固定式），因爲一旦 decoupled trait（解耦用的特徵萃取機制）定義完畢，你就不可以在演算法中覆寫（overridden）它。不過某些情況下這樣的覆寫可能是我們想要的，例如當我們碰巧知道某一套浮點數可被安全地累計放進一個同型的（浮點數）變數中，「覆寫 traits」就有可能使我們的開發更高效。

原則上，解決方案是這樣：添加一個帶有預設值的 template parameter，預設值由我們的 traits template 決定。這麼一來大部分用戶可以忽略額外的 template argument，而特殊需求的用戶則可以覆寫（override）預先設定的累計型別。此方案惟一令人不快的是：function templates 不能擁有預設的 template arguments[61]。

目前就讓我們透過「把演算法規劃爲一個 class」來智取問題。這也說明了一個事實：traits 可用於 class templates 之中，而且至少像用於 function templates 那樣容易。我們的應用程式不足之處在於：class templates 無法進行 template argument deduction（模板引數推導），因此引數必須被明確提供出來。因此我們需要以下形式：

```
Accum<char>::accum(&name[0], &name[length])
```

來使用新修訂的「累計用 template」：

```
// traits/accum5.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits4.hpp"
```

[61] 這種狀況幾乎肯定會在 C++ *Standard* 修訂版中獲得改變，編譯器廠商也有可能在這份修訂標準尚未發佈之前先提供這個性質（見 13.3 節, p207）。

```
template <typename T,
          typename AT = AccumulationTraits<T> >
class Accum {
  public:
    static typename AT::AccT accum (T const* beg, T const* end) {
        typename AT::AccT total = AT::zero();
        while (beg != end) {
            total += *beg;
            ++beg;
        }
        return total;
    }
};

#endif // ACCUM_HPP
```

上述 template 的大多數用戶很可能從來都不需要明確提供第二個 template argument，因爲根據第一個 template argument 就可以獲得適當預設值。

通常我們會導入一些便捷函式（convenience functions）用以簡化介面：

```
template <typename T>
inline
typename AccumulationTraits<T>::AccT accum (T const* beg,
                                            T const* end)
{
    return Accum<T>::accum(beg, end);
}

template <typename Traits, typename T>
inline
typename Traits::AccT accum (T const* beg, T const* end)
{
    return Accum<T, Traits>::accum(beg, end);
}
```

15.1.4 Policies（策略）和 Policy Classes（策略類別）

到目前爲止，我們一直都將累計（accumulation）與求和（summation）混爲一談。顯然我們其實可以設想其他種類的累計。例如我們可以求給定之實值序列的乘積；如果被操作的實值是字串，我們可以將它們串接起來；甚至「尋找序列中的最大值」也可被歸結爲累計問題。在所有情況中，`accum()`惟一需要修改的就是 `total += *beg`。這個操作可被稱爲「累計運算」過程

中的一個 **policy**（策略）。因此所謂 **policy class** 就是這樣的 class：提供一個介面，從而得以在演算法中運用一或多個 policies[62]。

下面是個例子，示範如何在 Accum class template 中引入上述介面：

```
// traits/accum6.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits4.hpp"
#include "sumpolicy1.hpp"

template <typename T,
          typename Policy = SumPolicy,
          typename Traits = AccumulationTraits<T> >
class Accum {
  public:
    typedef typename Traits::AccT AccT;
    static AccT accum (T const* beg, T const* end) {
        AccT total = Traits::zero();
        while (beg != end) {
            Policy::accumulate(total, *beg);
            ++beg;
        }
        return total;
    }
};

#endif // ACCUM_HPP
```

其中用到的 SumPolicy 可以這樣定義：

```
// traits/sumpolicy1.hpp

#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP

class SumPolicy {
  public:
```

[62] 我們可以將它泛化爲一個 *policy parameter*，可以是個 class（正如我們所討論的）或是個 pointer to function。

```
    template<typename T1, typename T2>
    static void accumulate (T1& total, T2 const & value) {
        total += value;
    }
};

#endif // SUMPOLICY_HPP
```

在這個例子中，我們選擇讓 policy 成爲常規 class（而不是個 class template），它具有一個 static member function template（而且是隱式內聯, implicitly inline）。另一個替代方案將於稍後討論。

透過「指定不同的 policy」來進行累計動作，我們就可以完成多種計算。例如下面這個程式，其目的是計算某些數值的乘積（product）：

```
// traits/accum7.cpp

#include "accum6.hpp"
#include <iostream>

class MultPolicy {
  public:
    template<typename T1, typename T2>
    static void accumulate (T1& total, T2 const& value) {
        total *= value;
    }
};

int main()
{
    // 產生一個 array，內有 5 個整數值
    int num[] = { 1, 2, 3, 4, 5 };

    // 列印所有數值的乘積
    std::cout << "the product of the integer values is "
              << Accum<int,MultPolicy>::accum(&num[0], &num[5])
              << '\n';
}
```

然而上述程式的輸出結果不如預期：

```
the product of the integer values is 0
```

問題在於我們對初始值的不當選擇：初始值 0 對「求和」而言很合適，對「乘積」來說就不妥了（初始值爲 0，註定乘積結果也是 0）。這也說明了不同的 traits 和 policies 可能相互影響，同時也強調了謹慎設計 templates 的重要性。

從這個例子中我們可以意識到，accumulation loop（累計迴圈）的初始化應該成爲 accumulation policy 的一個成份。這個 policy 可以使用也可以不使用 trait `zero()`。其他替代方案也不該被遺忘 — 並不是什麼事都非得使用 traits 和 policies 解決不可。C++ 標準程式庫的 `accumulate()` 就是接受第三個 call argument 做爲初始值。

15.1.5 Traits 和 Policies 有何差異？

我們可以舉出相當合理的例子來支持一個事實：policies 只是 traits 的特例，反過來說 traits 只不過是「對 policy 的編碼（encode）」而已。

《*New Shorter Oxford English Dictionary*》（新簡潔牛津英文字典；[NewShorterOED]）這樣說：

- **trait**, n. ... *a distinctive feature characterizing a thing*
  某種獨特特徵，用以刻畫（描繪）事物。
- **policy**, n. ... *any course of action adopted as advantegous or expedient*
  爲了優勢或權宜（方便）而採納的任何行動方向（方針）

基於此，我們往往將所謂的 policy classes 限定爲這樣的 classes：對某種行爲進行編碼（encode），該種行爲很大程度與同組其他任何 template arguments 不相干（亦即正交, orthogonal）。這和 Andrei Alexandrescu 在其《*Modern C++ Design*》（[AlexandrescuDesign], p.8）書中的陳述一致[63]：

> *Policies have much in common with traits but differ in that they put less emphasis on type and more on behavior.*（Policies 和 traits 之間有許多共同點，但前者更強調行爲而非型別。）

traits 技術的引入者 Nathan Myers，提出了更開放的定義（見 [MyersTraits]）：

> *Traits class: A class used in place of template parameters. As a class, it aggregates useful types and constants; as a template, it provides an avenue for that "extra level of indirection" that solves all software problems.*（Traits class：一種被用來取代 template parameters 的 class。作爲一個 class，它聚合了有用的型別和常數；作爲一個 template，它爲「可用來解決所有軟體難題」的所謂「額外間接層」提供了一條康莊大道。）

通常我們傾向於使用如下（稍微模糊）的定義：

- **Traits** 表現一個 template parameter 的自然附加屬性（natural additional properties）。

[63] Alexandrescu 儼然成爲 policy classes 世界的主要代言人，他以 policy classes 爲基礎開發了一套豐富技術（譯註：主要表現於 "Loki" 程式庫）。

- **Policies** 表現泛化函式和型別間的「可設置行爲」（configurable behavior；通常帶有預設值）。

爲了進一步闡述兩種概念之間的可能區別，我們列出對 traits 的觀察：

- Traits 作爲 fixed traits（也就是說不經由 template parameters 傳遞）是有用的。
- Traits 參數通常具有十分自然的預設值（它們很少被覆寫，或不可被覆寫）。
- Traits 參數傾向於緊密依賴一或多個主參數（main parameters）。
- Traits 通常與型別和常數（而非成員函式）結合使用。
- Traits 傾向被收集於所謂的 traits templates 中。

對於 policy classes，我們列出如下觀察：

- 如果不以 template parameters 傳遞的話，policy classes 作用不大。
- Policy 參數無需擁有預設值，而且常被明確指定（雖然許多泛型組件有常用的預設值）。
- Policy 參數通常和 templates 的其他參數「不相干」（正交; orthogonal）。
- Policy classes 通常和成員函式結合使用。
- Policies 可被收集於 "plain"（單純的、簡樸的）classes 或 class templates 中。

當然，兩個術語之間的界限並不是那麼涇渭分明。例如 C++ 標準程式庫的 **character traits** 也定義了諸如比較、搬移、搜尋字元的功能性行爲。只要替換這些 character traits，你就可以在保持相同字元型別（譯註：意指設計上無太大改變）的情況下定義一個「可區分大小寫」的 string classes（見 [JosuttisStdLib] 11.2.14 節）。因此儘管它們被稱爲 traits，也具有一些 policies 相關屬性。

15.1.6 Member Templates vs. Template Template Parameters

爲實作一個 accumulation policy（譯註：用以決定累計動作是「加」或「乘」或其他…），我們選擇將 `SumPolicy` 和 `MultPolicy` 表現爲「擁有一個 member function template」的一般性 classes（譯註：正如 p.256 和 p.257 所列）。另一種作法是以 class templates 來設計 policy class interface，然後可以被拿來作爲 template template arguments 使用。例如我們可以將 `SumPolicy` 改寫爲一個 template：

```
// traits/sumpolicy2.hpp

#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP

template <typename T1, typename T2>
class SumPolicy {
  public:
    static void accumulate (T1& total, T2 const & value) {
        total += value;
```

```
    }
};

#endif // SUMPOLICY_HPP
```

然後我們可以修改 Accum 介面，俾能使用一個 template template parameter：

```
// traits/accum8.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits4.hpp"
#include "sumpolicy2.hpp"

//譯註：以下可與 p.256 之 accum6.hpp 比較。
template <typename T,
          template<typename,typename> class Policy = SumPolicy,
          typename Traits = AccumulationTraits<T> >
class Accum {
  public:
    typedef typename Traits::AccT AccT;
    static AccT accum (T const* beg, T const* end) {
        AccT total = Traits::zero();
        while (beg != end) {
            Policy<AccT,T>::accumulate(total, *beg);
            ++beg;
        }
        return total;
    }
};

#endif // ACCUM_HPP
```

對 traits 參數也可以如法炮製。（關於這個主題，亦存在其他可能方案。例如不再「將 AccT 明顯傳遞給 policy」，改爲傳遞 accumulation trait 並讓 policy 根據 traits 參數來決定其成果型別。）

透過 template template parameter 來存取 policy classes，主要的優點是，這使得 policy class 更容易攜帶若干狀態資訊（亦即 static 成員變數） — 只需利用一個取決於 template parameters 的型別即可。（在我們給的第一種作法中，static 成員變數不得不嵌於一個 member class template 內）

然而 template template parameter 作法亦有不利的一面，policy classes 如今必須被寫成 template，攜帶一組由我們的介面所定義的精確 template parameters。不幸的是這將造成 policies 不得帶有任何額外的 template parameters。舉個例子，我們可能希望爲 SumPolicy 添加一個 bool nontype template parameter，由它決定「求和動作」該使用 += 運算子或 + 運算子。我們只需將原本的 SumPolicy（譯註：p.256）修改如下即可：

```
// traits/sumpolicy3.hpp

#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP

template<bool use_compound_op = true>
class SumPolicy {
  public:
    template<typename T1, typename T2>
    static void accumulate (T1& total, T2 const & value) {
        total += value;
    }
};

template<>
class SumPolicy<false> {
  public:
    template<typename T1, typename T2>
    static void accumulate (T1& total, T2 const & value) {
        total = total + value;
    }
};

#endif // SUMPOLICY_HPP
```

但對於使用 template template parameters 的 Accum 實作品來說，這樣的修改就不再可能。

15.1.7 聯合多個 Policies 和/或 Traits

先前的開發顯示，traits 和 policies 並沒有完全消除「對多個 template parameters 的需求」，然而它們的確使參數數量減少了，使事情變得好管理了。接下來又出現一個有意思的問題：如何安排這些參數的順序呢？

一個簡單的策略是：根據其預設值被選用的可能性，以遞增順序安排參數順序。通常這意味 traits 參數跟在 policy 參數後面，因爲 policy 參數更常被客端程式碼覆寫。細心的讀者可能已經注意到了，我們的範例程式就是採用這種策略。

如果打算對程式碼添加大量複雜玩意兒，另一種作法是本質上允許以任何順序指定非預設引數（non-default arguments）。詳見 16.1 節, p.285。第 13 章也討論了將來可能出現、可簡化這方面設計的一些 templates 新性質。

15.1.8 以泛型迭代器（General Iterators）進行點計

結束這段對 traits 和 policies 的介紹前，看一個「加入泛型迭代器（非僅指標）處理能力」的 accum() 版本，應該頗具啓發意義。這種能力正是工業強度泛型組件所期望擁有的。有趣的是，這個版本仍然允許我們在呼叫 accum() 時使用指標，因爲 C++ 標準程式庫提供了所謂的 **iterator traits**（traits 真是無處不在！），於是我們可以將最初版本的 accum() 定義如下（不考慮後來對它的強化）：

```
// traits/accum0.hpp        //譯註：與 p.246 比較，以下灰色區爲差異處。

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include <iterator>

template <typename Iter>
inline
typename std::iterator_traits<Iter>::value_type
accum (Iter start, Iter end)
{
    typedef typename std::iterator_traits<Iter>::value_type VT;

    VT total = VT();  // assume VT() actually creates a zero value
    while (start != end) {
        total += *start;
        ++start;
    }
    return total;
}

#endif // ACCUM_HPP
```

iterator_traits 將迭代器的所有相關屬性封裝起來。由於它有一個「針對指標」的偏特化版本，所以這些 traits 可以很方便地和任何一般指標一塊兒被使用。下面展示標準程式庫實作品對於這種支援能力的可能實現手法：

```
namespace std {
   template <typename T>
   struct iterator_traits<T*> {
      typedef T                                value_type;
      typedef ptrdiff_t                        difference_type;
      typedef random_access_iterator_tag       iterator_category;
      typedef T*                               pointer;
      typedef T&                               reference;
   };
}
```

然而目前並不存在現成可用於「數值累計」型別可供迭代器指涉（*refers*），因此我們還是需要設計自己的 `AccumulationTraits`。

15.2 Type Functions（譯註：對比於 value function）

譯註：本節（含小節）較多保留了 types（型別）, values（數值）, functions（函式）等英文詞。

早先那個 traits 例子展示一個事實：你可以定義「因 types 而異」的行爲。這有別於你通常在程式中實作的東西。在 C 和 C++ 中，functions 可被更確切地稱爲 value functions：它們接受某些 values 做爲引數，並傳回一個 value 做爲結果。現在我們可以運用 templates 技術實作出所謂的 type functions：接受一些 types 作爲引數，並產生一個 type 或 constant 作爲結果。

`sizeof` 就是一個非常有用的內建 type function。它傳回一個常數，記錄特定引數（某個 type）的體積（以 bytes 爲單位）。class templates 也可以充當 type functions，此時的參數就是 template parameters，其結果則被萃取爲一個 member type 或 member constant。例如，運用 `sizeof` 運算子可製作出如下介面：

```
// traits/sizeof.cpp

#include <stddef.h>
#include <iostream>

template <typename T>
class TypeSize {
  public:
    static size_t const value = sizeof(T);
};

int main()
{
    std::cout << "TypeSize<int>::value = "
              << TypeSize<int>::value << std::endl;
}
```

稍後我們將開發一些更通用的 type functions，可根據上述方式被視爲 traits classes 使用。

15.2.1 決定元素型別（Elements Types）

看看另一個例子。假設我們有許多 container templates（容器模板）如 `vector<T>`、`list<T>`和 `stack<T>` 等等。我們希望有個 type function，在接受一個 container type（容器型別）之後可以輸出 element type（元素型別）。這可透過偏特化達成：

```
// traits/elementtype.cpp

#include <vector>
#include <list>
#include <stack>
#include <iostream>
#include <typeinfo>

template <typename T>
class ElementT;                          // 主模板（primary template）

template <typename T>
class ElementT<std::vector<T> > {        // 偏特化（partial specialization）
  public:
    typedef T Type;
};

template <typename T>
class ElementT<std::list<T> > {          // 偏特化（partial specialization）
  public:
    typedef T Type;
};

template <typename T>
class ElementT<std::stack<T> > {         // 偏特化（partial specialization）
  public:
    typedef T Type;
};

template <typename T>
void print_element_type (T const & c)
{
    std::cout << "Container of "
              << typeid(typename ElementT<T>::Type).name()
              << " elements.\n";
}
```

```
int main()
{
    std::stack<bool> s;
    print_element_type(s);
}
```

透過偏特化，我們得以實現這項功能而無需要求「container types 必須對 type function 有所了解」。然而在很多場合中，type function 往往和可施行於其上的 types 一塊設計，讓實作碼得以簡化。舉個例子，假設 container types 定義了一個 member type `value_type`（一如標準容器所爲），我們就可以編寫這樣的程式碼：

```
template <typename C>
class ElementT {
  public:
    typedef typename C::value_type Type;
};
```

這可以作爲一份預設實作品，它並不排斥那些「未曾定義 member type `value_type`」的任何 container types 特化體。儘管如此，爲 template type parameters 提供型別定義（typedef），使它們（因爲簡短或因爲別具命名意義）得以更容易在泛型程式碼中被取用，往往是明智之舉。下面程式碼勾勒出這種想法：

```
template <typename T1, typename T2, ... >
class X {
  public:
    typedef T1 ... ;
    typedef T2 ... ;
    ...
};
```

type function 的價值怎麼體現出來呢？它允許我們以 container type 來參數化某個 template，而且並不要求「必須一併提供 element type 的相關參數及其他特徵」。例如我們不再這麼寫：

```
template <typename T, typename C>
T sum_of_elements (C const& c);
```

因爲如果那麼寫，就需要以 `sum_of_elements<int>(list)` 之類的語法來明確指定 element type。我們可以改而這麼宣告：

```
template<typename C>
typename ElementT<C>::Type sum_of_elements (C const& c);
```

如此一來，element type 就可以由 type function 來決定了。

注意，traits 可被實作爲「對現有 types 的一個擴展」。因此你甚至可以定義 type functions 來處理基本型別（fundamental types）和封閉型程式庫中的型別（types of closed libraries）。

這種情況下 `ElementT` 被稱爲一個 traits class，因爲它被用來取得某給定之 container type `C` 的 trait（通常這樣的 class 會容納不止一個 trait）。因此 traits classes 並不僅僅侷限於描述容器參數特徵（characteristics of container parameters），還可以描述任何一種主參數（main parameters）。

15.2.2 確認是否為 Class Types

下面的 type function 可以協助我們判斷某個 type 是不是個 class type：

```
// traits/isclasst.hpp

template<typename T>
class IsClassT {
  private:
    typedef char One;
    typedef struct { char a[2]; } Two;
    template<typename C> static One test(int C::*);
    template<typename C> static Two test(...);
  public:
    enum { Yes = sizeof(IsClassT<T>::test<T>(0)) == 1 };
    enum { No = !Yes };
};
//譯註：此例在 VC6,VC7.1 無法通過。ICL7.1 可正確編譯並執行。
```

這個 template 使用 8.3.1 節, p.106 所說的 SFINAE（substitution-failure-is-not-an-error；替換失敗並非錯誤）原理。SFINAE 的運用關鍵是找到一個對 class types 無效但對其他 types 有效的型別構件（type construct）；反之亦可。對於 class types 我們可以倚賴這樣的觀察結論：只有當 `C` 是個 class type 時，pointer-to-member（像是 `int C::*`）這樣的型別構件才有效。

以下程式使用了這個 type function 來測試一些 types 和 objects 究竟是不是 class types：

```
// traits/isclasst.cpp

#include <iostream>
#include "isclasst.hpp"

class MyClass {
};
```

```
struct MyStruct {
};

union MyUnion {
};

void myfunc()
{
}

enum E { e1 } e;

// 把型別當做模板引數（template argument）傳進去檢驗
template <typename T>
void check()
{
    if (IsClassT<T>::Yes) {
        std::cout << " IsClassT " << std::endl;
    }
    else {
        std::cout << " !IsClassT " << std::endl;
    }
}

// 把型別當做函式呼叫引數（function call argument）傳進去檢驗
template <typename T>
void checkT (T)
{
    check<T>();
}

int main()
{
    std::cout << "int:     ";
    check<int>();

    std::cout << "MyClass: ";
    check<MyClass>();
```

```
    std::cout << "MyStruct:";
    MyStruct s;
    checkT(s);

    std::cout << "MyUnion: ";
    check<MyUnion>();

    std::cout << "enum:    ";
    checkT(e);

    std::cout << "myfunc():";
    checkT(myfunc);
}
```

程式輸出如下：

```
int:         !IsClassT
MyClass:     IsClassT
MyStruct:    IsClassT
MyUnion:     IsClassT
enum:        !IsClassT
myfunc():    !IsClassT
```

15.2.3 References（引用）和 Qualifiers（飾詞）

譯註：這裡的 reference 是指常與 pointer 相提並論的那個東西，qualifiers（飾詞）指的是 `const`, `volatile` 之類的屬性修飾關鍵字。

考慮下面的 function templates 定義：

```
// traits/apply1.hpp

template <typename T>
void apply (T& arg, void (*func)(T))
{
    func(arg);
}
```

再考慮以下程式碼（試圖使用上面那個 function templates）：

```
// traits/apply1.cpp

#include <iostream>
#include "apply1.hpp"
```

```
void incr (int& a)
{
    ++a;
}

void print (int a)
{
    std::cout << a << std::endl;
}

int main()
{
    int x = 7;
    apply (x, print);
    apply (x, incr);
}
//譯註：此例在 VC6,VC7.1,ICL7.1 上均無法編譯。
//      三個編譯器都未支持最新增補之 C++ Standard 規格。
```

程式中的呼叫動作：

```
apply (x, print)
```

沒問題，`T` 被 `int` 取代，`apply()`的參數型別分別是 `int&` 和 `void(*)(int)`，對應於所得的引數型別。但是下面這個呼叫動作：

```
apply (x, incr)
```

就不那麼簡單了。要想匹配第二參數，`T` 必須被替換爲 `int&`，這就意味第一參數的型別必須是 `int& &`，但這並不是個合法的 C++ type。的確，在原始 *C++ Standard* 中這種替換並不合法，但正因爲上述這一類例子，後來的一份技術勘誤（對 *C++ Standard* 的一些小修改，見 [Standard02]）指出：`T&` 中的 `T` 如果被替換爲 `int&`，那麼 `T&` 等價於 `int&`[64]。

面對尚未實現此一新式「reference 替代規則」的 C++ 編譯器，我們可以爲它建立一個 type function 充當「*reference* 運算子」，若且惟若（*if and only if*）type 不是個 reference。我們還可以提供相反的操作：將「*reference* 運算子」剝除，若且惟若討論中的 type 是個 reference。如果我們興致夠高，還可以添加或剝除 `const` 飾詞[65]。這些都可以使用以下泛型定義之偏特化版本達成：

[64] 請注意，我們仍然不能寫 `int& &`。這相當於這樣的事實：`T const` 中的 `T` 可被替換爲 `int const`，但如果你明確寫出 `int const const`，無效！。

[65] 爲求簡化，這兒刻意不談 `volatile` 和 `const volatile` 飾詞，它們事實上可被如法炮製。

```
// traits/typeop1.hpp

template <typename T>
class TypeOp {           // primary template（主模板）
  public:
    typedef T          ArgT;
    typedef T          BareT;
    typedef T const    ConstT;
    typedef T &        RefT;
    typedef T &        RefBareT;
    typedef T const &  RefConstT;
};
```

首先，以一個偏特化版本處理 const types：

```
// traits/typeop2.hpp

template <typename T>
class TypeOp <T const> {    // 針對 const types 而設計的偏特化
  public:
    typedef T const    ArgT;
    typedef T          BareT;
    typedef T const    ConstT;
    typedef T const &  RefT;
    typedef T &        RefBareT;
    typedef T const &  RefConstT;
};
```

這個用來處理 reference types 的偏特化版本也有能力處理 reference-to-const types。於是，必要時可遞迴執行 TypeOp 以取得 "bare type"（剝除各種飾詞之後的 type）。對比於此，即使某個 template parameter 被「本身已是 const」的某個 type 替換，C++ 也允許我們將 const 飾詞施行於該 template parameter 身上。因此無論我們怎麼使用 const，都無需操心是否需要剝除 const 飾詞。

```
// traits/typeop3.hpp

template <typename T>
class TypeOp <T&> {       // 針對 references 而設計的偏特化
  public:
    typedef T &                           ArgT;
    typedef typename TypeOp<T>::BareT     BareT;
    typedef T const                       constT;
```

```
    typedef T &                             RefT;
    typedef typename TypeOp<T>::BareT &     RefBareT;
    typedef T const &                       RefConstT;
};
```

注意，references-to-void types 是不被允許的。不過有時候能夠處理諸如「無任何飾詞的 void」這類 types 還是頗有用處。下面的偏特化版本便是用來處理此事：

```
// traits/typeop4.hpp

template<>
class TypeOp <void> {     // 針對 void 而設計的偏特化
  public:
    typedef void            ArgT;
    typedef void            BareT;
    typedef void const      ConstT;
    typedef void            RefT;
    typedef void            RefBareT;
    typedef void            RefConstT;
};
```

有了這些，我們就可以將 apply template 改寫如下：

```
template <typename T>       //譯註：請與 p.268 比較。灰色爲差異處。
void apply (typename TypeOp<T>::RefT arg, void (*func)(T))
{
    func(arg);
}
```

而範例程式（譯註：p.269）將如預想般運轉。

記住，現在的 T 不再可從第一引數推導而來，因爲它如今出現於一個名稱飾詞（name qualifier）之中。現在這個 T 只能從第二引數推導而來，而後才被用來建立第一參數的 type。

15.2.4 Promotion Traits（型別晉升之特徵萃取）

譯註：promotion 在此的意思是「晉升、晉級」，意指（例如）由 char 晉升爲 int，或由 float 晉升爲 double。

到目前爲止，我們已經研究和開發「針對單一 type」的 type functions：只要給定一個 type，其他相關的 types 或 constants 便即獲得定義。通常我們還可以開發「與多個引數相依」的 type functions。舉個例子，當我們編寫所謂 **promotion traits** 這樣的 operator templates 時，這就非常有用。爲了刺激這個構想，讓我們編寫一個可將兩個 Array 容器相加的 function template：

```
template<typename T>
Array<T> operator+ (Array<T> const&, Array<T> const&);
```

這很好，但由於語言允許我們將一個 char 和一個 int 相加，我們十分希望諸如此類的混合操作也能實施於 arrays 身上。這麼一來我們就必須決定新版本的回返型別（return type）應該是什麼：

```
template<typename T1, typename T2>
Array<???>
operator+ (Array<T1> const&, Array<T2> const&);
```

promotion traits template 使我們能夠以如下方式填充上述宣告中的問號：

```
template<typename T1, typename T2>
Array<typename Promotion<T1, T2>::ResultT>
operator+ (Array<T1> const&, Array<T2> const&);
```

或使用如下方式：

```
template<typename T1, typename T2>
typename Promotion<Array<T1>, Array<T2> >::ResultT
operator+ (Array<T1> const&, Array<T2> const&);
```

此種想法在於爲 template Promotion 提供大量特化版本，以創建出一個滿足我們所需的 type function。另一個應用由 max() template 激發出來：當我們希望指出型別不同的兩數值中的較大者時，應該採用「較大」的那個型別（見 2.3 節, p.13）。

這個 template 並不存在真正的泛型定義。最好的選擇是：別去定義 primary class template（主模板）：

```
template<typename T1, typename T2>
class Promotion;
```

另一個必須採取的措施是，如果某個 type 比另一個 type 大，我們應該提昇至較大的那個 type。這可利用一個專門的 template IfThenElse（其中帶有一個 bool nontype template parameter）對 type parmeters 進行二選一：

```
// traits/ifthenelse.hpp    （譯註：VC6 無法編譯，因不支援偏特化）

#ifndef IFTHENELSE_HPP
#define IFTHENELSE_HPP

// primary template：根據第一引數導出第二或第三引數
template<bool C, typename Ta, typename Tb>
class IfThenElse;
```

```
// 偏特化：true 導出第二引數
template<typename Ta, typename Tb>
class IfThenElse<true, Ta, Tb> {
  public:
    typedef Ta ResultT;
};

// 偏特化：false 導出第三引數
template<typename Ta, typename Tb>
class IfThenElse<false, Ta, Tb> {
  public:
    typedef Tb ResultT;
};

#endif // IFTHENELSE_HPP
```

有了這些東西，我們就可以根據需要提昇之 types 的體積大小，在 `T1`、`T2` 和 `void` 之間創造出一個「三向選擇」：

```
// traits/promote1.hpp

// type promotion 的主模板（primary template）
template<typename T1, typename T2>
class Promotion {
  public:
    typedef typename
          IfThenElse<(sizeof(T1)>sizeof(T2)),
                     T1,
                     typename IfThenElse<(sizeof(T1)<sizeof(T2)),
                                         T2,
                                         void
                                        >::ResultT
                    >::ResultT ResultT;
};
```

primary template 所使用的「以大小爲根據的探索法」（size-based heuristic）有時可以有效運作，但需要檢驗。如果它選擇了一個錯誤 type，那麼必須有一個適當的特化版本用來推翻這個選擇。從另一方面說，如果兩個 types 完全一樣，我們可以安全地令它成爲 promoted type。下面的偏特化版本考慮了這一點：

```
// traits/promote2.hpp

// 針對「兩個 types 完全相同」而設計的偏特化版本。
template<typename T>
class Promotion<T,T> {
  public:
    typedef T ResultT;
};
```

我們需要很多特化版本來記錄語言基本（內建）型別的晉級（promotion）原則。以下巨集（macro）多少可以減少一些源碼數量：

```
// traits/promote3.hpp

#define MK_PROMOTION(T1,T2,Tr)                \
    template<> class Promotion<T1, T2> {      \
      public:                                 \
        typedef Tr ResultT;                   \
    };                                        \
                                              \
    template<> class Promotion<T2, T1> {      \
      public:                                 \
        typedef Tr ResultT;                   \
    };
```

這麼一來，就可以這樣加入各種晉級規則：

```
// traits/promote4.hpp

MK_PROMOTION(bool, char, int)
MK_PROMOTION(bool, unsigned char, int)
MK_PROMOTION(bool, signed char, int)
//...
```

這種方式相當直截了當，卻需要列舉數十個可能組合。各式各樣的替代方案也都可能存在，例如可以修改 `IsFundaT` template 和 `IsEnumT` template 來爲整數（integral）型別和浮點數型別定義 promotion type。這麼一來 `Promotion` 就只需針對「成果基礎型別」（resulting fundamental types）和用戶自定型別（user-defined types）完成特化即可。關於後者，你馬上就會看到。

一旦爲基礎（內建）型別（必要的話再加上 enum 型別）定義好 **Promotion**，其他晉級規則通常可以利用偏特化來表達。以先前的 `Array` 爲例：

```
// traits/promotearray.hpp

template<typename T1, typename T2>
class Promotion<Array<T1>, Array<T2> > {
  public:
    typedef Array<typename Promotion<T1,T2>::ResultT> ResultT;
};

template<typename T>
class Promotion<Array<T>, Array<T> > {
  public:
    typedef Array<typename Promotion<T,T>::ResultT> ResultT;
};
```

最後這個偏特化版本有若干特別值得注意的地方。乍見之下，好像先前針對型別一致（identical types）而設計的偏特化版本（`Promotion<T,T>`）已經處理了這種情況。不幸的是與偏特化版本`Promotion<T,T>`相比，偏特化版本 `Promotion<Array<T1>, Array<T2> >` 的偏特化工作既沒多做也沒少做（見 12.4 節, p.200）[66]。爲避免選擇 templates 時發生模棱兩可（歧義）情況，最後那個偏特化版本被加了進來，它比前兩個偏特化版本做了更多偏特化工作。

當更多 types 被加入時，爲了使 promotion（型別晉升）有意義，我們需要爲 `Promotion` template 加入更多的特化版本和偏特化版本。

15.3 Policy Traits

截至目前，我們所給的 traits templates 實例都用以決定 template parameters 的屬性（properties），例如它們表示哪一種 type？在混合型別操作（mixed-type operations）中應被提昇爲哪一種 type？這樣的 traits 被稱爲 property traits。

另有一些 traits 負責定義某些 types 應該如何被處置，我們稱此爲 policy traits。這使我們聯想先前討論過的 policy classes 概念（我們也早已指出，traits 和 policies 之間的區別並非絕對地涇渭分明），但 policy traits 傾向於「和 template parameter 之間」有更多彼此關聯的獨特屬性，而 policy classes 通常和其他 template parameters 保持獨立。

Property traits 往往以 type functions 實現，policy traits 則通常將 policy 封裝於成員函式。讓我們看第一個例子，那是一個 type function，定義了一個 policy 用以傳遞惟讀參數（read-only parameters）。

[66] 爲了弄明白這一點，請試著設法找到 `T` 的某個替代物，使後者變成前者，或者爲 `T1, T2` 分別找到替代物，使前者變成後者。

15.3.1 唯讀的參數型別（Read-only Parameter Types）

在 C 和 C++ 中，函式的 call arguments 預設採用 *by value*（傳值）方式來傳遞，這意味「呼叫者手上的引數值」被拷貝到「被呼叫者所控制的位置」。大多數程式員都知道此舉對大型結構而言代價高昂 — 大型結構更適合採用 *by reference-to-const*（C 語言則是 *by pointer-to-const*）來傳遞引數。不過，對於較小結構來說，整個分際並不總是那麼清晰，而且從效率觀點來看，最佳機制取決於程式碼所處的實際架構（exact architecture）。雖然這件事情在大多數情況下並非多麼關鍵，但有時候即使面對小型結構也必須謹慎處理。

面對 templates，事情變得更爲棘手。我們不知道將被用來替換 template parameters 的那個 type 到底有多大。此外，最終決定也不僅僅取決於大小：一個伴有「代價高昂之 *copy* 建構式」的小型結構可能會讓你醒悟：以 *by reference-to-const* 傳遞惟讀參數還是比較有道理的。

正如先前所暗示，使用一個「身爲 type function」的 policy traits template，可以相當方便地處理這個問題。此一 type function 將一個預期的引數型別 `T` 映射到一個最佳參數型別 `T` 或 `T const&`。初步規劃時，你可以在 primary template 中以 *by value* 方式傳遞「尺碼不大於兩個指標」的 types，並以 *by reference-to-const* 方式傳遞其他所有東西：

```
template<typename T>
class RParam {
  public:
    typedef typename IfThenElse<sizeof(T)<=2*sizeof(void*),
                                T,
                                T const&>::ResultT Type;
};
//譯註：定義一個 Type 型別，若 T 尺碼大於兩個指標大小，Type 就是 T，否則就是 T const&
```

另一方面，對於某些容器型別，雖然其 `sizeof` 傳回值可能並不大（譯註：因爲 `sizeof` 傳回的只是容器本身狀態，不含元素），但可能帶有代價高昂之 *copy* 建構式，所以我們需要許多像下面這樣的特化和偏特化版本：

```
template<typename T>
class RParam<Array<T> > {
  public:
    typedef Array<T> const& Type;
};
//譯註：對 Array<T> 而言，Type 將是 Array<T> const&
```

這一類 types 在 C++ 中司空見慣，爲求安全最好是在 primary template 中將 nonclass types 標示爲「以 *by value* 方式傳遞」，然後在需要更佳效率表現時選擇性地加入 class types（primary template 可使用 p.266 所示的 `IsClassT<>` 來鑑定 class types）：

```
// traits/rparam.hpp

#ifndef RPARAM_HPP
#define RPARAM_HPP

#include "ifthenelse.hpp"
#include "isclasst.hpp"

template<typename T>
class RParam {
  public:
    typedef typename IfThenElse<IsClassT<T>::No,
                                T,
                                T const&>::ResultT Type;
};

#endif // RPARAM_HPP
```

不論哪一種方式，policy 現在可集中於 traits template 定義式中，客端程式可用它達到良好的效果。舉個例子，假設我們有兩個 classes，其中之一具體指明「惟讀引數較適合以 *call by value* 傳遞」：

```
// traits/rparamcls.hpp

#include <iostream>
#include "rparam.hpp"

class MyClass1 {
  public:
    MyClass1 () {
    }
    MyClass1 (MyClass1 const&) {
        std::cout << "MyClass1 copy constructor called\n";
    }
};

class MyClass2 {
  public:
    MyClass2 () {
    }
    MyClass2 (MyClass2 const&) {
        std::cout << "MyClass2 copy constructor called\n";
    }
};
```

```
// 運用 RParam<>告訴大家：MyClass2 objects 將以 by value 方式傳遞
template<>
class RParam<MyClass2> {
  public:
    typedef MyClass2 Type;
};
```

現在你可以宣告一些函式，針對惟讀引數使用 RParam<>，然後試著呼叫那些函式：

```
// traits/rparam1.cpp

#include "rparam.hpp"
#include "rparamcls.hpp"

// 以下函式允許參數以 by value 或 by reference 方式傳遞
template <typename T1, typename T2>
void foo (typename RParam<T1>::Type p1,
          typename RParam<T2>::Type p2)
{
    //...
}

int main()
{
    MyClass1 mc1;
    MyClass2 mc2;
    foo<MyClass1,MyClass2>(mc1,mc2); //譯註：注意，如下所說，無法自動引數推導。
}
```

不幸的是，使用 RParam 會出現一些重大缺點。首先函式的宣告明顯較爲凌亂。更讓人討厭的是：像上述 foo() 這樣的函式，無法經由引數推導（argument deduction）被喚起，因爲 template parameters 只出現在函式參數的飾詞（qualifiers）上，因此呼叫端必須明確指定 templates arguments。

一個笨拙而龐大的解決辦法是：使用 inline wrapper function template，而我們假設這個 inline function 會被編譯器的優化機制除掉（譯註：之所以說「假設」，乃因 inline 成功與否完全由編譯器決定）。例如：

```
// traits/rparam2.cpp

#include "rparam.hpp"
#include "rparamcls.hpp"
```

```
// 以下函式允許參數以 by value 或 by reference 方式傳遞
template <typename T1, typename T2>
void foo_core (typename RParam<T1>::Type p1,
               typename RParam<T2>::Type p2)
{
   //...
}

// 下面是個 wrapper，用以免除客戶端「明確（顯式）指定 template parameter」的責任
template <typename T1, typename T2>
inline
void foo (T1 const & p1, T2 const & p2)
{
   foo_core<T1,T2>(p1,p2);
}

int main()
{
   MyClass1 mc1;
   MyClass2 mc2;
   foo(mc1,mc2);  // 此式效果相當於 foo_core<MyClass1,MyClass2>(mc1,mc2)
}
```

15.3.2 拷貝（Copying）、置換（Swapping）和搬移（Moving）

讓我們繼續效率方面的話題。本節介紹一個 policy traits template，用在 (1) 拷貝（copying）或 (2) 置換（swapping）(3) 搬移（moving）元素時，根據其型別選擇最佳操作機制。

通常我們會認爲，「拷貝操作」以 *copy* 建構式和 *copy-assignment* 運算子處理即可。這對單一元素來說無疑是正確的。但也可能存在這樣的事實：拷貝某類型的大量元素時，存在一種比「不斷重複呼叫該型別之 *copy* 建構式或 *copy-assignment* 運算子」更顯著高效的作法。

同樣道理，對置換（swapping）或搬移（moving）而言，某些類型的元素可以採用比以下傳統操作更高效的作法：

```
T tmp(a);
a = b;
b = tmp;
```

容器型別（container types）通常便是屬於此類。

有時候拷貝是不被允許的，置換或搬移則沒有問題。第 20 章談到 Utilities（工具程式）時，我們開發了一個所謂的 smart pointer（靈巧指標），就具有這個屬性。

因此，將此領域的決策集中於一個便利的 traits template 會很有用。針對泛型定義，我們區分 class types 和 nonclass types 兩大類，面對後者我們無需掛心「用戶自定的 *copy* 建構式和 *copy assignment* 運算子」。這次我們使用繼承機制，在兩個 traits 實作品之間進行選擇。

```
// traits/csmtraits.hpp

template <typename T>
class CSMtraits : public BitOrClassCSM<T, IsClassT<T>::No > {
};
```

這麼一來，實作任務就完全委託給 `BitOrClassCSM<>`的特化版本（"CSM" 代表 "copy"、"swap"、"move"）。第二個 template parameters 用以指示「位元逐一拷貝」（bitwise copying）能否安全實作出不同種類的操作。這個泛型定義保守地假設 class types 不能被安全地「位元逐一拷貝」，但如果已知某特定的 class type 是個 POD（plain old data）type，那就很容易將 `CSMtraits` class 特化以獲得更佳效率：

```
template<>
class CSMtraits<MyPODType>
 : public BitOrClassCSM<MyPODType, true> {
};
```

預設情況下 `BitOrClassCSM` template 包含兩個偏特化版本。primary template 以及「不進行位元逐一拷貝」的偏特化安全版本展示如下：

```
// traits/csm1.hpp

#include <new>
#include <cassert>
#include <stddef.h>
#include "rparam.hpp"

// primary template
template<typename T, bool Bitwise>
class BitOrClassCSM;

// 偏特化版本，用以對物件進行安全拷貝（safe copying）
template<typename T>
class BitOrClassCSM<T, false> {
  public:
    static void copy (typename RParam<T>::ResultT src, T* dst) {
        // 將一份資料拷貝（copy）到另一份身上
        *dst = src;
```

```
}

static void copy_n (T const* src, T* dst, size_t n) {
    // 將 n 份資料拷貝（copy）到另外的 n 份資料身上
    for (size_t k = 0; k<n; ++k) {
        dst[k] = src[k];
    }
}

static void copy_init (typename RParam<T>::ResultT src,
                       void* dst) {
    // 將一份資料拷貝（copy）到未初始化的儲存空間上
    ::new(dst) T(src);
}

static void copy_init_n (T const* src, void* dst, size_t n) {
    // 將 n 份資料拷貝（copy）到未初始化的儲存空間上
    for (size_t k = 0; k<n; ++k) {
        ::new((void*)((char*)dst+k)) T(src[k]);
    }
}

static void swap (T* a, T* b) {
    // 置換（swap, 對調）兩份資料
    T tmp(*a);
    *a = *b;
    *b = tmp;
}

static void swap_n (T* a, T* b, size_t n) {
    // 置換（swap, 對調）n 份資料
    for (size_t k = 0; k<n; ++k) {
        T tmp(a[k]);
        a[k] = b[k];
        b[k] = tmp;
    }
}
```

```
    static void move (T* src, T* dst) {
        // 搬移（move）一份資料到另一份身上
        assert(src != dst);
        *dst = *src;
        src->~T();
    }

    static void move_n (T* src, T* dst, size_t n) {
        // 搬移（move）n 份資料到另 n 份身上
        assert(src != dst);
        for (size_t k = 0; k<n; ++k) {
            dst[k] = src[k];
            src[k].~T();
        }
    }

    static void move_init (T* src, void* dst) {
        // 搬移（move）一份資料到未初始化儲存空間上
        assert(src != dst);
        ::new(dst) T(*src);
        src->~T();
    }

    static void move_init_n (T const* src, void* dst, size_t n) {
        // 搬移（move）n 份資料到未初始化儲存空間上
        assert(src != dst);
        for (size_t k = 0; k<n; ++k) {
            ::new((void*)((char*)dst+k)) T(src[k]);
            src[k].~T();
        }
    }
};
```

術語 *move*（搬移）在這兒的意思是「一個 value 從某地被移轉（*transferred*）到另一地」；原 value 不復存在（或更準確地說，原位置可能被銷毀）。至於 *copy*（拷貝）則是保證操作後的來源位置和目標位置同時存在有效且相同的 value。請不要和標準 C 程式庫中的 `memcpy()` 和 `memmove()` 搞混淆了 — 在那個情況下 *move* 意味來源區和目標區有可能重疊，*copy* 則否。我們的 CSM traits

總是假設來源區和目標區不重疊。在一個工業強度程式庫中，或許還應該加入 *shift*（平移）操作，用以表達「在一塊連續記憶區內移動物件」的 policy（也就是 `memmove()`所支援的那種操作）。但是爲求簡化，我們節略了這一點。

我們的 policy traits template 成員函式都是 static。任何情況下幾乎總是如此，因爲成員函式意圖施行於「隸屬參數型別」的物件上，而非「隸屬 traits class type」的物件上。

另一個偏特化版本令此 traits 作用於「得以進行位元逐一拷貝」的型別身上：

```
// traits/csm2.hpp

#include <cstring>
#include <cassert>
#include <stddef.h>
#include "csm1.hpp"

// 偏特化版本，針對 fast bitwise copying 撰寫
template <typename T>
class BitOrClassCSM<T,true> : public BitOrClassCSM<T,false> {
  public:
    static void copy_n (T const* src, T* dst, size_t n) {
        // 將 n 份資料拷貝（copy）到另外的 n 份資料身上
        std::memcpy((void*)dst, (void*)src, n);
    }

    static void copy_init_n (T const* src, void* dst, size_t n) {
        // 將 n 份資料拷貝（copy）到未初始化儲存空間上
        std::memcpy(dst, (void*)src, n);
    }

    static void move_n (T* src, T* dst, size_t n) {
        // 搬移（move）n 份資料到另 n 份身上
        assert(src != dst);
        std::memcpy((void*)dst, (void*)src, n);
    }

    static void move_init_n (T const* src, void* dst, size_t n) {
        // 搬移（move）n 份資料到未初始化儲存空間上
        assert(src != dst);
        std::memcpy(dst, (void*)src, n);
    }
};
```

以上多用了一層繼承，用以簡化此類 traits 的實作。這些當然不是惟一可能的作法，事實上某些特定平台（例如）可能希望引入 inline 組合語言碼（以便運用硬體優勢）。

15.4 後記

Nathan Myers 是第一個將 traits parameters 觀念正式化的人。他最初將此想法提呈給 C++ 標準委員會時，是將它當作「在標準程式庫組件（如輸入資料流和輸出資料流; I/O streams）中如何處理字元型別」的一種定義工具。當時他稱之爲 baggage templates，並註明它們相當於 traits。然而有些 C++ 委員會成員不喜歡 baggage 這個術語，反而提倡以 traits 爲名。從那時起，術語 traits 便獲得了廣泛的使用。

客端程式碼通常根本無需處理 traits，因爲預設的 traits classes 可以滿足絕大多數常見需求，而且由於它們是預設的 template arguments，它們根本無需出現在客端程式碼之中。這將有利於爲 default traits templates 提供較長的描述性名稱。當客端程式碼透過「提供一個訂製的 traits 引數」來修改 template 行爲時，將 resulting specializations（最後所得的特化體）以 `typedef` 定出一個「和訂製行爲相稱」的名稱，是個好習慣。這種情況下我們可以賦予 traits class 一個較長的、而且不會把源碼搞得很難看的描述性名稱。

我們的討論似乎顯示 traits templates 非得是 class templates 不可。事實並非如此。如果只需提供單一 policy trait，其實可使用 function template 傳入。例如：

```
template <typename T, void (*Policy)(T const&, T const&)>
class X;
```

不過，traits 的最初目標就是爲了減輕 secondary template arguments（次要模板引數）的包袱。如果一個 template parameter 內只封裝一個 trait，就達不到效果了。這足以說明 Myers 當初對術語 baggage 的偏愛是有道理的。第 22 章提供排序準則（ordering criterion）時我們將重啓這個問題。

標準程式庫定義了一個 class template `std::char_traits`，它被用作 policy traits parameter。此外爲了讓演算法很容易適應於它所獲得的 STL iterators，標準程式庫提供了一個非常簡明的 `std::iterator_traits` property traits template（並被用於標準程式庫介面）。template `std::numeric_limits` 也可被拿來當作一個 property traits template，但在標準程式庫中它顯然沒有得到適當地運用。class template `std::unary_function` 和 `std::binary_function` 歸屬於相同分類，都是非常簡單的 type functions：僅僅只是將引數重新定義（typedef）爲成員名稱，從而使這些名稱對 functors（或稱 function objects，見 22 章）有意義。最後要說的是，標準容器的記憶體配置問題是以一個 policy traits class 來處理的，而 template `std::allocator` 正是標準程式庫爲此目的而提供的一個標準組件。

> 譯註：如果不曾研究過 STL 源碼，幾乎可以斷定無法理解上述文字，因爲上述文字所描述的組件都是底層的、不與客戶端接觸的。任何 C++ 編譯器均附帶 STL 源碼，市面上也有 STL 源碼剖析相關書籍，建議一觀。

很多程式員和不少書籍作者都開發了 policy classes 技術，Andrei Alexandrescu 更使得術語 policy classes 膾炙人口。相比於本書簡短的一節來說，Andrei 的著作《*Modern C++ Design*》更加細緻地探討了這個主題（見 [AlexandrescuDesign]）。

C

參考書目和資源
Bibliography

附錄 C 列出書中提及、採納和引用的資源。如今，編程領域中的很多發展和進步都發生於電子論壇，因此毫不令人驚訝的是，我們不但可以找出許多書籍和文章，也可以找出許多相關的 Web 網站。我們當然不至於宣稱我們所給的清單近乎全面，不過我們的確認爲它們對 C++ templates 有著重大的貢獻。

Web 網址通常比書籍和文章更易有所變化。這兒列出的 Internet 連接點將來可能無效。因此我們在以下網址（但願它夠穩定）爲本書列出實際的連接清單：

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

列出書籍、文章和 Web 網址之前，讓我們先介紹由所謂新聞群組（newsgroups）提供的更具互動性的資源。

C.1 新聞群組（Newsgroups）

Usenet 是一個規模巨大、種類互異的電子論壇（通常稱爲 newsgroups）的總集合。其中一些 newsgroups 有主持人，這意味被提交的每一個帖子都經過某種形式的檢查，以保證其恰當性。

有一些 Usenet 群組專注於 C++ 語言的討論。事實上本書展示的許多最高級技術，首先就是發表於其中某些群組。有時候技術便是透過這些群組的合作討論而開發出來。

下面這些 Usenet newsgroups 專注討論 C++、*C++ Standard* 和 C++ 標準程式庫：

- C++ 細部指導（Tutorial level）（無主持人） `alt.comp.lang.learn.c-c++`
- C++ 一般面向（無主持人） `comp.lang.c++`
- C++ 一般面向（有主持人） `comp.lang.c++.moderated`
- *C++ Standard* 相關面向（有主持人） `comp.std.c++`

假如你不曾造訪 Usenet newsgroups server，可使用 Google Usenet archive 協助你：

```
http://groups.google.com
```

C.2 書籍和 Web 網站

[AlexandrescuDesign]
Andrei Alexandrescu
Modern C++ Design 《C++ 設計新思維》,侯捷/於春景合譯, 碁峰 2003
Generic Programming and Design Patterns Applied
Addison-Wesley, Reading, MA, 2001

[AusternSTL]
Matthew H. Austern
Generic Programming and the STL 《泛型程式設計與 *STL*》,侯捷/黃俊堯合譯, 碁峰 2000
Using and Extending the C++ Standard Template Library
Addison-Wesley, Reading, MA, 1999

[BCCL]
Jeremy Siek
The Boost Concept Check Library
`http://www.boost.org/libs/concept_check/concept_check.htm`

[Blitz++]
Todd Veldhuizen
Blitz++: Object-Oriented Scientific Computing
`http://www.oonumerics.org/blitz`

[Boost]
The Boost Repository for Free, Peer-Reviewed C++ Libraries
`http://www.boost.org`

[BoostCompose]
Boost Compose Library
`http://www.boost.org/libs/compose`

[BoostSmartPtr]
Smart Pointer Library
`http://www.boost.org/libs/smart_ptr`

[BoostTypeTraits]
Type Traits Library
`http://www.boost.org/libs/type_traits`

[CargillExceptionSafety]
Tom Cargill
Exception Handling: A False Sense of Security
Available at: http://www.awprofessional.com/meyerscddemo/demo/magazine/index.htm
C++ Report, November-December 1994

[CoplienCRTP]
James O. Coplien
Curiously Recurring Template Patterns
C++ Report, February 1995

[CoreIssue115]
Core Issue 115 of the C++ Standard
`http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg_toc.html`

[CzarneckiEisenecker GenProg]
Krzysztof Czarnecki, Ulrich W. Eisenecker
Generative Programming
Methods, Tools, and Applications
Addison-Wesley, Reading, MA, 2000

[DesignPatternsGoV]
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns 《物件導向設計模式》,葉秉哲譯, 培生 2001
Elements of Reusable Object-Oriented Software
Addison-Wesley, Reading, MA, 1995

[EDG]
Edison Design Group
Compiler Front Ends for the OEM Market
`http://www.edg.com`

[EllisStroustrupARM]
Margaret A. Ellis, Bjarne Stroustrup
The Annotated C++ Reference Manual (ARM)
Addison-Wesley, Reading, MA, 1990

[JosuttisAutoPtr]
Nicolai M. Josuttis
auto_ptr and auto_ptr_ref
`http://www.josuttis.com/libbook/auto_ptr.html`

[JosuttisOOP]
Nicolai M. Josuttis
Object-Oriented Programming in C++
John Wiley and Sons Ltd, 2002

[JosuttisStdLib]
Nicolai M. Josuttis
The C++ Standard Library 《C++ 標準程式庫》,侯捷/孟岩合譯, 碁峰 2002
A Tutorial and Reference
Addison-Wesley, Reading, MA, 1999

[KoenigMooAcc]
Andrew Koenig, Barbara E. Moo
Accelerated C++
Practical Programming by Example
Addison-Wesley, Reading, MA, 2000

[LambdaLib]
Jaakko J··arvi, Gary Powell
LL, The Lambda Library
`http://www.boost.org/libs/lambda/doc`

[LippmanObjMod]
Stanley B. Lippman
Inside the C++ Object Model 《深度探索 C++ 物件模型》,侯捷譯, 碁峰 1998
Addison-Wesley, Reading, MA, 1996

[MeyersCounting]
Scott Meyers
Counting Objects In C++
C/C++ Users Journal, April 1998

[MeyersEffective]
Scott Meyers
Effective C++ 《*Effective C++* 中文版》,侯捷譯, 培生 2000
50 Specific Ways to Improve Your Programs and Design (2nd Edition)
Addison-Wesley, Reading, MA, 1998

[MeyersMoreEffective]
Scott Meyers
More Effective C++ 《*More Effective C++* 中文版》,侯捷譯, 培生 2000
35 New Ways to Improve Your Programs and Designs
Addison-Wesley, Reading, MA, 1996

[MTL]
Andrew Lumsdaine, Jeremy Siek
MTL, The Matrix Template Library
`http://www.osl.iu.edu/research/mtl`

[MusserWangDynaVeri]
D. R. Musser, C. Wang
Dynamic Verification of C++ Generic Algorithms
IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997

[MyersTraits]
Nathan C. Myers
Traits: A New and Useful Template Technique
`http://www.cantrip.org/traits.html`

[NewMat]
Robert Davies
NewMat10, A Matrix Library in C++
`http://www.robertnz.com/nm_intro.htm`

[NewShorterOED]
Leslie Brown, et al.
The New Shorter Oxford English Dictionary (fourth edition)
Oxford University Press, Oxford, 1993

[POOMA]
POOMA: A High-Performance C++ Toolkit for Parallel Scientific Computation
`http://www.pooma.com`

[Standard98]
ISO
Information Technology-Programming Languages-C++
Document Number ISO/IEC 14882-1998
ISO/IEC, 1998

[Standard02]
ISO
Information Technology-Programming Languages-C++
(as amended by the first technical corrigendum)
Document Number ISO/IEC 14882-2002
ISO/IEC, expected late 2002

[StroustrupC++PL]
Bjarne Stroustrup
The C++ Programming Language, Special ed. 《C++ 程式語言經典本》,葉秉哲譯, 儒林
Addison-Wesley, Reading, MA, 2000

[StroustrupDnE]
Bjarne Stroustrup
The Design and Evolution of C++
Addison-Wesley, Reading, MA, 1994

[StroustrupGlossary]
Bjarne Stroustrup
Bjarne Stroustrup's C++ Glossary
`http://www.research.att.com/~bs/glossary.html`

[SutterExceptional]
Herb Sutter
Exceptional C++ 《*Exceptional C++* 中文版》,侯捷譯, 培生 2000
47 Engineering Puzzles, Programming Problems, and Solutions
Addison-Wesley, Reading, MA, 2000

[SutterMoreExceptional]
Herb Sutter
More Exceptional C++
40 New Engineering Puzzles, Programming Problems, and Solutions
Addison-Wesley, Reading, MA, 2001

[UnruhPrimeOrig]
Erwin Unruh
Original Metaprogram for Prime Number Computation
`http://www.erwin-unruh.de/primorig.html`

[VandevoordeSolutions]
David Vandevoorde
C++ Solutions
Addison-Wesley, Reading, MA, 1998

[VeldhuizenMeta95]
Todd Veldhuizen
Using C++ Template Metaprograms
C++ Report, May 1995

[VeldhuizenPapers]
Todd Veldhuizen
Todd Veldhuizen's Papers and Articles about Generic Programming and Templates
`http://osl.iu.edu/~tveldhui/papers`

D

詞彙/術語表

Glossary

這份詞彙/術語表是本書所論主題之最重要術語的彙編。[StroustrupGlossary] 也提供了一份供 C++ 程式員使用的全面術語表。

abstract class（抽象類別）
一種無法據以創建（*create*; 產生/建立/創造）具象物件（實體）的 class。這種 classes 適合將不同 classes 的共通屬性集結於單一型別之中，亦適合定義多型介面（polymorphic interface）。由於這種 classes 被當做 base classes 使用，所以有時候會出現縮寫字 ABC，代表 abstract base class（抽象基礎類別）。

ADL
argument-dependent lookup（相依於引數的查詢）縮寫。所謂 ADL 是這樣一個過程：在 namespaces 和 classes 內查詢某個函式或運算子的名稱。這些名稱出現於呼叫式中，而接受查詢的 namespaces 和 classes 則是以某種方式與函式呼叫引數相關聯。由於歷史因素，這個術語有時被稱爲 extended Koenig lookup 或乾脆簡稱爲 Koenig lookup（後者也用於「只針對運算子的 ADL」）。

angle bracket hack
一種非標準特性，允許編譯器接受兩個連續的 `>` 字元作爲兩個右角括號（它們之間原本應該插入至少一個空格）。例如算式 `vector<list<int>>` 原本不合法，經由 angle bracket hack 的處理，它被視同合法的 `vector<list<int> >`。

angle brackets（角括號）
字元 `<` 和 `>` 被用於 template 引數列或參數列的邊界符號時，即稱爲角括號。（譯註：而不是指「大於」或「小於」符號）

ANSI
美國國家標準學會（American National Standard Institute）的縮寫。這是一個非官方、非盈利組織，努力生產各類標準規格。其中一個名爲 J16 的小組委員會是 C++ 標準化的背後驅動力。ANSI 和國際標準組織（international standards organization，ISO）緊密合作。

argument（引數）
一個用以替換「程式性物體（programmatic entity）之參數」的（廣義）值。例如在呼叫式 `abs(-3)` 中，`-3` 便是引數。某些編程社群將引數稱爲實參（actual parameters），而將參數（parameters）稱爲形參（formal parameters）。

argument-dependent lookup（相依於引數的查詢）
見 ADL。

class（類別）
對某種類型之物件的描述。class 爲物件（*objects*）定義了一套特徵，包括資料（或稱屬性 *property*、成員變數 *data members*）和操作（*operations*；或稱方法 *methods* 或成員函式 *member functions*）。在 C++ 中 classes 是帶有成員的一種結構，其成員可以是函式，並服從某種取用限制。它們以關鍵字 `class` 或 `struct` 進行宣告。

class template（類別模板）
一種構件，用來表示「一整群類別族系」（a family of classes）。它規定某種樣式（pattern），只要以特定物體（entities）替換其 template 參數，便可從中生成真正的 class。Class template 有時被稱爲參數化類別（"parameterized" classes），儘管後者具有更一般化的含義。

class type（類別型別）
任何以關鍵字 `class`、`struct` 或 `union` 進行宣告的 C++ 型別。

collection class（群集類別）
用於管理「一組物件」的 class。在 C++ 中群集類別也稱爲容器（containers）。

constant-expression（常數-算式）
計算值（value）由編譯器在編譯期計算而得。有時被稱爲真常數（true constant），以避免和常數算式（constant expression；中間無連字號）混淆。後者包括「不能由編譯器在編譯期計算結果」的常數算式。

`const` member function（const 成員函式）
只可由常數物件和暫時物件呼叫的成員函式，它通常並不修改 `*this` 物件的成員。

container（容器）
參見 collection class（群集類別）

conversion operator（轉換運算子；轉型運算子）
一種特殊的成員函式，它定義物件如何被隱式（或顯式）轉換爲另一種型別。其宣告型式爲 `operator` *`type`*`()`。

CRTP
curiously recurring template pattern（奇特遞迴模板範式）的縮寫。指的是這樣一種程式寫法：class `X` 衍生自一個「以 `X` 爲 template 引數」的 base class。

curiously recurring template pattern（奇特遞迴模板範式）
參見 CRTP。

decay（退化）
從一個 array 或 function 隱式轉換至一個 pointer。例如字串字面值 "Hello" 的型別爲 `char const[6]`，但在很多 C++ 情境中它被隱式轉換爲一個 `char const*` 指標（指向該字串的首字元）。

declaration（宣告）
一種 C++ 構件，用以將某個名稱引入（或重新引入）C++ 作用域（scope）。參見 definition。

deduction（推導）
從 template 被使用的情境中「暗自（逕自）決定 template argument」的過程。完整的術語是 template argument deduction（模板引數推導）。

definition（定義）
一種 declaration，使已宣告物體（declared entity）的細節曝光。對變數而言，會迫使保留（獲得）其儲存空間。對 class 和 function 而言，是指其 declarations 帶有「以大括號圍起來的本體（body）」。對外部變數（external variable）而言，意味一個不帶 extern 關鍵字的 declaration，或是一個具有初值設定器（initializer）的 declaration。

dependent base class（受控基礎類別）
「與某個 template parameter 相依」的 base class。程式中存取此種 class 的成員時要特別小心。參見 two-phase lookup（兩段式查詢）。

dependent name（受控名稱）
「含義取決於某 template parameter」的識別名稱。例如，當 `A` 或 `T` 是 template parameter 時，`A<T>::x` 就是個 dependent name。如果呼叫式中某個引數的型別取決於某個 template parameter，那麼該呼叫式中的函式名稱也是 dependent。例如在呼叫式 `f((T*)0)` 中，如果 `T` 是個 template parameter，`f` 就是 dependent。然而 template parameter 名稱並不被認爲是 dependent。參見 two-phase lookup（兩段式查詢）。

digraph（雙字元語彙單元）
兩個連續字元的組合體，等價於 C++ 程式碼中另一個單字元。digraphs 的用途在於允許「缺少某些字元的鍵盤」得以輸入 C++ 源碼。當一個左角括號（`<`）後跟一個 *scope resolution* 運算子（`::`）而沒有插入必要空格時，就會意外形成雙字元語彙單元 `<:` — 儘管相對來說這很少被用到。

dot-C file（dot-C 檔案）
變數和 noninline 函式定義的座落檔案。程式的大多數「可執行的」（而非只是「宣告的」; declarative）程式碼正常來說都應該被放進 dot-C 檔案。它們之所以被命名爲 dot-C 檔案，因爲通常以 .cpp, .C, .c, .cc 或 .cxx 爲副檔名。參見 header file（表頭檔）和 translation unit（編譯單元）。

EBCO
empty base class optimization（空基礎類別優化）的縮寫。大多數現代編譯器都實作有此優化機制。有了它，一個 "empty" base class subobject（「空」基礎類別的子物件）就不會佔用任何儲存空間。

empty base class optimization（空基礎類別優化）
參見 EBCO。

explicit instantiation directive（顯式具現指令）
一個 C++ 構件，惟一用途是建立一個 POI（point of Instantiation; 具現點）。

explicit specialization（顯式特化）
一種 C++構件，爲某個 template 宣告或定義一個替代性（alternative）定義式。最初的那個最泛化的版本稱爲 primary template（主模板）。如果此一替代性定義仍需倚賴一或多個 template parameters，它就被稱爲 partial specialization（偏特化），否則就稱爲 full specialization（全特化）。

expression template（算式模板）
一種 class template，用以表現某算式的一部分。該 template 本身表示某種特定操作（譯註：例如加減乘除等等），template parameter 則表示操作施行對象（運算元）的種類。

friend name injection（friend 名稱植入）
當函式被宣告爲 friend 時，「使該函式名稱可見」的一個處理過程。

full specialization（全特化）
參見 explicit specialization（顯式特化）。

function object（函式物件）
參見 functor（仿函式）。

function template（函式模板）
一種構件（construct），用以表示「一整群函式族系」（a family of functions）。它規定某種樣式（pattern），只要以特定物體（entities）替換其 template 參數，便可從中生成真正的函式。注意，function template 是一個 template 而非一個 function。有時它也被稱爲參數化函式（"parameterized" functions），儘管後者具有更一般化的含義。

functor（仿函式）
可以「函式呼叫語法」進行呼叫的物件（也稱爲 function object; 函式物件）。在 C++ 中指的是 pointers to functions 或 references to functions，或是任何擁有 `operator()` 成員的 class。

header file（表頭檔）
意欲「透過`#include`指令成爲編譯單元一部分」的檔案。這樣的檔案通常包含變數和函式的宣告（它們可被不只一個編譯單元引用），以及 types（型別）、inline 函式、templates（模板）、constants（常數）和 macros（巨集）等定義。它們通常以 .hpp, .h, .H, .hh 或 .hxx 做爲副檔名。又稱爲含入檔（include files）。參見 dot-C file 和 translation unit（編譯單元）。

include file（含入檔）
參見 header file（表頭檔）。

indirect call（間接呼叫）
函式呼叫形式之一。未到呼叫動作真正發生（執行期），不知道該呼叫哪一個函式。

initializer（初值設定器）
一種構件（construct），用以規定如何初始化一個具名物件（named object）。例如以下的式子：
```
std::complex<float> z1 = 1.0, z2(0.0, 1.0);
```
初值設定器是 `=1.0` 和 `(0.0, 1.0)`。

initializer list（初值列，初始值列表）
一個封閉於小括號內部、以逗號分隔的算式列表，用以初始化物件或陣列。在建構式中可用以定義 members（成員）和 base class（基礎類別）的初值。

injected class name（植入類別名稱）
class 名稱在它自己的作用域（scope）中是可見的。對 class template 來說，在 template 作用域內，如果其名稱之後沒有跟著一個 template argument list 的話，該 template 名稱將被視爲一個 class 名字。

instance（實體）
在 C++ 編程領域中，instance 有兩層含義：物件導向方面的意思是「某個 class 的實體」，也就是 object（物件），是某 class 的實現結果。例如 C++ 中的 `std::cout` 是 class `std::ostream` 的一個實體。另一層意思（本書所採用）是 "a template instance"：透過「以特定值替換所有 template parameters」而得到的 classes、functions 或 member functions。在這個意義下 instance 也被稱爲 specialization（特化體），後者常被誤以爲是 explicit specialization（顯式特化體）。

instantiation（具現化）
以實際值替換 template parameters，從而自一個 template 創建出常規的（non-template）classes、functions 或 member functions 的過程。另一層意思（本書不採用）是：創建 class 的一份實體/物件（參見 instance）。

ISO
國際標準組織（International Organization for Standardization）的全球通用縮寫。名爲 WG21 的 ISO 工作小組正是 C++標準化和持續發展的背後驅動力。

iterator（迭代器）
一種知道如何巡訪（遍歷; traverse）元素序列的物件。通常那些元素屬於一個群集（參見 collection class）。

linkable entity（可聯結物）
包括 non-inline 函式或成員函式、global 變數或 static 成員變數，以及從 template 生成的任何此類東西。

lvalue（左值）
在最初的 C 語言中，如果算式（expression）可出現於 *assignment*（賦值）運算子左側的話，它就被稱爲一個左值。反過來說，只能出現於 *assignment* 運算子右側者，就稱爲 rvalue（右值）。這個定義不再適用於新一代 C/C++。如今 lvalue 可被認爲是一個 locator value，意指「透過名稱或位址（可以是 pointers、references 或 array access）」而非透過純粹計算來標示出某物件」的算式。lvalue 不必是可修改的（例如常數物件就是一種不可修改的 lvalue）。所有不是 lvalue 的算式都是 rvalue。透過 `T()` 顯式創建出來的暫時物件，或作爲函式呼叫結果的暫時物件，都是 rvalue。

member class template（成員類別模板）
一種構件，用來表示「一整個族系的成員類別」（a family of member classes）。這是一種「宣告於另一個 class 或 class template 內」的 class template，它有自己的一套 template parameters，此點與「class template 內的 member class」不相同。

member function template（成員函式模板）
一種構件，用來表示「一整個族系的成員函式」（a family of member functions）。它有自己的一套 template parameters，此點與「class template 內的 member function」不同。這種東西非常類似 function template，但其所有 template parameters 被替代後產出的是個 member function（而不是個 ordinary function）。member function template 不可以是 virtual。

member template（成員模板）
意指 member class template 或 member function template。

nondependent name（非受控名稱）
一個不取決於 template parameters 的名稱。參見 dependent name 和 two-phase lookup。

ODR
one-definition rule（單一定義規則）的縮寫。這個規則對 C++ 程式內的定義式（definitions）施加了一些約束。詳見 7.4 節, p.90 和附錄 A。

one-definition rule（單一定義規則）
參見 ODR。

overload resolution（重載決議）
當函式候選者多於一個時（它們通常擁有相同名稱），「選擇/決定 喚起哪個函式」的過程。

parameter（參數）
一種佔位物體(placeholder entity)，會於程式的某一點被實際值(一個引數)替換。macro 參數和 template 參數的替換發生於編譯期，function call 參數的替換發生於執行期。某些編程社群將參數稱爲形參（formal parameters），將引數（arguments）稱爲實參（actual parameters）。參見 argument。

parameterized class（參數化類別）
意指 class template，或「嵌套（nested）於某 class template 內」的 class。它們都是參數化的，因爲除非 template argument 獲得指定，否則它們不會對應至某個獨一無二的 class。

parameterized function（參數化函式）
意指 function template 或 member function template 或 class template 的成員函式。它們都是參數化的，因爲除非 template argument 獲得指定，否則它們不會對應至某個獨一無二的函式（或成員函式）。

partial specialization（偏特化）
一種構件（construct），用以爲 template 定義一個替代性定義式，而其中仍具有某些可替代參數。最初那個最泛化的版本稱爲 primary template（主模板）。這一份替代性定義仍然相依於某些 template parameters。這種構件目前僅對 class templates 有效。參見 explicit specialization（顯式特化）。

POD（簡樸舊式資料）
plain old data (type) 的縮寫。POD 是這樣一種型別：其定義不牽扯某些 C++ 特性諸如 virtual（虛擬）成員函式、存取關鍵字 public, private 等等。每個 C `struct` 都是個 POD。

POI（具現點）
point of instantiation（具現點）的縮寫。POI 是源碼中的某個位置，該處將以 template arguments 替換 template parameters，從而將 template（或其成員）概念性地展開。現實中並非每個 POI 都需要發生展開行爲。參見 explicit instantiation directive（顯式具現指令）。

point of instantiation（具現點）
參見 POI。

policy class（策略類別）
一個 class 或 class template，其成員用以描述泛型組件(generic component)之可組態行爲(configurable behavior)。此物通常透過 template arguments 傳遞。例如一個排序用的 template 可能需要一個 "ordering policy"。Policy classes 亦被稱爲 policy templates 或簡稱爲 policies。參見 traits template。

polymorphism（多型）
「將某一（根據名稱確認之）操作施行於不同種類的物件身上」的能力。C++ 傳統物件導向概念中的 polymorphism（亦稱爲 run-time polymorphism 或 dynamic polymorphism）是透過「在 derived class 中覆寫虛擬函式」實現出來的。C++ templates 還支援所謂的 static polymorphism（靜態多型）。

precompiled header（預編譯表頭）
一種經過處理的源碼形式，可以迅速被編譯器載入（loaded）。預編譯表頭底部的（underlying）程式碼必須是編譯單元（translation unit）的第一部分（換句話說不能從編譯單元的某個中間點開始）。通常一個預編譯表頭對應多個表頭檔（header files）。使用預編譯表頭，可大幅改善建置（build）C++ 大型程式的時間。

primary template（主模板/原始模板）
一個「非偏特化版本」（not a partial specialization）的 template。

qualified name（資格限定名稱/資格修飾詞）
包含 *scope*（作用域）修飾符號（::）的名稱。

reference counting（引用計數）
一種資源管理策略，記錄（計數）當下有多少個物體（entities）正在引用（*referring to*）某特定資源。一旦數目降爲 0，資源就可被釋放掉。

rvalue（右值）
參見 lvalue（左值）。

source file（源碼檔案）
意指表頭檔（header file）或 dot-C 檔案。

specialization（特化體/特化版本）
以實值替換 template parameters 的結果。特化體可經由具現化（instantiation）或顯式特化（explicit specialization）創造出來。這個術語有時被錯誤地和 explicit specialization（顯式特化版本）混爲一談。參見 instance（實體）。

template（模板）
一種構件（construct），用以表示「一整個家族的 class 或 function」。它規定某種樣式（pattern），只要以特定物體（entities）替換其 template 參數，便可從中生成真正的 classes 或 functions。本書中這個術語不包括「僅僅由於是某個 class template 的成員而被參數化」的 functions、classes 和 static data members。參見 class template（類別模板）、parameterized class（參數化類別）、function template（函式模板）和 parameterized function（參數化函式）。

template argument（模板引數）
「用以替換 template parameters」的一個值（value）。此「值」通常是一種型別（type），儘管 constant values（常數）和 templates（模板）也是合法的 template arguments。

template argument deduction（模板引數推導）
參見 deduction（推導）。

template-id（模板識別符號）
template 名稱連同「緊隨其後之角括號內的 template arguments」的組合（例如 `std::list<int>`）。

template parameter（模板參數）
templates 中的泛型佔位符號（generic placeholder）。最常見的 template parameters 是 type parameters，代表型別；nontype parameters 則用以表示某特定型別的常數值；template template parameters（雙重模板參數）表現的是 class templates（類別模板）。

traits template（「特徵萃取」模板）
一種 templates，其成員用以描述 template arguments 的特徵（characteristics; traits）。通常 traits templates 的用途是避免 template parameters 過量。參見 policy class。

translation unit（編譯單元）
一個 dot-C 檔案，帶有以`#include` 指令含入的所有「表頭檔」和「標準程式庫表頭檔」，並扣除被條件編譯指令（例如 `#if`）排除掉的程式段落。簡單地說，它可以被視爲「對一個 dot-C 檔案進行預處理」後的結果。參見 dot-C file 和 header file。

true constant（真常數）
參見 constant-expression（常數-算式）。

tuple（三部合成構件）
C `struct` 概念的一種泛化版本，其成員可經由編號來存取。

two-phase lookup（兩段式查詢）
template 內的名稱查詢機制。所謂兩段式是指（1）當 templates 定義式被編譯器第一次遇上時（2）當 template 被具現化時。Nondependent names（非受控名稱）只在第一階段被查詢，但此階段不考慮 nondependent base classes（非受控基礎類別）。至於 dependent names（受控名稱），擁有 *scope* 修飾符號（`::`）者只在第二階段被查詢，不帶 *scope* 修飾符號（`::`）者可能在兩階段都被查詢，但第二階段只執行 argument-dependent lookup（相依於引數的查詢）。

user-defined conversion（用戶自定轉換）
由程式員定義的型別轉換動作，可以是個「以單引數喚起」的建構式，也可以是個 *conversion*（轉型）運算子。除非帶有關鍵字 `explicit`，否則「單引數建構式」可能會發生隱式型別轉換。

whitespace
在 C++中，這是對源程式碼之各種記號（包括標識符號 identifiers、字面值 literals、語彙單元符號 symbols 等等）進行分界的區隔物。除了傳統的空格（blank space）、換行（new line）和水平定位（horizontal tabulation）等字元外，還包括註釋（comments）。其他的空格字元（例如 page feed control character）有時也是合法的。