前言

Preface

Templates 概念自應用於 C++，迄今已逾三十年。C++ template 明載於 1990 年出版的「The Annotated C++ Reference Manual」（ARM；詳見 [*EllisStroustrupARM*]）裡，亦見諸於更早的專門出版物。然而十多年過去了，我們發現始終缺乏探討其基礎概念與進階技術的專著，即便它是如此的迷人、難解、且威力強大。藉由本書的第一版，我們希望達成這個目標，寫一本關於 templates 的書（或許我們有點自以為是吧）。

自第一版出版於 2002 歲末以來，C++ 改變了不少。新一輪的 C++ 標準增添了新特性，而 C++ 社群的持續創新也發明了基於 template 的新編程技術。因此，本書的第二版維持和第一版相同的目標，但這次針對的是「現代 C++」。

編寫本書的我們有著不同的背景，亦懷抱不同目的。David（或「Daveed」）是一位富有經驗的編譯器實作者，積極參與推進核心語言發展的 C++ 標準委員會工作小組。他對精確、仔細地描述所有關於 templates 的強大特性（和問題）感興趣。Nico 是一位「通俗」* 的應用程式開發者、C++ 標準委員會程式庫工作組成員。他著重於理解所有能夠應用於日常工作、並從中獲益的 templates 技術。Doug 是一位 template 程式庫開發者，後轉而從事編譯器實作及程式語言設計。他關注於收集、分類、與驗證成千上萬用以建構 template 程式庫的技術。最後，我們希望對整個社群、以及讀者您分享這些知識，以避免更多的誤解、混淆、與焦慮。

因此，你將會讀到觀念層面的介紹，輔以日常範例和對 template 精確行為的詳實描述。從 template 基本原理出發，乃至「template 編程藝術」。你將會發現（或重新發現）一些技術，如：靜態多型（static polymorphism）、type traits（型別特徵萃取）、metaprogramming（後設編程）、expression templates（陳述式模板）。你會對 C++ 標準程式庫有更深一層的理解，因為裡頭的程式碼全面應用了 templates。

我們從編寫此書中獲益良多、同時也樂在其中。衷心希望您在閱讀此書時也能有同樣體驗。好好享受吧！

---

* 譯註：指其特別關心一般應用程式開發

關於本書

About This Book

本書的第一版出版於十五年前，我們初心是想寫一本 C++ template 的專門指南，以期它能有助於培訓 C++ 程式設計師們。那個計劃成功了：每每從讀者們那裡聽到我們的書令他獲益、見到我們的書一次次的被推薦為參考書、以及普遍得到好評時，我們都極其滿足。

第一版雖已一把年紀，其中大多數內容仍然對現代 C++ 程式設計師們有用。然而不可否認，C++ 語言本身仍在持續進化——集大成於「現代 C++」標準，C++11、C++14、和 C++17，因為它們，第一版內容勢必要做些修正了。

故於本書的第二版，我們的願景仍然不變：提供一本對 C++ template 的專門指南，包含可靠的參考資料和深入淺出的教程。然而這次我們面對的是「現代 C++」，比起第一版時的語言版本，它明顯更大、更（依然）狂野。

我們同時也敏銳地察覺到，自第一版出版後，C++ 程式設計資源有了些許變化（往好的方面）。例如，許多專門探討以 template 進行應用程式開發的書籍開始出現。更重要的是，更多關於 C++ template 與衍生技術的資訊更容易在網上取得了，這也包括相關技術的進階範例。因此在第二版，我們決定收錄更多不同的技術，以滿足各式各樣的應用場景。

有些在第一版中提及的技術顯得過時了，現在 C++ 語言提供許多更直接的方法能達成同樣的效果。這些內容被我們刪去（或改放在附註），取而代之的是一些應用最新語言特性的技巧。

我們已與 C++ template 共度了二十餘年，然而 C++ 程式設計師社群仍經常發現一些嶄新的基本見解，其能滿足我們的軟體開發需求。我們對本書的期許正是分享這些知識，以及幫助讀者對語言有新一層的理解，甚至可能的話，發現下個 C++ 的主要技術。

閱讀本書之前，您應該知道的事

為了充份理解本書，您最好已經對 C++ 有基本認識。我們會描述某個語言特徵的細節，而非語言本身的基礎知識。您應該熟悉 Classes （類別）與繼承概念，您也應該能夠使用 C++ 標準程式庫中的構件來撰寫 C++ 程式，如 IOStreams 和各式容器（containers）等。您也應該熟悉「現代 C++」的基本特性，如 `auto`、`decltype`、搬移語意（move semantics）、以及 lambdas 。然而若有必要，我們也會討論一些與 template 並不直接相關的微妙議題。這確保了文章對專家或是中等程度的程式設計師都同樣地好理解。

我們主要針對 C++ 語言標準於 2011、2014、和 2017 年的修訂版本。然而在執筆當下，C++17 標準才剛出爐，因此我們假設多數讀者並不熟悉它的細節。所有修訂版本對 template 的行為及用法均有重大的影響，因此我們針對那些會造成重大影響的新特性提供簡短的介紹。然而，我們的目標並非介紹現代 C++ 標準，也非提供一份（自 [*C++98*] 和 [*C++03*] 開始）前後版本標準差異的詳細描述。相反地，我們專注於 templates 如何在 C++ 中被設計及應用，並以現代 C++ 標準（[*C++11*]、[*C++14*]、[*C++17*]）做為基礎。我們偶爾也會在現代 C++ 標準相較先前標準採用或偏好不同技術時舉例說明。

全書結構

我們的目標是提供讀者使用 templates 的必要資訊、並能得益於其威力。同時提供有經驗的程式設計師們推動語言革新的所需資訊。為了達成這個目的，我們將全書分為以下**幾篇**：

1. 第一篇介紹 templates 的基本概念，以教程（tutorial）的型式呈現。
2. 第二篇介紹語言細節，為 template 相關構件的便捷參考。
3. 第三篇闡述 C++ templates 支援的基礎設計和編程技術，從直白的想法到老練的慣用手法都有。

每一篇都包含數個章節。此外，我們提供了一些附錄，其中包含但不限於 template 相關內容（如 C++ 重載決議機制的概述）。另有一篇附錄記述了 templates 的根本性擴充，其已被包含於未來標準的草案中（預計為 C++20）。

第一篇裡的章節需要依序閱讀。例如第三章便是奠基於第二章所提及的內容。至於其他幾篇的章節彼此之間就不大相關，在閱讀時交互查閱有助於讀者穿梭於不同主題之中。

最後，我們提供了一份相當完整的索引，方便讀者以自己的方式，跳脫順序閱讀本書。

如何閱讀本書

如果你是一位 C++ 程式設計師，並且希望學習或是複習 templates 的基本概念，請仔細閱讀第一篇：基本認識。即便你已經相當熟悉 templates 了，快速略讀第一篇也有助於熟悉本書的寫作風格與慣用術語。它同時也提到當使用 templates 時如何有系統地組織你的程式碼。

視乎你偏好的學習方法，你可能想於第二篇了解盡可能多的 template 細節，或是先閱讀第三篇裡的實際編程技巧（並在想了解微妙的語言細節時回頭查閱第二篇）。若你購買本書是為了解決具體的日常難題時，後者或許更適合你。

附錄包含許多在本書正文中常提及的有用資訊，我們同樣試著讓它們讀起來更加有趣。

依我們的經驗，學習新事物的最好方法是從範例出發。因此全書包含了大量範例，有些僅以數行程式碼說明抽象概念，有些則是以完整的程式示範內容的具體應用。後面這種範例會將含有該程式碼的檔案標明於 C++ 註釋中，你可以在本書的網站上找到這些檔案：

`http://www.tmplbook.com`

關於編程風格的幾點說明

C++ 程式設計師們有各自的編程風格，我們也一樣：司空見慣的問題像是在哪兒加上空白、分隔符號（大、小括號）等等。我們盡可能大體上保持一致，即便有時我們可能會在某些時候做點妥協。例如，在教程小節，我們偏好大量使用使用空白及較具體的命名方式以增進可讀性；但在深入討論時，較緊湊的程式碼可能更合適。

我們希望您留意，在宣告型別（types）、參數（parameters）、和變數（variables）時，我們有一個較不尋常的習慣。以下幾種顯然都是可能的寫法：

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

雖然可能比較少見，我們決定使用 `int const` 而非 `const int` 來表示「常整數（constant integer）」。我們基於兩點理由使用這樣的順序。首先，它能簡單地回答以下問題：「何者是常數（*what* is constant）」？答案永遠是：`const` 修飾符前面的那一個。確實如此，即便下面兩式等價：

```
const int N = 100;       // 一般人慣用的寫法
int const N = 100;       // 本書使用的寫法
```

但以下式子並不存在等價的型式：

```
int* const bookmark;     // 指標本身不可更動，但它所指的值可以
```

若是將 const 修飾符置於指標運算子 * 之前*，意思就改變了。在本例中，指標本身是個常數，而不是它指向的那個 int。

我們的第二個理由與語法替換原則（syntactical substitution principle）有關，在處理 template 時經常會遇到。考慮下列兩個使用 typedef 關鍵字做的型別（type）宣告[1] **：

```
typedef char* CHARS;
typedef CHARS const CPTR;        // 指向 chars 的 const 指標
```

或是在使用 using 關鍵字時：

```
using CHARS = char*;
using CPTR  = CHARS const;       // 指向 chars 的 const 指標
```

當我們將 CHARS 文字代換為它的定義時，第二條宣告式的原意依舊不變：

```
typedef char* const CPTR;        // 指向 chars 的 const 指標
```

或是：

```
using CPTR = char* const;        // 指向 chars 的 const 指標
```

然而，如果我們將 const 置於被修飾物之前，這個原則就不再適用了。若我們把先前的兩個型別定義式改寫成這樣：

```
typedef char* CHARS;
typedef const CHARS CPTR;        // 指向 chars 的 const 指標
```

現在若再次將 CHARS 做文字代換，會產生完全不同的型別。

```
typedef const char* CPTR;        // 指向 const chars 的指標
```

對於 volatile 修飾詞，也會有同樣的狀況。

此外，我們總是會在 & 符號和參數名稱中間插入空白：

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

這樣做是為了刻意把參數型別和參數名稱分開。像以下的宣告方式肯定會使人困惑：

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

根據繼承自 C 語言的規則，這裡的 a 是一個指標，而 b 是一般的 char。為了避免混淆，我們避免宣告多個變數於同一行。

---

1 注意在 C++ 裡，typedef 實際上是定義了一個「型別別名（type alias）」，而非一個新的型別（見 2.8 節，第 38 頁）。例如：

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

* 譯註：即 int const* bookmark。

** 譯註：參考 C++ 標準，這裡使用「宣告」而非「定義」，見 C++ 標準 3.1 節。

這是一本主要針對語言特性的書，然而 C++ 標準程式庫裡包含了許多技術、特性、以及 helper templates（輔助模板）。為了兼顧兩者，我們將說明 templates 相關技術如何用來實作某個程式庫構件，同時使用標準程式庫工具以建立更為複雜的範例。因此我們不僅會使用如 `<iostream>` 和 `<string>` 之類的標頭檔（它們應用 templates 技術，但通常不會用來定義其他 templates），同時也會使用 `<cstddef>`、`<utilities>`、`<functional>`、和 `<type_traits>`（它們提供了能用來實作更複雜 templates 的基礎元件）。

此外，附錄 D 是一份關於 C++ 標準程式庫中重要 template 工具的參考資料。其中包含所有 standard type traits（標準型別特徵萃取）的詳細描述。這些工具經常會在複雜的 template 程式設計中派上用場。

C++11、C++14 與 C++17 標準

C++ 標準最初發佈於 1998 年，隨後在 2003 年發表了一份技術勘誤（*technical corrigendum*），用以對初始版本做小幅度修正和澄清。這份「舊式 C++ 標準」被稱做 C++98 或 C++03。

C++11 標準是第一個由 ISO C++ 標準委員會負責的 C++ 重大改版，替語言帶來了豐富的新特性。本書介紹了其中某些與 templates 相互影響的新特性，包含：

- Variadic templates（可變參數模板）
- Alias templates（別名模板）
- Move semantics（搬移語意）、rvalue references（右值參考）、與完美轉發（perfect forwarding）
- Standard type traits（標準型別特徵萃取）

後續的 C++14 和 C++17 都引進了某些新的語言特性，但改變幅度並沒有像 C++11 那樣巨大[2]。本書介紹與 templates 產生互動的新特性包含了（但不僅限於此）：

- Variable templates（變數模板，C++14）
- 泛型 Lambda 表示式（C++14）
- Class template 引數推導（C++17）
- 編譯期 `if`（C++17）
- 摺疊表示式（C++17）

我們更進一步提到 *concepts*（作為 templates 的介面），它預計會被包含在即將推出的 C++20 標準中。

---

2 標準化委員會現行目標大約是每三年發佈一次新標準。顯然，這樣會壓縮新增大量特性的可用時間，不過也使得改變可以更快地分享給更多的編程社群。這樣一來，橫跨一定時間的重大特性開發，可能會被散布在多個標準之中。

在寫作的當下，C+11 和 C++14 標準已經被主流編譯器廣泛地支援，同時 C++17 也獲得了相當程度的支援。即便如此，各家編譯器對於不同語言特性的支援差異很大。部分讀者會編譯本書中大多數的程式碼，不過有些編譯器可能會無法處理部分範例。不過，我們預料這個問題將會很快獲得解決，因為世界各地的程式設計人員都需要供應商支援語言標準。

即便如此，C++ 程式語言仍舊可能隨著時間經過而持續進化。C++ 社群中的專家們（不管他們是否參與 C++ 標準化委員會）討論著改良語言的各種方法，也已經有一些改良方案影響了 templates。第 17 章會介紹這個領域的一些趨勢。

範例程式碼和補充資訊

你可以從本書官網取得所有的範例程式以及與本書相關的其他資訊，網址如下：

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

意見回饋

我們竭誠歡迎您的建設性意見，無分讚美或批評。我們耗盡心力將本書呈現給您，希望您會覺得它是一本優秀的作品。不過，在某個時刻我們不得不停筆、停止校閱和修訂，否則便永遠無法推出產品。您可能會因此發現一些錯誤、內文前後不一致、欠佳的表達方式、或是缺了某些主題。您的回饋使我們有機會透過官網告知所有讀者，同時在後續版本中改進。

聯絡我們最好的方式是透過電子郵件。你可以在本書官網找到電子郵件地址：

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

請確認在回報任何意見前，先行檢查過本書官網上的已知勘誤資訊（errata）。感謝您。

1

函式模板

Function Templates

本章主要介紹 *function templates*（函式模板）。Function templates 是一種參數化（parameterized）的函式，用以表現一整個函式家族。

1.1 初識 Function Templates

Function templates 提供適用不同型別的函式行為。換句話說，單一 function template 能表現一整個函式家族。這種寫法看起來就像個普通的函式——除了一些函式內的元素尚未被決定之外。而這些未定的元素被「參數化」了。為了說明，讓我們瞧瞧一個簡單的例子。

1.1.1 定義 Template

以下是一個用以回傳兩數中較大者的 function template。

basics/max1.hpp

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

這個 template 定義代表了一整個函式家族，能夠回傳兩參數 a 和 b 中的較大者[1*]。參數的型別仍然是未定的，以 *template parameter*（模板參數）T 表示。如範例所示，template parameters 必須用以下的語法宣告。

---

1 注意。參考 [*StepanovNotes*]，以上的 max() template 試圖回傳“b < a ? a : b”，而非“a < b ? b : a”，以確保函式行為在兩相異參數 a、b 等值時仍然正確。

* 譯註：考慮一個由小到大排序的數列 [a, b]，你通常會假設 max(a, b) 回傳的是 b，即便 a、b 等值也應該如此。但若以上述的“a < b ? b : a”來實作，回傳值會是前面的 a，這樣的結果會令人意外。所以作者才會說用“b < a ? a : b”比較正確。

```
template< 一列以逗號區隔的參數 >
```

在我們的範例裡，參數列放的是 `typename T`。注意現在 < 和 > 符號作為括號成對使用；我們稱之為**角括號**（*angle brackets*）。而關鍵字 `typename` 引入了一個 *type parameter*（**型別參數**）。這是目前為止在 C++ 程式中最常見的 template parameter 形式，但也可能出現其他形式的參數，我們將在後面討論它們（見第三章）。

這兒的 type parameter 是 `T`，你也可以使用任何的**識別字**（*identifier*）作為參數名稱，但一般習慣使用字母 `T`*。Type parameter 可以是任意型別，由函式**呼叫者**（*caller*）在使用函式時決定。你可以使用任何型別（包括**基本型別**（*fundamental type*）、class（類別）等），只要其支援在 template 內所使用到的運算即可。在這個例子中，型別 `T` 必須能夠支援 < **運算子**（*operator*），因為 `a` 和 `b` 使用了 < 運算子進行比較。還有一點比較隱晦的是，`max()` 的定義暗示：因函式回傳值所需，型別 `T` 必須是**可複製的**（*copyable*）[2]。

因為歷史因素，你也可以使用關鍵字 `class` 取代 `typename` 來定義一個 type parameter。關鍵字 `typename` 在開發的相對晚期才納入 C++98 標準。在那之前，`class` 關鍵字是唯一一種引入 type parameter 的方法，並一直保留至今。因此，template `max()` 也能用以下方式定義：

```
template<class T>
T max (T a, T b)
{
    return b < a ? a : b;
}
```

語意上，這樣定義並無任何區別。所以即使你在這兒使用了 `class`，仍然能夠使用**任何**型別作為 *template arguments*（模板引數），用以代入參數。然而，這種使用 `class` 的方式可能造成誤解（`T` 並非只能使用 class 型別代入），因此您在這裡使用 `typename` 較好。然而，不像宣告 class 型別那樣，關鍵字 `struct` 並不能取代 `typename` 用以宣告 type parameters。

1.1.2 使用 Template

以下程式示範如何使用 `max()` function template：

basics/max1.cpp

```
#include "max1.hpp"
#include <iostream>
#include <string>
```

[2] 在 C++17 前，為了能夠傳入引數，型別 `T` 的必要條件是它可複製。但是從 C++17 開始，你也可以選擇傳入**暫存值**（temporaries；即 rvalues（右值），見附錄 B），即便該型別未支援複製建構子（copy constructor）和搬移建構子（move constructor）也可以。

* 譯註：T 代表 Type。

```
int main()
{
  int i = 42;
  std::cout << "max(7,i):   " << ::max(7,i) << '\n';

  double f1 = 3.4;
  double f2 = -6.7;
  std::cout << "max(f1,f2): " << ::max(f1,f2) << '\n';

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

這個程式裡 `max()` 被呼叫了三次：第一次引數是兩個 `int`s、第二次引數是兩個 `double`s、最後一次則是兩個 `std::string`s。每一次呼叫時，兩數的較大值都會被計算出來。因此，程式會有以下的輸出：

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

注意每次呼叫 `max()` 時前面都加上了 `::`。這是為了確保我們喚起的 `max()` template 是定義於 *global namespace*（全域命名空間）裡的那一個。在**標準程式庫**（*standard library*）內，同樣存在著一個 `std::max()` template，在某些情況下這個版本可能會被喚起，或是導致**歧義**（*ambiguity*）的發生[3]。

Templates 並非被編譯成一個能處理任何型別的一份**個體程式碼**（entities）。相反的，每一處對應不同型別而使用到 template 的地方，都會有不同的個體程式碼被產生[4]。因此，`max()` 會對應這三種型別，分別編譯出三個版本。例如，第一次呼叫 `max()` 時

```
int i = 42;
… max(7,i) …
```

使用了以 `int` 作為 template parameter `T` 的 function template。因此，它的語意等同於呼叫以下程式碼：

```
int max (int a, int b)
{
    return b < a ? a : b;
}
```

---

3 例如：若某個引數型別定義於 `std` 命名空間中（如 `std::string`）。基於 C++ 的查詢規則，存在於 global 和 `std` 命名空間中的 `max()` 都會被找到，因而引發歧義（見附錄 C）。

4 「通用個體程式碼」（one-entity-fits-all）概念上可行，實行上卻有困難（這會降低執行期（run time）效能）。當前所有語言規則都遵循相同的設計原則：不同 template arguments 會產生不同的個體程式碼。

以具體型別取代 template parameters 的過程稱為**實體化**（*instantiation*），它會產生一份 template 的**實體**（*instance*）[5]。

請注意，我們只需使用 function template 就能夠觸發此種實體化。實體化過程並不需要程式設計師的額外關注。

同理，另兩個對 `max()` 的呼叫也能夠實體化 `max` template 對應於 `double` 及 `std::string` 的不同版本，彷彿它們各自被宣告與實作一般。

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

值得留意的是，`void` 型別同樣是個合法的 template argument，能據以產生合法的程式碼。例如：

```
template<typename T>
T foo(T*)
{
}

void* vp = nullptr;
foo(vp);                          // OK：推導出 void foo(void*)
```

1.1.3 兩段式轉譯（Two-Phase Translation）

嘗試以某個型別實體化 template 時，若該型別並未對所有用到的**運算**（operation）提供支援，則會導致**編譯期錯誤**（compile-time error）。例如：

```
std::complex<float> c1, c2;       // 未提供 < 運算子
…
::max(c1,c2);                     // 編譯時出錯啦
```

因此，「編譯」templates 實際上分成兩階段。

1. 尚未實體化的**定義時期**（*definition time*），忽略 template parameters 並進行程式碼的正確性檢查。過程包含：
   - 檢查語法錯誤。像是沒加上分號之類的問題。
   - 檢查是否使用了與 template parameters 無關的未知名稱（型別名稱、函式名稱等等）。
   - 檢查和 template parameters 無關的**靜態斷言**（static assertions）。
2. 於**實體化時期**（*instantiation time*）template 程式碼會（再次）被檢查，以確保所有程式碼均正確。換言之，所有依賴於 template parameters 的部分都會在此時特別被重新檢驗。

[5] 「實體（*instance*）」和「實體化（*instantiate*）」兩個名詞在**物件導向程式設計**（*object-oriented programming*，*OOP*）中有著不同的應用場景——它們用以表現一個 class 的具體**物件**（*concrete object*）。然而，這本書關注的是 templates，故除非我們特別標明，否則這些名詞均用以表現「作用於」templates 時的涵義。

例如：

```
template<typename T>
void foo(T t)
{
  undeclared();      // 如果找不到 undeclared()，則導致第一階段編譯期錯誤
  undeclared(t);     // 如果找不到 undeclared(T)，則導致第二階段編譯期錯誤
  static_assert(sizeof(int) > 10,    // 若 sizeof(int)<=10，必定失敗
                "int too small");
  static_assert(sizeof(T) > 10,      // 以 size <=10 的 T 進行實體化才會失敗
                "T too small");
}
```

名稱經過兩次檢查的這個行為叫做 *two-phase lookup*（兩段式查詢），在 14.3.1 節（第 249 頁）有更深入的討論。

注意並不是所有的編譯器都會在第一階段做完整的檢查[6]。因此你可能要到 template 程式碼被實體化至少一次後方能見到問題。

編譯和連結

兩段式轉譯引發了一個實際運用 template 時會遇到的重要問題：當使用 function template 會觸發實體化時，編譯器（在某些時候）會需要知道 template 的定義。這破壞了一般函式在編譯及連結時期的預設分野：在編譯函式呼叫處時，只需要其宣告式就夠了。第 9 章會討論處理這個問題的方法。現在我們先採用簡單的作法：把所有的 template 實作都放在標頭檔（header file）中。

1.2 Template 引數推導

當呼叫一個像 `max()` 這樣的 function template 時，我們傳入的引數會決定 template parameters。如果將兩個 `int`s 作為參數型別 `T` 傳入，C++ 編譯器會作出結論：`T` 必須是個 `int`。

然而，`T` 可能僅僅只是參數型別的「部分組成」。例如，如果我們用 *constant reference*（常數參考）來宣告 `max()`：

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

若傳入的引數是 `int`，則 `T` 同樣會被推導為 `int`。因為此時函式參數匹配於 `int const&`。

6 例如，某些版本的 Visual C++ 編譯器（像是 Visual Studio 2013 和 2015）會放過和 template parameters 無關的未宣告名稱以及部分語法瑕疵（像是忘了分號）。

型別推導期間的型別轉換

記住：當型別推導時，自動型別轉換會受到限制。

- 當參數宣告為*以 reference（參考）呼叫*（call by reference）時，即便是最直觀的轉型也不會在型別推導時實行。以同一個 template parameter `T` 宣告的兩個引數，型別必須完全相同。
- 當參數宣告為*以值呼叫*（call by value）時，只支援最直觀的*退化*（*decay*）轉換：冠詞如 `const` 或 `volatile` 將被忽略、references 會被轉為對應的型別*、原始陣列（raw arrays）或函式會被轉為對應的指標型別。以同一個 template parameter `T` 宣告的兩個引數，*退化後*（*decayed*）的型別必須相同。

舉例：

```
template<typename T>
T max (T a, T b);
…
int i = 17;
int const c = 42;
max(i, c);              // OK：T 被推導為 int
max(c, c);              // OK：T 被推導為 int
int& ir = i;
max(i, ir);             // OK：T 被推導為 int
int arr[4];
max(&i, arr);           // OK：T 被推導為 int*
```

下面則是出錯的例子：

```
max(4, 7.2);            // 錯誤：T 可以被推導為 int 或 double
std::string s;
max("hello", s);        // 錯誤：T 可以被推導為 char const[6] 或 std::string
```

有三個方法可以處理這樣的錯誤：

1. 把兩個引數轉為相同型別：
   ```
   max(static_cast<double>(4), 7.2);      // OK
   ```
2. 明確指定（描述）`T` 的型別，以避免編譯器試著推導型別：
   ```
   max<double>(4, 7.2);                   // OK
   ```
3. 讓各個參數擁有不同型別**。

1.3 節（第 9 頁）會闡述這些解決方案，7.2 節（第 108 頁）與第 15 章會詳細討論型別推導時的型別轉換原則。

預設引數的型別推導

同時注意，型別推導不會作用於 *default call arguments*（預設呼叫引數），例如：

---

* 譯註：去掉 `&`。

** 初版譯註：如第一個是 `T1`，第二個是 `T2`。

```
template<typename T>
void f(T = "");
…
f(1);                       // OK：T 將被推導為 int，故呼叫的是 f<int>(1)
f();                        // 錯誤：無法推導 T
```

為了支援這種用法，你必須替 template parameter 也宣告一個預設引數。1.4 節（第 13 頁）會有更進一步的討論。

```
template<typename T = std::string>
void f(T = "");
…
f();                        // OK
```

1.3 多個 Template Parameters

到目前為止，function template 會具備兩種不同的參數：

1. *Template parameters*（模板參數）：宣告於 function template 名稱前的角括號中。

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

2. *Call parameters*（呼叫參數）：宣告於 function template 名稱後的小括號中。

   ```
   T max (T a, T b)              // a 和 b 是 call parameters
   ```

你可以有任意多個 template parameters。例如，你可以這樣定義 `max()` template，讓兩個 call parameters 可能擁有不同的型別。

```
template<typename T1, typename T2>
T1 max (T1 a, T2 b)
{
    return b < a ? a : b;
}
…
auto m = ::max(4, 7.2);          // OK，但回傳型別由第一個引數決定
```

能夠傳入不同型別的參數到 `max()` template 看來很吸引人，但就這個例子而言，它引發了一個問題：如果你用了其中一個型別作為**回傳型別**（return type），無論呼叫者願不願意，另一個代入的引數都可能被轉型成這個型別。如此一來，回傳型別就依賴於 call argument 的順序。66.66 和 42 取最大值得到的結果會是 `double` 66.66，而 42 和 66.66 取最大值則會得到 `int` 66。

C++ 提供了不同方法來解決這個問題：

- 引入第三個 template parameter 作為回傳型別。
- 讓編譯器來決定回傳型別。
- 用兩個參數型別的「**共通型別**（common type）」來宣告回傳型別。

接下來讓我們討論以上幾種方案。

1.3.1 將 Template Parameters 用於回傳型別

先前的討論示範了 *template* 引數推導（*template argument deduction*），讓我們得以用呼叫一般函式的語法（syntax）來呼叫 function templates，我們毋需特別標明每個 template 參數的型別。

我們同時也提到，若想特別標明 template parameters 的型別，也是可行的：

```
template<typename T>
T max (T a, T b);
…
::max<double>(4, 7.2);  // 將 T 實體化為 double
```

萬一當 template 和 call parameters 之間沒有明顯關聯，並且當編譯器無法推導出 template parameters 時，你必須在呼叫時特別標明 template argument。例如，你可以引入第三個 template argument 型別以定義 function template 的回傳型別：

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

然而，template 引數推導並不會考慮回傳值[7]，且 `RT` 並不會出現在函式 call parameters 的型別之中。因此 `RT` 無法被推導出來[8]。

是故，你必須特別標明整個 template argument list（模板引數列表），例如：

```
template<typename T1, typename T2, typename RT>
RT max (T1 a, T2 b);
…
::max<int,double,double>(4, 7.2);    // OK，但很囉嗦
```

到目前為止，我們已經看過特別寫出所有 template arguments 的函式、也見過沒有標明 template arguments 的例子。不過還有一種寫法：我們可以只寫出第一個引數、並讓推導機制決定剩下的部分。一般來說，你必須標明所有無法被隱式推導決定的引數型別。所以如果你調整一下例子裡 template parameters 的順序，呼叫時就只需要標明回傳型別即可：

```
template<typename RT, typename T1, typename T2>
RT max (T1 a, T2 b);
…
::max<double>(4, 7.2);  // OK：回傳型別是 double，T1 和 T2 會被推導出來。
```

在這個例子裡，呼叫 `max<double>` 時明確定義了 `RT` 為 `double`，而參數 `T1` 和 `T2` 會依據引數被推導為 `int` 和 `double`。

---

7 推導可以被視為重載決議機制（overload resolution）的一部分；無論是推導或重載決議，都不會倚賴回傳值來區分不同的呼叫。唯一的例外是轉型運算子成員函式（conversion operator members）。（初版譯註：轉型運算子函式名稱形式為：`operator type()`，其中 type 可為任意型別；無需另外指出回傳型別，因為函式名稱已經表現出回傳型別。）

8 在 C++ 裡，回傳型別同樣無法依據呼叫時的上下文（context）推導出來。

這些 max() 的修正版本並沒有帶來什麼明顯的好處。對於單一參數的這個版本，如果傳入的兩個引數型別不同，你可以指定參數型別（及回傳型別）。因此為保持程式碼簡單易懂，我們在接下來的章節討論其他 template 議題時，預設使用這個單一參數版本。

關於推導過程的細節，詳見第 15 章。

1.3.2 推導回傳型別

如果回傳型別依賴於 template parameters，最簡單也最好的推導方法就是讓編譯器來決定。自 C++14 起，不宣告回傳型別也沒問題（但你仍然要宣告回傳型別為 auto）。

basics/maxauto.hpp

```
template<typename T1, typename T2>
auto max (T1 a, T2 b)
{
  return b < a ? a : b;
}
```

事實上，我們使用 auto 時並沒有加上對應的 *trailing return type*（**後置回傳型別**，會隨著 -> 符號出現在句末），這件事意味著實際的回傳型別必須依靠函式本體（body）裡的回傳陳述句（return statement）推導出來。當然函式本體要真的能推導出回傳型別才行。因此，程式碼必須存在、並且多個回傳陳述句也要一致。

在 C++14 以前，如果想讓編譯器決定回傳型別，基本上只能讓函式實作內容成為函式宣告的一部分。C++11 裡可以藉助以下功能：*trailing return type* 語法允許我們使用 call parameters。也就是說，我們能**宣告**（*declare*）operator?: 產生的結果為回傳型別：

basics/maxdecltype.hpp

```
template<typename T1, typename T2>
auto max (T1 a, T2 b) -> decltype(b<a?a:b)
{
  return b < a ? a : b;
}
```

這裡的回傳型別由運算子 ?: 的規則決定，看來有點複雜，但一般會產生合乎直覺的結果。例如，即使 a 和 b 有著不同的算術型別（arithmetic type），得到的結果也會是一個共通（common）的算術型別。

```
template<typename T1, typename T2>
auto max (T1 a, T2 b) -> decltype(b<a?a:b);
```

注意上式是一份宣告（*declaration*），方便編譯器使用 operator?: 所陳述的規則，在編譯期利用參數 a 和 b 找出 max() 的回傳型別。函式實作部分並不一定要和宣告式長得一樣。事實上，在宣告式裡以 true 作為 operator?: 的條件也行：

```
template<typename T1, typename T2>
auto max (T1 a, T2 b) -> decltype(true?a:b);
```

但不論怎麼看，這份定義都有一個顯著的缺點：回傳型別有可能是一個 reference 型別，因為在某些情況下 T 可能是個 reference。考慮到這點，你應該回傳 T 退化後的型別，看起來會像下面這樣：

basics/maxdecltypedecay.hpp

```
#include <type_traits>

template<typename T1, typename T2>
auto max (T1 a, T2 b) -> typename std::decay<decltype(true?a:b)>::type
{
  return b < a ? a : b;
}
```

在這裡我們使用了 type traits（型別特徵萃取）：std::decay<>，其定義於標準程式庫的 <type_traits> 標頭檔（見附錄 D.4，第 731 頁），它能回傳記錄於成員變數 type 裡的最終型別。因為成員變數 type 是一個型別，取用時必須在陳述式之前加上 typename 關鍵字（見 5.1 節、第 67 頁）。

注意初始化（initialization）型別 auto 的過程總是伴隨著退化。當回傳型別用上了 auto，回傳值同樣會受到影響。下面的程式碼描述了以 auto 作為回傳型別時的行為，在這裡 a 是以 i 的退化後型別（int）來宣告的：

```
int i = 42;
int const& ir = i;       // ir 是 i 的 reference
auto a = ir;             // a 是一個以 int 型別宣告的新物件
```

1.3.3 回傳共通型別

自 C++11 開始，標準程式庫提供了方法以選擇「更加泛用（the more general type）」的型別。std::common_type<>::type 提供了傳入兩個（或兩個以上）不同型別引數時的「共通型別（common type）」。例如：

basics/maxcommon.hpp

```
#include <type_traits>

template<typename T1, typename T2>
std::common_type_t<T1,T2> max (T1 a, T2 b)
{
  return b < a ? a : b;
}
```

std::common_type 又是一個 *type trait*，定義於 <type_traits>。它提供了一個有著 type 成員變數的結構，並藉以取得最終型別。它主要的用法會像這樣：

```
typename std::common_type<T1,T2>::type  // C++11 起接受的寫法
```

然而，自 C++14 起，traits 用起來更簡單了，只要在 trait 名稱後面加上 _t，同時 typename 關鍵字和後面的 ::type 都可以省略（細節詳見 2.8 節，第 40 頁）。現在回傳型別可以簡單這樣定義：

```
std::common_type_t<T1,T2>                    // 與上式相同，C++14 開始可以這樣寫
```

std::common_type<> 實作上使用了一些巧妙的 template 編程技巧，在 26.5.2 節（第 622 頁）會討論到。基本上，它會根據 ?: 的語言規則、或是具體型別的特化（specializations）來選擇最終型別。無論是 ::max(4, 7.2) 還是 ::max(7.2, 4) 都會得出相同的結果：7.2，型別為 double。注意 std::common_type<> 同樣也會退化。更詳細的內容請見附錄 D.5（第 732 頁）。

1.4 Default Template Arguments（預設模板引數）

Template parameter 也可以定義預設值，稱為 *default template arguments*（預設模板引數），能夠在任何一種 template 上使用[9]，甚至可以引用出現過的 template parameters。

舉個例子，如果你想要定義回傳型別，同時支援多個不同型別的參數（像上一節討論的那樣）。你可以引進一個 template parameter RT 作為回傳型別，並設定兩個引數的共通型別作為其預設值。現在我們再次有著不同方案：

1. 直接使用 operator?:。但是 operator?: 必須放在 call parameters a 和 b 之前，故我們無法對 a、b 進行比較，只能使用它們的型別 T1 和 T2：

   basics/maxdefault1.hpp

   ```
   #include <type_traits>

   template<typename T1, typename T2,
            typename RT = std::decay_t<decltype(true ? T1() : T2())>>
   RT max (T1 a, T2 b)
   {
     return b < a ? a : b;
   }
   ```

   再次留意這裡用了 std::decay_t<>，以確保回傳值不是個 reference[10]。

---

9 在 C++11 之前，因為 function templates 演化過程裡的老毛病，default template arguments 只能存在於 class templates 中。

10 再次提醒，於 C++11 必須得用 typename std::decay<...>::type，而不能寫成 std::decay_t<...>（見 2.8 節，第 40 頁）。

同時注意，以上實作要求傳入型別的預設建構子（default constructor）能夠被呼叫。另一種解法是使用 std::declval，可是這麼做會使這條宣告式變得更加複雜。你可以在 11.2.3 節（第 166 頁）找到例子。

2. 亦可使用 type trait：std::common_type<> 來標明預設回傳型別：

basics/maxdefault3.hpp

```
#include <type_traits>

template<typename T1, typename T2,
         typename RT = std::common_type_t<T1,T2>>
RT max (T1 a, T2 b)
{
  return b < a ? a : b;
}
```

再次當心 std::common_type<> 會導致退化，故回傳值不能是 reference。

無論如何，現在呼叫者可以使用回傳型別的預設值了。

```
auto a = ::max(4, 7.2);
```

或是在其他型別引數的後面顯式地（explicitly）加註回傳型別也可以。

```
auto b = ::max<double,int,long double>(7.2, 4);
```

再次遇到老問題，現在就算只想給回傳型別，也得把三個型別都寫出來。其實我們想要的是，當回傳型別放在第一個 template parameter 的情況下，還能夠在需要時從引數型別中推導出回傳型別。原則上，只替第一個 function template parameter 設定預設引數，並忽略後續參數，也是可行的。

```
template<typename RT = long, typename T1, typename T2>
RT max (T1 a, T2 b)
{
  return b < a ? a : b;
}
```

有了以上定義，我們可以這樣呼叫函式：

```
int i;
long l;
…
max(i, l);                  // 回傳 long（template parameter 的預設引數，作為回傳型別）
max<int>(4, 42);            // 在明確要求時，回傳 int
```

然而這個做法只有在 template parameter 擁有「天然的預設型別（natual default）」時可行。這裡我們想要的是讓 template parameter 的預設引數由出現過的 template parameters 決定。原則上做得到，但得倚靠 type traits 技巧並且會讓 traits 定義變得更複雜，在 26.5.1 小節（第 621 頁）有相關的討論。

基於以上原因，最好也最簡單的做法是讓編譯器自行推導出回傳型別，像 1.3.2 節（第 11 頁）敘述的那樣。

1.5 重載 Function Templates

如同一般函式，function templates 也能被**重載**（overloaded）。亦即一個函式名稱可以有好幾份不同的函式定義，當該名稱於函式呼叫被提及時，C++ 編譯器得決定哪一份定義會被喚起。即使不牽涉 templates，該決策機制也相當複雜。這一節，我們會討論使用 templates 時的重載行為。如果你對（沒有 templates 時的）基本重載規則還不是很熟悉，建議你先閱讀附錄 C。我們提供了一份對重載決議機制相當詳盡的講解。

以下的小程式示範了如何重載 function template：

basics/max2.cpp

```
// 取兩個 ints 中較大者：
int max (int a, int b)
{
  return b < a ? a : b;
}

// 取任意兩個相同型別數值中較大者：
template<typename T>
T max (T a, T b)
{
  return b < a ? a : b;
}

int main()
{
  ::max(7, 42);           // 呼叫 nontemplate 版本處理兩個 ints
  ::max(7.0, 42.0);       // 呼叫 max<double>（由引數推導決定）
  ::max('a', 'b');        // 呼叫 max<char>（由引數推導決定）
  ::max<>(7, 42);         // 呼叫 max<int>（由引數推導決定）
  ::max<double>(7, 42); // 呼叫 max<double>（不會發生引數推導）
  ::max('a', 42.7);       // 呼叫 nontemplate 版本處理兩個 ints
}
```

如範例所示，nontemplate function（非模板函式）可以與 function template 同時存在、共享相同的函式名稱、並且以相同型別被實體化*。在其他條件都相同的情況下，比起由 template 衍生的實體，重載決議機制偏好使用 nontemplate 版本。例子裡的第一條呼叫式便符合這個原則：

```
::max(7, 42);    // 具備兩個 int 引數，完全吻合 nontemplate function 宣告式
```

* 譯註：如 nontemplate 版本和 max<int> 版本都接受 int 參數，它們的兩份實體也會同時存在。

如果 template 生成的函式更加合適，則 template 版本會被選中。第二和第三次對 max() 的呼叫示範了此一原則：

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

Template 版本在這裡更合適，因為不需要將 double 或是 char 轉型成 int（相關重載決議機制詳見附錄 C.2，第 682 頁）。

標明空的 template argument list 也是可行的。此語法代表僅有 template 會參與決議過程（意即不使用 nontemplate 版本），不過所有 template parameters 都需要藉由 call arguments 推導出來：

```
::max<>(7, 42);      // 呼叫 max<int>（由引數推導決定）
```

自動型別轉換只作用於普通函式裡的參數，而不會作用於被推導出來的 template parameters。在上例最後一次呼叫時，因為 'a' 和 42.7 都需要轉換為 int，所以使用 nontemplate function：

```
::max('a', 42.7);    // 只有 nontemplate function 允許非直觀的（nontrivial）型別轉換
```

舉個有趣的例子，透過重載 max() template 來做到顯式指定回傳型別：

basics/maxdefault4.hpp

```
template<typename T1, typename T2>
auto max (T1 a, T2 b)
{
  return b < a ? a : b;
}
template<typename RT, typename T1, typename T2>
RT max (T1 a, T2 b)
{
  return b < a ? a : b;
}
```

現在我們可以用以下方式呼叫 max()：

```
auto a = ::max(4, 7.2);                  // 套用第一個 template
auto b = ::max<long double>(7.2, 4);     // 套用第二個 template
```

然而，當我們呼叫下式時：

```
auto c = ::max<int>(4, 7.2);             // 錯誤：兩個 function templates 都符合
```

因為兩個 templates 都與呼叫式匹配，重載決議無所適從，從而導致歧義錯誤（ambiguity error）發生。因此，在重載 function templates 時，應該保證對所有呼叫都僅有一個符合的版本。

下面的例子十分有用，重載 max() template 來處理指標和普通 C-strings*：

basics/max3val.cpp

```
#include <cstring>
#include <string>

// 取任意兩個相同型別中較大者：
template<typename T>
T max (T a, T b)
{
  return b < a ? a : b;
}

// 取兩個指標所指之物較大者：
template<typename T>
T* max (T* a, T* b)
{
  return *b < *a ? a : b;
}

// 取兩個 C-strings 中較大者：
char const* max (char const* a, char const* b)
{
  return std::strcmp(b,a) < 0 ? a : b;
}

int main ()
{
  int a = 7;
  int b = 42;
  auto m1 = ::max(a,b);      // 呼叫接受兩個 ints 型別的 max()

  std::string s1 = "hey";
  std::string s2 = "you";
  auto m2 = ::max(s1,s2);    // 呼叫接受兩個 std::strings 型別的 max()

  int* p1 = &b;
  int* p2 = &a;
  auto m3 = ::max(p1,p2);    // 呼叫接受兩個指標型別的 max()

  char const* x = "hello";
  char const* y = "world";
  auto m4 = ::max(x,y);      // 呼叫接受兩個 C-strings 型別的 max()
}
```

注意對於所有 max() 重載版本，我們都以傳值（call-by-value）方式傳入引數。一般而言，在不同的 function templates 重載版本間最好只存在「必要的差異」**。你應當將不同版本的

* 譯註：C 語言型別字串，即 char*。

** 譯註：同時在其餘非必要的部分讓各版本保持一致。

差異限制在參數數目上、或是顯式標明不同的 template parameters 型別，否則意想不到的副作用將找上門。舉個例子，如果你在實作 `max()` template 時選擇傳入引數的 reference，並且在重載版本裡允許兩個 C-strings 以傳值方式傳入，你將無法利用有著三個引數的重載版本來取得三個 C-strings 中的最大者：

basics/max3ref.cpp

```
#include <cstring>

// 取兩個任意型別數值中較大者（call-by-reference）
template<typename T>
T const& max (T const& a, T const& b)
{
  return b < a ? a : b;
}

// 取兩個 C-strings 中較大者（call-by-value）
char const* max (char const* a, char const* b)
{
  return std::strcmp(b,a) < 0 ? a : b;
}

// 取三個任意型別數值中最大者（call-by-reference）
template<typename T>
T const& max (T const& a, T const& b, T const& c)
{
  return max (max(a,b), c);          // 若 max(a,b) 呼叫傳值版本，會造成錯誤
}

int main ()
{
  auto m1 = ::max(7, 42, 68);        // OK

  char const* s1 = "frederic";
  char const* s2 = "anica";
  char const* s3 = "lucas";
  auto m2 = ::max(s1, s2, s3);       // 執行期錯誤（未定義的行為）
}
```

問題出在如果你用三個 C-strings 呼叫 `max()`，以下述句會導致執行期錯誤：

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

因為對於 C-strings，`max(a,b)` 會*創建一個新的 temporary local value（區域暫存值），並且回傳其 reference。但是該暫存值在 return 述句執行完後馬上就失效了，留給 `main()`

* 譯註：以傳值版本。

的只是一個 dangling reference（懸置參考）。不幸的是，這種錯誤相當難以察覺，在各種情況下都不會輕易現身[11]。

相較之下，在 main() 裡第一次對 max() 的呼叫不會造成以上問題。引數 7、42、和 68 雖然都會創建暫存值，但這些暫存值都是創建於 main() 之中，會一直存活到該陳述句（statement）結束。

這僅僅是複雜的重載決議機制導致非預期行為的眾多例子之一。此外，也請確保所有函式重載版本都在呼叫前被宣告。因為當呼叫對應函式時，若有些重載函式無法在當前範圍內被找到，也會造成一些問題。例如，如果在定義「三個引數」版本的 max() 之前，找不到特別針對「二個 ints 引數」寫的 max() 宣告式版本，就會在使用「三個引數」的 template 版本時，喚起「兩個引數」版本的 template：

basics/max4.cpp

```
#include <iostream>

// 取兩個任意型別中的較大者：
template<typename T>
T max (T a, T b)
{
  std::cout << "max<T>() \n";
  return b < a ? a : b;
}

// 取三個任意型別中的最大者：
template<typename T>
T max (T a, T b, T c)
{
  return max (max(a,b), c); // 即便對於 ints，也喚起 template 版本
}                           // 因為下面的宣告太晚出現了

// 取兩個 ints 型別中的較大者：
int max (int a, int b)
{
  std::cout << "max(int,int) \n";
  return b < a ? a : b;
}

int main()
{
  ::max(47,11,33);       // 唉呀：這裡會喚起 max<T>()，而非 max(int,int)
}
```

我們會在 13.2 節（第 217 頁）討論更多細節。

11 合格的編譯器大都無法阻止這段程式碼通過編譯。

1.6 難道不能這樣寫？

即使像是上面這些 function template 的簡單例子也可能引出更多疑問，不過有三個相當常見的問題，我們得先在這裡簡要討論一下。

1.6.1 傳值與傳參考，哪個好？

你可能會想，為什麼我們通常會以值傳遞引數的方式宣告函式，而不是用傳 references 的方式呢？一般來說，除非是低成本的簡單型別（像是基本型別或是 `std::string_view`），不然傳 reference 通常是比較建議的做法，因為這樣不會額外創造出不必要的**副本物件**（copies）。

然而，因為幾點原因，傳值通常是比較好的做法：

- 語法簡單。
- 編譯器能更好的優化它們。
- 搬移語義經常能降低複製的成本。
- 有時甚至完全沒有複製或搬移發生。

對於 templates，還有以下特定方面的考量：

- Template 可能同時用於簡單和複雜的型別，如果考慮複雜型別而選擇傳 reference，可能會對簡單型別帶來副作用。
- 藉由 `std::ref()` 和 `std::cref()` 的幫助，使用者仍然可以自行選擇在呼叫函式時以 reference 方式傳遞引數（見 7.3 節，第 112 頁）。
- 雖然將 string literals（字串文字）和原始陣列傳入函式通常會造成問題，但傳入它們的 references 被認為會造成更大的問題。

以上幾點會在第 7 章詳細討論。除非某些功能有傳 references 的特別需求，不然本書通常會使用傳值方式傳入引數。

1.6.2 為何不寫 `inline` ？

一般而言，function templates 宣告時不用特別加註 `inline`（內嵌）關鍵字。和一般的 noninline 函式不同，我們可以於標頭檔給出 noninline function templates 的完整定義，並且在多個編譯單元（translations units）裡引用該檔案。

唯一的例外是對特定型別進行 template 全特化（full specializations）時，這樣做的話最終程式碼就會失去泛型特性（因為全部的 template parameters 都被定義好了）。參見 9.2 節（第 140 頁）以獲得更多細節。

根據嚴謹的語言定義，`inline` 關鍵字僅僅代表一份函式定義可以在整個程式裡多次出現。然而它同時也是**對編譯器的一個提示**：最好將對此函式的呼叫替換為「被呼叫函式的程式碼」，意即在該處將程式碼「內嵌展開（expanded inline）」。這樣做在某些情況下可以產生更高

效的程式碼，但也可能造成程式碼在其他狀況下沒有效率。當今的編譯器通常滿擅長在沒有 inline 關鍵字暗示時判斷函式是否內嵌。不過當 inline 出現時，是否生效仍然取決於編譯器的決定。

1.6.3 為何不寫 constexpr ？

自 C++11 起，你可以利用 constexpr（常數陳述）關鍵字寫出在編譯期做計算的程式碼。這點對許多 templates 來講都管用。

例如，要能夠在編譯期利用函式取最大值，你必須用以下方式宣告：

basics/maxconstexpr.hpp

```
template<typename T1, typename T2>
constexpr auto max (T1 a, T2 b)
{
  return b < a ? a : b;
}
```

你可以在有編譯期上下文（compile-time context）的地方使用 max() function templates，像是當你需要宣告原始陣列的大小時：

```
int a[::max(sizeof(char),1000u)];
```

或是宣告 std::array<> 的大小時：

```
std::array<std::string, ::max(sizeof(char),1000u)> arr;
```

注意我們特意用 unsigned int 型別傳入 1000，避免引發在 template 內比較 signed 和 unsigned 數值時的警告訊息。

8.2 節（第 125 頁）會討論其他使用 constexpr 的例子。然而，為了讓我們專注在基本問題上，我們通常會在討論其他 template 特性時忽略 constexpr。

1.7 總結

- Function templates 定義了一整個函式家族，用以處理不同的 template arguments
- 當你將引數傳給依賴於 template parameters 的函式參數時，function templates 為了能被對應的參數型別實體化，會推導該 template parameters。
- 你可以明確指定排在前面的 template parameters。
- 你可以替 template parameters 定義預設引數。這些引數可以參考出現過的 template parameters，後面也可以跟著沒有預設引數的其他參數。
- 你可以重載 function templates。
- 使用 function templates 重載其他 function templates 時，應該確保在任何呼叫處都只有一個符合的 template 版本。

- 重載 function templates 時，盡量保持各版本間僅存在「必要的差異」。
- 在呼叫 function templates 之前，確保編譯器看得到所有重載版本。