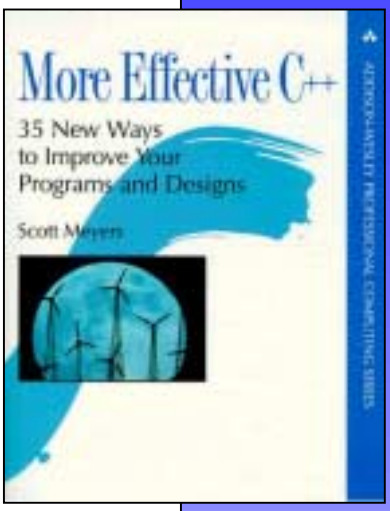
More Effective C++



國際中文版

35 New Ways to Improve Your Programs and Designs

改善程式技術與設計思維的 35 個有效新作法

Scott Meyers 著
侯捷 譯

o. Never treat arrays polymorphically
o. Virtualizing constructors and non-member functions
o. Limiting the number of objects of a class
o. Requiring or prohibiting heap-based objects
o. Smart pointers
o. Reference counting
o. Proxy classes
o. Double dispatching
o. Distinguish between prefix and postfix forms of increment and decrement operators
o. Remember the 80-20 rule
o. Consider using lazy evaluation
o. Understand the origin of temporary objects
o. Facilitate the return value optimization
o. Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI
o. ...

Pearson PTR

More Effective C++ 的榮耀：

這是一本多方面發人深省的 C++ 書籍：不論在你偶爾用到的語言特性上，或是在你自以爲十分熟悉的語言特性上。只有深刻瞭解 C++ 編譯器如何解釋你的碼，你才有可能以 C++ 語言寫出穩健強固的軟體。本書是協助你獲得此等層級之瞭解過程中，一份極具價值的資源。讀過本書之後，我感覺像是瀏覽了 C++ 程式大師所檢閱過的碼，並獲得許多極具價值的洞見。

—Fred Wild, Vice President of Technology,
Advantage Software Technologies

本書內含大量重要的技術，這些技術是撰寫優良 C++ 程式所不可或缺的。本書解釋如何設計和實作這些觀念，以及潛伏在其他某些替代方案中的陷阱。本書亦含晚近加入之 C++ 特性的詳細說明。任何人如果想要好好地運用這些新特性，最好買一本並且放在隨手可得之處，以備查閱。

—Christopher J. Van Wyk, Professor
Mathematics and Computer Science, Drew University

一本具備工業強度的最佳書籍。對於已經閱讀過 *Effective C++* 的人，這是完美的續集。

—Eric Nagler, C++ Instructor and Author,
University of California Santa Cruz Extension

More Effective C++ 是一本無微不至而且價值不扉的書籍，是 Scott 第一本書 *Effective C++* 的續集。我相信每一位專業的 C++ 軟體開發人員都應該讀過並記憶 *Effective C++* 和 *More Effective C++* 兩本書內的各種招式，以及其中重要（並且有時候不可思議）的語言面向。我強烈推薦這兩本書給軟體開發人員、測試人員、管理人員…，每個人都可以從 Scott 專家級的知識與卓越的表達能力中獲益。

—Steve Burkett, Software Consultant

More Effective C++

國際 中文版

改善程式技術與設計思維的 35 個有效新作法

Scott Meyers

侯捷 譯

For Clancy,
my favorite enemy within.

獻給每一位對 C++/OOP 有所渴望的人

正確的觀念、重於一切

－ 侯 捷 －

譯序（侯捷）

C++ 是一個難學易用的語言！

C++ 的難學，不僅在其廣博的語法，以及語法背後的語意，以及語意背後的深層思維，以及深層思維背後的物件模型；C++ 的難學，還在於它提供了四種不同（但相輔相成）的程式設計思維模式：procedural-based, object-based, object-oriented, generic paradigm。

世上沒有白吃的午餐。又要有效率，又要有彈性，又要前瞻望遠，又要回溯相容，又要能治大國，又要能烹小鮮，學習起來當然就不可能太簡單。

在如此龐大複雜的機制下，萬千使用者前仆後續的動力是：一旦學成，妙用無窮。

C++ 相關書籍之多，車載斗量；如天上繁星，如過江之鯽。廣博如四庫全書者有之（*The C++ Programming Language*、*C++ Primer*），深奧如重山複水者有之（The *Annotated C++ Reference Manual*, *Inside the C++ Object Model*），細說歷史者有之（*The Design and Evolution of C++*, *Ruminations on C++*），獨沽一味者有之（*Polymorphism in C++*, *Genericity in C++*），獨樹一幟者有之（*Design Patterns*, *Large Scale C++ Software Design*, *C++ FAQs*），程式庫大全有之（*The C++ Standard Library*），另闢蹊徑者有之（*Generic Programming and the STL*），工程經驗之累積亦有之（*Effective C++*, *More Effective C++*, *Exceptional C++*）。

這其中，「工程經驗之累積」對已具 C++ 相當基礎的程式員而言，有著致命的吸引力與立竿見影的幫助。Scott Meyers 的 *Effective C++* 和 *More Effective C++* 是此類佼佼，Herb Sutter 的 *Exceptional C++* 則是後起之秀。

這類書籍的一個共通特色是輕薄短小，並且高密度地納入作者浸淫於 C++/OOP 領域多年而廣泛的經驗。它們不但開展讀者的視野，也爲讀者提供各種 C++/OOP 常見問題或易犯錯誤的解決模型。某些小範圍主題諸如「在 base classes 中使用 virtual destructor」、「令 `operator=` 傳回 `*this` 的 reference」，可能在百科型 C++ 語言書籍中亦曾概略提過，但此類書籍以深度探索的方式，讓我們瞭解問題背後的成因、最佳的解法、以及其他可能的牽扯。至於大範圍主題，例如 smart pointers, reference counting, proxy classes，double dispatching, 基本上已屬 design patterns 的層級！

這些都是經驗的累積和心血的結晶。

我很高興將以下三本極佳書籍，規劃爲一個系列，以精裝的形式呈現給您：

1. *Effective C++* 2/e, by Scott Meyers, AW 1998
2. *More Effective C++*, by Scott Meyers, AW 1996
3. *Exceptional C++*, by Herb Sutter, AW 1999

不論外裝或內容，中文版比其英文版兄弟毫不遜色。本書不但與原文本頁頁對譯，保留索引，並加上精裝、書籤條、譯註、書籍交叉參考[1]、完整範例碼[2]、讀者服務[3]。

這套書對於您的程式設計生涯，可帶來重大幫助。製作這套書籍使我感覺非常快樂。我祈盼（並相信）您在閱讀此書時擁有同樣的心情。

侯捷 2000/05/15 于新竹.臺灣
`jjhou@ccca.nctu.edu.tw`
`http://www.jjhou.com`

[1] *Effective C++* 2/e 和 *More Effective C++* 之中譯，事實上是以 Scott Meyers 的另一個產品 *Effective C++ CD* 爲本，不僅資料更新，同時亦將 CD 版中兩書之交叉參考保留下來。這可爲讀者帶來旁徵博引時的莫大幫助。

[2] 書中程式多爲片段。我將陸續完成完整的範例程式，並在 Visual C++,C++Builder, GNU C++ 上測試。請至侯捷網站（`http://www.jjhou.com`）下載。

[3] 歡迎讀者對本書範圍所及的主題提出討論，並感謝讀者對本書的任何誤失提出指正。來信請寄侯捷電子信箱（`jjhou@ccca.nctu.edu.tw`）。

目 錄

致謝

中文版 略

譯註：藉此版面提醒讀者，本書之中如果出現「條款 5」這樣的參考指示，指的是本書條款 5。如果出現「條款 E5」這樣的參考指示，E 是指 *Effective C++ 2/e*）

導讀

對 C++ 程式員而言，日子似乎有點過於急促。雖然只商業化不到 10 年，C++ 卻儼然成爲幾乎所有主要電算環境的系統程式語言霸主。面臨程式設計方面極具挑戰性問題的公司和個人，不斷投入 C++ 的懷抱。而那些尚未使用 C++ 的人，最常被詢問的一個問題則是：你打算什麼時候開始用 C++。C++ 標準化已經完成，其所附帶之標準程式庫幅員廣大，不僅涵蓋 C 函式庫，也使之相形見絀。這麼一個大型程式庫使我們有可能在不必犧牲移植性的情況下，或是在不必從頭撰寫常用演算法和資料結構的情況下，完成琳琅滿目的各種複雜程式。C++ 編譯器的數量不斷增加，它們所供應的語言性質不斷擴充，它們所產生的碼品質也不斷改善。C++ 開發工具和開發環境愈來愈豐富，威力愈來愈強大，穩健強固（robust）的程度愈來愈高。商業化程式庫幾乎能夠滿足各個應用領域中的寫碼需求。

一旦語言進入成熟期，而我們對它的使用經驗也愈來愈多，我們所需要的資訊也就隨之改變。1990 年人們想知道 C++ 是什麼東西。到了 1992 年，他們想知道如何運用它。如今 C++ 程式員問的問題更高級：我如何能夠設計出適應未來需求的軟體？我如何能夠改善程式碼的效率而不折損正確性和易用性？我如何能夠實作出語言未能直接支援的精巧機能？

這本書中我要回答這些問題，以及其他許多類似問題。

本書告訴你如何更具實效地設計並實作 C++ 軟體：讓它行爲更正確；面對異常情況時更穩健強固；更有效率；更具移植性；將語言特性發揮得更好；更優雅地調整適應；在「混合語言」開發環境中運作更好；更容易被正確運用；更不容易被誤用。簡單地說就是如何讓軟體更好。

本書內容分爲 35 個條款。每個條款都在特定主題上精簡摘要出 C++ 程式設計社群所累積的智慧。大部份條款以準則的型式呈現，附隨的說明則闡述這條準則爲什麼存在，如果不遵循會發生什麼後果，以及什麼情況下可以合理違反該準則。

所有條款被我分爲數大類。某些條款關心特定的語言性質，特別是你可能罕有使用經驗的一些新性質。例如條款 9~15 專注於 exceptions（就像 Tom Cargill, Jack Reeves, Herb Sutter 所發表的那些雜誌文章一樣）。其他條款解釋如何結合語言的不同特性以達成更高階目標。例如條款 25~31 描述如何限制物件的個數或誕生地點，如何根據一個以上的物件型別產生出類似虛擬函式的東西，如何產生 smart pointers 等等。其他條款解決更廣泛的題目。條款 16~24 專注於效率上的議題。不論哪一條款，提供的都是與其主題相關且意義重大的作法。在 *More Effective C++* 一書中你將學習到如何更實效更精銳地使用 C++。大部份 C++ 教科書中對語言性質的大量描述，只能算是本書的一個背景資訊而已。

這種處理方式意味，你應該在閱讀本書之前便熟悉 C++。我假設你已瞭解類別（classes）、保護層級（protection levels）、虛擬函式、非虛擬函式，我也假設你已通曉 templates 和 exceptions 背後的概念。我並不期望你是一位語言專家，所以涉及較罕見的 C++ 特性時，我會進一步做解釋。

本書所談的 C++

我在本書所談、所用的 C++，是 ISO/ANSI 標準委員會於 1997 年 11 月完成的 C++ 國際標準最後草案（Final Draft International Standard）。這暗示了我所使用的某些語言特性可能並不在你的編譯器(s) 支援能力之列。別擔心，我認爲對你而言唯一所謂「新」特性，應該只有 templates，而 templates 如今幾乎已是各家編譯器的必備機能。我也運用 exceptions，並大量集中於條款 9~15。如果你的編譯器(s) 未能支援 exceptions，沒什麼大不了，這並不影響本書其他部份帶給你的好處。但是，聽我說，縱使你不需用到 exceptions，亦應閱讀條款 9~15，因爲那些條款（及其相關篇幅）檢驗了某些不論什麼場合下你都應該瞭解的主題。

我承認，就算標準委員會授意某一語言特性或是贊同某一實務作法，並非就保證該語言特性已出現在目前的編譯器上，或該實務作法已可應用於既有的開發環境

上。一旦面對「標準委員會所議之理論」和「真正能夠有效運作之實務」間的矛盾，我便兩者都加以討論，雖然我其實比較更重視實務。由於兩者我都討論，所以當你的編譯器(s) 和 C++ 標準不一致時，本書可以協助你，告訴你如何使用目前既有的架構來模擬編譯器(s) 尚未支援的語言特性。而當你決定將一些原本繞道而行的解決辦法以新支援的語言特性取代時，本書亦可引導你。

注意當我說到編譯器(s) 時，我使用複數。不同的編譯器對 C++ 標準的滿足程度各不相同，所以我鼓勵你在至少兩種編譯器(s) 平台上發展程式碼。這麼做可以幫助你避免不經意地依賴某個編譯器專屬的語言延伸性質，或是誤用某個編譯器對標準規格的錯誤闡示。這也可以幫助你避免使用過度先進的編譯器技術，例如獨家廠商才做得出的某種語言新特性。如此特性往往實作不夠精良（臭蟲多，要不就是表現遲緩，或是兩者兼具），而且 C++ 社群往往對這些特性缺乏使用經驗，無法給你應用上的忠告。雷霆萬鈞之勢固然令人興奮，但當你的目標是要產出可靠的碼，恐怕還是步步爲營（並且能夠與人合作）得好。

本書用了兩個你可能不甚熟悉的 C++ 性質，它們都是晚近才加入 C++ 標準之中。某些編譯器支援它們，但如果你的編譯器不支援，你可輕易以你所熟悉的其他性質來模擬它們。

第一個性質是型別 `bool`，其值必爲關鍵字 `true` 或 `false`。如果你的編譯器尚未支援 `bool`，有兩個方法可以模擬它。第一個方法是使用一個 global `enum`：

```
enum bool { false, true };
```

這允許你將參數爲 `bool` 或 `int` 的不同函式加以多載化（overloading）。缺點是，內建的「比較運算子（comparison operators）」如 `==, <, >=,` 等等仍舊傳回 `int`s。所以以下程式碼的行爲不如我們所預期：

```
void f(int);
void f(bool);
int x, y;
...
f( x < y );      // 呼叫 f(int)，但其實它應該呼叫 f(bool)
```

一旦你改用真正支援 `bool` 的編譯器，這種 enum 近似法可能會造成程式行爲的改變。

另一種作法是利用 typedef 來定義 bool，並以常數物件做爲 true 和 false：

```
typedef int bool;
const bool false = 0;
const bool true = 1;
```

這種手法相容於傳統的 C/C++ 語意。使用這種模擬法的程式，在移植到一個支援有 bool 型別的編譯器平台之後，行爲並不會改變。缺點則是無法在函式多載化（overloading）時區分 bool 和 int。以上兩種近似法都有道理，請選擇最適合你的一種。

第二個新性質，其實是四個轉型運算子：static_cast, const_cast, dynamic_cast, 和 reinterpret_cast。如果你不熟悉這些轉型運算子，請翻到條款 2 仔細閱讀其中內容。它們不只比它們所取代的 C 舊式轉型做得更多，也更好。書中任何時候當我需要執行轉型動作，我都使用新式的轉型運算子。

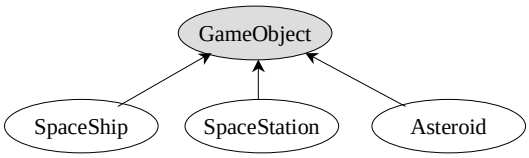
C++ 擁有比語言本身更豐富的東西。是的，C++ 還有一個偉大的標準程式庫（見條款 E49）。我儘可能使用標準程式庫所提供的 string 型別來取代 char* 指標，而且我也鼓勵你這麼做。string objects 並不比 char*-based 字串難操作，它們的好處是可以免除你大部份的記憶體管理工作。而且如果發生 exception 的話（見條款 9 和 10），string objects 比較沒有 memory leaks（記憶體遺失）的問題。實作良好的 string 型別甚至可和對應的 char* 比賽效率，而且可能會贏（條款 29 會告訴你箇中故事）。如果你不打算使用標準的 string 型別，你當然會使用類似 string 的其他 classes，是吧？是的，用它，因爲任何東西都比直接使用 char* 來得好。

我將儘可能使用標準程式庫提供的資料結構。這些資料結構來自 Standard Template Library（"STL" — 見條款 35）。STL 包含 bitsets, vectors, lists, queues, stacks, maps, sets, 以及更多東西，你應該儘量使用這些標準化的資料結構，不要情不自禁地想寫一個自己的版本。你的編譯器或許沒有附 STL 給你，但不要因爲這樣就不使用它。感謝 Silicon Graphics 公司的熱心，你可以從 SGI STL 網站下載一份免費產品，它可以和多種編譯器搭配。

如果你目前正在使用一個內含各種演算法和資料結構的程式庫，而且用得相當愉快，那麼就沒有必要只爲了「標準」兩個字而改用 STL。然而如果你在「使用 STL」和「自行撰寫同等功能的碼」之間可以選擇，你應該讓自己傾向使用 STL。記得程式碼的重用性嗎？STL（以及標準程式庫的其他組件）之中有許多碼是十分值得重複運用的。

慣例與術語

任何時候如果我談到 inheritance（繼承），我的意思是 public inheritance（見條款 E35）。如果我不是指 public inheritance，我會明白地指明。繪製繼承體系圖時，我對 base-derived 關係的描述方式，是從 derived classes 往 base classes 畫箭頭。例如，下面是條款 31 的一張繼承體系圖：



這樣的表現方式和我在 *Effective C++* 第一版（注意，不是第二版）所採用的習慣不同。現在我決定使用這種最廣被接受的繼承箭頭畫法：從 derived classes 畫往 base classes，而且我很高興事情終能歸於一統。此類示意圖中，抽象類別（abstract classes，例如上圖的 `GameObject`）被我加上陰影而具象類別（concrete classes，例如上圖的 `SpaceShip`）未加陰影。

Inheritance（繼承機制）會引發「pointers（或 references）擁有兩個不同型別」的議題，兩個型別分別是靜態型別（static type）和動態型別（dynamic type）。Pointer 或 reference 的「靜態型別」是指其宣告時的型別，「動態型別」則由它們實際所指的物件來決定。下面是根據上圖所寫的一個例子：

```
GameObject *pgo =              // pgo 的靜態型別是 GameObject*，
  new SpaceShip;               // 動態型別是 SpaceShip*
Asteroid *pa = new Asteroid;   // pa 的靜態型別是 Asteroid*，
                               // 動態型別也是 Asteroid*。
pgo = pa;                  // pgo的靜態型別仍然（永遠）是 GameObject*，
                           // 至於其動態型別如今是 Asteroid*。
```

```
GameObject& rgo = *pa;          // rgo 的靜態型別是 GameObject，
                                // 動態型別是 Asteroid。
```

這些例子也示範了我喜歡的一種命名方式。pgo 是一個 pointer-to-GameObject；pa 是一個 pointer-to-Asteroid；rgo 是一個 reference-to-GameObject。我常常以此方式來爲 pointer 和 reference 命名。

我很喜歡兩個參數名稱：lhs 和 rhs，它們分別是 "left-hand side" 和 "right-hand side" 的縮寫。爲了瞭解這些名稱背後的基本原理，請考慮一個用來表示分數（rational numbers）的 class：

```
class Rational { ... };
```

如果我想要一個用以比較兩個 Rational objects 的函式，我可能會這樣宣告：

```
bool operator==(const Rational& lhs, const Rational& rhs);
```

這使我得以寫出這樣的碼：

```
Rational r1, r2;
...
if (r1 == r2) ...
```

在呼叫 operator== 的過程中，r1 位於 "==" 左側，被繫結於 lhs，r2 位於 "==" 右側，被繫結於 rhs。

我使用的其他縮寫名稱還包括：ctor 代表 "constructor"，dtor 代表 "destructor"，RTTI 代表 C++ 對 runtime type identification 的支援（在此性質中，dynamic_cast 是最常被使用的一個零組件）。

當你配置記憶體而沒有釋放它，你就有了 memory leak（記憶體遺失）問題。Memory leaks 在 C 和 C++ 中都有，但是在 C++ 中，memory leaks 所遺失的還不只是記憶體，因爲 C++ 會在物件被產生時，自動呼叫 constructors，而 constructors 本身可能亦配有資源（resources）。舉個例子，考慮以下程式碼：

```
class Widget { ... };           // 某個 class — 它是什麼並不重要。
Widget *pw = new Widget;        // 動態配置一個 Widget 物件。
...                             // 假設 pw 一直未被刪除（deleted）。
```

這段碼會遺失記憶體，因爲 pw 所指的 Widget 物件從未被刪除。如果 Widget constructor 配置了其他資源（例如 file descriptors, semaphores, window handles,

database locks），這些資源原本應該在 `Widget` 物件被摧毀時釋放，現在也像記憶體一樣都遺失掉了。爲了強調在 C++ 中 memory leaks 往往也會遺失其他資源，我在書中常以 resource leaks 一詞取代 memory leaks。

你不會在本書中看到許多 inline 函式。並不是我不喜歡 inlining，事實上我相信 inline 函式是 C++ 的一項重要性質。然而決定一個函式是否應被 inlined，條件十分複雜、敏感、而且與平台有關（見條款 E33）。所以我儘量避免 inlining，除非其中有個關鍵點非使用 inlining 不可。當你在本書之中看到一個 non-inline 函式，並不意味我認爲把它宣告爲 `inline` 是個壞主意，而只是說，它「是否爲 inline」與當時討論的主題無關。

有一些傳統的 C++ 性質已明白地被標準委員會排除。這樣的性質被明列於語言的最後撤除名單，因爲新性質已經加入，取代那些傳統性質的原本工作，而且做得更好。這本書中我會檢視被撤除的性質，並說明其取代者。你應該避免使用被撤除的性質，但是過度在意倒亦不必，因爲編譯器廠商爲了挽留其客戶，會盡力保存回溯相容性，所以那些被撤除的性質大約還會存活好多年。

所謂 `client`，是指你所寫的程式碼的客戶。或許是某些人（程式員），或許是某些物（classes 或 functions）。舉個例子，如果你寫了一個 `Date` class（用來表現生日、最後期限、耶穌再次降臨日等等），任何使用了這個 class 的人，便是你的 client。任何一段使用了 `Date` class 的碼，也是你的 clients。Clients 是重要的。事實上 clients 是遊戲的主角。如果沒有人使用你寫的軟體，你又何必寫它呢？你會發現我很在意如何讓 clients 更輕鬆，通常這會導至你的行事更困難，因爲好的軟體「以客爲尊」。如果你譏笑我太過濫情，不妨反躬自省一下。你曾經使用過自己寫的 classes 或 functions 嗎？如果是，你就是你自己的 client，所以讓 clients 更輕鬆，其實就是讓自己更輕鬆，利人利己。

當我討論 class template 或 function templates 以及由它們所產生出來的 classes 或 functions 時，請容我保留偷懶的權利，不一一寫出 templates 和其 instantiations（具現體）之間的差異。舉個例子，如果 `Array` 是個 class template，有個型別參數 `T`，我可能會以 `Array` 代表此 template 的某個特定具現體（instantiation），

雖然其實 `Array<T>` 才是正式的 class 名稱。同樣道理，如果 `swap` 是個 function template，有個型別參數 `T`，我可能會以 `swap` 而非 `swap<T>` 表示其具現體。如果這樣的簡短表示法在當時情況下不夠清楚，我便會在表示 template 具現體時加上 template 參數。

臭蟲報告，意見提供，內容更新

我盡力讓這本書技術精準、可讀性高，而且有用，但是我知道一定仍有改善空間。如果你發現任何錯誤 — 技術性的、語言上的、錯別字、或任何其他東西 — 請告訴我。我會試著在本書新刷中修正之。如果你是第一位告訴我的人，我會很高興將你的大名登錄到本書致謝文（acknowledgments）內。如果你有改善建議，我也非常歡迎。

我將繼續收集 C++ 程式設計的實效準則。如果你有任何這方面的想法並願意與我分享，我會十分高興。請將你的建議、你的見解、你的批評、以及你的臭蟲報告，寄至：

```
Scott Meyers
c/o Editor-in-Chief, Corporate and Professional Publishing
Addison-Wesley Publishing Company
1 Jacob Way
Reading, MA 01867
U. S. A.
```

或者你也可以送電子郵件到 `mec++@awl.com`。

我維護有一份本書第一刷以來的修訂記錄，其中包括錯誤修正、文字修潤、以及技術更新。你可以從本書網站取得這份記錄，以及其他與本書相關的資訊。你也可以透過 anonymous FTP，從 `ftp.awl.com` 的 `cp/mec++` 目錄中取得它。如果你希望擁有這份資料，但無法上網，請寄申請函到上述地址，我會郵寄一份給你。

這篇序文有夠長的，讓我們開始正題吧。

基礎議題

基礎議題。是的，pointers（指標）、references（物件化身）、casts（型別轉換）、arrays（陣列）、constructors（建構式）— 再沒有比這些更基礎的議題了。幾乎最簡單的 C++ 程式也會用到其中大部份特性，而許多程式會用到上述所有特性。

儘管你可能已經十分熟悉語言的這一部份，有時候它們還是會令你吃驚。特別是對那些從 C 轉戰到 C++ 的程式員，因爲 references, dynamic casts, default constructors 及其他 non-C 性質背後的觀念，往往帶有一股黝暗陰鬱的色彩。

這一章描述 pointers 和 references 的差異，並告訴你它們的適當使用時機。本章介紹新的 C++ 轉型（casts）語法，並解釋爲什麼新式轉型法比舊式的 C 轉型法優越。本章也檢驗 C 的陣列概念以及 C++ 的多型（polymorphism）概念，並說明爲什麼將這兩者混合運用是不智之舉。最後，本章討論 default constructors（預設建構式）的正方和反方意見，並提出一些建議作法，讓你迴避語言的束縛（因爲在你不需 default constructors 的情況下，C++ 也會給你一個）。

只要留心下面各條款的各項忠告，你將向著一個很好的目標邁進：你所生產的軟體可以清楚而正確地表現出你的設計意圖。

條款 1：仔細區別 pointers 和 references

Pointers 和 references 看起來很不一樣（pointers 使用 "*" 和 "->" 運算子，references 則是使用 "."），但它們似乎做類似的事情。不論 pointers 或是 references 都讓你得以間接參考到其他物件。那麼，何時使用哪一個？你心中可有一把尺？

首先你必須認知一點，沒有所謂的 null reference。一個 reference 必須總是代表某

個物件。所以如果你有一個變數，其目的是用來指向（代表）另一個物件，但是也有可能它不指向（代表）任何物件，那麼你應該使用 pointer，因爲你可以將 pointer 設爲 null。換個角度看，如果這個變數總是必須代表一個物件，也就是說如果你的設計並不允許這個變數爲 null，那麼你應該使用 reference。

『但是等等』你說，『下面這樣的東西，底層意義是什麼呢？』

```
char *pc = 0;        // 將 pointer 設定爲 null
char& rc = *pc;      // 讓 reference 代表 null pointer 的提領值
```

唔，這是有害的行爲，其結果無可預期（C++ 對此沒有定義），編譯器可以產生任何可能的輸出，而寫出這種程式碼的人，應該與大眾隔離，直到他們允諾不再有類似行爲。如果你在你的軟體中還需擔心這類事情，我建議你還是完全不要使用 references 的好，要不就是另請一個比較高明的程式員來負責這類事情。從現在起，我們將永遠不再考慮「reference 成爲 null」的可能性。

由於 reference 一定得代表某個物件，C++ 因此要求 references 必須有初值：

```
string& rs;               // 錯誤！references 必須被初始化
string s("xyzzy");
string& rs = s;           // 沒問題，rs 指向 s
```

但是 pointers 就沒有這樣的限制：

```
string *ps;            // 未初始化的指標，有效，但風險高
```

「沒有所謂的 null reference」這個事實意味使用 references 可能會比使用 pointers 更富效率。這是因爲使用 reference 之前不需測試其有效性：

```
void printDouble(const double& rd)
{
   cout << rd;     // 不需測試 rd；它一定代表某個 double
}
```

如果使用 pointers，通常就得測試它是否爲 null：

```
void printDouble(const double *pd)
{
  if (pd) {      // 檢查是否爲 null pointer
    cout << *pd;
  }
}
```

Pointers 和 references 之間的另一個重要差異就是，pointers 可以被重新設值，指向另一個物件，reference 卻總是指向它最初獲得的那個物件：

```
string s1("Nancy");
string s2("Clancy");

string& rs = s1;          // rs 代表 s1
string *ps = &s1;         // ps 指向 s1
rs = s2;                  // rs 仍然代表 s1，
                          // 但是 s1 的值現在變成了 "Clancy"。
ps = &s2;                 // ps 現在指向 s2；
                          // s1 沒有變化。
```

一般而言，當你需要考慮「不指向任何物件」的可能性時，或是考慮「在不同時間指向不同物件」的能力時，你就應該採用 pointer。前一種情況你可以將 pointer 設爲 null，後一種情況你可以改變 pointer 所指對象。而當你確定「總是會代表某個物件」，而且「一旦代表了該物件就不再能夠改變」，那麼你應該選用 reference。

還有其他情況也需要使用 reference，例如當你實作某些運算子的時候。最常見的例子就是 `operator[]`。這個運算子很特別地必須傳回某種「能夠被當做 assignment 設值對象」的東西：

```
vector<int> v(10);   // 產生一個 int vector，大小爲 10；
                     // vector 是 C++ 標準程式庫（見條款 35）
                     // 提供的一個 template。
v[5] = 10;           // assignment 的設值對象是 operator[] 的傳回值
```

如果 `operator[]` 傳回 pointer，上述最後一個句子就必須寫成這樣子：

```
*v[5] = 10;
```

但這使 `v` 看起來好像是個以指標形成的 `vector`，事實上它不是。爲了這個因素，你應該總是令 `operator[]` 傳回一個 reference。條款 30 有一個例外，十分有趣。

因此，讓我做下結論：當你知道你需要指向某個東西，而且絕不會改指向其他東西，或是當你實作一個運算子而其語法需求無法由 pointers 達成，你就應該選擇 references。任何其他時候，請採用 pointers。

條款 2：最好使用 C++ 轉型運算子

想想低階轉型動作。它幾乎像 `goto` 一樣地被視爲程式設計上的賤民。儘管如此，它卻仍能夠苟延殘喘，因爲當某種情況愈來愈糟，轉型可能是必要的。是的，當某種情況愈來愈糟，轉型是必要的。

不過，舊式的 C 轉型方式並非是唯一選擇。它幾乎允許你將任何型別轉換爲任何其他型別，這是十分拙劣的。如果每次轉型都能夠更精確地指明意圖，更好。舉個例子，將一個 pointer-to-const-object 轉型爲一個 pointer-to-non-const-object（也就是說只改變物件的常數性），和將一個 pointer-to-base-class-object 轉型爲一個 pointer-to-derived-class-object（也就是完全改變了一個物件的型別），其間有很大的差異。傳統的 C 轉型動作對此並無區分（這應該不會造成你的驚訝，因爲 C 式轉型是爲 C 設計的，不是爲了 C++）。

舊式轉型的第二個問題是它們難以辨識。舊式轉型的語法結構是由一對小括弧加上一個物件名稱（辨識符號）組成，而小括弧和物件名稱在 C++ 的任何地方都有可能被使用。因此我們簡直無法回答最基本的轉型相關問題：「這個程式中有使用任何轉型動作嗎？」。因爲人們很可能對轉型動作視而不見，而諸如 `grep` 之類的工具又無法區分語法上極類似的一些非轉型寫法。

爲解決 C 舊式轉型的缺點，C++ 導入四個新的轉型運算子（cast operators）：`static_cast`, `const_cast`, `dynamic_cast` 和 `reinterpret_cast`。以大部份使用目的而言，面對這些運算子你唯一需要知道的便是，過去習慣的寫碼型式：

```
(type) expression
```

現在應該改爲這樣：

```
static_cast<type>(expression)
```

舉個例子，假設你想要將一個 `int` 轉型爲一個 `double` 以強迫一個整數運算式導出一個浮點數值出來。採用 C 舊式轉型，可以這麼做：

```
int firstNumber, secondNumber;
...
double result = ((double)firstNumber)/secondNumber;
```

如果採用新的 C++ 轉型法，應該這麼寫：

```
double result = static_cast<double>(firstNumber)/secondNumber;
```

這種型式十分容易被辨識出來，不論是對人類或是對工具程式而言。

static_cast 基本上擁有與 C 舊式轉型相同的威力與意義，以及相同的限制。例如你不能夠利用 static_cast 將一個 struct 轉型爲 int 或將一個 double 轉型爲 pointer；這些都是 C 舊式轉型動作原本就不可以完成的任務。static_cast 甚至不能夠移除算式的常數性（constness），因爲有一個新式轉型運算子 const_cast 專司此職。

其他新式 C++ 轉型運算子使用於更集中（範圍更狹窄）的目的。const_cast 用來改變算式中的常數性（constness）或變易性（volatileness）。使用 const_cast，便是對人類（以及編譯器）強調 ，透過這個轉型運算子，你唯一打算改變的是某物的常數性或變易性。這項意願將由編譯器貫徹執行。如果你將 const_cast 應用於上述以外的用途，那麼轉型動作會被拒絕。下面是個例子：

```
class Widget { ... };
class SpecialWidget: public Widget { ... };
void update(SpecialWidget *psw);
SpecialWidget sw;               // sw 是個 non-const 物件，
const SpecialWidget& csw = sw;  // csw 卻是一個代表 sw 的 reference，
                                // 並視之爲一個 const 物件。

update(&csw);        // 錯誤！不能將 const SpecialWidget*
                     // 傳給一個需要 SpecialWidget* 的函式。

update(const_cast<SpecialWidget*>(&csw));
                     // 可！&csw 的常數性被明白去除了。也因此，
                     // csw（亦即 sw）在此函式中可被更改。

update((SpecialWidget*)&csw);
                     // 情況同上，但使用的是較難辨識
                     // 的 C 舊式轉型語法。

Widget *pw = new SpecialWidget;
update(pw);          // 錯誤！pw 的型別是 Widget*，但
                     // update() 需要的卻是 SpecialWidget*。

update(const_cast<SpecialWidget*>(pw));
                     // 錯誤！const_cast 只能用來影響
                     // 常數性或變易性，無法進行繼承體系
                     // 的向下轉型（cast down）動作。
```

顯然，const_cast 最常見的用途就是將某個物件的常數性去除掉。

第二個特殊化的轉型運算子 dynamic_cast，用來執行繼承體系中「安全的向下轉型或跨系轉型動作」。也就是說你可以利用 dynamic_cast，將「指向 base class objects 之 pointers 或 references」轉型爲「指向 derived（或 sibling base）class objects 之 pointers 或 references」，並得知轉型是否成功[1]。如果轉型失敗，會以一個 null 指標（當轉型對象是指標）或一個 exception（當轉型對象是 reference）表現出來：

```
Widget *pw;
...
update(dynamic_cast<SpecialWidget*>(pw));
                    // 很好，傳給 update() 一個指標，指向
                    // pw 所指之 SpecialWidget — 如果 pw
                    // 真的指向這樣的東西；否則傳過去的
                    // 將是一個 null 指標。

void updateViaRef(SpecialWidget& rsw);

updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
                    // 很好，傳給 updateViaRef() 的是
                    // pw 所指的 SpecialWidget — 如果
                    // pw 真的指向這樣的東西；否則
                    // 丟出一個 exception。
```

dynamic_cast 只能用來協助你曳航於繼承體系之中。它無法應用在缺乏虛擬函式（請看條款 24）的型別身上，也不能改變型別的常數性（constness）：

```
int firstNumber, secondNumber;
...
double result =
dynamic_cast<double>(firstNumber)/secondNumber;
                    // 錯誤！未涉及繼承機制
const SpecialWidget sw;
...
update(dynamic_cast<SpecialWidget*>(&sw));
                    // 錯誤！dynamic_cast 不能改變常數性
```

如果你想爲一個不涉及繼承機制的型別執行轉型動作，可使用 static_cast；要改變常數性（constness），則必須使用 const_cast。

最後一個轉型運算子是 reinterpret_cast。這個運算子的轉換結果幾乎總是與編譯平台息息相關。所以 reinterpret_casts 不具移植性。

[1] dynamic_cast 的第二個（與第一個不相干）的用途是找出被某物件佔用的記憶體的起始點。我將在條款 27 解釋這項能力。

reinterpret_cast 的最常用途是轉換「函式指標」型別。假設有一個陣列，內放的都是函式指標，有特定的型別：

```
typedef void (*FuncPtr)();     // FuncPtr 是個指標，指向某個函式；
                               // 後者無需任何引數，傳回值爲 void。
FuncPtr funcPtrArray[10];      // funcPtrArray 是個陣列，
                               // 內有 10 個 FuncPtrs
```

假設由於某種原因，你希望將以下函式的一個指標放進 funcPtrArray 中：

```
int doSomething();
```

如果沒有轉型，不可能辦到這一點，因爲 doSomething 的型別與 funcPtrArray 所能接受的不同。funcPtrArray 內各函式指標所指之函式的傳回值是 void，但 doSomething 的傳回值卻是 int：

```
funcPtrArray[0] = &doSomething;     // 錯誤！型別不符
```

使用 reinterpret_cast，可以強迫編譯器瞭解你的意圖：

```
funcPtrArray[0] =                   // 這樣便可通過編譯
  reinterpret_cast<FuncPtr>(&doSomething);
```

函式指標的轉型動作，並不具移植性（C++ 不保證所有的函式指標都能以此方式重新呈現），某些情況下這樣的轉型可能會導至不正確的結果（見項目 31），所以你應該儘量避免將函式指標轉型，除非你已走投無路，像是被逼到牆角，而且有一隻刀子抵住你的喉嚨。一隻銳利的刀子，呃，非常銳利的刀子。

如果你的編譯器尚未支援這些新式轉型動作，你可以使用傳統轉型方式來取代 static_cast, const_cast 和 reinterpret_cast。甚至可以利用巨集（macros）來模擬這些新語法：

```
#define static_cast(TYPE,EXPR)      ((TYPE)(EXPR))
#define const_cast(TYPE,EXPR)       ((TYPE)(EXPR))
#define reinterpret_cast(TYPE,EXPR)  ((TYPE)(EXPR))
```

上述新語法的使用方式如下：

```
double result = static_cast(double, firstNumber)/secondNumber;
update(const_cast(SpecialWidget*, &sw));
funcPtrArray[0] = reinterpret_cast(FuncPtr, &doSomething);
```

這些近似法當然不像其本尊那麼安全，但如果你現在就使用它們，一旦你的編譯器開始支援新式轉型，程式升級的過程便可簡化。

至於 dynamic_cast，沒有什麼簡單方法可以模擬其行爲，不過許多程式庫提供了一些函式，用來執行繼承體系下的安全轉型動作。如果你手上沒有這些函式，而卻必須執行這類轉型，你也可以回頭使用舊式的 C 轉型語法，但它們不可能告訴你轉型是否成功。當然，你也可以定義一個巨集，看起來像 dynamic_cast，就像你爲其他轉型運算子所做的那樣：

```
#define dynamic_cast(TYPE,EXPR)    (TYPE)(EXPR)
```

這個近似法並非執行真正的 dynamic_cast，所以它無法告訴你轉型是否成功。

我知道，我知道，這些新式轉型運算子看起來又臭又長。如果你實在看它們不順眼，值得安慰的是 C 舊式轉型語法仍然繼續可用。然而這麼一來也就喪失了新式轉型運算子所提供的嚴謹意義與易辨識度。如果你在程式中使用新式轉型法，比較容易被解析（不論是對人類或是對工具而言），編譯器也因此得以診斷轉型錯誤（那是舊式轉型法偵測不到的）。這些都是促使我們捨棄 C 舊式轉型語法的重要因素。至於可能的第三個因素是：讓轉型動作既醜陋又不易鍵入（typing），或許未嘗不是件好事。

條款 3：絕對不要以多型（polymorphically）方式處理陣列

繼承（inheritance）的最重要性質之一就是：你可以透過「指向 base class objects」的 pointers 或 references，來操作 derived class objects。如此的 pointers 和 references，我們說其行爲是多型（polymorphically）— 猶如它們有多重型別似地。C++ 也允許你透過 base class 的 pointers 和 references 來操作「derived class objects 所形成的陣列」。但這一點也不值得沾沾自喜，因爲它幾乎絕不會如你所預期般地運作。

舉個例子，假設你有一個 class BST（意思是 binary search tree）以及一個繼承自 BST 的 class BalancedBST：

```
class BST { ... };
class BalancedBST: public BST { ... };
```

在一個真正具規模的程式中，這樣的 classes 可能會被設計爲 templates，不過這不是此處重點；如果加上 template 各種語法，反而使程式更難閱讀。針對目前的討論，我假設 BST 和 BalancedBST 都只內含 ints。

現在考慮有個函式，用來列印 BSTs 陣列中的每一個 BST 的內容：

```
void printBSTArray(ostream& s, const BST array[], int numElements)
{
  for (int i = 0; i < numElements; ++i) {
    s << array[i];             // 假設 BST objects 有一個
  }                            // operator<< 可用。
}
```

當你將一個由 BST 物件組成的陣列傳給此函式，沒問題：

```
BST BSTArray[10];
...
printBSTArray(cout, BSTArray, 10);         // 運作良好
```

然而如果你將一個 BalancedBST 物件所組成的陣列交給 printBSTArray 函式，會發生什麼事：

```
BalancedBST bBSTArray[10];
...
printBSTArray(cout, bBSTArray, 10);        // 可以正常運作嗎？
```

你的編譯器會毫無怨言地接受它，但是看看這個迴圈（就是稍早出現的那一個）：

```
for (int i = 0; i < numElements; ++i) {
  s << array[i];
}
```

array[i] 其實是一個「指標算術運算式」的簡寫：它代表的其實是 *(array+i)。我們知道，array 是個指標，指向陣列起始處。array 所指記憶體和 array+i 所指記憶體兩者相距多遠？答案是 i*sizeof(陣列中的物件)，因爲 array[0] 和 array[i] 之間有 i 個物件。爲了讓編譯器所產生的碼能夠正確走訪整個陣列，編譯器必須有能力決定陣列中的物件大小。很容易呀，參數 array 不是被宣告爲「型別爲 BST」的陣列嗎？所以陣列中的每個元素必然都是 BST 物件，所以 array 和 array+i 之間的距離一定是 i*sizeof(BST)。

至少你的編譯器是這麼想的。但如果你交給 printBSTArray 函式一個由 BalancedBST 物件組成的陣列，你的編譯器就會被誤導。這種情況下它仍假設陣列中每一元素的大小是 BST 的大小，但其實每一元素的大小是 BalancedBST 的大小。由於 derived classes 通常比其 base classes 有更多的 data members，所以 derived class objects 通常都比其 base class objects 來得大。因此我們可以合理地

預期一個 BalancedBST object 比一個 BST object 大。如果是這樣，編譯器爲 printBSTArray 函式所產生的指標算術運算式，對於 BalancedBST objects 所組成的陣列而言就是錯誤的。至於會發生什麼結果，無可預期。無論如何，結果不會令人愉快。

如果你嘗試透過一個 base class 指標，刪除一個由 derived class objects 組成的陣列，那麼上述問題還會以另一種不同面貌出現。下面是你可能做出的錯誤嘗試：

```
// 刪除一個陣列，但是首先記錄一個有關此刪除動作的訊息
void deleteArray(ostream& logStream, BST array[])
{
  logStream << "Deleting array at address "
            << static_cast<void*>(array) << '\n';
  delete [] array;
}
BalancedBST *balTreeArray =         // 產生一個 BalancedBST 陣列
  new BalancedBST[50];
...
deleteArray(cout, balTreeArray);    // 記錄此一刪除動作
```

雖然你沒有看到，但其中一樣有「指標算術運算式」的存在。是的，當陣列被刪除，陣列中每一個元素的 destructor 都必須被喚起（見條款 8），所以當編譯器看到這樣的句子：

```
delete [] array;
```

必須產生出類似這樣的碼：

```
// 將 *array 中的物件以其建構次序的相反次序加以解構
for (int i = the number of elements in the array - 1; i >= 0; --i)
{
  array[i].BST::~BST();         // 呼叫 array[i] 的 destructor
}
```

如果你這麼寫，便是一個行爲錯誤的迴圈。編譯器如果產生類似的碼，當然同樣是個行爲錯誤的迴圈。C++ 語言規格中說，透過 base class 指標刪除一個由 derived classes objects 構成的陣列，其結果未有定義。我們知道所謂「未定義」的意思就是：執行之後會產生苦惱。簡單地說，多型（polymorphism）和指標算術不能混合運用。陣列物件幾乎總是會涉及指標的算術運算，所以陣列和多型不要混合運用。

注意，如果你避免讓一個具象類別（如本例之 BalancedBST）繼承自另一個具象

類別（如本例之 BST），你就不太能夠犯下「以多型方式來處理陣列」的錯誤。一如條款 33 所說，設計你的軟體使「具象類別不要繼承自另一個具象類別」，可以帶來許多好處。我鼓勵你翻開條款 33，好好看看完整內容。

條款 4：非必要不使用 default constructor

所謂 default constructor（也就是說可以不給任何引數就喚起者）是 C++ 一種「無中生有」的方式。Constructors 用來將物件初始化，所以 default constructors 的意思是在沒有任何外來資訊的情況將物件初始化。有時候這很可以想像，例如數值之類的物件，可以被合理地初始化爲 0 或一個無意義值。其他諸如指標之類的物件（條款 28）亦可被合理地初始化爲 null 或無意義值。資料結構如 linked lists, hash tables, maps 等等，可被初始化爲空容器。

但是並非所有物件都落入這樣的分類。有許多物件，如果沒有外來資訊，就沒有辦法執行一個完全的初始化動作。例如一個用來表現聯絡簿欄位的 class，如果沒有獲得外界指定的人名，產生出來的物件將毫無意義。在某些公司，所有儀器設備都必須貼上一個識別號碼；爲這種用途（用以模塑出儀器設備）而產生的物件，如果其中沒有供應適當的 ID 號碼，將毫無意義。

在一個完美的世界中，凡可以「合理地從無到有產出物件」的 classes，都應該內含 default constructors，而「必須有某些外來資訊才能產出物件」的 classes，則不必擁有 default constructors。但我們的世界畢竟不是完美的世界，所以我們必須納入其他考量。更明確地說，如果 class 缺乏一個 default constructor，當你使用這個 class 時便會有某些限制。

考慮下面這個針對公司儀器而設計的 class，在其中，儀器識別代碼是一定得有的一個 constructor 引數：

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber);
  ...
};
```

由於 EquipmentPiece 缺乏 default constructor，其運用可能在三種情況下出現

問題。第一個情況是在產生陣列的時候。一般而言沒有任何方法可以將物件陣列指定爲 constructor 的引數，所以幾乎不可能產生一個由 EquipmentPiece objects 構成的陣列：

```
EquipmentPiece bestPieces[10]; // 錯誤！無法呼叫 EquipmentPiece ctors

EquipmentPiece *bestPieces =
  new EquipmentPiece[10];      // 錯誤！另有一些問題。
```

有三個方法可以側面解決這個束縛。一個方法是使用 non-heap 陣列，於是便能夠在定義陣列時給予必要的引數：

```
int ID1, ID2, ID3, ..., ID10;    // 變數，用來放置儀器識別代碼
...
EquipmentPiece bestPieces[] = {  // 很好，ctor 獲得了必要的引數
  EquipmentPiece(ID1),
  EquipmentPiece(ID2),
  EquipmentPiece(ID3),
  ...,
  EquipmentPiece(ID10)
};
```

不幸的是此法無法延伸至 heap 陣列。

更一般化的作法是使用「指標陣列」而非「物件陣列」：

```
typedef EquipmentPiece* PEP;  // PEP 是個指向 EquipmentPiece 的指標

PEP bestPieces[10];                // 很好，不需呼叫 ctor。
PEP *bestPieces = new PEP[10];     // 也很好。
```

陣列中的各指標可用來指向一個個不同的 EquipmentPiece object：

```
for (int i = 0; i < 10; ++i)
  bestPieces[i] = new EquipmentPiece( ID Number );
```

此法有兩個缺點。第一，你必須記得將此陣列所指之所有物件刪除掉。如果你忘了，就會出現 resource leak（資源遺失）問題。第二，你需要的記憶體總量比較大，因爲你需要一些空間用來放置指標，還需要一些空間用來放置 EquipmentPiece objects。

「過度使用記憶體」這個問題可以避免，方法是先爲此陣列配置 raw memory，然後使用 "placement new"（見條款 8）在這塊記憶體上建構 EquipmentPiece objects：

```
// 配置足夠的 raw memory，給一個預備容納 10 個 EquipmentPiece
// objects 的陣列使用；條款 8 對於 operator new[] 有詳細說明。
void *rawMemory =
  operator new[](10*sizeof(EquipmentPiece));

// 讓 bestPieces 指向此塊記憶體，使這塊記憶體
// 被視爲一個 EquipmentPiece 陣列
EquipmentPiece *bestPieces =
  static_cast<EquipmentPiece*>(rawMemory);

// 利用 "placement new"（見條款 8）建構這塊
// 記憶體中的 EquipmentPiece objects。
for (int i = 0; i < 10; ++i)
  new (&bestPieces[i]) EquipmentPiece( ID Number );
```

注意，你還是必須供應 constructor 一個引數，做爲每個 `EquipmentPiece` objects 的初值。這項技術（以及「由指標構成陣列」的主意）允許你在「缺乏 default constructor」的情況下仍能產生物件陣列；但並不意味你可以因此迴避供應 constructor 引數。噢，那是不可能的。如果可能，constructors 的目標便受到了嚴厲的挫敗，因爲 constructors 保證物件會被初始化。

placement `new` 的缺點是，大部份程式員不怎麼熟悉它，維護起來比較困難。此外你還得在陣列內的物件結束生命時，以手動方式呼叫其 destructors，最後還得以呼叫 `operator delete[]`（見條款 8）的方式釋放 raw memory：

```
// 將 bestPieces 中的各個物件，以其建構次序的相反次序解構掉
for (int i = 9; i >= 0; --i)
  bestPieces[i].~EquipmentPiece();

// 釋放 raw memory
operator delete[](rawMemory);
```

如果你對 `rawMemory` 採用一般的陣列刪除語法，程式行爲將無可預期。因爲，刪除一個非以 `new` operator 獲得的指標，其結果未有定義：

```
delete [] bestPieces;      // 未有定義！因爲 bestPieces
                           // 並非來自 new operator。
```

關於 `new` operator 和 placement `new`，以及它們如何與 constructors 和 destructors 互動，細節請見條款 8。

Classes 如果缺乏 default constructors，帶來的第二個缺點是，它們將不適用於許多 template-based container classes。對那些 tempaltes 而言，具現化（*instantiated*）的「標的型別」需要有 default constructors。這是一個普遍的共同需求，因爲在那些 templates 內幾乎總是會產生一個以「template 型別參數」做爲型別而架構起來的陣列。例如一個爲 `Array` class 而寫的 template 可能看起來像這樣：

```
template<class T>
class Array {
public:
  Array(int size);
  ...
private:
  T *data;
};

template<class T>
Array<T>::Array(int size)
{
  data = new T[size];    // 陣列中的每個元素都呼叫 T::T()
  ...
}
```

大部份情況下，如果謹慎設計 template，可以消除對 default constructor 的需求。例如標準的 `vector` template（會產生出行爲類似「可擴展陣列」的各種 classes），就不要求其型別參數擁有一個 default constructor。不幸的是許多 templates 的設計什麼都有，獨缺謹慎。因此缺乏 default constructors 的 classes 將不相容於許多（不夠嚴謹的）templates。當 C++ 程式員學得更多的 template 設計技術與觀念，這個問題的重要性應該會降低。至於這一天什麼時候才會到來，唔，每個人猜測的都不一樣。

到底「要還是不要」提供一個 default constructor 呢？就像哈姆雷特的難題一樣，to be or not to be？進退維谷的情況下，最後一個考量點和 virtual base classes（見條款 E43）有關。Virtual base classes 如果缺乏 default constructors，與之合作將是一種刑罰。因爲 virtual base class constructors 的引數必須由欲產生之物件的衍生層次最深（所謂 most derived）的 class 提供。於是，一個缺乏 default constructor 的 virtual base class，要求其所有的 derived classes — 不論距離多麼遙遠 — 都必須知道、瞭解其意義、並且提供 virtual base class 的 constructors 引數。喔，derived classes 的設計者既不期望也不欣賞這樣的要求。

由於「缺乏 default constructors」帶來諸多束縛，有些人便認爲所有 classes 都應該有 default constructors — 甚至即使其 default constructor 沒有足夠資訊將物件做完整的初始化。依照這樣的哲學，`EquipmentPiece` 可能會被修改如下：

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber = UNSPECIFIED);
  ...
private:
  static const int UNSPECIFIED;     // 一個魔術數字，
                                    // 意味沒有被指定 ID 值
};
```

這就允許 `EquipmentPiece` objects 被這樣產生出來：

```
EquipmentPiece e;                   // 現在，這樣可行。
```

這幾乎總是會造成 class 內的其他 member functions 益形複雜，因爲這便不再保證一個 `EquipmentPiece` object 的所有欄位都有富意義的初值。如果「一個無 ID 值的 `EquipmentPiece` object」竟然是可以生存的，大部份 member functions 便必須檢查 ID 是否存在，否則就會出現大問題。通常，這部份的實作策略並不明朗，許多編譯器選擇的解決辦法是：什麼都不做，僅提供便利性：它們丟出一個 exception，或是呼叫某函式將程式結束掉。這樣的事情一旦發生，我們實在很難認爲軟體的整體品質會因爲「爲一個不需要 default constructor 的 class 畫蛇添足地加上一個 default constructor」而獲得提昇。

添加無意義的 default constructors，也會影響 classes 的效率。如果 member functions 必須測試欄位是否真被初始化了，其呼叫者便必須爲測試行爲付出時間代價，並爲測試碼付出空間代價，因爲可執行檔和程式庫都變大了。萬一測試結果爲否定，對應的處理程序又需要一些空間代價。如果 class constructors 可以確保物件的所有欄位都會被正確地初始化，上述所有成本便都可以免除。如果 default constructors 無法提供這種保證，那麼最好是避免讓 default constructors 出現。雖然這可能會造成可用的 classes 個數有某種限制，但同時也帶來一種保證：當你真的使用了這樣的 classes，你可以預期它們所產生的物件會被完全地初始化，實作上亦富效率。

技術

Techniques, or Idioms, or Patterns

本書大部份篇幅與程式設計的準則有關。此等準則雖然很重要，但沒有一個程式員可以光靠準則討生活。就像電視劇 Felix the Cat 所演的，「無論何時只要遭逢困境，他就打開他的錦囊，其中必有妙計」。唔，如果一個卡通人物可以有一個錦囊妙袋，C++ 程式員也可以有。請把這一章想像是你的錦囊妙袋的一個開端。

設計 C++ 軟體時，有一些問題會不斷重複出現。例如，如何讓 constructors 以及 non-member functions 像虛擬函式一樣地作用？如何限制 class 的實體（物件）個數？如何阻止物件被產生於 heap 內？如何保證物件被產生於 heap 內？如何能夠產生某種物件，使它在「其他某些 class 的 member functions」被呼叫時，自動執行某些動作？如何令不同的物件共享同一份資料結構，卻讓使用者錯以爲每個物件各自有一份資料？如何區分 `operator[]` 的讀/寫用途？如何產生一個虛擬函式，使其行爲視多個（而非單一）物件的動態型別而定？

所有這些（以及其他更多）問題都在本章獲得解答。本章描述 C++ 程式員常常遭遇的一些問題的解決辦法，這些解法都已獲得證明。我把這樣的解法稱爲 techniques（技術），也有人稱之爲 idioms（慣用法）或 patterns（模型、樣式）。不論你如何稱呼它們，當你每天與軟體開發過程中的各種小衝突搏鬥時，本章提供的資訊可以帶給你很多幫助。它也應該使你覺悟，不論你打算做什麼事，C++ 幾乎都有某種方法可以完成它。

條款 25：將 constructor 和 non-member functions 虛擬化

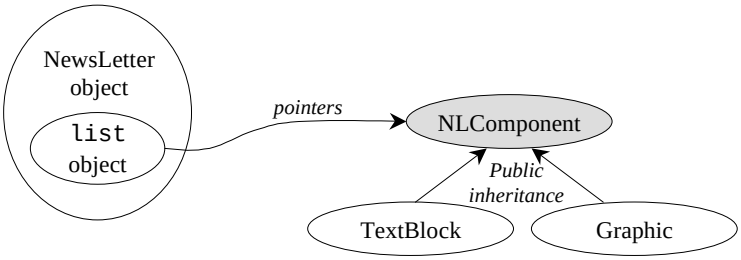
第一次面對 "virtual constructors" 時，似乎不覺得有什麼道理可言。是的，當你手上

有一個物件的 pointer 或 reference，而你不知道該物件的真正型別是什麼的時候，你會呼叫 virtual function（虛擬函式）以完成「因型別而異的行爲」。當你尚未獲得物件，但已經確知需要什麼型別的時候，你會呼叫 constructor 以建構物件。那麼，誰能夠告訴我什麼是 virtual constructors 呢？

很簡單。雖然 virtual constructors 似乎有點荒謬，但它們很有用。（如果你認爲荒謬的想法都是沒有用的，你如何解釋現代物理學的成功？）假設你寫了一個軟體，用來處理時事新聞，其內容由文字和圖形構成。你可以把程式組織成這樣：

```
class NLComponent {                    // 抽象基礎類別，用於時事消息
public:                                // 的組件（components）身上，
  ...                                  // 其中內含至少一個純虛擬函式。
};

class TextBlock: public NLComponent {
public:
  ...                                  // 沒有內含任何純虛擬函式
};

class Graphic: public NLComponent {
public:
  ...                                  // 沒有內含任何純虛擬函式
};

class NewsLetter {                     // 一份時事通訊係由一系列的
public:                                // NLComponent 物件構成
  ...
private:
  list<NLComponent*> components;
};
```

這些 classes 彼此間的關係如下：



NewsLetter 所使用的 list class 由 Standard Template Library 提供，後者是 C++ 標準程式庫的一部份（見條款 E49 和條款 35）。list 物件的行爲就像雙向串列（doubly linked lists）— 儘管它們不一定得以雙向串列實作出來。

NewsLetter 物件尚未開始運作的時候，可能儲存於磁碟中。爲了能夠根據磁碟上的資料產出一份 Newsletter，如果我們讓 NewsLetter 擁有一個 constructor 並以 istream 做爲引數，會很方便。這個 constructor 將從 stream 讀取資料以便產生必要的核心資料結構：

```
class NewsLetter {
public:
  NewsLetter(istream& str);
  ...
};
```

此 constructor 的虛擬碼（pseudo code）可能看起來像這樣：

```
NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    read the next component object from str;
    add the object to the list of this
    newsletter's components;
  }
}
```

或者，如果將棘手的東西搬移到另一個名爲 readComponent 的函式，就變成這樣：

```
class NewsLetter {
public:
  ...
private:
  // 從 str 讀取下一個 NLComponent 的資料，
  // 產生組件（component），並傳回一個指標指向它。
  static NLComponent * readComponent(istream& str);
  ...
};

NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    // 將 readComponent 傳回的指標加到 components list 尾端；
    // "push_back" 是一個 list member function，用來將物件安插
    // 到 list 尾端。
    components.push_back(readComponent(str));
  }
}
```

思考一下，readComponent 做了些什麼事。它產生一個新物件，或許是個 TextBlock，或許是個 Graphic，視讀入的資料而定。由於它產生新的物件，所以行爲彷若 constructor，但由於它能夠產生不同型別的物件，所以我們稱它爲一個 virtual

constructor。所謂 virtual constructor 是某種函式，視其獲得的輸入，可產生不同型別的物件（譯註：此行爲亦或被稱爲 dynamic creation）。Virtual constructors 在許多情況下有用，其中之一就是從磁碟（或網路或磁帶等等）讀取物件資訊。

有一種特別的 virtual constructor — 所謂 virtual copy constructor — 也被廣泛地運用。Virtual copy constructor 會傳回一個指標，指向其喚起者（某物件）的一個新副本。基於這種行爲，virtual copy constructors 通常以 `copySelf` 或 `cloneSelf` 命名，或者像下面一樣命名爲 `clone`。很少有其他函式能夠比這個函式有更直接易懂的實作方式了：

```
class NLComponent {
public:
  // 宣告 virtual copy constructor
  virtual NLComponent * clone() const = 0;
  ...
};

class TextBlock: public NLComponent {
public:
  virtual TextBlock * clone() const    // virtual copy constructor
  { return new TextBlock(*this); }
  ...
};

class Graphic: public NLComponent {
public:
  virtual Graphic * clone() const     // virtual copy constructor
  { return new Graphic(*this); }
  ...
};
```

如你所見，class 的 virtual copy constructor 就只是呼叫真正的 copy constructor 而已。"copy" 這層意義對這兩個函式而言是一樣的。如果真正的 copy constructor 執行的是淺層複製（shallow copy），virtual copy constructor 也一樣。如果真正的 copy constructor 執行的是深層複製（deep copy），virtual copy constructor 也一樣。如果真正的 copy constructor 做了某些煞費苦心的動作如 reference counting（參用計數）或 copy-on-write（「寫入時才複製」，見條款 29），virtual copy constructor 也一樣。啊，一貫性，多麼奇妙的事情。

注意上述實作手法乃利用「虛擬函式之傳回型別」規則中的一個寬鬆點，那是晚近才被接納的一個規則。當 derived class 重新定義其 base class 的一個虛擬函式時，不再需要一定得宣告與原本相同的傳回型別。如果函式的傳回型別是個指標（或

reference），指向一個 base class，那麼 derived class 的函式可以傳回一個指標（或 reference），指向該 base class 的一個 derived class。這並不會造成 C++ 的型別系統門戶洞開，卻可準確宣告出像 virtual copy constructors 這樣的函式。這也就是爲什麼縱使 `NLComponent` 的 `clone` 函式的傳回型別是 `NLComponent*`，`TextBlock` 的 `clone` 函式卻可以傳回 `TextBlock*` 而 `Graphic` 的 `clone` 函式可以傳回 `Graphic*` 的原因。

`NLComponent` 擁有一個 virtual copy constructor，於是我們現在可以爲 `NewsLetter` 輕鬆實作出一個（正常的）copy constructor：

```
class NewsLetter {
public:
  NewsLetter(const NewsLetter& rhs);         // copy constructor
  ...
private:
  list<NLComponent*> components;
};

NewsLetter::NewsLetter(const NewsLetter& rhs)
{
  // 迭代巡訪 rhs 的 list，運用每個元素的 virtual copy constructor，
  // 將元素複製到此物件的 components list 中。以下程式碼的細部討論，
  // 請見條款 35。
  for (list<NLComponent*>::const_iterator it =
           rhs.components.begin();
       it != rhs.components.end();
       ++it) {

    // "it" 指向 rhs.components 的目前元素，
    // 所以呼叫該元素的 clone 函式以取得該元素的一份副本，
    // 然後將該副本加到本物件的 components list 尾端。
    components.push_back((*it)->clone());
  }
}
```

我知道，除非你熟悉 Standard Template Library，否則上面這段碼看起來很詭異。但是其觀念很簡單：只要迭代巡訪即將被複製的那個 `NewsLetter` 物件的 components list，並針對其中的每一個組件（component）呼叫其 virtual copy constructor 即可。在這裡，我們需要一個 virtual copy constructor，因爲這個 components list 內含 `NLComponent` 物件指標，但我們知道每個指標真正指向的是一個 `TextBlock` 或是 `Graphic`。我們希望無論指標真正指向什麼，我們都可以複製它；virtual copy constructor 可以達到這個目標。

將 Non-Member Functions 的行爲虛擬化

就像 constructors 無法真正虛擬化一樣，non-member functions（見條款 E19）也是。然而就像我們認爲應該能夠以某個函式建構出不同型別的新物件一樣，我們也認爲應該可以讓 non-member functions 的行爲視其參數的動態型別而不同。舉個例子，假設你希望爲 `TextBlock` 和 `Graphic` 實作出 output 運算子，明顯的一個辦法就是讓 output 運算子虛擬化。然而，output 運算子（`operator<<`）接獲一個 `ostream&` 做爲其左端引數，因此它不可能成爲 `TextBlock` 或 `Graphic` classes 的一個 member function。

（是可以啦，但是看看會發生什麼事：

```
class NLComponent {
public:
  // output operator 的非傳統宣告
  virtual ostream& operator<<(ostream& str) const = 0;
  ...
};

class TextBlock: public NLComponent {
public:
  // virtual output operator（也是打破傳統）
  virtual ostream& operator<<(ostream& str) const;
};

class Graphic: public NLComponent {
public:
  // virtual output operator（也是打破傳統）
  virtual ostream& operator<<(ostream& str) const;
};

TextBlock t;
Graphic g;
...
t << cout;          // 透過 virtual operator<<，在 cout 身上
                    // 列印出 t。注意此語法與傳統不符。

g << cout;          // 透過 virtual operator<< 在 cout 身上
                    // 列印出 g。注意此語法與傳統不符。
```

Clients 必須把 stream 物件放在 "<<" 符號的右手端，而那和傳統的 output 運算子習慣不符。如果要回到正常的語法型式，我們必須將 `operator<<` 從 `TextBlock` 和 `Graphic` classes 身上移除，但如果這麼做，我們就不再能夠將它宣告爲 virtual）

另一種作法是宣告一個虛擬函式（例如 print）做爲列印之用，並在 TextBlock 和 Graphic 中定義它。但如果我們這麼做，TextBlock 和 Graphic 物件的列印語法就和其他型別物件的列印語法不一致，因爲其他型別都仰賴 operator<< 做爲輸出之用。

這些解法沒有一個令人滿意。我們真正需要的是一個 non-member function，它呼叫 operator<<，以展現像 print 虛擬函式一般的行爲。這一段「需求描述」其實已經非常接近其「作法描述」，是的，讓我們同時定義 operator<< 和 print，並令前者呼叫後者：

```
class NLComponent {
public:
  virtual ostream& print(ostream& s) const = 0;
  ...
};

class TextBlock: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...
};

class Graphic: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...
};

inline
ostream& operator<<(ostream& s, const NLComponent& c)
{
  return c.print(s);
}
```

顯然，non-member functions 的虛擬化十分容易：寫一個虛擬函式做實際工作，再寫一個什麼都不做的非虛擬函式，只負責呼叫虛擬函式。當然啦，爲了避免此一巧妙安排蒙受函式呼叫所帶來的成本，你可以將非虛擬函式 inline 化（見條款 E33）。

現在你知道如何讓 non-member functions 視其某個引數而虛擬化了。你可能會想，有沒有可能讓它們根據一個以上的引數而虛擬化？可以，但是不容易。有多困難呢？請翻到條款 31，那個條款專門討論這個問題。

條款 26：限制某個 class 所能產生的物件數量

好吧，你爲物件瘋狂！但有時候你會想要對你的狂熱有所節制。舉個例子，你的系統只有一部印表機，所以你希望將印表機的物件個數限制爲 1。或者你只有 16 個 file descriptors（檔案描述器）可用，所以你必須確定絕不會有更多的 file descriptor objects 被產生出來。如何辦到這樣一件事？如何限制物件的個數呢？

如果以數學歸納法證明，我們可以從 n= 1 開始，然後一步一步上去。幸運的是這既非證明題，也不需用到歸納法。此外，從 n= 0 開始應該是比較有益的，所以我們從 0 開始討論 — 也就是說，如何阻止物件被產生出來？

允許零個或一個物件

每當即將產生一個物件，我們確知一件事情：會有一個 constructor 被喚起。「阻止某個 class 產出物件」的最簡單方法就是將其 constructors 宣告爲 private：

```
class CantBeInstantiated {
private:
  CantBeInstantiated();
  CantBeInstantiated(const CantBeInstantiated&);
  ...
};
```

此舉移除了每一個人產出物件的權利。但我們也可以選擇性地解除這項約束。舉個例子，假設我想爲印表機設計一個 class，我希望設下「只能存在一台印表機」的約束；我們可以將印表機物件封裝在某個函式內，如此一來每個人都能夠取用印表機，但只有唯一一個印表機物件會被產出：

```
class PrintJob;     // 前置宣告（forward declaration），見條款 E34。
class Printer {
public:
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...
friend Printer& thePrinter();
```

```
private:
  Printer();
  Printer(const Printer& rhs);
  ...
};

Printer& thePrinter()
{
  static Printer p;    // 唯一的一個印表機物件
  return p;
}
```

此設計有三個成份。第一，Printer class 的 constructors 性屬 private，可以壓制物件的誕生。第二，全域函式 thePrinter 被宣告爲此 class 的一個 friend，致使 thePrinter 不受 private constructors 的約束。第三，thePrinter 內含一個 static Printer 物件，意思是只有一個 Printer 物件會被產生出來。

一旦 client 需要和系統中唯一那台印表機打交道，就呼叫 thePrinter。由於此函式傳回一個 reference，代表一個 Printer 物件，所以 thePrinter 可以用在任何需要 Printer 物件的地方：

```
class PrintJob {
public:
  PrintJob(const string& whatToPrint);
  ...
};

string buffer;
...                        // 把資料放進 buffer 內
thePrinter().reset();
thePrinter().submitJob(buffer);
```

當然啦，將 thePrinter 加入全域命名空間之中，可能會令你不悅。你可能會說，『是的，此全域函式看起來像極一個全域變數，而全域變數是一種拙劣的技巧，我比較喜歡將與印表機相關的所有機能都放進 Printer class 內』。唔，我絕不會和這種人爭辯，是的，thePrinter 可以輕易成爲 Printer 的一個 static member function，完成你的願望。這同時也消除了 friend 的必要性（畢竟它往往被視爲一種黏糊糊的關係）。改用 static member function 之後，Printer 看起來像這樣：

```
class Printer {
public:
  static Printer& thePrinter();
  ...
private:
  Printer();
  Printer(const Printer& rhs);
  ...
};

Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

現在，clients 取用印表機時，會稍微冗長些：

```
Printer::thePrinter().reset();
Printer::thePrinter().submitJob(buffer);
```

另一個作法是把 `Printer` 和 `thePrinter` 從全域空間中移走，放進一個 namespace 內（見條款 E28）。Namespaces 是晚近才加入 C++ 的性質。任何東西，包括 classes、structs、typedefs、函式、變數、物件等等，如果可被宣告於全域空間，也必定可被宣告於某個 namespace 內。「物件位於 namespace 內」這個事實並不會影響其行爲，但卻得以避免發生名稱牴觸問題。如果把 `Printer` class 和 `thePrinter` 函式放進一個 namespace，我們就不必擔心其他人碰巧也選用相同的名稱；是的，namespace 可以阻止名稱衝突。

從語法上看，namespaces 像是一個 classes，只不過沒有所謂的 public, protected 或 private 區段；每樣東西都是 public。下面便是把 `Printer` 和 `thePrinter` 放進一個名爲 `PrintingStuff` 的 namespace 中：

```
namespace PrintingStuff {
  class Printer {   // 這個 class 位於 PrintingStuff namespace 內
  public:
    void submitJob(const PrintJob& job);
    void reset();
    void performSelfTest();
    ...
    friend Printer& thePrinter();
```

```
private:
   Printer();
   Printer(const Printer& rhs);
   ...
 };

 Printer& thePrinter()  // 這個函式也位於 PrintingStuff namespace 內
 {
    static Printer p;
    return p;
 }
}                        // 這是 namespace 的結尾
```

有了這個 namespace，clients 可以使用全修飾名稱（也就是包含 namespace 的名稱）來取用 thePrinter：

```
PrintingStuff::thePrinter().reset();
PrintingStuff::thePrinter().submitJob(buffer);
```

也可以利用 using declaration 來少打幾個字：

```
using PrintingStuff::thePrinter;    // 將 "thePrinter" 名稱從
                                    // namespace "PrintingStuff"
                                    // 匯入目前的 scope 中。
thePrinter().reset();               // 現在 thePrinter 可以像
thePrinter().submitJob(buffer);     // 區域名稱一樣地被使用。
```

在此 thePrinter 實作碼中，有兩個精細的地方值得探討。第一，形成唯一一個 Printer 物件的，是函式中的 static 物件，而非 class 中 static 物件。這點很重要。「class 擁有一個 static 物件」的意思是，縱使從未被用到，它也會被建構（及解構）。相反的，「函式擁有一個 static 物件」的意思是，此物件在函式第一次被呼叫時才產生。如果該函式從未被呼叫，這個物件也就絕不會誕生（然而你必須付出代價，在函式每次被呼叫時檢查看看物件是否需要誕生）。C++ 的一個哲學基礎是，你不應該爲你並未使用的東西付出任何代價，而「將印表機這類物件定義爲函式內的一個 static」，正是固守此哲學的一種作法。這是你應該儘可能堅持的一個哲學。

讓印表機成爲一個 class static 而非一個 function static，另有一個缺點，那就是它的初始化時機。我們確切知道一個 function static 的初始化時機：在該函式第一次被呼叫時、並且在該 static 被定義處。至於一個 class static（或是 global static，如果你不認爲那很拙劣的話）則不一定在什麼時候初始化。C++ 對於「同一編譯單元內的

statics」的初始化次序是有提出一些保證，但對於「不同編譯單元內的 statics」的初始化次序沒有任何說明（見條款 E47）。事實上，這成爲無數頭痛問題的來源。Function statics，如果堪用的話，可讓我們避免這些頭痛問題。此例之中它是堪用的，那我們又何必多費心思呢？

第二個細微點是函式的「static 物件與 inlining 的互動」。再次看看 `thePrinter` 的 non-member 版本：

```
Printer& thePrinter()
{
  static Printer p;
  return p;
}
```

除了第一次被喚起（彼時 `p` 必須被建構），這是個僅有一行的函式 — 由述句 `"return p;"` 構成。如果說 inlining 有什麼好候選人，這個函式再適合不過了。但它卻未被宣告爲 `inline`。爲什麼不？

停下來思考一下，爲什麼你要將一個物件宣告爲 static？通常是因爲你只需要唯一一份物件，對嗎？現在思考一下 `inline` 是什麼意思？觀念上，它意味編譯器應該將每一個呼叫動作以函式本體取代之，但對於 non-member functions，它還意味其他一些事情。它意味這個函式有內部聯結（internal linkage）。

通常你不需要操心此等語言上的符咒，但是有一件事必須記住：函式如果帶有內部聯結，可能會在程式中被複製，也就是說程式的目的碼（object code）可能會對帶有內部聯結的函式複製一份以上的碼，而此複製行爲也包括函式內的 static 物件。結果呢，如果你有一個 inline non-member function 並於其中內含一個 local static 物件，你的程式可能會擁有多份該 static 物件的副本。所以，千萬不要產生內含 local static 物件的 inline non-member functions[1]。

但或許你認爲「產生一個函式，傳回一個 reference 指向某隱藏物件」，是限制物件個數的一條錯誤路線。或許你認爲比較好的作法是簡單地計算目前存在的物件個數，並於外界申請太多物件時，在 constructor 內丟出一個 exception。換句話說，你或

[1] 1996 年七月，ISO/ANSI 標準委員會把 inline 函式的預設聯結（linkage）由內部（internal）改爲外部（external），所以我描述的這個問題已經消除 — 至少文件上如此。你的編譯器可能尚未符合標準，所以你最好還是不要以 inline 函式搭配 static 資料。

許認爲我們應該這樣處理印表機的誕生：

```
class Printer {
public:
  class TooManyObjects{};       // 當外界申請太多物件時，
                                // 丟出這種 exception class。
  Printer();
  ~Printer();
  ...
private:
  static size_t numObjects;

  Printer(const Printer& rhs); // 有著「印表機個數永遠爲 1」的限制，
                                // 所以絕不允許複製行爲（見條款 E27）
};                              // （譯註：所以放在 private 區）
```

現在的想法是，利用 numObjects 來追蹤記錄目前存在多少個 Printer 物件。這個數值將在 constructor 中累加，並在 destructor 中遞減。如果外界企圖建構太多 Printer 物件，我們就丟出一個型別爲 TooManyObjects 的 exception：

```
// class static 的義務性定義
// （譯註：意指 class statics 除了於 class 之內宣告，還需於 class 之外定義）
size_t Printer::numObjects = 0;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal construction here;

  ++numObjects;
}

Printer::~Printer()
{
  perform normal destruction here;

  --numObjects;
}
```

這種限制物件誕生的方法有許多吸引人的理由。至少它非常直接易懂 — 每個人應該都能夠瞭解其行爲。另一個理由是，它很容易被一般化，使物件的最大數量可以設定爲 1 以外的值。

不同的物件建構狀態

但是這種策略也有問題。假設我們有一台特別的印表機，譬如說彩色印表機好了。其 class 和一般印表機的 class 有許多共同點，所以當然我們會使用繼承機制：

```
class ColorPrinter: public Printer {
  ...
};
```

現在假設我們的系統安裝了一台一般印表機和一台彩色印表機：

```
Printer p;
ColorPrinter cp;
```

上述物件定義帶給我們多少個 Printer 物件？答案是兩個：一個是 p，另一個是 cp 的「Printer 成份」。一旦執行，在 cp 的「base class 成份」建構當時，會有一個 TooManyObjects exception 被丟出。對許多程式員而言，這既不是他們要的，也不是他們所預想得到的。「避免具象類別（concrete class）繼承其他的具象類別」這一設計準則可使你免受此問題之苦；此設計哲學之來龍去脈，請看條款 33。

當其他物件內含 Printer 物件時，類似問題再度發生：

```
class CPFMachine {        // 針對那些可影印、可列表、可傳真的機器
private:

  Printer p;              // 針對列表功能
  FaxMachine f;           // 針對傳真功能
  CopyMachine c;          // 針對影印功能
  ...
};

CPFMachine m1;            // 沒問題
CPFMachine m2;            // 丟出一個 TooManyObjects exception
```

問題出在 Printer 物件可於三種不同狀態下生存：(1) 它自己，(2) 衍生物的「base class 成份」，(3) 內嵌於較大物件之中。這些不同狀態的呈現，把「追蹤目前存在之物件個數」的意義嚴重弄混了。你心裡頭所想的「目前存在的物件個數」可能和編譯器所想的不同。

通常你只對上述的狀態 (1) 感興趣，而你希望限制的也是這類物件的個數。如果採用原先的 Printer class 策略的話，這種約束很容易達成，因爲 Printer constructors

是 private，而如果沒有宣告任何 friend 的話，「帶有 private constructors 之 classes」不能夠被用來當做 base classes，也不能夠被內嵌於其他物件內。

「帶有 private constructors 之 class 不得被繼承」這一事實導出「禁止衍生」的一般性體制，此體制不必一定得和「有限個物件」聯想在一起。舉個例子，假設你需要 class FSA，用來表現有限個數的「狀態機器」（state machines，此等機器在許多情況下有用，其中一些導出使用者介面的設計）。更進一步假設你希望允許任意數量的 FSA 物件得被產生，但你也希望確保沒有任何 class 繼承自 FSA（這麼做的理由之一是得以爲「FSA 中出現 nonvirtual destructor」辯解。條款 E14 曾解釋爲什麼 base classes 通常需要 virtual destructors，條款 24 曾解釋爲什麼不帶有虛擬函式的 classes 會比帶有虛擬函式者導至較小的物件）。你可以設計 FSA 如下，同時滿足上述兩個條件：

```
class FSA {
public:
  // pseudo（假冒的）constructors
  static FSA * makeFSA();
  static FSA * makeFSA(const FSA& rhs);
  ...
private:
  FSA();
  FSA(const FSA& rhs);
  ...
};

FSA * FSA::makeFSA()
{ return new FSA(); }

FSA * FSA::makeFSA(const FSA& rhs)
{ return new FSA(rhs); }
```

不像 thePrinter 函式總傳回一個 reference 代表唯一物件，這裡的每一個 makeFSA pseudo-constructor 都傳回一個指標，指向一個獨一無二的物件。此即允許產生無限個數的 FSA 物件。

很好，但是每一個 pseudo-constructor 都呼叫 new，意味呼叫者必須記得喚起 delete，否則就會出現資源遺失（resource leak）問題。如果呼叫者希望在退離目前的 scope 時能夠讓 delete 自動被喚起，可將 makeFSA 傳回的指標儲存於一個 auto_ptr 物件內（見條款 9）；此種物件退離 scope 時會自動刪除其所指之物：

```
// 間接呼叫 default FSA constructor
auto_ptr<FSA> pfsa1(FSA::makeFSA());

// 間接呼叫 FSA copy constructor
auto_ptr<FSA> pfsa2(FSA::makeFSA(*pfsa1));

...          // 像正常指標一樣地使用 pfsa1 和 pfsa2，
             // 但是不必操心其刪除事宜
```

允許物件生生滅滅

現在我們知道如何設計「只允許單一物件」的 class 了。我們知道「追蹤某特定 class 的物件個數」會因「物件的 constructors 可於三種情況下被呼叫」而有點棘手。我們知道只要令 constructors 成爲 private，便可以消除那些令我們混淆的物件計數。現在值得再做最後一項觀察。我們利用 `thePrinter` 函式將唯一物件的各種處理行爲封裝起來，此法雖然符合期望地限制了 `Printer` 物件的個數爲 1，卻也限制我們在每次執行此程式時只能有唯一一個 `Printer` 物件。因此我們不可能寫出這樣的碼：

```
create Printer object p1;
use p1;
destroy p1;
create Printer object p2;
use p2;
destroy p2;
...
```

上述動作並未在同一時間產生一個以上的 `Printer` 物件，但是它在程式的不同地點使用了不同的 `Printer` 物件。如果這樣竟然不被允許，似乎不合理。畢竟我們並未違反「只有一台印表機」的條件。什麼辦法可以讓它合法？

有。我們唯一需要做的就是將稍早的物件計數（object-counting）碼和先前所見的虛假建構式（pseudo-constructors）結合起來：

```
class Printer {
public:
  class TooManyObjects{};
  // pseudo-constructor
  static Printer * makePrinter();
  ~Printer();
```

```
void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...
private:
  static size_t numObjects;
  Printer();
  Printer(const Printer& rhs);      // 不要定義此函式，因爲我們絕不
};                                  // 允許複製行爲（見條款 E27）。

// class static 的義務性定義
size_t Printer::numObjects = 0;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }
  proceed with normal object construction here;
  ++numObjects;
}

Printer * Printer::makePrinter()
{ return new Printer; }
```

如果「一旦太多物件被申請，即丟出一個 exception」的概念對你帶來不當困擾，你可以改令 pseudo-constructor 傳回一個 null 指標。當然，clients 因而必須在對此指標做任何動作之前先檢查其值。

Clients 使用這個 `Printer` class，就像使用任何其他 class 一樣，只不過他們必須呼叫 pseudo-constructor，取代真正的 constructor：

```
Printer p1;                     // 錯誤！default ctor 是 private。
Printer *p2 =
  Printer::makePrinter();       // 沒問題，間接呼叫 default ctor。
Printer p3 = *p2;               // 錯誤！copy ctor 是 private。

p2->performSelfTest();          // 所有其他函式都像平常一般地呼叫之。
p2->reset();
...
delete p2;                      // 避免資源遺失。如果 p2 是個 auto_ptr，
                                // 此一動作就不需要。
```

這項技術很容易被泛化爲「任意個數（不限一個）的物件」。我們唯一需要做的就是將原本寫死的常數 1 改爲 class 專屬的一個數值，然後將複製物件的限制去掉。例如，下面是修訂後的 Printer class，允許最多 10 個 Printer 物件同時存在：

```
class Printer {
public:
  class TooManyObjects{};

  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);
  ...

private:
  static size_t numObjects;
  static const size_t maxObjects = 10;      // 見以下說明

  Printer();
  Printer(const Printer& rhs);
};

// class statics 的義務性定義
size_t Printer::numObjects = 0;
const size_t Printer::maxObjects;

Printer::Printer()
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }
  ...
}

Printer::Printer(const Printer& rhs)
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }
  ...
}

Printer * Printer::makePrinter()
{ return new Printer; }

Printer * Printer::makePrinter(const Printer& rhs)
{ return new Printer(rhs); }
```

如果你的編譯器不允許你在 class 定義區中對 Printer::maxObjects 做宣告，不要驚訝。更明確地說，在 class 定義區內爲該變數指定初值 10，可能無法通過編譯。

在 class 定義區中爲 static const members（須爲整數型別如 ints, chars, enums 等）指定初值，是晚近才加入 C++ 的一個特性，某些編譯器尚未允許這麼做。如果你的編譯器至今尚未支援，鎮定些，不妨將 maxObjects 宣告爲「無具名 private enum」內的一個列舉元（enumerator）：

```
class Printer {
private:
  enum { maxObjects = 10 };    // 在此 class 內部，
  ...                          // maxObjects 的值爲常數 10。
};
```

或是將 const static 初始化，就像面對 non-const static member 一樣：

```
class Printer {
private:
  static const size_t maxObjects;   // 沒有給予初值
  ...
};
// 此行放在某個實作檔中
const size_t Printer::maxObjects = 10;
```

這些和先前原來的碼有相同的效果，但是「明白指定初值」比較容易被其他程式員瞭解。一旦你的編譯器支援「在 class 定義區內爲 const static members 指定初值」，你就應該多多利用這項能力。

一個用來計算物件個數的 Base Class

除了「statics 的初始化」尚有爭議，上述作法基本上已經具備魅惑眾生的能力，只不過還有一點需要再嘮叨一下。如果我們有許多像 Printer 那樣的 classes，其物件個數需要有所限制，我們必須一再一再地爲每一個 class 撰寫相同的碼。那實在是平凡到令人心靈麻痺。既然我們有一個特選的（簡直嚇嚇叫的）語言如 C++，好歹應該能夠將這樣的程序自動化吧。難道沒有一種辦法可以將「計算物件個數」的概念封裝起來並包裝到 class 內嗎？

我們可以輕易完成一個 base class，做爲物件計數之用，並讓諸如 Printer 之類的 classes 繼承它。但我們甚至可以做得更好。我們實際上可以某種方法封裝整個計數工具，我的意思不只包括用來處理物件個數的函式，還包括計數器本身。當我們在條款 29 檢驗所謂的 reference counting（參用計數）技術時，我們還會看到類似需求。此設計之詳細驗證，請看我發表於 *C/C++ Users Journal*, April 1998 的文章 "Counting Objects in C++"。

Printer class 內的計數器是 static numObjects，所以我們必須將這個變數搬到一個用來計算物件個數的 class 內。然而，我們也必須確保計算對象的每一個 class 都各自有一個計數器。設計一個「計數用的 class template」，可幫助我們自動產生適量的計數器，因爲我們可以令計數器爲「template 所產生之 classes」內的一個 static member：

```
template<class BeingCounted>
class Counted {
public:
  class TooManyObjects{};      // 此乃可能被丟出的 exceptions
  static int objectCount() { return numObjects; }
protected:
  Counted();
  Counted(const Counted& rhs);
  ~Counted() { --numObjects; }
private:
  static int numObjects;
  static const size_t maxObjects;
  void init();                 // 用以避免 ctor 碼重複出現
};

template<class BeingCounted>
Counted<BeingCounted>::Counted()
{ init(); }

template<class BeingCounted>
Counted<BeingCounted>::Counted(const
Counted<BeingCounted>&)
{ init(); }

template<class BeingCounted>
void Counted<BeingCounted>::init()
{
  if (numObjects >= maxObjects) throw TooManyObjects();
  ++numObjects;
}
```

此 template 所產生之 classes，只被設計用來做爲 base classes，因此才有所謂的 protected constructors 和 destructor 存在。注意 private member function init，它用來避免在兩個 Counted constructors 內出現重複的碼。

現在我們修改 Printer class，讓它運用 Counted template：

```
class Printer: private Counted<Printer> {
public:
  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);

  ~Printer();

  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...

  using Counted<Printer>::objectCount;       // 見以下說明
  using Counted<Printer>::TooManyObjects;    // 見以下說明

private:
  Printer();
  Printer(const Printer& rhs);
};
```

「Printer 利用 Counted template 以追蹤目前存在多少個 Printer 物件」這件事情，坦白說除了 Printer 作者以外，和任何人都無關。此等實作細目最好保持爲 private，這就是爲什麼此處使用 private inheritance 的原因（見條款 E42）。另一種作法是在 Printer 和 Counted<Printer> 之間使用 public inheritance，但是這麼一來我們就不得不給予 Counted classes 一個 virtual destructor（否則我們就得冒險：如果有人透過 Counted<Printer>* 指標刪除了一個 Printer 物件，會有不正確的行爲。見條款 E14）。條款 24 已經說得很清楚，虛擬函式出現在 Counted 中，會影響到 Counted 所衍生的類別物件的大小及其記憶體佈局。我們不希望吞下這個額外負擔，private inheritance 可以讓我們避而遠之。

Counted 的大部份作爲都已隱藏起來不讓 Printer 的使用者知曉，這相當正確，但是某些使用者可能希望知道有多少個 Printer 物件存在。Counted template 提供了一個 objectCount 函式，供應這份資訊，但是該函式在 Printer 中變成了 private 屬性，因爲我們用的是 private inheritance。爲了恢復其 public 存取層級，我們採用一個 using declaration：

```
class Printer: private Counted<Printer> {
public:
  ...
  using Counted<Printer>::objectCount;  // 讓此函式對於 Printer 的
  ...                                   // 使用者而言成爲 public。
};
```

這絕對合法，但如果你的編譯器尚未支援 namespaces，就不會接受它。如果是這樣，你可以使用舊式的存取宣告語法：

```
class Printer: private Counted<Printer> {
public:
  ...
  Counted<Printer>::objectCount;    // 讓 objectCount 在 Printer 中
  ...                               // 成爲 public。
};
```

上述傳統語法和先前使用 using declaration 的意義相同，但 C++ 標準規格已明白宣佈不再支援此一傳統語法。class `TooManyObjects` 有著和 `objectCount` 相同的處理方式，因爲 `Printer` 的使用者必須能夠處理 `TooManyObjects` — 如果他們希望能夠捕捉該種 exceptions 的話。

一旦 `Printer` 繼承自 `Counted<Printer>`，便可以完全忘記「物件計數」這回事。`Printer` 自己完全不必操心，就好像另外有人專門爲它服務似的，那人（其實是 `Counted<Printer>`）做掉了一切。`Printer` constructor 現在看起來像這樣：

```
Printer::Printer()
{
  proceed with normal object construction;
}
```

此處有趣的並非是你所看到的，而是你所看不到的。沒有任何檢驗動作用來查看物件個數是否超額，也沒有在 constructor 中增加物件個數。所有那些動作如今都由 `Counted<Printer>` constructors 處理掉了，而由於 `Counted<Printer>` 是 `Printer` 的一個 base class，我們知道在 `Printer` constructor 被呼叫之前，總是會有一個 `Counted<Printer>` constructor 被喚起。如果外界索求太多物件，`Counted<Printer>` constructor 就會丟出一個 exception，而 `Printer` constructor 也就不會被喚起。好極了，不是嗎？

或許吧。但是有一個不夠結實的尾巴需要綁緊一點，那是關於 `Counted` 內的 statics 義務性定義。是的，我們很容易處理 `numObjects` — 只要將以下兩行放進 `Counted` 的某個實作檔即可：

```
template<class BeingCounted>            // 定義 numObjects，
int Counted<BeingCounted>::numObjects;  // 並自動初始化爲 0
```

但 `maxObjects` 的情況就有點棘手。我們應該將此變數初始化爲什麼值呢？如果我們希望允許最多 10 台印表機，就應該將 `Counted<Printer>::maxObjects` 初始化

爲 10。如果我們希望允許最多 16 個 file descriptor objects，就應該將 `Counted<FileDescriptor>::maxObjects` 初始化爲 16。怎麼做？

我們採用無爲而治的態度：什麼都不做。我們不爲 `maxObjects` 提供初值，取而代之的是，我們要求 class 的使用者提供適當的初始化行爲。`Printer` 的作者必須在某個實作檔中加入這行：

```
const size_t Counted<Printer>::maxObjects = 10;
```

同樣道理，`FileDescriptor` 的作者必須加上這一行：

```
const size_t Counted<FileDescriptor>::maxObjects = 16;
```

如果這些作者忘記爲 `maxObjects` 提供一個適當定義，會發生什麼事？很簡單，他們會在聯結時期獲得錯誤訊息，因爲 `maxObjects` 未有定義。倘若我們對此一需求有足夠的說明，`Counted` 的使用者於是會對自己說一聲『喔』，然後回頭加上必要的初始化動作。

條款 27：要求（或禁止）物件 座落於 heap 之中

有時候你會想要做這樣的安排：讓某種型別的物件有「自殺」能力，也就是說能夠 "`delete this`"。 如此安排很明顯要求該型別之物件被配置於 heap 內。另些時候你可能想要擁有某種確定性，保證某一型別絕不會發生記憶體遺失（memory leaks），原因是沒有任何一個該型別的物件從 heap 中配置出來。假設你在嵌入式系統上工作，這類系統如果發生記憶體遺失，是非常麻煩的事，因爲 heap 空間極爲寶貴。我們有沒有可能完成一些碼，要求或禁止物件產生於 heap 之中呢？通常可以，但結果證明，所謂「在 heap 之中」可能比你所想像的更含糊。

要求物件產生於 heap 之中（譯註：所謂的 Heap-Based Objects）

讓我們從「限制物件必須誕生於 heap」開始。爲了厲行此等限制，你必須找出一個方法，阻止 clients 不得使用 `new` 以外的方法產生物件。這很容易辦到。是的，non-heap objects 會在其定義點自動建構，並在其壽命結束時自動解構，所以只要讓那些被隱喻喚起的建構動作和解構動作不合法，就可以了。

欲令這些呼叫動作不合法，最直接了當的方式就是將 constructors 和 destructor 宣告爲 private。但這實在太過了。沒有理由讓它們都成爲 private。比較好的辦法是讓 destructor 成爲 private 而 constructors 仍爲 public。如此一來，利用條款 26 所介紹

的程序，你可以導入一個 pseudo-（假冒的）destructor 函式，用來呼叫真正的 destructor。Clients 則呼叫這個 pseudo-destructor 以摧毀他們所產生的物件。

例如，假設我們希望確保「表現無限精度」的數值物件只能誕生於 heap 之中，我們可以這麼做：

```
class UPNumber {
public:
  UPNumber();
  UPNumber(int initValue);
  UPNumber(double initValue);
  UPNumber(const UPNumber& rhs);

  // pseudo（假冒的）destructor。這是一個 const member function，
  // 因爲 const 物件也可能需要被摧毀。
  void destroy() const { delete this; }
  ...
private:
  ~UPNumber();   // 譯註：注意，dtor 位於 private 區內。
};
```

Clients 於是應該這麼寫：

```
UPNumber n;                     // 錯誤！（雖然合法，但當 n 的 dtor
                                // 稍後被隱喻喚起，就不合法了）
UPNumber *p = new UPNumber;     // 良好。
...
delete p;                       // 錯誤！企圖呼叫 private destructor。
p->destroy();                   // 良好。
```

另一個辦法就是將所有 constructors 都宣告爲 private。這個想法的缺點是，class 常常有許多個 constructors，其中包括 copy constructor，也可能包括 default constructor；class 的作者必須記住將它們每一個都宣告爲 private。如果這些函式係由編譯器產生，編譯器產生的函式總爲 public（見條款 E45）。所以比較容易的辦法還是只宣告 destructor 爲 private，因爲一個 class 只能有一個 destructor。

只要限制 destructor 或 constructors 的運用，便可阻止 non-heap objects 的誕生。但是，就如條款 26 所說，它也妨礙了繼承（inheritance）和內含（containment）：

```
class UPNumber { ... };         // 將 dtor 或 ctors 宣告爲 private。

class NonNegativeUPNumber:      // 譯註：這是繼承（inheritance）。
  public UPNumber { ... };      // 錯誤！dtor 或 ctors 無法通過編譯。
```

```
class Asset {
private:
  UPNumber value;              // 譯註：這是內含（containment）。
  ...                          // 錯誤！dtor 或 ctors 無法通過編譯。
};
```

這些困難都可以克服。令 UPNumber 的 destructor 成爲 protected（並仍保持其 constructors 爲 public），便可解決繼承問題。至於「必須內含 UPNumber 物件」之 classes，可以修改爲「內含一個指標，指向 UPNumber 物件」：

```
class UPNumber { ... };              // 注意，將 dtor 宣告爲 protected

class NonNegativeUPNumber:
  public UPNumber { ... };           // 現在沒問題了；derived classes
                                     // 可以取用 protected members
class Asset {
public:
  Asset(int initValue);
  ~Asset();
  ...
private:
  UPNumber *value;
};

Asset::Asset(int initValue)
: value(new UPNumber(initValue))     // 沒問題
{ ... }

Asset::~Asset()
{ value->destroy(); }                // 也沒問題
```

判斷某個物件是否位於 Heap 內

如果我們採用上述策略，我們必須再次斟酌所謂「在 heap 內」的意義。假設 class 的定義概略如上，我們可以合法定義一個 non-heap NonNegativeUPNumber 物件：

```
NonNegativeUPNumber n;           // 沒問題
```

現在，NonNegativeUPNumber 物件 n 的「UPNumber 成份」並不位於 heap 內。可以嗎？答案視 class 的設計及其實作而定，但是讓我們假設那不可以，所有 UPNumber 物件 — 甚至是其衍生類別物件中的「base class 成份」— 都必須在 heap 中。我們如何厲行這樣的約束？

沒有什麼簡單的辦法。UPNumber constructor 不可能知道它之所以被喚起是否是爲了產生某個 heap-based object 的「base class 成份」。也就是說，沒有什麼辦法可以讓

UPNumber constructor 偵測出以下狀態有什麼不同：

```
NonNegativeUPNumber *n1 =
  new NonNegativeUPNumber;          // 在 heap 內
NonNegativeUPNumber n2;             // 不在 heap 內
```

或許你不相信我。或許你認爲你可以在 new operator、operator new、以及「new operator 所呼叫之 constructor」三方互動關係（條款 8）上玩把戲。或許你認爲只要把 UPNumber 修改成這樣就可以打敗它們：

```
class UPNumber {
public:
  // 如果產生一個 non-heap object，就丟出 exception。
  class HeapConstraintViolation {};

  static void * operator new(size_t size);

  UPNumber();
  ...
private:
  static bool onTheHeap;  // 此一 flag 用來在 ctors 內指示
  ...                     // 正建構中的物件是否位於 heap。
};

// class static 的義務性定義
bool UPNumber::onTheHeap = false;

void *UPNumber::operator new(size_t size)
{
  onTheHeap = true;                 // 譯註：此爲新增動作
  return ::operator new(size);      // 譯註：此爲原就該有的動作
}
UPNumber::UPNumber()
{
  if (!onTheHeap) {
    throw HeapConstraintViolation();
  }

  proceed with normal construction here;

  onTheHeap = false;                // 清除 flag，供下一個物件使用。
}
```

這裡面沒有什麼太深的技術，只是利用了一個觀念：當物件被配置於 heap 內，operator new 會被呼叫起來配置生鮮（raw）記憶體，然後會有一個 constructor 被呼叫起來將物件初始化於該記憶體中。更明確地說，operator new 將 onTheHeap 設爲 true，而每一個 constructor 都會檢查 onTheHeap 的值，看看建構中的物件的

記憶體是否由 operator new 配置而來。如果不是，就丟出一個型別爲 HeapConstraintViolation 的 exception。否則建構程序就如常繼續下去。一旦建構完成，onTheHeap 會被設爲 false，爲下一個將被建構的物件重新設定好預設值。

這個主意夠好的了，但它不可行！是的，考慮以下這份可能的程式碼：

```
UPNumber *numberArray = new UPNumber[100];
```

第一個問題是，陣列記憶體乃由 operator new[] 而非 operator new 配置。不過你依然可以爲它撰寫一個自有版本（如果你的編譯器支援的話），做相同的手腳。比較麻煩的是 numberArray 有 100 個元素，所以應該有 100 次 constructor 呼叫動作。但是整個程序卻只有一次記憶體配置動作，所以 100 次 constructors 動作中，只有第一次動作時 onTheHeap 的值爲 true。第二次呼叫 constructor 時，會有一個 exception 被丟出，令你苦惱不已。

即使沒有陣列，前述的位元設立動作也可能失敗。是的，考慮以下程式碼：

```
UPNumber *pn = new UPNumber(*new UPNumber);
```

此處於 heap 內產生了兩個 UPNumbers，並讓 pn 指向其中一個。換句話說某物件以另一物件的值做爲初值。這段碼會造成資源遺失（resource leak），但是讓我們忽略之。我們看看以下動作執行時會發生什麼事：

```
new UPNumber(*new UPNumber)
```

這裡內含兩個 new operator 呼叫動作，並造成兩個 operator new 和兩個 UPNumber constructors 呼叫動作（見條款 8）。程式員通常期望這些函式呼叫的執行次序如下：

1. 爲第一個物件呼叫 operator new
2. 爲第一個物件呼叫 constructor
3. 爲第二個物件呼叫 operator new
4. 爲第二個物件呼叫 constructor

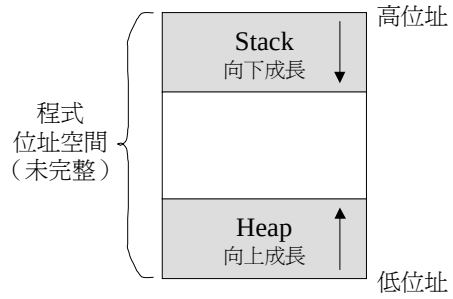
但 C++ 並不保證這麼做。某些編譯器產生出來的函式呼叫次序是這樣子：

1. 爲第一個物件呼叫 operator new
2. 爲第二個物件呼叫 operator new
3. 爲第一個物件呼叫 constructor
4. 爲第二個物件呼叫 constructor

產出這樣的碼，對編譯器而言並非錯誤，但是前述「在 `operator new` 中設立位元」的計倆卻因此失敗。因爲步驟 1 和步驟 2 所設立的位元在步驟 3 中被清除掉了，造成步驟 4 所建構的物件認爲它不處於 heap 之中 — 雖然它其實是。

這些困難並不至於造成「令每一個 constructor 檢驗 `*this` 是否位於 heap 內」的基本觀念無效。它們只是顯示，在 `operator new`（或 `operator new[]`）內檢查某個位元是否被設立起來，並不是「決定 `*this` 是否位於 heap 內」的一個可靠作法。我們需要更好的辦法。

如果你對此絕望了，可能會被誘入「不具移植性」的領域中。例如，你可能決定利用許多系統都有的一個事實：程式的位址空間以線性序列組織而成，其中 stack（堆疊）高位址往低位址成長，heap（堆積）由低位址往高位址成長：



有的系統是這樣，有的系統不是這樣。如果你的系統的確以此方式來組織應用程式的記憶體，你或許認爲你可以藉由以下函式決定某個位址是否位於 heap 內：

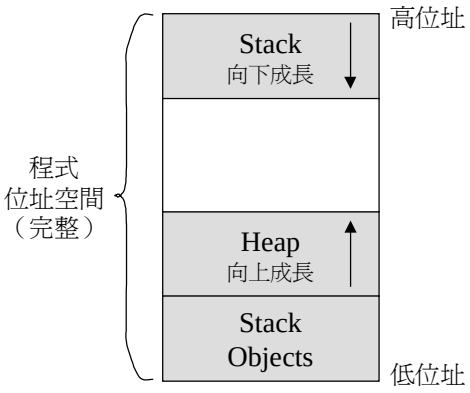
```
// 不正確的企圖，決定某個位址是否位於 heap 之內
bool onHeap(const void *address)
{
  char onTheStack;     // local stack variable

  return address < &onTheStack;
}
```

這個函式背後的觀念很有趣。在 `onHeap` 函式內，`onTheStack` 是個區域變數。所以

它被置於 stack 內。當 `onHeap` 被呼叫，其 stack frame（亦即其 activation record）會被放在 stack 的最頂端，而由於此架構中的 stack 係向低位址成長，所以 `onTheStack` 的位址一定比其他任何一個位於 stack 中的變數（或物件）更低。因此，如果參數 `address` 比 `onTheStack` 的位址更低，它就不可能位於 stack，那就一定是位於 heap。

這樣的邏輯是正確的，但還不夠完善。基本問題在於物件可能被配置於三個地方，而不是兩個地方。是的，stack 和 heap 都可以持有物件，但我們不要忘了 static 物件。Static 物件在程式執行期間只初始化一次。所謂 static 物件，不只涵蓋明白宣告爲 `static` 的物件，也包括 global scope 和 namespace scope（見條款 E47）內的物件。如此物件必須安置於某處，而所謂某處既不是 stack 也不是 heap。

它們究竟在哪裡？視系統而定！但在「stack 和 heap 的安排如上頁所示」的許多系統中，它們被置於 heap 之下。上頁的記憶體佈局圖雖然說出了系統中的許多真象，但沒有說出完整的真象。如果把 static 物件納入圖中，看起來便是這樣：



突然間 `onHeap` 運作失敗的原因變得再清楚不過：它無法區分 heap 物件和 static 物件：

```
void allocateSomeObjects()
{
  char *pc = new char;    // heap object：onHeap(pc) 會傳回 true。
```

```
    char c;                 // stack object：onHeap(&c) 會傳回 false。
    static char sc;         // static object：onHeap(&sc) 會傳回 true，
    ...                     // 這是錯誤的結果。
}
```

現在，你可能拼命想找出什麼方法可以區分 heap objects 和 stack objects，而在拼命的過程中你可能會欣然接受和「不具移植性」這個惡魔打交道的機會。但難道你已經饑渴到寧願接受不保證四海皆準的答案嗎？當然不，所以我知道你必將拒絕這個雖蠱惑人但其實不足取的「位址比較法」。

令人悲苦的是，不只沒有一個「絕對具移植性」的辦法可以決定某物件是否位於 heap 內，甚至沒有一個「頗具移植性」作法可以在大部份時候有用。如果你要絕對而確實地說某個位址是否位於 heap 之中，你就一定得走入不可移植的、因實作系統而異的陰暗角落。沒錯，就是這樣。所以你最好重新設計你的軟體，避免需要判斷某個物件是否位於 heap 內。

如果你發現自己被「物件是否位於 heap 內」困住，可能是因爲你想要知道爲它喚起 `delete` 是否安全。通常這樣的刪除動作是以聲名狼籍的 "`delete this`" 型式呈現。然而，此動作是否安全，和「指標是否指向一個位於 heap 內的物件」是兩回事。因爲，並非所有指向 heap 內的指標都可以被安全地刪除。再次考慮一個 `Asset` 物件，內含有一個 `UPNumber` 物件：

```
class Asset {
private:
  UPNumber value;
  ...
};

Asset *pa = new Asset;
```

顯然 `*pa`（及其成員）位於 heap 內。同樣明顯的是，對著一個指向 `pa->value` 的指標進行刪除動作是不安全的，因爲沒有一個這樣的指標是以 `new` 獲得。

幸運的是，判斷「指標的刪除」是否安全，比判斷「指標是否指向 heap 內的物件」簡單一些，因爲前者的判斷依據就只是：此指標是否由 `new` 傳回。由於我們可以自行寫一個 `operator new`（見條款 E8~E10），所以這問題比較容易解決。下面是一個可能的解決辦法：

```
void *operator new(size_t size)
{
  void *p = getMemory(size);   // 呼叫某個函式以配置記憶體，
                               // 並處理記憶體不足的情況。
  add p to the collection of allocated addresses;

  return p;
}

void operator delete(void *ptr)
{
  releaseMemory(ptr);     // 將記憶體釋還給自由空間（free store）

  remove ptr from the collection of allocated addresses;
}

bool isSafeToDelete(const void *address)
{
  return whether address is in collection of
  allocated addresses;
}
```

很簡單，operator new 負責把一些條目（entries）加到一個由「動態配置而得的位址」所形成的集合中，operator delete 則負責把這些條目移除；isSafeToDelete 負責搜尋該集合，看看某個位址是否在其中。如果這些 operator new 和 operator delete 函式都在全域範圍內，這應該對所有型別（甚至是內建型別）都管用。

實際運用上，有三件事情可能會消減我們對此設計的狂熱。第一，我們極端不願意在全域空間內定義任何東西，尤其是帶有預設意義的函式如 operator new 和 operator delete。我們知道，全域空間只有一個，該空間內帶有「正規型式」（意指特定之參數型別，見條款 E9）的 operator new 和 operator delete 也只有一個。稍早的作爲會使那些「正規型式」被我們所奪。如果那麼做，便是宣告我們的軟體不相容於其他「也實作有全域版之 operator new 和 operator delete」的任何軟體（例如許多物件導向資料庫系統）。

第二個考慮是效率：如果沒有必要，何必爲所有的「heap 應用」承擔沉重的簿記工作（以追蹤傳回的位址）呢？

最後一個考量雖然俗氣卻很重要：似乎不可能實作出一個總是能夠有效作用的 isSafeToDelete 函式。困難在於，當物件涉及多重或虛擬的基礎類別時，會擁有多

個位址（譯註：此稱爲非自然多型，unnatural polymorphism，見《多型與虛擬》第二章），所以不能保證「交給 isSafeToDelete 的位址」和「被 operator new 傳回的位址」是同一個 — 縱使論及的物件的確是配置於 heap 內。細節請見條款 24 和 31。

我們想要的，是這些函式提供的機能，但又不附帶全域命名空間的污染問題、額外的義務性負荷、以及正確性的疑慮。幸運的是，C++ 以 abstract mixin base class（抽象混合式基礎類別）型式，完全滿足了我們的需求。

所謂 abstract base class 是一個不能夠被實體化的 base class。也就是說它至少有一個純虛擬函式。所謂 mixin（"mix in"）class 則提供有一組定義完好的能力，能夠與其 derived class 所可能提供的其他任何能力（見條款 E7）相容。如此的 classes 幾乎總是 abstract。我們於是可以形成一個所謂的 abstract mixin base class，用以爲 derived classes 提供「判斷某指標是否以 oeprator new 配置出來」的能力。以下便是這樣一個 class：

```
class HeapTracked {              // mixin class；追蹤並記錄
public:                          // 被 operator new 傳回的指標。
  class MissingAddress{};        // exception class；見以下說明
  virtual ~HeapTracked() = 0;
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
  bool isOnHeap() const;
private:
  typedef const void* RawAddress;
  static list<RawAddress> addresses;
};
```

這個 class 使用 C++ 標準程式庫（見條款 E49 和條款 35）提供的 list 資料結構，記錄所有由 operator new 傳回的指標。Operator new 函式負責配置記憶體並將條目（entries）加入 list 內；operator delete 負責釋放記憶體並從 list 身上移除條目；isOnHeap 決定某物件的位址是否在 list 內。

HeapTracked class 的實作很簡單，因爲真正的記憶體配置動作和釋放動作交給全域的 operator new 函式和 operator delete 函式完成，而 list class 所擁有函式又可以順利完成安插、移除、搜尋等行爲。下面是 HeapTracked 的完整實作內容：

```
// static class member 的義務性定義
list<RawAddress> HeapTracked::addresses;
```

```
// HeapTracked 的 destructor 是個純虛擬函式，俾使這個
// class 成爲抽象類別（見條款 E14）。但這個 destructor
// 仍然必須有定義，所以我們提供一個空定義。
HeapTracked::~HeapTracked() {}

void * HeapTracked::operator new(size_t size)
{
  void *memPtr = ::operator new(size);  // 取得記憶體

  addresses.push_front(memPtr);         // 將其位址置於 list 頭部。

  return memPtr;
}

void HeapTracked::operator delete(void *ptr)
{
  // 獲得一個 "iterator"，用以找出哪一筆 list 元素內含 ptr。
  // 細節見條款 35。
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), ptr);

  if (it != addresses.end()) {  // 如果找到符合條件的元素（條目，entry），
    addresses.erase(it);        // 移除之，並
    ::operator delete(ptr);     // 釋放記憶體。
  } else {                      // 否則表示 ptr 不是 operator new 所配置，
    throw MissingAddress();     // 於是丟出一個 exception。
  }
}

bool HeapTracked::isOnHeap() const
{
  // 取得一個指標，指向 *this 所佔記憶體的起始處；細節描述於下。
  const void *rawAddress = dynamic_cast<const void*>(this);

  // 在位址串列中搜尋「被 operator new 傳回的指標」
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), rawAddress);

  return it != addresses.end();     // 傳回「找到與否」訊息
}
```

這些碼都很直接易懂，但如果你不熟悉 Standard Template Library 的 list class 和其他組件的話，情況可能比較麻煩些。條款 35 解釋了每一樣東西，但是上述註解應該也足夠解釋本例所發生的事情。

剩下唯一可能對你造成困惑的，就是以下述句（在 isOnHeap 函式內）：

```
const void *rawAddress = dynamic_cast<const void*>(this);
```

稍早我說過，凡涉及「多重或虛擬基礎類別」之物件，會擁有多個位址，並因此使全域函式 isSafeToDelete 的撰寫變得更複雜。此問題也在 isOnHeap 中令我們苦惱，但由於 isOnHeap 只施行於 HeapTracked 物件身上，我們可以利用 dynamic_cast 運算子（見條款 2）的特殊性質來消除這個問題。只要簡單地將指標「動態轉型」爲 void*（或 const void* 或 volatile void* 或 const volatile void* — 如果你無法在日常飲食中獲得足夠修飾養份的話☺），便會獲得一個指標，指向「原指標所指物件」的記憶體起始處。不過，dynamic_cast 只適用於那種「所指物件至少有一個虛擬函式」的指標身上，我們命運多舛的 isSafeToDelete 函式卻必須對任何型別的指標都起作用，因此 dynamic_cast 無法幫助它。isOnHeap 比較有所選擇（只測試指向 HeapTracked 物件的指標），所以將 this 動態轉型爲 const void*，會爲我們帶來一個指標，指向現行物件的記憶體起始處。那便是 HeapTracked::operator new 必須傳回的指標 — 如果物件記憶體最初是由 HeapTracked::operator new 配置的話。只要你的編譯器支援 dynamic_cast 運算子，這項技術就有移植性。

有了這樣的 class，即使是 BASIC 程式員☺ 也可以爲任何 class 加上「追蹤指標（指向 heap 配置體）」的能力。他們唯一需要做的就是令 class 繼承 HeapTracked。舉個例子，如果我們希望能夠判斷某個 Asset 物件指標是否指向一個 heap-based object，我們可以修改 Asset class 的定義，以 HeapTracked 做爲其 base class：

```
class Asset: public HeapTracked {
private:
  UPNumber value;
  ...
};
```

然後我們便可以查詢 Asset* 指標如下：

```
void inventoryAsset(const Asset *ap)
{
  if (ap->isOnHeap()) {
    ap is a heap-based asset — inventory it as such;
  }
  else {
    ap is a non-heap-based asset — record it that way;
  }
}
```

像 HeapTracked 這樣的 mixin class 有個缺點，那就是它不能夠使用於內建型別身

上，因爲 int 和 char 這類型別並不繼承自任何東西。此外，使用諸如 HeapTracked 這樣的 class，最常見的理由便是要決定是否能夠 "delete this"，而你絕不會想要對內建型別這麼做，因爲這些型別根本沒有 this 指標。

禁止物件產生於 heap 之中

「檢驗物件是否位於 heap 內」的種種規劃終於告一個段落了。光譜的另一端則是「阻止物件被配置於 heap 內」。這個題目的前景稍微光明一些。一般而言有三種可能：(1) 物件被直接實體化、(2) 物件被實體化爲 derived class objects 內的「base class 成份」、(3) 物件被內嵌於其他物件之中。讓我一一討論。

欲阻止 clients 直接將物件實體化於 heap 之中，很容易，因爲此等物件總是以 new 產生出來，而你可以讓 client 無法呼叫 new。雖然你不能夠影響 new operator 的能力（那是語言內建的），但你可以利用一個事實：new operator 總是呼叫 operator new（見條款 8），而後者是你可以自行宣告的。更明確地說，你可以將它宣告爲 private。舉個例子，如果你不希望 clients 將 UPNumber 物件產生於 heap 內，可以這麼做：

```
class UPNumber {
private:
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
  ...
};
```

現在 clients 只能夠做某些被允許的事：

```
UPNumber n1;                  // 可以
static UPNumber n2;           // 也可以
UPNumber *p = new UPNumber;   // 錯誤！企圖呼叫 private operator new
```

將 operator new 宣告爲 private 應該足夠了，但如果 operator new 屬性爲 private 而 operator delete 卻爲 public，實在有點奇怪，所以除非有一個好理由將它們拆夥，否則最好是將它們宣告於 class 的同一存取層級內。如果你也想禁止「由 UPNumber 物件所組成的陣列」位於 heap 內，可以將 operator new[] 和 operator delete[]（見條款 8）亦宣告爲 private。（operator new 和 operator delete 之間的契合力，比許多人想像的還要強。它們之間有一個罕爲人知的關係，請看我發表於 *C/C++ Users Journal*, April 1998 的文章 "Counting Objects in C++" 內的方塊說明）

有趣的是，將 operator new 宣告爲 private，往往也會妨礙 UPNumber 物件被實體

化爲 heap-based derived class objects 的「base class 成份」。那是因爲 operator new 和 operator delete 都會被繼承，所以如果這些函式不在 derived class 內宣告爲 public，derived class 繼承的便是其 base(s) 所宣告的 private 版本：

```
class UPNumber { ... };              // 如上
class NonNegativeUPNumber:           // 假設此 class 未宣告 operator new
  public UPNumber {
  ...
};
NonNegativeUPNumber n1;              // 沒問題
static NonNegativeUPNumber n2;       // 也沒問題
NonNegativeUPNumber *p =             // 錯誤！企圖呼叫 private operator new
  new NonNegativeUPNumber;
```

如果 derived class 宣告有一個屬於自己的 operator new（譯註：且爲 public），當 client 將 derived class objects 配置於 heap 內時，該 operator new 函式會被喚起，因此我們必須另覓良法以求阻止「UPNumber 的 base class 成份」的誕生。類似情況，當我們企圖配置一個「內含 UPNumber 物件」的物件，「UPNumber 的 operator new 乃爲 private」這一事實並不會帶來什麼影響：

```
class Asset {
public:
  Asset(int initValue);
  ...
private:
  UPNumber value;                    // 譯註：containment.
};

Asset *pa = new Asset(100);          // 沒問題，呼叫的是
                                     // Asset::operator new 或
                                     // ::operator new，而非
                                     // UPNumber::operator new。
```

對實用目的而言，這把我們帶回熟悉的情境。我們曾經希望「如果一個 UPNumber 物件被建構於 heap 以外，那麼就在 UPNumber constructors 內丟出 exception」，這次我們希望的是「如果物件被產生於 heap 內的話，就丟出一個 exception」。然而，就像沒有任何具移植性的作法可以判斷某位址是否位於 heap 內一樣，我們也沒有具移植性的作法可以判斷它是否不在 heap 內。這應該不令人驚訝，畢竟如果我們能夠確定某個位址在 heap 內，當然我們就能夠確定某個位址不在 heap 內。既然我們做不到前者，當然我們也做不到後者。

條款 28：Smart Pointers（精靈指標）

所謂 smart pointers，是「看起來、用起來、感覺起來都像內建指標，但提供更多機能」的一種物件。它們有各式各樣的用途，包括資源的管理（見條款 9, 10, 25 和 31）以及自動的重複寫碼工作（見條款 17 和 29）。

當你以 smart pointers 取代 C++ 的內建指標（亦即所謂的 dumb pointers），你將獲得以下各種指標行爲的控制權：

- **建構和解構（Construction and Destruction）**。你可以決定 smart pointer 被產生以及被摧毀時發生什麼事。通常我們會給 smart pointers 一個預設值 0，以避免「指標未獲初始化」的頭痛問題。某些 smart pointers 有責任刪除它們所指的物件 — 當指向該物件的最後一個 smart pointer 被摧毀時。這是消除資源遺失問題的一大進步。
- **複製和設值（Copying and Assignment）**。當一個 smart pointer 被複製、或涉及設值動作時，你可以控制發生什麼事。某些 smart pointer 會希望在此時刻自動爲其所指之物進行複製或設值動作，也就是執行所謂的深層複製（deep copy）。另一些 smart pointer 則可能只希望指標本身被複製或設值就好。還有一些則根本不允許複製和設值。不論你希望什麼樣的行爲，smart pointers 都可以讓你如願。
- **提領（Dereferencing）**。當 client 提領（取用）smart pointer 所指之物時，你有權決定發生什麼事情。例如你可以利用 smart pointers 協助實作出條款 17 所說的 lazy fetching 策略。

Smart pointers 係由 templates 產生出來。由於它就像內建指標一樣，所以它必須有強烈的型別性（strongly typed）；使用者可利用 template 參數表明其所指物件的型別。大部份 smart pointer templates 看起來像這樣子：

> 譯註：Scott Meyers 發表於 *Dr. Dobbs' Journal*, October 1999 的文章："Implementing `operator->*` for Smart Pointers" 非常值得一觀，可視爲本條款的一份補充資料。

```
template<class T>                    // template，用來產生
class SmartPtr {                     // smart pointer objects
public:
  SmartPtr(T* realPtr = 0);          // 產生一個 smart ptr，指向一個
                                     // 原本已由 dumb ptr 指向之物。
                                     // 未初始化之 ptrs 被預設爲 0（null）
  SmartPtr(const SmartPtr& rhs);     // 複製一個 smart ptr
  ~SmartPtr();                       // 摧毀一個 smart ptr

  // 對一個 smart ptr 做設值（assignment）動作
  SmartPtr& operator=(const SmartPtr& rhs);

  T* operator->() const;             // 提領（dereference）smart ptr，
                                     // 以獲得其所指之物的一個 member。
  T& operator*() const;              // 提領（dereference）smart ptr。

private:
  T *pointee;                        // smart ptr 所指之物
};
```

注意，這裡的 copy constructor 和 assignment operator 都是 public。如果你的 smart pointer classes 不允許被複製或設值，你應該將它們宣告爲 private（見條款 E27）。兩個提領（dereferencing）運算子被宣告爲 const，因爲提領「指標所指之物」並不會改變指標本身（雖然可能導致指標所指之物的改變）。此外，每個 smart pointer-to-T 都內含有一個 dumb pointer-to-T，後者才是實踐指標行爲的真正主角。

在進入 smart pointer 的實作細節之前，應該先看看 clients 如何使用 smart pointers。考慮一個分散式系統，其中某些物件位於本機（local），某些位於遠端（remote）。本機物件的存取通常比遠端物件的存取簡單而且快速，因爲遠端存取可能需要 remote procedure calls（RPC）或其他某種與不同機器溝通的方法。

對於應用程式而言，如果本機物件和遠端物件的處理方式有所不同，是件挺麻煩的事兒。讓所有物件都好像位於本機端，應該是比較方便的作法。程式庫可以提供 smart pointers，實現此一幻象：

```
template<class T>                    // 這個 template 所產生的 smart ptrs
class DBPtr {                        // 用來指向分散式資料庫（DB）內的物件
public:
  DBPtr(T *realPtr = 0);             // 產生一個 smart ptr，指向一個 DB 物件 —
                                     // 如果已有一個本機端的 dumb 指標指向它。
```

```
  DBPtr(DataBaseID id);          // 產生一個 smart ptr，指向一個 DB 物件 —
                                 // 如果知道一個獨一無二的 DB 識別代碼的話。
  ...                            // 其他的 smart ptr 函式
};

class Tuple {                    // class，用來表現資料庫中
public:                          // 的一筆資料（tuples）。
  ...
  void displayEditDialog();      // 呈現一個圖形式對話盒，允許
                                 // 使用者輸入 tuple。
  bool isValid() const;          // 檢驗 *this 是否有效。
};

// class template，用來在一個 T 物件被修改時，
// 完成運轉記錄（log entries）。詳見以下說明。
template<class T>
class LogEntry {
public:
  LogEntry(const T& objectToBeModified);
  ~LogEntry();
};

void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);       // 爲以下編輯動作完成運轉記錄；
                                    // 詳見以下說明。
  // 反覆顯示編輯對話盒，直到獲得有效值爲止。
  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

editTuple 所編輯的資料可能實際上位於遠端，但是 editTuple 撰寫者不需操心這樣的事情；smart pointer class 會對系統隱瞞此事。程式員關心的是，所有資料（上述的 tuples）除了其宣告型式外，都應該像內建指標一樣地被使用。

請注意，editTuple 函式內使用了一個 LogEntry 物件。傳統的設計是以「開始運轉記錄」和「結束運轉記錄」等函式呼叫動作，將 displayEditDialog 呼叫動作包圍起來。這裡的設計則顯示，LogEntry constructor 開啓一筆運轉記錄，而其 destructor 結束該筆運轉記錄。一如條款 9 所解釋，利用物件（而非明白呼叫某函式）來開始和結束運轉記錄，在面對 exceptions 時是一種比較穩健的作法。所以你應該

讓自己習慣使用諸如 LogEntry 這樣的 classes。此外，產生單一一個 LogEntry 物件，也比加上個別呼叫動作以分別啓動和結束一筆運轉記錄更容易些。

如你所見，使用 smart pointer 和使用 dumb pointer，兩者之間沒有太大差別。這正是封裝（encapsulation）的有效證據。我們可以想像，smart pointers 的使用者能夠把它們視同 dumb pointers。我們即將見到，有時候這樣的取代行爲會更透明無痕。

Smart Pointers 的建構、設值、解構

Smart pointer 的建構行爲通常明確易解：確定一個標的物（通常是利用 smart pointer 的 constructor 引數），然後讓 smart pointer 內部的 dumb pointer 指向它。如果尚未決定標的物，就將內部指標設爲 0，或是發出一個錯誤訊息（可能是丟出 exception）。

Smart pointer 的 copy constructor、assignment operator(s) 和 destructor 的實作，多少會因爲「擁有權」的觀念而稍微複雜。如果一個 smart pointer 擁有它所指的物件，它就有責任在本身即將被摧毀時刪除該物件。前提是這個 smart pointer 所指物件係動態配置而得 — 當我們使用 smart pointers 時，這個假設十分普遍（如何確定此一假設爲真？見條款 27）。

考慮 C++ 標準程式庫提供的 auto_ptr template。一如條款 9 解釋，auto_ptr 物件是個 smart pointer，用來指向一個誕生於 heap 內的物件，直到該 auto_ptr 被摧毀爲止。當摧毀情事發生，auto_ptr 的 destructor 會刪除其所指物。auto_ptr template 可能實作如下：

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *ptr = 0): pointee(ptr) {}
  ~auto_ptr() { delete pointee; }
  ...
private:
  T *pointee;
};
```

在「同一物件只可被一個 auto_ptr 擁有」的前提下，上述作法可以有效運作。但一旦 auto_ptr 被複製或被設值，會發生什麼事？

```
auto_ptr<TreeNode> ptn1(new TreeNode);
auto_ptr<TreeNode> ptn2 = ptn1;         // 呼叫 copy ctor;
                                        // 會發生什麼事？
```

```
auto_ptr<TreeNode> ptn3;
ptn3 = ptn2;                                  // 呼叫 operator=；
                                              // 會發生什麼事？
```

如果我們只是複製其內的 dumb pointer，會導至兩個 auto_ptrs 指向同一物件。這會造成麻煩，因爲每一個 auto_ptr 都會在被摧毀時刪除其所指之物。這表示物件會被刪除兩次。如此的雙殺動作，其結果未有定義（但往往災情慘重）。

另一種作法是以 new 運算子爲所指物件產生一個新副本。這就保證我們不會有多個 auto_ptrs 指向同一物件，但這可能使新物件的誕生（以及後來的解構）形成無法讓人接受的效率衝擊。猶有進者，我們不一定能夠確知產生什麼樣型別的物件，因爲一個 auto_ptr<T> 不一定得指向一個型別爲 T 的物件；它可以指向一個 T 衍生型別的物件。Virtual constructors（見條款 25）可以協助解決這個問題，但要在一個像 auto_ptr 這麼目標廣泛的 class 內使用該技術，似乎不甚合適。

如果我們禁止 auto_ptr 被複製和設值，問題就消除了。但是 auto_ptr 採用一個更富彈性的解法：當 auto_ptr 被複製或被設值，其物件擁有權會移轉：

```
template<class T>
class auto_ptr {
public:
  ...
  auto_ptr(auto_ptr<T>& rhs);       // copy constructor

  auto_ptr<T>&                      // assignment operator
  operator=(auto_ptr<T>& rhs);
  ...
};

template<class T>
auto_ptr<T>::auto_ptr(auto_ptr<T>& rhs)      // copy constructor
{
  pointee = rhs.pointee;       // 將 *pointee 的擁有權轉移至 *this

  rhs.pointee = 0;             // rhs 不再擁有任何東西
}
```

```
template<class T>              // assignment operator
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
  if (this == &rhs)            // 如果自己指派給自己，
    return *this;              // 不做任何事情。

  delete pointee;              // 刪除目前擁有之物。

  pointee = rhs.pointee;       // 將 *pointee 的擁有權移轉給 *this。
  rhs.pointee = 0;             // rhs 不再擁有任何東西。

  return *this;
}
```

注意，assignment 運算子在掌握一個新物件的擁有權之前，必須先刪除它所擁有的物件。如果沒有這麼做，該物件就絕不會被刪除。記住，只有 auto_ptr 物件才「擁有」它所指之物件。

由於 auto_ptr 的 copy constructor 被喚起時，物件擁有權移轉了，所以以 by value 方式傳遞 auto_ptrs 往往是個非常糟的主意。下面是其原因：

```
// 此函式往往導至災難
void printTreeNode(ostream& s, auto_ptr<TreeNode> p)
{ s << *p; }

int main()
{
  auto_ptr<TreeNode> ptn(new TreeNode);
  ...
  printTreeNode(cout, ptn);        // 以 by value 方式傳遞 auto_ptr
  ...
}
```

當 printTreeNode 的參數 p 被初始化（藉由 auto_ptr 的 copy constructor），ptn 所指物件的擁有權被移轉至 p。當 printTreeNode 結束，p 即將離開其生存空間，於是其 destructor 會刪除它所指（也是 ptn 原本所指）之物。然而 ptn 不再指向任何東西（其內部的 dumb pointer 是 null），所以 printTreeNode 被呼叫之後，任何人如果使用 ptn 都會導至未定義的行爲。因此，只有當你確定要將物件擁有權移轉給函式的某個參數時，才應該以 by value 方式傳遞 auto_ptrs。事實上很少人會想要這麼做。

這並非意味你不能夠以 auto_ptrs 做爲參數，這只是意味 pass-by-value 並不適用。Pass-by-reference-to-const 才是適當的途徑：

```
// 此函式的行爲直觀多了
void printTreeNode(ostream& s,
                   const auto_ptr<TreeNode>& p)
{ s << *p; }
```

在此函式中，p 是個 reference 而不是個物件，所以不會有 constructor 被用來爲 p 設立初值。當 ptn 被傳遞給這一版的 printTreeNode，它保留了所指之物的擁有權，於是在 printTreeNode 被呼叫之後，ptn 還是可以安全地被使用。換句話說，以 by reference-to-const 的方式傳遞 auto_ptrs，可避免 pass-by-value 所造成的危險。（至於寧願使用 pass-by-reference 而儘量不使用 pass-by-value 的其他理由，請看條款 E22）。

複製和設值時「將擁有權從一個 smart pointer 移轉至另一個 smart pointer」的概念，很有趣。更有趣的是 copy constructor 和 assignment operator 的非傳統式宣告。它們原本通常需要 const 參數，但上面的碼顯示並非如此。事實上上述程式碼在複製和設值時改變了參數內容，換句話說如果 auto_ptr 物件被複製，或是成爲設值動作的源端，它們會被修改。

這是千真萬確的事情。如果 C++ 有足夠的彈性讓你全權處理，不是很好嗎？如果 C++ 語言要求 copy constructors 和 assignment operators 一定得採用 const 參數，你或許必須將參數的常數性加以轉型（見條款 E21），要不就得另起爐灶玩另一個遊戲。但現在，你只需確實說出你想說的就好：當 smart pointer 被複製，或是身爲設值動作的源端，它會被改變。這可能不怎麼直觀，但的確簡單、直接、而且正確。

如果你對 auto_ptr member functions 的驗證感興趣，你可能會想看一份完整實作品。本書 291~294 頁有一份完整實作品，在那裡你會看到，C++ 標準程式庫中的 auto_ptr template 的 copy constructors 和 assignment operators，其彈性比此處所描述的更高；在標準的 auto_ptr template 內，那些函式都是 member function templates，而不只是 member functions。（關於 Member function templates，本條款稍後會介紹。你也可以在條款 E25 讀到它）

Smart pointer 的 destructor 通常看起來像這樣：

```
template<class T>
SmartPtr<T>::~SmartPtr()
{
  if (*this owns *pointee) {
    delete pointee;
  }
}
```

有時候沒有必要在其中做測試動作 — 例如當 auto_ptr 總是擁有其所指物時。另一些時候，上述測試又可能有點複雜，例如一個運用了參用計數（reference counting，見條款 29）之 smart pointer，必須在決定「是否有權力刪除其所指之物」前先調整計數器。當然，某些 smart pointers 很像 dumb pointers：它們本身被摧毀時，對所指之物並沒有影響。

實作 Dereferencing Operators（提領運算子）

現在讓我們把注意力放在 smart pointers 的核心：operator* 和 operator-> 函式身上。前者傳回所指物件。觀念上十分簡單：

```
template<class T>
T& SmartPtr<T>::operator*() const
{
  perform "smart pointer" processing;
  return *pointee;
}
```

這個函式首先做任何必要的初始化動作、或是讓 pointee 獲得有效值的任何動作。例如，如果程式採用了 lazy fetching（緩式取值）設計（見條款 17），上述函式可能需要爲 pointee 變幻出一個新物件。如果 pointee 有效，operator* 函式就傳回一個 reference，代表被指之物。

注意，傳回值是 reference 形式。如果傳回的是物件，雖然編譯器允許你這麼做，結果會慘不忍睹。記住，pointee 不需非得指向型別爲 T 的物件不可；它也可以指向一個 T 衍生型別的物件。若真如此而且你的 operator* 函式傳回的是個 T 物件，而非一個 reference（代表真正的 derived class object），你的函式便是傳回一個錯誤型別的物件！這涉及所謂的切割（slicing）問題，見條款 E22 和條款 13。這麼一來，以該物件呼叫虛擬函式，將不會喚起被指物的動態型別相應函式。你的 smart pointer 也就因此沒有在本質上適當地支援虛擬函式，這樣的指標又哪能多麼地「精靈」呢？此外，傳回 reference，效率比較好，因爲其中不需建構暫時物件（見條款 19）。正確性與效率都到手了，真令人開心。

如果你是個多慮的人，你可能會想，萬一有人對著一個 null smart pointer（也就是其內嵌之 dumb pointer 爲 null）呼叫 `operator*`，該怎麼辦。輕鬆點，你可以做你想做的任何事情。「提領 null 指標」是個無定義的行爲，所以無所謂錯誤與否。想要丟出一個 exception 嗎？丟吧。想要呼叫 `abort` 嗎？呼吧。想要走入記憶體將每個 byte 都設爲你的生日除以 256 的餘數嗎？行！雖然不是很棒，但就目前的 C++ 語言而言，你有完全的自由。

`operator->` 的故事和 `operator*` 差不多，但檢驗 `operator->` 之前，讓我們回憶一下呼叫此函式的一個不尋常意義。再次考慮 `editTuple` 函式，其中用到一個 smart pointer-to-`Tuple`：

```
void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);

  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

其中的述句：

```
pt->displayEditDialog();
```

會被編譯器解釋爲：

```
(pt.operator->())->displayEditDialog();
```

這意味不論 `operator->` 傳回什麼，對它施行 `->` 運算子都必須是合法的。因此 `operator->` 只可能傳回兩種東西：一個 dumb pointer（指向某物件），或是一個 smart pointer。大部份時候你會想要傳回一個普通的 dumb pointer，於是你可以實作 `operator->` 如下：

```
template<class T>
T* SmartPtr<T>::operator->() const
{
  perform "smart pointer" processing;
  return pointee;
}
```

這可以運作良好。由於此函式傳回一個指標，所以透過 `operator->` 喚起虛擬函式，會有應有的表現。

對許多用途而言，這便是你需要知道的關於 smart pointers 的全部。例如條款 29 的參用計數（reference-counting）所使用的 smart pointers 便不超過目前所討論的範圍。然而如果你想要將 smart pointers 推往更高遠的境界，你必須知道更多有關於 dumb pointer 的行爲，並知道 smart pointers 如何能夠（或不能夠）模擬那些行爲。如果你的座右銘是「大部份人在此下馬休息，但我不」，那麼下面的材料是爲你準備的。

測試 Smart Pointers 是否爲 Null

有了先前討論的函式，我們可以產生、摧毀、複製、指派（設值）、提領 smart pointers。但有一件事我們沒辦法做，那就是判斷它是否爲 null：

```
SmartPtr<TreeNode> ptn;
...
if (ptn == 0) ...                    // 錯誤！
if (ptn) ...                         // 錯誤！
if (!ptn) ...                        // 錯誤！
```

這是個嚴重的限制。

爲我們的 smart pointer classes 加上一個 `isNull` 函式很容易。但那並不能解決「smart pointers 無法如 dumb pointers 那般自然地測試是否爲 null」這個枷鎖。另一種作法是，提供一個隱式型別轉換運算子，允許上述測試動作得以通過編譯。這類轉換的傳統目標是 `void*`：

```
template<class T>
class SmartPtr {
public:
  ...
  operator void*();                  // 如果 smart ptr 是 null，傳回零，
  ...                                // 否則傳回非零值。
};

SmartPtr<TreeNode> ptn;
...
if (ptn == 0) ...                    // 現在可以了。
if (ptn) ...                         // 也可以了。
if (!ptn) ...                        // 沒問題。
```

這很類似 iostream classes 提供的一種轉換功能，同時也解釋了爲什麼你可以寫出這樣的碼：

```
ifstream inputFile("datafile.dat");

if (inputFile) ...      // 測試 inputFile 是否成功地被開啓
```

就像所有型別轉換函式一樣，此處這個也有相同的缺點。在大部份程式員都認爲「函式呼叫失敗」的情況下，它卻讓它成功（見條款 5）。更明確地說，它允許你把 smart pointers 拿來和完全不同的型別做比較：

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;
...
if (pa == po) ...         // 竟然可以過關！
```

是的，即使我們並沒有撰寫以 `SmartPtr<Apple>` 和 `SmartPtr<Orange>` 爲參數的 `operator==` 函式，但由於兩個 smart pointers 都可以被隱式轉換爲 `void*` 指標，而針對語言內建指標，又存在有一個內建的比較函式，所以上式可通過編譯。這種行爲使得隱式轉換函式非常危險（再次請看條款 5，並閱讀再三，直到能夠倒背如流）。

以「轉換至 `void*`」爲基調，可衍生出各種變形。某些設計者喜歡轉換爲 `const void*`，另一些人喜歡轉換爲 `bool`，但沒有一個可以消除上述「竟然允許不同型別互相比較」的問題。

有種差強人意的作法，允許你提供「測試 nullness」的合理語法，而且能夠將「意外引起不同型別之 smart pointers 相互比較」的機會降到最低，那就是將「`!` 運算子」多載化。是的，對你的 smart pointer classes 撰寫 `operator!`，並在其喚起者（某個 smart pointer）是 null 的情況下，傳回 `true`：

```
template<class T>
class SmartPtr {
public:
  ...
  bool operator!() const;    // 只有當 smart ptr 是 null 時才傳回 true
  ...
};
```

這使你的 clients 得以這麼寫：

```
SmartPtr<TreeNode> ptn;
...
if (!ptn) {               // 沒問題
  ...                     // ptn 是 null
}
else {
  ...                     // ptn 不是 null
}
```

但不能這麼寫：

```
if (ptn == 0) ...         // 還是錯誤

if (ptn) ...              // 也錯誤
```

唯一風險就是這樣：

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;
...
if (!pa == !po) ...        // 啊呀，這竟然可通過編譯
```

幸運的是程式員並不常常寫出這樣的碼。iostream 程式庫不但允許隱式轉換爲 `void*`，還提供有一個 `operator!`，但這兩個函式都只測試互異的少許 stream 狀態（譯註：彼此可轉換者）。在 C++ 標準程式庫中（見條款 E49 和條款 35），「隱式轉換爲 `void*`」已被「隱式轉換爲 `bool`」取代，而 `operator bool` 總是傳回 `operator!` 的反相。

將 Smart Pointers 轉換爲 Dumb Pointers

有時候你可能會希望將 smart pointers 加入已使用 dumb pointers 的應用軟體或程式庫中。例如你的分散式資料庫系統不可能一開始就是分散式，所以你可能會有一些老舊函式並非使用 smart pointers：

```
class Tuple { ... };              // 如前

void normalize(Tuple *pt);        // 將 *pt 放進標準型式中。
                                  // 注意，用的是 dumb pointer。
```

如果你呼叫 `normalize` 並指定一個 smart pointer-to-`Tuple` 給它，會發生什麼事：

```
DBPtr<Tuple> pt;
...
normalize(pt);                    // 錯誤！
```

這個呼叫動作之所以失敗，是因爲目前沒有辦法可以將一個 `DBPtr<Tuple>` 轉換爲一個 `Tuple*`。如果這樣就可以：

```
normalize(&*pt);                    // 難看，但合法
```

但我希望你同意，那動作真是難看。

如果我們爲 smart pointer-to-T template 加上一個隱式型別轉換運算子，使之可轉換爲 dumb pointer-to-T，先前那個呼叫便可以成功：

```
template<class T>                   // 如前
class DBPtr {
public:
  ...
  operator T*() { return pointee; } // 譯註：新增的轉換運算子
  ...
};

DBPtr<Tuple> pt;
...
normalize(pt);                      // 現在這就成功了
```

上述函式一旦加上，nullness 測試問題也就消除了：

```
if (pt == 0) ...                    // 沒問題，將 pt 轉換爲一個 Tuple*
if (pt) ...                         // 同上
if (!pt) ...                        // 同上
```

不過這樣的轉換也有黑暗面（總是如此啦，你看過條款 5 了吧）。它們使 clients 得以輕易地直接對 dumb pointers 做動作，因而迴避了 smart pointer 當初的設計目的：

```
void processTuple(DBPtr<Tuple>& pt)
{
  Tuple *rawTuplePtr = pt;          // 將 DBPtr<Tuple> 轉換爲 Tuple*

  use rawTuplePtr to modify the tuple;
}
```

通常，smart pointer 所提供的靈巧行爲應該是你的一個基礎設計思維，所以「允許 clients 直接使用 dumb pointers」往往會導至災難。舉個例子，如果 `DBPtr` 實作出條款 29 所展示的參用計數（reference-counting），那麼「允許 clients 直接操控 dumb pointers」幾乎一定會導至簿記方面的錯誤，造成參用計數所用的資料結構崩解。

即使你提供了一個隱式轉換運算子，可以將 smart pointer 轉換爲其內部的 dumb pointer，你的 smart pointer 還是無法真正地和 dumb pointer 交換。因爲「從 smart pointer 轉換爲 dumb pointer」是一種使用者自定的轉換行爲，而編譯器禁止一次施行一個以上這類轉換。舉個例子，假設你有一個 class，用來表現「已處理過某筆特定資料（tuple）」的所有客戶：

```
class TupleAccessors {
public:
  TupleAccessors(const Tuple *pt);       // pt 代表一筆資料（tuple）
  ...                                    // 而其處理者是我們所在乎的。
};
```

一如往常，TupleAccessors 的「單一引數 constructor」也可以做爲型別轉換運算子使用，將 Tuple* 轉換爲 TupleAccessors（見條款 5）。現在考慮一個函式，用來將兩個 TupleAccessors 物件內的資訊合併在一起：

```
TupleAccessors merge(const TupleAccessors& ta1,
                     const TupleAccessors& ta2);
```

由於 Tuple* 可能被隱式轉換爲一個 TupleAccessors，所以以兩個 dumb pointers Tuple* 做爲引數來呼叫 merge 函式，沒有問題：

```
Tuple *pt1, *pt2;
...
merge(pt1, pt2);     // 沒問題，兩個指標都被轉換爲 TupleAccessors 物件
```

然而如果以兩個 smart pointers DBPtr<Tuple> 呼叫之，就會失敗：

```
DBPtr<Tuple> pt1, pt2;
...
merge(pt1, pt2);     // 錯誤！無法將 pt1 和 pt2 轉換爲 TupleAccessors 物件
```

那是因爲從 DBPtr<Tuple> 轉換爲 TupleAccessors 需要兩個使用者自定轉換（第一個將 DBPtr<Tuple> 轉換爲 Tuple*，第二個將 Tuple* 轉換爲 TupleAccessors），而此等轉換情節是 C++ 所不允許的。

Smart pointer classes 如果提供隱式轉換至 dumb pointer，便是打開了一個難纏臭蟲的門戶。考慮這樣的碼：

```
DBPtr<Tuple> pt = new Tuple;
...
delete pt;
```

這應該無法編譯，畢竟 pt 不是指標，而是物件。你不能夠刪除一個物件。只有指標才能夠被刪除，不是嗎？

是的。但回憶條款 5 所說，編譯器會尋找隱式型別轉換，儘可能讓函式呼叫成功。再回憶條款 8 所言，delete 運算子會導至 destructor 和 operator delete（兩者都是函式）被喚起。編譯器希望那些函式呼叫動作都能成功，所以在上述的 delete 述句中，它們暗自將 pt 轉換爲一個 Tuple*，然後刪除。這幾乎一定會弄糟你的程式。

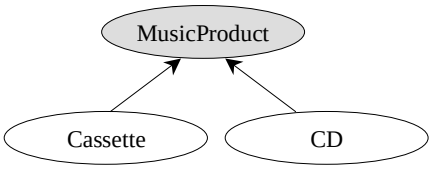
如果 pt 擁有其所指之物，該物件現在被刪除了兩次，一次是在 delete 呼叫點，一次是在 pt 的 destructor 被喚起時。如果 pt 並未擁有物件，其他人會擁有。那個人可能會 deleted pt，那沒問題。然而如果 pt 所指物件的擁有者不是 delete pt 的人，我們可以預期正牌擁有者稍後會再次刪除該物件。上述第一種情況和第三種情況會導至物件被刪除兩次，而將物件刪除一次以上會導至未定義的行爲。

這個臭蟲特別邪惡，容易致命，因爲隱藏在 smart pointers 背後的整個概念就是讓它們看起來及感覺起來儘量像 dumb pointers，而愈接近這個目標，你的 client 愈是可能忘記他們正在使用 smart pointers。因此，如果他們繼續像以前一樣地認爲，爲了避免資源遺失，必須在呼叫 new 之後對應呼叫 delete，誰又能夠責怪他們呢？

底線很簡單：不要提供對 dumb pointers 的隱式轉換運算子，除非不得已。

Smart Pointers 和「與繼承有關的」型別轉換

假設我們有一個 public inheritance 繼承體系，模塑出消費性音樂產品：




```
class MusicProduct {
public:
  MusicProduct(const string& title);
  virtual void play() const = 0;
  virtual void displayTitle() const = 0;
  ...
};
```

```
class Cassette: public MusicProduct {
public:
  Cassette(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};

class CD: public MusicProduct {
public:
  CD(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};
```

更進一步假設有個函式，給它一個 MusicProduct 物件，它就會顯示標題並播放之：

```
void displayAndPlay(const MusicProduct* pmp, int numTimes)
{
  for (int i = 1; i <= numTimes; ++i) {
    pmp->displayTitle();
    pmp->play();
  }
}
```

我們可能這樣使用這個函式：

```
Cassette *funMusic = new Cassette("Alapalooza");
CD *nightmareMusic = new CD("Disco Hits of the 70s");

displayAndPlay(funMusic, 10);
displayAndPlay(nightmareMusic, 0);
```

沒什麼令人驚訝的。但如果我們將 dumb pointers 以其 smart pointers 取代，看看會發生什麼事：

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int numTimes);

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));

displayAndPlay(funMusic, 10);          // 錯誤！
displayAndPlay(nightmareMusic, 0);     // 錯誤！
```

如果 smart pointers 是那麼聰明靈巧，爲什麼上述兩行無法通過編譯？

此之所以無法通過編譯，是因爲沒辦法將 SmartPtr<CD> 或 SmartPtr<Cassette> 轉換爲 SmartPtr<MusicProduct>。編譯器所看見的，是三個各不相干的 classes — 彼

此之間沒有任何關係。誰說編譯器應該另作他想呢？畢竟情況並非「`SmartPtr<CD>` 或 `SmartPtr<Cassette>` 繼承自 `SmartPtr<MusicProduct>`」。由於這些 classes 之間沒有繼承關係，我們很難期望編譯器會將其中一種型別轉換爲另一個型別。

幸運的是，有個辦法可以繞個彎解除此一束縛，而其觀念很簡單（雖然實作上不簡單）：令每一個 smart pointer class 有個隱式型別轉換運算子（見條款 5），用來轉換至另一個 smart pointer class。舉個例子，上述音樂產品的繼承體系中，你可以爲 `Cassette` 和 `CD` 的 smart pointer classes 加上一個 `SmartPtr<MusicProduct>` 運算子：

```
class SmartPtr<Cassette> {
public:
  operator SmartPtr<MusicProduct>()  // 譯註：這便是新增的轉換運算子
  { return SmartPtr<MusicProduct>(pointee); }
  ...
private:
  Cassette *pointee;
};

class SmartPtr<CD> {
public:
  operator SmartPtr<MusicProduct>()  // 譯註：這便是新增的轉換運算子
  { return SmartPtr<MusicProduct>(pointee); }
  ...
private:
  CD *pointee;
};
```

此作法有兩個缺點。第一，你必須一一爲每一個「`SmartPtr` class 具現體」加入上述特殊式子，如此一來才有機會加上必要的隱式型別轉換運算子。但這對 template 而言無異是一大諷刺。第二，你可能必須加上許多如此這般的轉換運算子，因爲你所指的物件可能位於繼承體系的底層，而你必須爲物件直接繼承或間接繼承之每一個 base class 提供一個轉換運算子。如果你認爲你只需爲每個直接繼承的 base class 提供隱式型別轉換運算子就好，請三思。由於編譯器禁止一次執行一個以上的「使用者自定之型別轉換函式」，所以它們不能夠將一個 smart pointer-to-`T` 轉換爲一個 smart pointer-to-**indirect-base-class-of**-`T` — 除非它們能夠在單一步驟中完成它。

如果有辦法讓編譯器爲你完成所有這些隱式型別轉換函式，便可以節省許多時間。謝天謝地，由於新近加入的一個語言擴充性質，你真的可以辦到。這個擴充性質就是將 nonvirtual member function 宣告爲 templates，你可以利用它來產生 smart pointer 的

轉換函式，像這樣：

```
template<class T>         // template class，用於
class SmartPtr {          // smart pointers-to-T 物件。
public:
  SmartPtr(T* realPtr = 0);

  T* operator->() const;
  T& operator*() const;

  template<class newType>           // template function，用於
  operator SmartPtr<newType>()      // 隱式轉換運算子。
  {
    return SmartPtr<newType>(pointee);
  }
  ...
};
```

現在，穩住，這不是魔術 — 不過也很接近了。上述 template 運作如下。（稍後我會給一個例子，所以如果本段剩餘的部份讀起來刻板枯燥，請不要灰心。在你看過實例之後，我保證你會更有感覺。）假設編譯器有個 smart pointer-to-T 物件，而它需要將此物件轉換爲 smart pointer-to-**base-class-of**-T。編譯器檢驗 class 內部對 SmartPtr<T> 的定義，看看其中是否宣告有必要的轉換運算子，結果沒有（不可能有：上述 template 未曾宣告任何轉換運算子）。編譯器於是再檢查看看是否有哪一個 member function template 是它可以具現化的，俾使得以完成外界要求的轉換行爲。它發現了這樣一個 template（其型別參數爲 newType），所以它將此 template 具現化，並將 newType 繫結於其 base class 的型別參數 T 身上，T 正是轉換目標。此時唯一的問題就是，被具現出來的 member function 是否可通過編譯。爲了讓它得被編譯，將 dumb pointer pointee 交給「smart pointer-to-**base-of**-T 的 constructor」這一動作必須合法。pointee 的型別是 T，所以將它轉換爲一個指標，指向自己「以 public 方式繼承或以 protected 方式繼承」之 base classes，當然合法。因此，上述的型別轉換運算子可通過編譯，而從 smart pointer-to-T 至 smart pointer-to-**base-of**-T 的隱式轉換因此得以成功。

看一個實際範例會比較有感覺。讓我們回到 CDs, Cassettes 和 MusicProducts 繼承體系。稍早我們看到下列程式碼無法編譯，因爲沒有任何途徑可以讓編譯器將「指向 CDs 或 Cassettes」的 smart pointers 轉換爲「指向 MusicProducts」的 smart pointers：

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));

displayAndPlay(funMusic, 10);           // 原本是錯誤的
displayAndPlay(nightmareMusic, 0);      // 原本是錯誤的
```

如果使用修訂版的 smart pointer class，令其帶有 member function template，做爲隱式型別轉換運算子之用，上述程式碼便能成功。爲什麼？看看這個呼叫：

```
displayAndPlay(funMusic, 10);
```

`funMusic` 物件屬於 `SmartPtr<Cassette>` 型別，而 `displayAndPlay` 函式期望得到的是個 `SmartPtr<MusicProduct>` 物件。編譯器偵測到型別不吻合，於是尋找某種方法，企圖將 `funMusic` 轉換爲一個 `SmartPtr<MusicProduct>` 物件。編譯器在 `SmartPtr<MusicProduct>` class 內企圖尋找一個「單一引數之 constructor」（見條款 5），且其引數型別爲 `SmartPtr<Cassette>`，但是沒有找到。於是再接再勵在 `SmartPtr<Cassette>` class 內尋找一個隱式型別轉換運算子，希望可以產出一個 `SmartPtr<MusicProduct>` class。但是也失敗了。接下來編譯器再試圖尋找一個「可具現化以導出合宜之轉換函式」的 member function template。這一次它們在 `SmartPtr<Cassette>` 找到了這樣一個東西，當它被具現化並令 `newType` 繫結至 `MusicProduct` 時，產生了所需函式。於是編譯器將該函式具現化，導出以下函式碼：

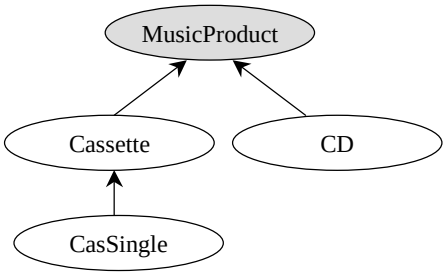
```
SmartPtr<Cassette>::operator SmartPtr<MusicProduct>()
{
  return SmartPtr<MusicProduct>(pointee);
}
```

這可編譯嗎？這裡面什麼都沒有，只是呼叫 `SmartPtr<MusicProduct>` constructor，並以 `pointee` 做爲引數。所以真正的問題是，你是否可以以一個 `Cassette*` 指標建構出一個 `SmartPtr<MusicProduct>` 物件。`SmartPtr<MusicProduct>` constructor 期望獲得一個 `MusicProduct*` 指標，但現在我們面對的是 dumb pointer 型別間的轉換，所以很明顯地 `Cassette*` 可被交給一個期望獲得 `MusicProduct*` 的函式。因此 `SmartPtr<MusicProduct>` 的建構會成功，而 `SmartPtr<Cassette>` 至 `SmartPtr<MusicProduct>` 的轉換也會成功。太棒了，這就是 smart pointer 的隱式型別轉換。還有什麼能夠比這更簡單？

猶有進者，還有什麼能夠比這更具威力？不要被此例誤導，以爲此法只適用於繼承體系的向上轉型。事實上此法對於指標型別之間的任何合法隱式轉換都能成功。如果你

手上有個 dumb pointer 型別爲 `T1*`，另一個 dumb pointer 型別爲 `T2*`，只要你能夠將 `T1*` 隱式轉換爲 `T2*`，你便能夠將 smart pointer-to-`T1` 隱式轉換爲 smart pointer-to-`T2`。

這項技術完全賦予你所想要的行爲 — 幾乎啦。假設我擴充 `MusicProduct` 繼承體系，加上一個新的 `CasSingle` class。修訂後的繼承體系如下：



現在考慮這份碼：

```
template<class T>              // 如上，其中包括針對轉換運算子
class SmartPtr { ... };        // 而設計的 member template。

void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);

void displayAndPlay(const SmartPtr<Cassette>& pc,
                    int howMany);

SmartPtr<CasSingle> dumbMusic(new CasSingle("Achy Breaky Heart"));

displayAndPlay(dumbMusic, 1);       // 錯誤！
```

此例之中，`displayAndPlay` 被多載化，函式之一接受 `SmartPtr<MusicProduct>` 物件，函式之二接受 `SmartPtr<Cassette>` 物件。當我們喚起 `displayAndPlay` 並給予一個 `SmartPtr<CasSingle>`，我們預期喚起的是 `SmartPtr<Cassette>` 函式，因爲 `CasSingle` 直接繼承自 `Cassette` 而僅只間接繼承自 `MusicProduct`。如果面對的是 dumb pointers，當然如此抉擇。但我們的 smart pointers 沒有那麼機靈，它們把 member functions 拿來做爲轉換運算子使用，而 C++ 的理念是，對任何轉換函式的呼叫動作，無分軒輊一樣地好。於是，`displayAndPlay` 的呼叫動作成爲一種模稜兩可的行爲，因爲從 `SmartPtr<CasSingle>` 轉換至 `SmartPtr<Cassette>`，並不比轉換至 `SmartPtr<MusicProduct>` 更好。

利用 member templates 來轉換 smart pointer ，有另外兩個缺點。第一，目前支援 member templates 的編譯器還不多，此技術雖好，移植性不高。未來情勢當然會改變，但是沒有人知道未來有多麼遙遠。第二，其間涵蓋的技術並不那麼簡單，你必須熟悉 (1) 函式呼叫的引數匹配規則、(2) 隱式型別轉換函式、(3) template functions 的暗自具現化、(4) member function templates 等技術。憐憫憐憫那些從未見過這些深層技術，卻被要求以它們來維護或改善程式的可憐人吧。這項技術很機靈，無庸置疑，但是太過先進反而可能是件危險的事情。

讓我們停止在這塊尚未廣被開墾的蠻荒大地上繼續投注精力吧。我們真正想要知道的是如何能夠使「smart pointer classes 的行爲」在「與繼承相關的型別轉換」上，能夠和 dumb pointers 一樣。答案很簡單：不能夠。Daniel Edelson 下過一個註解：smart pointers 雖然 *smart*，卻不是 *pointers*。是的，我們所能做的最好情況就是使用 member templates 來產生轉換函式，然後再在其中出現模稜兩可的時候使用轉型動作（見條款 2）。這並不完美，但是夠好。Smart pointers 提供了精巧的功能，而「必須使用轉型動作以避免模稜兩可」則是我們有時候必須付出的一個小小代價。

Smart Pointers 與 const

回憶過去使用 dumb pointers 的時光，`const` 可用來修飾被指之物，或是指標本身，或是兩者兼具（見條款 E21）：

```
CD goodCD("Flood");

const CD *p;                          // p 是一個 non-const 指標，
                                      // 指向一個 const CD object。

CD * const p = &goodCD;               // p 是一個 const 指標，
                                      // 指向一個 non-const CD object；
                                      // 由於 p 是 const，所以必須有初值。

const CD * const p = &goodCD;         // p 是一個 const 指標，
                                      // 指向一個 const CD object。
```

很自然地我們希望 smart pointers 也有相同的彈性。不幸的是面對 smart pointers 只有一個地方可以放置 `const`：只能施行於指標身上，不能及於其所指物件：

```
const SmartPtr<CD> p =                // p 是一個 const smart ptr，
  &goodCD;                            // 指向一個 non-const CD object。
```

矯正之道似乎很簡單 — 產生一個 smart pointer，指向一個 const CD：

```
SmartPtr<const CD> p =              // p 是一個 non-const smart ptr，
  &goodCD;                          // 指向一個 const CD object。
```

現在我們可以對 const 以及 non-const 的物件及指標，產生四種組合：

```
SmartPtr<CD> p;                           // non-const object,
                                          // non-const pointer.
SmartPtr<const CD> p;                     // const object,
                                          // non-const pointer.
const SmartPtr<CD> p = &goodCD;           // non-const object,
                                          // const pointer.
const SmartPtr<const CD> p = &goodCD;     // const object,
                                          // const pointer.
```

啊呀，這塊蛋糕上面有一隻蒼蠅耶。如果使用 dumb pointers，我們可以 non-const 指標做爲 const指標的初值，也可以「指向 non-const 物件」之指標做爲「指向 const 物件」之指標的初值; 設值（指派，assignment）規則亦類似。例如：

```
CD *pCD = new CD("Famous Movie Themes");
const CD * pConstCD = pCD;                    // 可以
```

但是如果我們在 smart pointers 身上做相同動作，看看會發生什麼事：

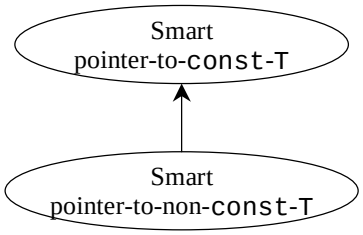
```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");
SmartPtr<const CD> pConstCD = pCD;            // 可以嗎？
```

SmartPtr<CD> 和 SmartPtr<const CD> 是完全不同的型別。就目前的編譯器而言，它們之間毫無關係，所以沒有理由認爲它們之間可以彼此指派（設值）。唯一能夠讓兩個型別被視爲「可互相指派（設值）」的情況是，如果你提供有一個函式，能夠將 SmartPtr<CD> 物件轉換爲 SmartPtr<const CD> 物件。如果你的編譯器支援 member templates，你可以使用先前所示的技術，自動產出你所需要的隱式型別轉換運算子（稍早我還特別提醒過，只要對應之 dumb pointers 可以轉換成功，這項技術就可以成功。我可不是隨便說說的。轉換如果涉及 const，也不例外）。如果你沒有這樣的編譯器，你必須多跳幾次火圈。

型別轉換如果涉及 const，便是一條單行道：從 non-const 轉換至 const 是安全的，從 const 轉換至 non-const 則不安全。此外，你能夠對 const 指標所做的任何事情，也都可以對 non-const 指標進行，但你還可以對後者做更多事情（例如設

值，assignment）。同樣道理，你能夠對 pointer-to-const 所做的任何事情，也都可以對 pointer-to-non-const 做，但是面對後者，你還可以另做其他事情（例如設值，assignment）。

這些規則聽起來和 public inheritance 的規則（見條款 E35）很類似。是的，你可以將 derived class object 轉換爲 base class object，反之不然。你可以對 derived class object 做「可對 base class object 進行」的任何事情，但通常還可以對 derived class object 做更多事情。實作 smart pointers 時我們可以利用這種類似性質，令每一個 smart pointer-to-T class 公開繼承一個對應的 smart pointer-to-const-T class：




```
template<class T>              // smart pointers，用於 const 物件
class SmartPtrToConst {
  ...                       // 一般都會有的 smart pointer member functions
protected:
  union {
    const T* constPointee;     // 給 SmartPtrToConst 使用
    T* pointee;                // 給 SmartPtr 使用
  };
};

template<class T>              // smart pointers，用於 non-const 物件
class SmartPtr:
  public SmartPtrToConst<T> {
  ...                          // 沒有 data members
};
```

使用這樣的設計，smart pointer-to-non-const-T 物件必須內含一個 dumb pointer-to-non-const-T，而 smart pointer-to-const-T 必須內含一個 dumb pointer-to-const-T。一個天真的想法是放一個 dumb pointer-to-const-T 於 base class 中，並放一個 dumb pointer-to-non-const-T 於 derived class 中。然而這是一種浪費，因爲 SmartPtr 物

件會因此內含兩個 dumb pointers：其一繼承自 `SmartPtrToConst`，其二是 `SmartPtr` 本身所有。

這個問題可藉由 C 語言的舊武器 union 獲得解決。union 在 C++ 中的表現就像在 C 一樣。此一 union 的存取層級應該是 protected，俾使兩個 classes 都可取用。其中內含兩個必要的 dumb pointer 型別：`constPointee` 指標供 `SmartPtrToConst<T>` 物件使用，`pointee` 指標則供 `SmartPtr<T>` 物件使用。我們因此獲得了兩個互異指標的優點，又不必配置多餘空間（條款 E10 有另一個關於 union 的例子）。這正是 union 漂亮之處。當然啦，兩個 classes 的 member functions 必須約束自己，只使用適當的指標。編譯器無法協助你厲行這項規範。這是使用 union 的風險。

運用這個新設計，我們獲得了我們希望的行爲：

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");
SmartPtrToConst<CD> pConstCD = pCD;     // 沒問題。
```

評估

該結束 smart pointers 這個主題了，但是離開之前，應該問自己一個問題：如此大費周張，值得嗎？特別是如果你的編譯器尚未支援 member function templates 的話。

答案通常是肯定的。例如，條款 29 所展示的參用計數（reference-counting），就可以利用 smart pointers 加以簡化。然而，就如本條款的例子所示，某些 smart pointers 的用途受到諸如「nullness 之測試、dumb pointers 之轉換、以繼承爲本之轉換、對 pointers-to-`consts` 之支援」等等限制。而且 smart pointers 可能不容易實作、瞭解、維護。欲對運用了 smart pointers 的程式碼除錯，會比對運用了 dumb pointers 的程式碼除錯更困難。你絕對無法成功設計出一個泛用型 smart pointer，可以完全無間隙地取代 dumb pointer。

儘管如此，smart pointers 能夠爲你完成一些原本極端困難達成的效果。Smart pointers 應該被明智而謹慎地運用，但每位 C++ 程式員一定都會在某些時候或某種場合下發現，它們的確很有用。

條款 29：Reference counting（參引計數）

Reference counting 這項技術，允許多個等值物件共享同一實值。此技術之發展有兩個動機，第一是爲了簡化 heap objects 周邊的簿記工作。一旦某個物件以 new 配置出來，記錄「物件的擁有者」是件重要的事，因爲他（也只有他）有責任刪除該物件。但是程式執行過程中，物件的擁有權可能移轉（例如以指標做爲函式引數），所以記錄物件的擁有權並非是件輕鬆的工作。類似 auto_ptr 那樣的 classes（見條款 9）可以協助此類工作，但是經驗顯示，大部份程式仍然沒有好好地利用它。Reference counting 可以消除「記錄物件擁有權」的負荷，因爲當物件運用了 reference counting 技術，它便擁有它自己。一旦不再有任何人使用它，它便自動摧毀自己。也因此，reference counting 架構出垃圾收集機制（garbage collection）的一個簡單型式。

Reference counting 的第二個發展動機則只是爲了實現一種常識。如果許多物件有相同的值，將那個值儲存多次是件愚蠢的事。最好是讓所有等值物件共享一份實值就好。這麼做不只節省記憶體，也使程式速度加快，因爲不再需要建構和解構同值物件的多餘副本。

Reference counting 架構在一大堆有趣的細節上面。其中或許談不上什麼真理，但是靠著這些細節，才有辦法成功實作出 reference counting 技術。鑽研細節之前，讓我們先掌握基礎面。首先讓我們看看當初如何造成許多同值物件。下面是一種可能：

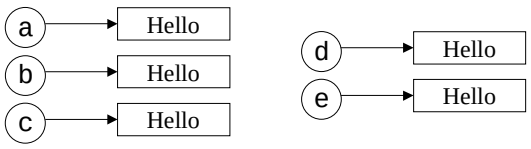
```
class String {                          // 標準的 string 型別有可能運用
public:                                 // 本條款技術，但非絕對必要。
  String(const char *value = "");
  String& operator=(const String& rhs);
  ...
private:
  char *data;
};

String a, b, c, d, e;

a = b = c = d = e = "Hello";
```

很顯然物件 a～e 都有相同的值 "Hello"。該值的呈現方式視 String class 如何實

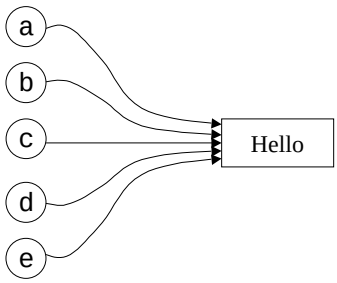
作而定，通常每一個 `String` 物件會攜帶自己的一份資料。例如，`String` 的指派（assignment）運算子可能實作如下：

```
String& String::operator=(const String& rhs)
{
  if (this == &rhs) return *this;            // 見條款 E17

  delete [] data;                            // 譯註：刪除舊資料
  data =   new char[strlen(rhs.data) + 1];   // 譯註：配置新空間
  strcpy(data, rhs.data);                    // 譯註：設定新值

  return *this;                              // 見條款 E15
}
```

有了這份實作碼，我們可以將上述五個物件及其所含資料想像如下：



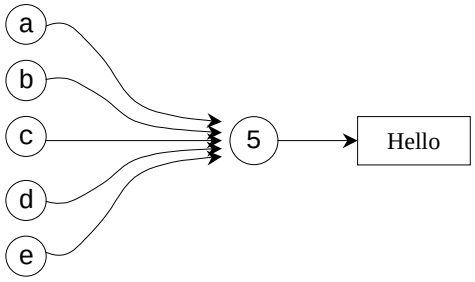
此法之累贅與浪費，清晰可見。理想情況下我們希望把這張圖改變爲：



這裡只存在一份 `"Hello"`，所有「以 `"Hello"` 爲實值」的 `String` 物件，共享同一份資料實體。

實際上不可能到達這樣的理想，因爲我們必須追蹤記錄有多少個物件共享此值。如果上述物件 `a` 稍後被指派一個異於 `"Hello"` 的值，我們不可以摧毀 `"Hello"`，因爲其他四個物件仍然需要它。從另一個角度說，如果只有一個物件的值是 `"Hello"` 而且此物件即將離開其生存空間，不再有任何物件擁有該值，那麼我們就必須摧毀該值以免資源遺失。

由於需要某些資訊用以記錄「目前共享同一實值」的物件個數，我們的理想圖應該做點修改，將一個 reference count（參用計數器）納入：



有人稱此數值爲 use count，但我不採用這個詞。啊，是的，無可避免，C++ 有許多技術上的黨派之爭 ☺。

Reference Counting（參用計數）之實作

產生一個 reference-counted `String` class 並不困難，但必須注意許多細節。我將帶你走過這類 class 最常見的一些 member functions。在此之前，認知我們「需要某個空間，用以爲每一個 `String` 儲存參用次數」是很重要的。那個空間不可以設計在 `String` 物件內，因爲我們是要爲每一個字串值準備一個參用次數，而不是爲每一個字串物件準備。這暗示了「物件實值」和「參用次數」之間有一種耦合（coupling）關係，所以我打算產生一個 class，不但儲存參用次數，也儲存它們所追蹤的物件實值。此 class 被我命名爲 `StringValue`。由於其存在的唯一目的就是協助實作 `String` class，所以我把它巢狀放進 `String` 的 private 段落內。此外，如果讓所有的 `String` member functions 都能夠存取 `StringValue`，會帶來很大的方便，所以我將 `StringValue` 宣告爲 struct。這是一個值得注意的技巧：將一個 struct 巢狀放進一個 class 的 private 段落內，可以很方便地讓該 class 的所有 members 有權處理這個 struct，而又能夠禁止任何其他人存取這個 struct（當然，class 的 friends 不在此限）。

我們的基本設計看起來像這樣：

```
class String {
public:
  ...                    // 一般的 String member functions 安排在這裡
private:
  struct StringValue { ... };    // 持有一個參用次數（reference count）
                                 // 以及一個字串實值。
  StringValue *value;            // String 的值
};
```

或許你認爲應該給上述 class 一個不一樣的名稱（RCString 如何？），強調它是以 reference counting 技術完成，但是 class 的實作細目不應該是客戶關心的焦點，客戶只對 class 的公開介面感興趣。上述 String reference-counting 版本的操作介面與 non-reference-counted 版本完全一致，所以何必弄縐一池春水地將「實作細目上的決定」嵌入「原本用來彰顯抽象概念」的 classes 名稱內呢？真的沒有必要，所以我們不這麼做。

下面是 StringValue 的定義：

```
class String {
private:
  struct StringValue {
    int refCount;
    char *data;

    StringValue(const char *initValue);
    ~StringValue();
  };
  ...
};

String::StringValue::StringValue(const char *initValue)
: refCount(1)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::~StringValue()
{
  delete [] data;
}
```

這便是全部，很明顯其中尚未實作出 reference-counted 字串該有的完整機能。就拿一件事來說，既沒有 copy constructor 也沒有 assignment operator（見條款 E11），

甚至沒有 refCount 欄位的處理函式。別擔憂，這些遺漏的機能將由 String class 補足。StringValue 的主要目的是提供一個地點，將「某特定值」以及「共享該值的 String 物件個數」關連起來。這樣的 StringValue 應該就夠用了。

現在我準備完成 String 的 member functions。讓我們從 constructors 開始：

```
class String {
public:
  String(const char *initValue = "");
  String(const String& rhs);
  ...
};
```

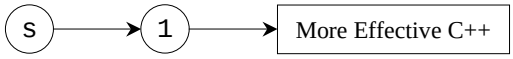
第一個 constructor 的實作儘可能簡單。我們根據傳進來的 char* 字串產生一個新的 StringValue 物件，然後讓建構中的這個 String 物件指向新鑄造出來的那個 StringValue：

```
String::String(const char *initValue)
: value(new StringValue(initValue))
{}
```

於是，以下運用：

```
String s("More Effective C++");
```

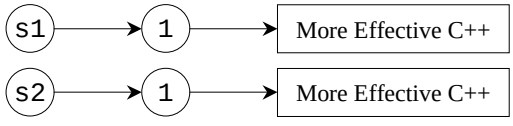
會導至形勢如下的資料結構：



「分開建構，但擁有相同初值」的 String 物件，並不共享同一個資料結構。所以以下運用型式：

```
String s1("More Effective C++");
String s2("More Effective C++");
```

會導至這樣的資料結構：



只要令 String（或 StringValue）追蹤現有的 StringValue 物件，並僅僅在「面對真正獨一無二的字串時」才產生新的 StringValue 物件，上圖所顯示的重複空間

便可消除。這樣的精雕細琢有點恐怖也有點令人憎恨，就留給讀者做練習吧。

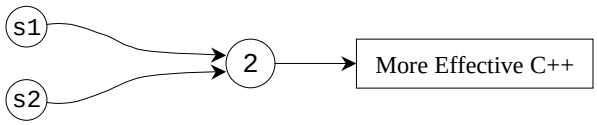
至於 String copy constructor，不僅不令人恐怖憎恨，還很有效率：當 String 物件被複製，新產生的 String 物件共享同一個 StringValue 物件：

```
String::String(const String& rhs)
: value(rhs.value)
{
  ++value->refCount;
}
```

如果以圖形表示，下面這樣的碼：

```
String s1("More Effective C++");
String s2 = s1;
```

會導至這樣的資料結構：



這比傳統的 non-reference-counted String class 效率高。因爲它不需要配置記憶體給字串實值的第二個副本使用，也不需要於稍後釋還記憶體，更不需要將字串實值拷貝到記憶體中。這裡只需將指標複製一份，並將參用次數加 1。

String destructor 也很容易完成，因爲大部份時間它什麼事也沒做。只要某個 StringValue 的參用次數不是 0，就表示至少有一個 String 物件正在使用該值；因此它一定不能被摧毀。如果被解構之 String 是該值的唯一使用者，也就是如果該值的參用次數爲 1，String destructor 才應該摧毀 StringValue 物件：

```
class String {
public:
  ~String();
  ...
};

String::~String()
{
  if (--value->refCount == 0) delete value;
}
```

讓我拿它來和 non-reference-counted 版的 destructor 效率比較一下。後者總是呼叫 `delete` 並幾乎確定有不低的執行成本。但是如果不同的 `String` 物件擁有相同的值，上述作法除了將計數器減 1 並與 0 相比之外，什麼也沒做。

如果，在此時刻，reference counting 的訴求並不明顯，上述作法也不會讓你特別費神。

這就是 `String` 的建構和解構。現在讓我們移動位置，考慮 `String` 的指派（assignment）運算子：

```
class String {
public:
  String& operator=(const String& rhs);
  ...
};
```

當使用者寫下這樣的碼：

```
s1 = s2;          // s1 和 s2 都是 String 物件
```

指派結果應該是：`s1` 和 `s2` 指向同一個 `StringValue` 物件，該物件的參用次數應該在指派過程中加 1。此外，`s1` 於指派動作之前所指的 `StringValue` 物件的參用次數應該減 1，因爲 `s1` 不再擁有該值。如果 `s1` 原本是該值的唯一擁有者，該值此刻就面臨被摧毀的時機了。在 C++ 中，一切看起來是這樣子：

```
String& String::operator=(const String& rhs)
{
  if (value == rhs.value) {         // 如果數值相同，什麼也不做。
    return *this;                   // 這很類似對 &rhs 的慣常測試
  }                                 // （見條款 E17）。

  if (--value->refCount == 0) {     // 如果沒有其他人使用，
    delete value;                   // 就摧毀 *this 的數值。
  }

  value = rhs.value;                // 令 *this 共享 rhs 的數值。
  ++value->refCount;

  return *this;
}
```

Copy-on-Write（塗寫時才複製）

爲了完整驗證 reference-counted 字串，讓我們繼續考慮中括號運算子（[]），它允許字串的個別字元被讀取或被塗寫：

```
class String {
public:
  const char&
    operator[](int index) const;          // 針對 const Strings

  char& operator[](int index);            // 針對 non-const Strings
...
};
```

此函式的 const 版實作起來十分直接，因爲那是個唯讀動作；字串內容不受影響：

```
const char& String::operator[](int index) const
{
  return value->data[index];
}
```

（這個函式以 C++ 的傳統堂而皇之地執行索引值穩健測試 — 其實是什麼都沒測試。如果你希望對參數做更大程度的有效性測試，應該很容易加入）

operator[] 的 non-const 版本就有完全不同的故事。這個函式可能用來讀取一個字元，也可能用來塗寫一個字元：

```
String s;
...
cout << s[3];                         // 這是一個讀取動作
s[5] = 'x';                           // 這是一個塗寫動作
```

我們希望讀取動作和塗寫動作的處理不一樣。讀取動作的處理可以和上述的 operator[] const 版相同，但是塗寫動作必須以完全不同的方式完成。

當我們修改某個 String 的值時，必須小心避免更動「共享同一個 StringValue 物件」的其他 String 物件。不幸的是，C++ 編譯器無法告訴我們 operator[] 被用於讀取或塗寫。所以我們必須悲觀地假設 non-const operator[] 的所有呼叫都是用於塗寫。（條款 30 介紹的 proxy classes 可以幫助我們區分讀取或塗寫）

爲了安全實作出 non-const operator[]，我們必須確保沒有其他「共享同一 StringValue」的 String 物件因塗寫動作而改變。簡單地說，任何時候當我們傳

回一個 reference，指向 String 的 StringValue 物件內的一個字元時，我們必須確保該 StringValue 物件的參用次數確確實實爲 1。以下是我的作法：

```
char& String::operator[](int index)
{
  // 如果本物件和其他 String 物件共享同一實值，
  // 就分割（複製）出另一個副本供本物件自己使用。
  if (value->refCount > 1) {
    --value->refCount;                    // 將目前實值的參用次數減 1，
                                          // 因爲我們不再使用該值。
    value =                               // 爲自己做一份新副本。
      new StringValue(value->data);
  }
  // 傳回一個 reference，代表我們這個「絕對未被共享」的
  // StringValue 物件內的一個字元。
  return value->data[index];
}
```

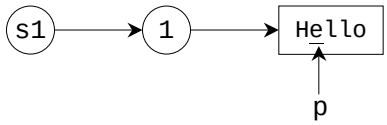
「和其他物件共享一份實值，直到我們必須對自己所擁有的那一份實值進行塗寫動作」，這個觀念在電腦科學領域中有很長的歷史。特別是在作業系統領域，各行程（processes）之間往往允許共享某些記憶體分頁，直到它們打算修改屬於自己的那一頁。這項技術是如此普及，因而有一個專用名稱：copy-on-write（塗寫時才複製）。這是提昇效率的一般化作法（也就是 lazy evaluation，緩式評估，見條款 17）中的一劑特效藥。

Pointers, References, 以及 Copy-on-Write

實作出 copy-on-write，使我們幾乎得以同時保留效率和正確性。但是有一個徘徊不去的問題困擾著我們。考慮以下的碼：

```
String s1 = "Hello";
char *p = &s1[1];
```

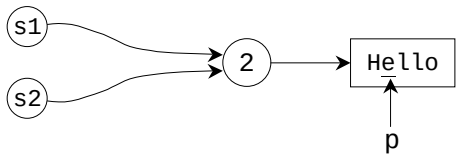
此時資料結構如下：



現在加上新的一行：

```
String s2 = s1;
```

`String` copy constructor（譯註：p188）會造成 `s2` 共享 `s1` 的 `StringValue`，所以最新資料結構圖如下：



但是這暗示，下面的句子會讓人不愉快：

```
*p = 'x';                    // 同時修改 s1 和 s2！
```

`String` copy constructor 沒辦法偵測出這個問題，因爲它沒辦法知道目前存在一個指標指向 `s1` 的 `StringValue` 物件。這個問題不限指標，如果有人將「`String` 的 non-`const` `operator[]` 傳回值」的 reference 儲存起來，也會發生同樣的困擾。

至少有三個方法可以處理此問題。第一個方法是忽略之，假裝不存在。此種鋸箭法很不幸普遍爲「實作有 reference-counted 字串」的 class 程式庫採用。如果你正在使用 reference-counted 字串，不妨試試上述例子，看看是否出現相同的困擾。如果你不確定你所存取的是否爲 reference-counted 字串，無論如何請試試此例。因爲，透過封裝的神奇魔力，很可能你正在使用此類型別而不自知。

並非所有產品都會對此問題視而不見。有些產品知道這個問題，雖然無力（或不打算）解決，卻會在文件中多少做些說明：『不要那麼做。如果你不聽，後果無可預期』。如果你逞強做了 — 不論有意或無意 — 並抱怨結果，他們便說：『唔，我不是早告訴過你別那麼做了嗎』。這樣的產品往往擁有良好效率，但離實用性還有一大段距離。

第三個方法實作上並不困難，但會降低物件之間共享的實值個數。基本觀念是：爲每一個 `StringValue` 物件加上一個旗標變數，用以指示可否被共享。一開始我們先豎立此旗標（表示物件可被共享），但只要 non-`const` `operator[]` 作用於物件實值身上，就將旗標清除。一旦旗標被清除（設立爲 `false`），可能永遠不再改變狀態[2]。

[2] C++ 標準程式庫（見條款 E49 和 35）所提供的 `string` 型別使用上列 2, 3 兩種作法的組合。Non-`const` `operator[]` 傳回的 reference 在下一次呼叫「可能修改該字串」之函式前，保證有效。但是在那時機之後，使用該 reference（或是它所代表的字元）會導至未定義的結果。這允許字串的「可共享旗標」被重設爲 `true` — 只要呼叫了某個可能修改字串的函式。

下面是 StringValue 的修改版，包含一個「可共享」旗標：

```
class String {
private:
  struct StringValue {
    int refCount;
    bool shareable;              // 新增這個
    char *data;
    StringValue(const char *initValue);
    ~StringValue();
  };
...
};

String::StringValue::StringValue(const char *initValue)
:   refCount(1),
    shareable(true)              // 新增這個
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::~StringValue()
{
  delete [] data;
}
```

如你所見，沒有太多東西需要改變；修改的兩行已經標記出來。當然，String member functions 亦必須更新，以便將「可共享欄位」納入考量。以下是 copy constructor 的新作爲：

```
String::String(const String& rhs)
{
  if (rhs.value->shareable) {
    value = rhs.value;
    ++value->refCount;
  }
  else {
    value = new StringValue(rhs.value->data);
  }
}
```

所有其他的 String member functions 都應該以類似方式檢查「可共享欄位」。Non-const operator[] 是唯一將「可共享欄位」設爲 false 者：

```
char& String::operator[](int index)
{
  if (value->refCount > 1) {
    --value->refCount;
    value = new StringValue(value->data);
  }
  value->shareable = false;          // 新增此行
  return value->data[index];
}
```

如果我們使用條款 30 的 proxy class 技術，將 `operator[]` 的讀取和塗寫用途區分開來，通常便可降低「需被標記爲不可共享」之 `StringValue` 物件的個數。

一個 Reference-Counting（參用計數）基礎類別

Reference-counting 可用於字串以外的場合。任何 class 如果其不同的物件可能擁有相同的值，都適用此技術。然而重寫 class 以便運用 reference counting，可能是個大工程，大部份的我們該做的事情還很多。如果我們能夠在一個與外界無任何關聯的環境下撰寫 reference counting 碼（並測試及說明），然後在必要時機把它移植到 classes 身上，豈不甚妙？那當然是很好。好奇心改變了命運，是的，有一種辦法可以完成它（或至少完成大部份目標）。

第一個步驟是，首先產生一個 base class `RCObject`，做爲「reference-counted 物件」之用。任何 class 如果希望自動擁有 reference counting 能力，都必須繼承自這個 class。`RCObject` 將「參用計數器」本身以及「增減計數器」的函式封裝進來。此外還包括一個函式，用來將不再被使用（也就是其參用次數爲 0）的實值摧毀掉。最後，它還內含一個欄位，用來追蹤其值是否「可共享」，並提供查詢其值、將欄位設爲 `false` 等相關函式。沒有必要提供一個函式讓外界設定該欄位爲 `true`，因爲所有實值預設均爲可共享。一如先前所提示，一旦某個物件被貼上「不可共享」的標籤，就沒有辦法再恢復其「可共享」的身份了。

`RCObject` 定義如下：

```
class RCObject {
public:
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;
```

```
  void addReference();
  void removeReference();
  void markUnshareable();
  bool isShareable() const;
  bool isShared() const;
private:
  int refCount;
  bool shareable;
};
```

RCObjects 可被產生（做爲衍生物件的 base class 成份），亦可被摧毀；可被賦予更高的參用次數，也可移除目前的參用次數。它們的共享狀態可被查詢，也可被除能（disabled）；它們可以回報目前是否正被共享。這便是它們所提供的全部。當一個 class 封裝了這種「有參用計數能力」的概念，我們便能預期它們擁有上述各機能。請注意其中的 virtual destructor，表示這個 class 被設計做爲一個 base class（見條款 E14）。也請注意這個 destructor 是純虛擬函式，表示此 class 只被設計用來做爲 base class。

RCObject 的實作碼十分簡明扼要：

```
RCObject::RCObject()
: refCount(0), shareable(true) {}

RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}

RCObject& RCObject::operator=(const RCObject&)
{ return *this; }

RCObject::~RCObject() {}       // virtual dtors 必須被實作出來，
                               // 即使它們是純虛擬函式而且不做
                               // 任何事情（見條款 33 和條款 E14）
void RCObject::addReference() { ++refCount; }

void RCObject::removeReference()
{ if (--refCount == 0) delete this; }

void RCObject::markUnshareable()
{ shareable = false; }

bool RCObject::isShareable() const
{ return shareable; }

bool RCObject::isShared() const
{ return refCount > 1; }
```

說來或許古怪，我們在兩個 constructors 中都將 `refCount` 設爲 0。這似乎違反直覺。至少 `RCObject` 的生產者正在使用它，不是嗎！其實這是爲了簡化事情，俾使物件產生者自行將 `refCount` 設爲 1，所以此處不得不如此。很快我們便會看到其所導至的簡化效果。

另一件看似奇怪的事情是，不論我們正要複製的那個 `RCObject` 的 `refCount` 數值是多少，copy constructor 總是將 `refCount` 設爲 0。那是因爲我們正在產生一個新物件，用以表現某值，而該新值既然曰「新」，一定尚未被共享，並且只被其生產者參用。再一次強調，`refCount` 的生產者有責任爲 `refCount` 設定適當的值。

`RCObject` 的指派（assignment）運算子看起來十足是個危險份子：它什麼也沒做。坦白說，這個運算子不太可能被呼叫。要知道，`RCObject` 是針對「實值可共享」之物件而設計的一個 base class，在一個擁有 reference counting 能力的系統中，此等物件並不會被指派給另一個物件。先前例子中，我們並不認爲 `StringValue` 物件會被指派，我們只認爲 `String` 物件會被指派。這樣的指派動作中，`StringValue` 的實值不會有任何改變 — 只有 `StringValue` 的參用次數會被改變。

不過，或許將來會有繼承自 `RCObject` 的 classes，希望能夠對「reference-counted 實值」做指派（設值）動作也說不定（見條款 32 和條款 E16）。果真如此，`RCObject` 的 assignment 運算子應該做出正確的行爲，而其所謂正確的行爲就是啥也不做。爲什麼？想像我們允許 `StringValue` 物件之間做指派動作。假設有兩個 `StringValue` 物件 `sv1` 和 `sv2`，在一個指派動作中，`sv1` 和 `sv2` 的參用次數會發生什麼變化？

```
sv1 = sv2;                  // sv1 和 sv2 的參用次數會受怎樣的影響？
```

指派動作之前，已有某些數量的 `String` 物件指向 `sv1`；該數量不會因爲這個指派動作而有變化，因爲只有 `sv1` 的「實值」才會受此指派動作而改變。同樣情況，指派動作之前，已有某些數量的 `String` 物件指向 `sv2`；指派之後，該數量（`sv2` 的參用次數）亦不會有所變化。當 `RCObjects` 涉及指派動作，指向「左右兩方 `RCObject` 物件」的外圍物件（例如本例的 `String` 物件）的個數都不會受到影響，因此 `RCObject::operator=` 不應該改變參用次數。這就是爲什麼 `operator=` 啥都沒做的原因。違反直覺嗎？或許吧，但這是正確的行爲。

RCObject::removeReference 的責任不只在於將物件的 refCount 遞減，也在於摧毀物件 — 如果 refCount 的新值是 0 的話。完成的辦法是 deleteing this，而正如條款 27 所解釋，只有當 *this 是個 heap 物件，這個行爲才安全。爲了確保其成功，我們必須讓 RCObjects 只誕生於 heap 內。「保證誕生於 heap 內」的一般手法已於條款 27 末尾討論過，此處採用專屬手段，將在本條款最後結論中再描述。

爲了運用這個新的 reference-counting base class，我修改 StringValue，令它繼承 RCObject 的參用計數能力：

```
class String {
private:
  struct StringValue: public RCObject {
    char *data;

    StringValue(const char *initValue);
    ~StringValue();
  };
...
};

String::StringValue::StringValue(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::~StringValue()
{
  delete [] data;
}
```

這一版的 StringValue 幾乎與前一版完全相同，唯一的改變是 StringValue 的 member functions 不再處理 refCount 欄位，改由 RCObject 掌管。

「讓一個內層（nested）類別（StringValue）繼承另一個類別（RCObject），而後者與外圍（nesting）類別（String）完全無關」，如果你對此感到驚訝與陌生，不要覺得它不好。任何人初次見到這樣的安排都會覺得怪異，但這其實非常合適。nested class 和所有的 classes 沒有兩樣，所以它當然也能夠自由繼承其他任何 classes。假以時日，你再也不會對這樣的繼承關係多看兩眼。

自動操作 Reference Count（參用次數）

`RCObject` class 給了我們一個放置參用次數的空間，也給了我們一些 member functions，用來操作參用次數，但那些函式的呼叫動作還是一定得由我們手動式地安插到其他 class 內。並且還是有勞 `String` copy constructor 和 `String` assignment operator 呼叫 `StringValue` 物件所提供的 `addReference` 和 `removeReference`。這太笨拙了，我們希望能夠把那些呼叫動作移至一個可重複使用的 class 內，這麼一來就可以讓諸如 `String` 之類的 classes 的作者不必操心 reference counting 的任何細節。辦得到嗎？噢，C++ 不是支援重用性（reuse）嗎？

是的，的確可以辦到。並沒有什麼輕鬆的辦法可以讓所有與 reference-counting 相關的雜務都從應用性 classes 身上移走，但是有一個辦法可以爲大部份 classes 消除大部份雜務。（某些應用性 classes 可以去除 reference-counting 的所有相關雜務，但是本例的 `String` class 不是其中一員。`String` 有一個 member function 破壞了歡樂的氣氛，我想你不會太驚訝聽到那個老名字，是的，又是 non-`const` `operator[]` 惹的禍。不過，放心好了，我們終將馴服這頭怪獸）

注意，每個 `String` 物件都內含一個指標，指向 `StringValue` 物件，後者用以表現 `String` 的實值：

```
class String {
private:
  struct StringValue: public RCObject { ... };
  StringValue *value;              // 用以表現此一 String 的實值
  ...
};
```

我們必須能夠在任何時候，當任何有趣的事情發生於一個「指向 `StringValue` 物件的指標」身上時，操控該 `StringValue` 物件內的 `refCount` 欄位。所謂有趣的事情包括對指標的複製、重新指派、摧毀。如果我們能夠讓指標本身偵測這些事情，並自動執行對 `refCount` 欄位的操控動作，我們就可以自由自在無所罣礙了。不幸的是，指標是十分難搞的傢伙，欲藉由它們偵測任何事物，並自動對其偵測到的事物做反應，機率非常渺茫。幸運的是有個方法可以讓指標機靈起來：以行爲和形貌皆類似指標（但功能更多）的物件取代之。

此等物件稱爲 smart pointers，你可以在條款 28 中讀到其相關細節。以此刻的目的而言，知道以下事實就夠了：smart pointer 支援成員選取運算子（`->`）和提領運算子（`*`），就像真正的指標（我們常稱之爲 dumb pointers）一樣；它們也像 dumb pointers 一

樣具有強型特質（strongly typed），換句話說你不能夠令一個 smart pointer-to-T 指向一個型別 T 以外的物件。

下面這個 template 用來產生 smart pointers，指向 reference-counted 物件：

```
// template class，用於 smart pointers-to-T。
// T 必須支援 RCObject 介面，因此 T 通常繼承自 RCObject。
template<class T>
class RCPtr {
public:
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
  ~RCPtr();

  RCPtr& operator=(const RCPtr& rhs);

  T* operator->() const;       // 見條款 28
  T& operator*() const;        // 見條款 28
private:
  T *pointee;                  // dumb ptr
  void init();                 // 共同的初始化動作
};
```

這個 template 讓 smart pointer objects 控制其建構、設值、解構期間發生的事情。當此類事件發生，這些物件可以自動執行適當的處置行爲 — 以此處目的而言就是處理它們所指物件的 refCount 欄位。

舉個例子，當程式產生一個 RCPtr 時，其所指物件之參用次數必須加 1。沒有必要讓應用程式開發人員承擔這樣令人厭煩的細節，因爲 RCPtr constructors 可以自行處理。RCPtr 的兩個 constructors 函式碼幾乎完全相同，只有 member initialization lists 不同，所以我不打算寫兩次，我把相同的碼放進一個名爲 init 的 private member function 內，並令兩個 constructors 呼叫之：

```
template<class T>
RCPtr<T>::RCPtr(T* realPtr): pointee(realPtr)
{
  init();
}

template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs): pointee(rhs.pointee)
{
  init();
}
```

```
template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) {                    // 如果 dumb pointer 是 null，
    return;                              // 那麼 smart pointer 也是。
  }
  if (pointee->isShareable() == false) {     // 如果其值不可共享，
    pointee = new T(*pointee);               // 就複製一份。
  }
  pointee->addReference();          // 注意現在有了一個針對實值的新介面
}
```

將共同程式碼搬移到另一個函式中（如本例之 `init`），是軟體工程的楷模行爲，但是其光輝會因不正確的行爲而黯淡失色。本例這般行爲並不正確。問題出在當 `init` 需要爲實值產生一份新副本時（因原有副本不可共享），它執行以下的碼：

```
pointee = new T(*pointee);
```

`pointee` 的型別是 pointer-to-`T`，所以這個句子產生了一個新的 `T` 物件，並呼叫 `T` 的 copy constructor 進行初始化工作。如果 `RCPtr` 應用於 `String` class 中，`T` 便是 `String::StringValue`，所以上面的句子將呼叫 `String::StringValue` 的 copy constructor。然而我們並未爲該 class 宣告 copy constructor，所以編譯器會爲我們產

譯註：本中文版以 *Effective C++ CD* 爲本（見本書譯序），`CD` 版與紙本於此頁內容有不小變化，我採用內容較新的 `CD` 版，致篇幅略有變動，多出半頁空白。

另，`CD` 版對於本書 `p.209` 之 `RCIPtr` 實作碼，考慮更週詳，其程式碼多於紙本。爲求中英文版頁頁對應，我將 `p.209` 新增碼移至此處（以下）。

```
template<class T>                    // 實作出 copy-on-write (COW)
void RCIPtr<T>::makeCopy()           // 中的 copy 部份。
{
  if (counter->isShared()) {
    T *oldValue = counter->pointee;
    counter->removeReference();
    counter = new CountHolder;
    counter->pointee = new T(*oldValue);
    counter->addReference();
  }
}

template<class T>                              // non-const access;
T* RCIPtr<T>::operator->()                     // 需要 COW。
{ makeCopy(); return counter->pointee; }

template<class T>                              // non-const access;
T& RCIPtr<T>::operator*()                      // 需要 COW。
{ makeCopy(); return *(counter->pointee); }
```

生一個。這樣一個 copy constructor 將完全遵循「由 C++ 編譯器自動產生之 copy constructors」的規矩，只複製 StringValue 的 data 指標；它不會複製 data 指標所指之 char* 字串。如此行爲幾乎會在任何 class（不只是 reference-counted classes）內造成慘重災情，這也正是爲什麼你應該養成習慣，爲所有內含指標之 classes 撰寫 copy constructor（及 assignment operator）的原因（見條款 E11）。

如果想要讓 RCPtr<T> template 的行爲正確，前提是 T 必須擁有一個可對「T 所表現之實值」執行深層複製（deep copy）的 copy constructor。我們必須爲 StringValue 加上如此一個 constructor 以便搭配 RCPtr class：

```
class String {
private:
  struct StringValue: public RCObject {
    StringValue(const StringValue& rhs);
    ...
  };
  ...
};
String::StringValue::StringValue(const StringValue& rhs)
{
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);
}
```

「可執行深層複製行爲」之 copy constructor，並非是 RCPtr<T> 對 T 的唯一要求。它還要求 T 必須繼承自 RCObject，或至少提供 RCObject 的所有機能。由於 RCPtr 物件的設計僅僅用來指向 reference-counted 物件，所以這勉強可算是一個尚稱合理的假設。儘管如此，此一假設還是應該有良好的說明文件。

RCPtr<T> 的最後一個假設是，其物件所指的對象，型別需爲 T。這彷彿多此一舉，畢竟 pointee 被宣告爲型別 T*。但是 pointee 其實可能指向 T 的 derived class。例如，假設我們有一個 SpecialStringValue class，繼承自 String::StringValue：

```
class String {
private:
  struct StringValue: public RCObject { ... };
  struct SpecialStringValue: public StringValue { ... };
  ...
};
```

而我們最終擁有一個 String，內含一個 RCPtr<StringValue>，指向一個 SpecialStringValue 物件。此情況下，我們希望 init 函式中的這一行：

```
pointee = new T(*pointee);   // T 是 StringValue，但 pointee 真正
                             // 指向的卻是一個 SpecialStringValue。
```

呼叫的是 SpecialStringValue（而非 StringValue）的 copy constructor。我們可以使用 virtual copy constructor（見條款 25）來確保此事發生。本條款的 String 例子中，我們並不預期會有衍生自 StringValue 的 classes，所以也就不理會這個題目。

妥善處理了 RCPtr 的 constructors 之後，RCPtr 的其餘函式可以非常大的活潑度被迅速處理。RCPtr 的指派動作十分直接了當 — 雖然測試「新被指派的實值是否爲可共享」這一必要動作使事情稍稍複雜了些。幸運的是這個複雜度已經在先前「針對 RCPtr constructors 而設計的 init 函式」內處理掉了，所以我們再次利用 init：

```
template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {          // 如果實值沒有變化，就跳離。
    if (pointee) {
      pointee->removeReference();        // 移除現行實值上的參用次數。
    }
    pointee = rhs.pointee;               // 指向新的實值。
    init();                              // 如果可能，共享之；
  }                                      // 否則做一份屬於自己的副本。
  return *this;
}
```

Destructor 更簡單。當一個 RCPtr 被摧毀，只需簡單地移除 reference-counted 物件的參用次數就好：

```
template<class T>
RCPtr<T>::~RCPtr()
{
  if (pointee)pointee->removeReference();
}
```

如果此處壽終的 RCPtr 是標的物件的最後一個參用者，該物件將會被摧毀（在 RCObject 的 removeReference member function 內）。因此 RCPtr 物件不需操心「摧毀其所指實值」這檔事兒。

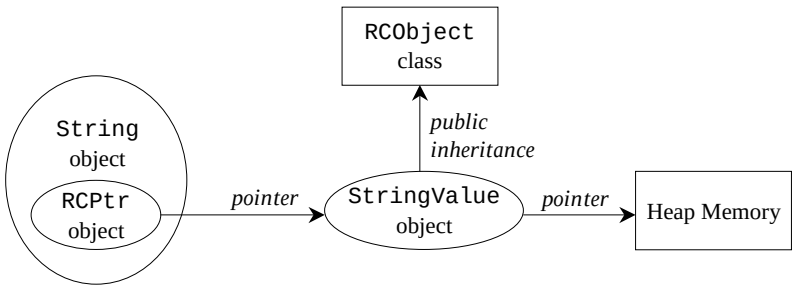
最後，RCPtr 的「指標模擬」運算子，和你在條款 28 所讀到的 smart pointer 標準作法完全相同：

```
template<class T>
T* RCPtr<T>::operator->() const { return pointee; }

template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

把所有努力放在一起

夠了！了結它吧！經過漫長的努力，我們終於能夠將所有成果放在一起， 以具有可重用性質的 RCObject 和 RCPtr classes 爲基礎，建造一個 reference-counted String class。如果夠幸運，你應該沒有忘記這是我們最初的目標。

每一個 reference-counted 字串均以此資料結構實作出來：



架構出上述資料結構的各個 classes，定義於下：

```
template<class T>                    // template class，用來產生
class RCPtr {                        // smart pointers-to-T objects；
public:                              // T 必須繼承自 RCObject。
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
  ~RCPtr();

  RCPtr& operator=(const RCPtr& rhs);

  T* operator->() const;
  T& operator*() const;

private:
  T *pointee;

  void init();
};
```

```
class RCObject {      // base class，用於 reference-counted objects
public:
  void addReference();
  void removeReference();
  void markUnshareable();
  bool isShareable() const;
  bool isShared() const;
protected:
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;
private:
  int refCount;
  bool shareable;
};
class String {       // 應用性 class，這是應用程式開發人員接觸的層面。
public:
  String(const char *value = "");
  const char& operator[](int index) const;
  char& operator[](int index);
private:
  // 以下 struct 用以表現字串實值
  struct StringValue: public RCObject {
    char *data;
    StringValue(const char *initValue);
    StringValue(const StringValue& rhs);
    void init(const char *initValue);
    ~StringValue();
  };
  RCPtr<StringValue> value;
};
```

其中大部份都只是將先前發展出來的碼重述一遍而已，所以不應該有任何東西讓你感到驚訝。仔細檢驗，你會發現我在 `String::StringValue` 身上多加了一個 `init` 函式；稍後你會看到，它的功用和 `RCPtr` 的同名函式相同：只是將 constructors 內重複的碼集中起來而已。

此 `String` class 的公開介面和我們在本條款一開始所用的那個有重大的差異。copy constructor 哪裡去了？assignment operator 哪裡去了？destructor 哪裡去了？某些東西似乎遺漏掉了。

不，沒有什麼東西被遺漏。事實上一切運作非常完美。如果你不瞭解爲什麼如此，顯然你需要好好再讀一讀 C++。

是的，我們並不需要那些函式！當然啦，`String` 物件的複製行爲還是受到支援，而且，是的，複製行爲仍然能夠正確處理底部的 reference-counted `StringValue` 物件，但是 `String` class 不需要提供單獨一行碼來讓這事發生，因爲編譯器爲 `String` 所產生的 copy constructor，會自動呼叫 `String` 內的 `RCPtr` member 的 copy constructor，而後者會執行對 `StringValue` 物件的所有必要處理，包括其參用次數。`RCPtr` 是一個 smart pointer，記得嗎？我們設計它就是爲了處理 reference counting 的細節，所以那就是它的工作。它也處理「指派」和「解構」行爲，這便是爲什麼 `String` 不需撰寫那些函式的原因。我們原先的目標是要將無法重用的 reference-counting 碼移出 `String` class 之外，移入「與上下情境無關」的 classes 內，俾使後者能夠被任何 class 使用。現在我們已經做到了（以 `RCObject` 和 `RCPtr` classes 的型式），所以當它們開始運轉，不要反而驚訝起來。

下面是 `RCObject` 的實作碼：

```
RCObject::RCObject()
: refCount(0), shareable(true) {}

RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}

RCObject& RCObject::operator=(const RCObject&)
{ return *this; }

RCObject::~RCObject() {}
void RCObject::addReference() { ++refCount; }

void RCObject::removeReference()
{ if (--refCount == 0) delete this; }

void RCObject::markUnshareable()
{ shareable = false; }

bool RCObject::isShareable() const
{ return shareable; }

bool RCObject::isShared() const
{ return refCount > 1; }
```

下面是 `RCPtr` 的實作碼：

```
template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) return;
  if (pointee->isShareable() == false) {
    pointee = new T(*pointee);
  }
  pointee->addReference();
}

template<class T>
RCPtr<T>::RCPtr(T* realPtr)
: pointee(realPtr)
{ init(); }

template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs)
: pointee(rhs.pointee)
{ init(); }

template<class T>
RCPtr<T>::~RCPtr()
{ if (pointee)pointee->removeReference(); }

template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {
    if (pointee) pointee->removeReference();
    pointee = rhs.pointee;
    init();
  }
  return *this;
}

template<class T>
T* RCPtr<T>::operator->() const { return pointee; }

template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

下面是 String::StringValue 的實作碼：

```
void String::StringValue::init(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}

String::StringValue::StringValue(const char *initValue)
{ init(initValue); }
```

```
String::StringValue::StringValue(const StringValue& rhs)
{ init(rhs.data); }

String::StringValue::~StringValue()
{ delete [] data; }
```

最後，所有的道路導往 String，以下是其實作碼：

```
String::String(const char *initValue)
: value(new StringValue(initValue)) {}

const char& String::operator[](int index) const
{ return value->data[index]; }

char& String::operator[](int index)
{
  if (value->isShared()) {
    value = new StringValue(value->data);
  }
  value->markUnshareable();
  return value->data[index];
}
```

如果你把現在的 String class 拿來和先前以 dumb pointers 發展的 String class 做比較，你會發現，第一，新版本精簡許多，因爲 RCPtr 做掉了許多原本落在 String 身上的 reference-counting 雜務。第二，目前還保留於 String 內的碼，與原先版本幾乎沒變，換句話說 smart pointer 幾乎毫無間隙地取代了 dumb pointer。事實上，唯一的改變只在 operator[]，其內呼叫 isShared 而不再直接檢驗 refCount，此外我們使用 smart object RCPtr，又消除了在 copy-on-write 時機下自行動手處理參用次數的需要。

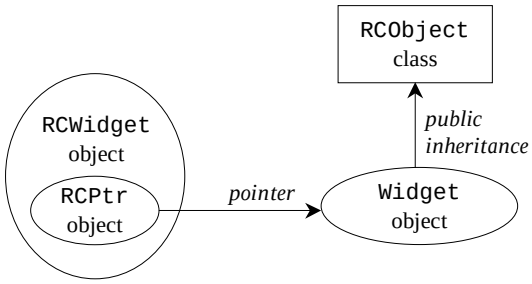
這一切都非常好。誰能寫出更少的碼來？誰能拒絕封裝帶來的美好生活？不過，底線決定於新的 String class 對其 clients 帶來的衝擊，而非 String 的任何實作細節 — 雖然那讓人眼睛爲之一亮。如果說沒有消息就是好消息的話，這裡的消息實在是非常好：String 的介面沒有任何改變。我們加上了 reference counting，我們加上了「讓個別的字串實值不再可共享」的能力，我們將參用計數的概念搬移到一個新的 base class 內，我們加上了 smart pointers（以便將參用次數的處理自動化），但是沒有一行 client 碼需要改變。當然，我們改變了 String class 的定義，所以如果 clients 希望取得 reference-counted 字串的好處，必須重新編譯和聯結，但是他們的投資（原始碼）完全獲得保障。你看，封裝是件多麼美妙的事情呀。

將 Reference Counting 加到既有的 Classes 身上

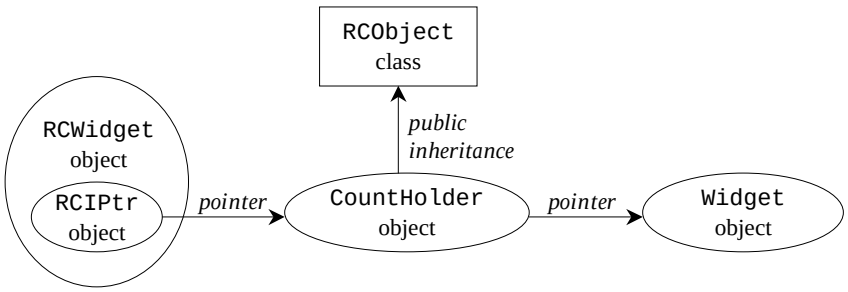
截至目前所討論的每一件事情，都必須動用到我們感興趣的那個 class 的原始碼。但如果我們想要將 reference counting 施行於程式庫中一個名爲 `Widget` 的 class 呢？程式庫的內容不是我們可以更改的，所以沒辦法讓 `Widget` 繼承自 `RCObject`，也就無法對 `Widget` 使用 smart `RCPtrs`。難道幸運之神已離我們而去？

不，幸運之神還站在我們這一邊。只要稍微修改設計，我們就可以爲任何型別加上 reference counting 能力。

首先，讓我們考慮，如果令 `Widget` 繼承自 `RCObject`，我們的設計看起來將是如何。這種情況下，我們必須增加一個 `RCWidget` class 給 clients 使用，但每件事情都極類似先前的 `String`/`StringValue` 例子：`RCWidget` 扮演 `String` 的角色，`Widget` 扮演 `StringValue` 的角色。整個設計看起來如下：



現在我們可以將「計算機科學領域中大部份問題得以解決」的原理施展出來。我們可以加上一層間接性。是的，我們增加一個新的 `CountHolder` class，用以持有參用次數，並令 `CountHolder` 繼承自 `RCObject`。我們也令 `CountHolder` 內含一個指標，指向一個 `Widget`。然後將 smart `RCPtr` template 以同樣靈巧的 `RCIPtr` template 取代，後者知道 `CountHolder` class 的存在。`RCIPtr` 的 `"I"` 意指 "indirect"（間接）。修改後的設計如下：



就像「StringValue 只是實作上的細目，不需給 String 的使用者知道」一樣，CountHolder 也是實作上的細目，不需給 RCWidget 的使用者知道」。事實上它是 RCIPtr 的實作細目，所以我把它巢狀放進 RCIPtr class 內部。RCIPtr 實作如下：

```
template<class T>
class RCIPtr {
public:
  RCIPtr(T* realPtr = 0);
  RCIPtr(const RCIPtr& rhs);
  ~RCIPtr();

  RCIPtr& operator=(const RCIPtr& rhs);

  const T* operator->() const;      // 稍後會解釋爲什麼
  T* operator->();                  // 這些函式以此方式宣告。
  const T& operator*() const;
  T& operator*();
private:
  struct CountHolder: public RCObject {
    ~CountHolder() { delete pointee; }
    T *pointee;
  };
  CountHolder *counter;
  void init();
  void makeCopy();                  // 見稍後說明
};

template<class T>
void RCIPtr<T>::init()
{
  if (counter->isShareable() == false) {
    T *oldValue = counter->pointee;
    counter = new CountHolder;
    counter->pointee = new T(*oldValue);
  }
  counter->addReference();
}

template<class T>
RCIPtr<T>::RCIPtr(T* realPtr)
: counter(new CountHolder)
{
  counter->pointee = realPtr;
  init();
}

template<class T>
RCIPtr<T>::RCIPtr(const RCIPtr& rhs)
: counter(rhs.counter)
{ init(); }
```

譯註：部份程式碼移至 **p200**，請見該頁說明。

```
template<class T>
RCIPtr<T>::~RCIPtr()
{ counter->removeReference(); }

template<class T>
RCIPtr<T>& RCIPtr<T>::operator=(const RCIPtr& rhs)
{
  if (counter != rhs.counter) {
    counter->removeReference();
    counter = rhs.counter;
    init();
  }
  return *this;
}

template<class T>                              // const access;
const T* RCIPtr<T>::operator->() const         // 不需要 COW。
{ return counter->pointee; }

template<class T>                              // const access;
const T& RCIPtr<T>::operator*() const          // 不需要 COW。
{ return *(counter->pointee); }
```

RCIPtr 和 RCPtr 之間有兩個差異。第一，「RCPtr 物件」直接指向實值，而「RCIPtr 物件」透過中介層「CountHolder 物件」指向實值。第二，RCIPtr 將 operator-> 和 operator* 多載化了，如此一來只要有 non-const access 發生於被指物身上，copy-on-write（「塗寫時才複製」）就會自動執行。

有了 RCIPtr，RCWidget 的實作就很容易，因爲 RCWidget 的每一個函式都只是透過底層的 RCIPtr 轉呼叫對應的 Widget 函式。如果 Widget 看起來像這樣：

```
class Widget {
public:
  Widget(int size);
  Widget(const Widget& rhs);
  ~Widget();
  Widget& operator=(const Widget& rhs);
  void doThis();
  int showThat() const;
};
```

那麼 RCWidget 應該定義如下：

```
class RCWidget {
public:
  RCWidget(int size): value(new Widget(size)) {}
  void doThis() { value->doThis(); }
  int showThat() const { return value->showThat(); }
private:
  RCIPtr<Widget> value;
};
```

注意 `RCWidget` constructor 如何以其所獲得的參數做爲引數，透過 `new` operator（見條款 8）呼叫 `Widget` constructor。注意 `RCWidget` 的 `doThis` 函式如何呼叫 `Widget` 的 `doThis` 函式，`RCWidget::showThat` 又是如何傳回其對應之 `Widget` 兄弟所傳回的東西。也請注意，`RCWidget` 沒有宣告 copy constructor，也沒有 assignment operator 和 destructor。就像先前的 `String` class 一樣，不再需要撰寫這些函式了。感謝 `RCIPtr` class，預設版本做了正確的事情。

如果你認爲，`RCWidget` 是如此制式，應該可以自動化生產之，你是對的。寫個程式，以 `Widget` 之類的 class 做爲輸入，產出 `RCWidget` 之類的 class 做爲輸出，似乎並不困難。如果你寫出這樣的程式，請讓我知道。

評估

讓我們從 widgets, strings（字串）, values（實值）, smart pointers 以及 reference-counting base classes 的種種細瑣中解脫出來。這樣我們才得以站定，在更寬闊的環境中回頭思索 reference counting。在那更寬闊的環境中，我們必須回答一個層次更高的問題：什麼時候適合使用 reference counting 技術？

Reference-counting 的實作並非不需成本。每一個擁有參用計數能力的實值都攜帶有一個參用計數器，而大部份操作都需要能夠以某種方式查驗或處理這個參用計數器。物件的實值因而需要更多記憶體；面對它們，有時候我們得執行更多程式碼。此外，reference-counted class 的底層原始碼比以前複雜得多。一個 un-reference-counted 字串類別通常可以很獨立，但本條款最後的那個 `String` class 必須在其他三個輔助類別（`StringValue`, `RCObject`, `RCPtr`）都存在的情況下才有用。是的，我們的複雜設計在「實值可被共享」時提供了更佳效率的承諾，它消除了「追蹤物件擁有權」的必要性，它也助長了 reference counting 這個構想及其實作品的可重用性。儘管如此，卻得寫出那四重奏般的 classes，並加以測試、說明、維護。比起單一類別的撰寫、測試、說明、維護，工作繁重多了 — 就算是呆伯特經理也看得出來。

Reference counting 是個最佳化技術，其適用前提是：物件**常常**共享實值（見條款 18）。如果這個假設失敗，reference counting 反而會賠上更多記憶體，執行更多程式碼。從另一個角度看，如果你的物件確實有「共同實值」的傾向，reference counting 應可同

時節省你的時間和空間。你的物件實值愈大，並且有愈多物件可同時共享實值，你所節省的記憶體就愈多。你在物件之間做的複製動作和指派動作愈頻繁，你所節省的時間就愈多。產生和摧毀一個實值的成本愈高，你所節省的時間也愈多。簡單地說，以下是使用 reference counting 改善效率的最適當時機：

- 相對之數目的物件共享相對少量的實值。如此的共享行爲通常是透過 assignment operators 和 copy constructors。物件/實值 數量比愈高，reference counting 帶來的利益愈大。
- 物件實值的產生或摧毀成本很高，或是它們使用了記憶體。不過即使這種情況，reference counting 還是不能爲你帶來任何利益，除非實值可被多個物件共享。

只有一個確切方法可以告訴你，你的程式是否滿足這些情況。這個方法不是漫天瞎猜，也不是仰賴直覺（見條款 16）。「確定程式是否受益於 reference counting」的可靠方法就是嚴謹地分析（profile）你的程式。於是你可以知道，產生和摧毀物件實值，是否是效率上的瓶頸；你也可以量測「物件/實值」比率。有了這些資料在手，你才能夠決定 reference counting 帶來的利益是否多於損失。

即使上述各狀態都符合，reference counting 的運用仍有可能不恰當。某些資料結構（例如 directed graphs）會導至自我參考（self-referential）或循環相依（circular dependency）結構。此等資料結構有一個傾向，它會滋生（*spawn*）出孤立的物件群，此等物件雖不爲任何人所用，其參用次數卻從不降爲 0。那是因爲「未使用」之結構內的每一個物件都被同結構內的至少另一個物件指向。工業水準的垃圾收集器（garbage collectors）以特殊技術來找出此類結構並消除它們，但是我們此處所驗證的簡易型 reference-counting 作法，並不容易含入此等特殊技術。

即使效率不是你的第一考量，reference counting 也可能吸引你。如果你發現你自己由於「不確定誰被允許刪除什麼東西」而茶不飲飯不思，reference counting 可用來減輕你的負擔。許多程式員光只是爲了這個因素就投身於 reference counting。

讓我們以先前懸而未決的一項技術來結束本條款。當 `RCObject::removeReference` 將物件的參用次數減 1，它會檢查新的計數是否爲 0。如果的確是 0，`removeReference`

會以 deleteing this 的方式摧毀這個物件。只有當物件以 new 配置而得，這才是一個安全的行爲。所以我們需要某種方法確保 RCObjects 只以 new 配置出來。

這一次我們以公約規範來達成目標。RCObject 的設計目的是用來做爲有參用計數能力之「實值物件」的基礎類別，而那些「實值物件」應該只被 RCPtr smart pointers 取用。此外，應該只有確知「實值物件」共享性的所謂「應用物件」才能將「實值物件」實體化。描述「實值物件」的那些 classes 不應該被外界看到。在我們的例子中，描述「實值物件」者爲 StringValue，我們令它成爲「應用物件」String 內的私有成員，以限制其用途。只有 String 才能夠產生 StringValue 物件，所以，確保所有 StringValue 物件皆以 new 配置而得，是 String class 作者的責任。

我們這種「限制 RCObjects 只能誕生於 heap」的作法，其實就是把責任交給一組有良好定義的 classes，並確保只有這組 classes 可以產出 RCObjects。其他任何 clients 都不可能意外（或惡意）地以不適當的方式產出 RCObjects。我們限制了 reference-counted 物件的產出權，而在此同時，我們也很清楚地給了一個附帶條件：物件誕生規則必須獲得遵守。

條款 30：Proxy classes（替身類別、代理人類別）

雖然你的岳母只有一個（就說是一維吧），這個世界通常卻非如此美好☺。不幸的是 C++ 尚未掌握這個事實。至少 C++ 語言在陣列的支援方面提不出什麼有力的證據證明它掌握了這一事實。你可以在 FORTRAN、BASIC、甚至在 COBOL 中產生二維陣列、三維陣列，乃至於 n 維陣列（好吧，我承認 FORTRAN 只允許最多七維陣列，但這不是重點），你能夠在 C++ 中這麼做嗎？唔，某種情況下才可以。

下面這樣做是合法的：

```
int data[10][20];                          // 二維陣列，大小 10 x 20
```

但如果以變數做爲陣列大小，就不可以：

```
void processInput(int dim1, int dim2)
{
  int data[dim1][dim2];                    // 錯誤！陣列的尺度（大小）
  ...                                      // 必須在編譯時期已知
}
```

C++ 甚至不允許一個與二維陣列相關的 heap-based 配置行爲：

```
int *data =
  new int[dim1][dim2];                          // 錯誤！
```

實作二 維陣列

多維陣列在任何語言中有用，在 C++ 中也一樣有用，所以想出某種辦法，相當程度地支援它們，是件重要的事情。C++ 中最廣泛也最標準的作法就是：產生一個 class，用以表現我們有需要而卻被語言遺漏的物件。我們可以爲二維陣列定義一個 class template：

```
template<class T>
class Array2D {
public:
  Array2D(int dim1, int dim2);
  ...
};
```

於是就可以這樣定義我們所要的陣列了：

```
Array2D<int> data(10, 20);           // 沒問題

Array2D<float> *data =
  new Array2D<float>(10, 20);        // 沒問題

void processInput(int dim1, int dim2)
{
  Array2D<int> data(dim1, dim2);      // 沒問題
  ...
}
```

然而，這些陣列物件的使用並不十分直接了當。爲了保持 C 和 C++ 共同的語法傳統，我們希望能夠以中括號表現陣列索引：

```
cout << data[3][6];
```

如何在 `Array2D` 中宣告所謂的索引運算子（indexing operator），俾能夠完成上述心願呢？

我們的第一個衝動可能是宣告 `operator[][]` 函式，像這樣：

```
template<class T>
class Array2D {
public:
  // 注意，以下宣告無法通過編譯
  T& operator[][](int index1, int index2);
  const T& operator[][](int index1, int index2) const;
  ...
};
```

然而我們很快學會抑制如此衝動，因爲根本沒有 `operator[][]` 這樣的東西，別以爲你的編譯器會疏忽這一點。（條款 7 有一份列表，列出所有運算子，以及其中可被多載化者）

如果願意忍受奇特的語法，你或許可以循著許多程式語言的作法，以小括號表現陣列索引。爲使用小括號，你必須將 `operator()` 多載化：

```
template<class T>
class Array2D {
public:
  // 以下宣告可通過編譯
  T& operator()(int index1, int index2);
  const T& operator()(int index1, int index2) const;
  ...
};
```

於是我們得以這樣的方式使用陣列：

```
cout << data(3, 6);
```

這很容易實作，也很容易推廣至任意維度。缺點是你的 `Array2D` 物件看起來一點也不像個內建陣列。上面那個對元素 (3, 6) 的存取動作，看起來倒像是個函式呼叫。

如果你拒絕讓你的陣列看起來像 FORTRAN 難民，你可能再回到「以中括號做爲索引運算子」的概念上。雖然沒有 `operator[][]` 這樣的東西，但你知道，這樣寫卻是合法的：

```
int data[10][20];
...
cout << data[3][6];         // 沒問題
```

這帶給你什麼啓示？

是這樣的，變數 data 並非真正是個二維陣列，它其實是 10 個元素所組成的一維陣列。每個元素本身又是 20 個元素所組成的一維陣列，所以算式 data[3][6] 真正的意思是 (data[3])[6]，亦即「data 的第四個元素所表現之陣列」中的第七個元素。簡單地說，第一個中括號導出另一個陣列，第二個中括號才取得次陣列中的某個元素。

我們可以在 Array2D class 中玩相同的把戲：將 operator[] 多載化，令它傳回一個 Array1D 物件。然後我們再對 Array1D 多載化其 operator[]，令它傳回原先二維陣列中的一個元素：

```
template<class T>
class Array2D {
public:
  class Array1D {
  public:
    T& operator[](int index);
    const T& operator[](int index) const;
    ...
  };

  Array1D operator[](int index);
  const Array1D operator[](int index) const;
  ...
};
```

於是下面動作就合法了：

```
Array2D<float> data(10, 20);
...
cout << data[3][6];         // 沒問題
```

在這裡，data[3] 獲得一個 Array1D 物件，而對該物件再施行 operator[]，獲得原二維陣列中 (3, 6) 位置的浮點數。

Array2D class 的使用者不需要知道 Array1D class 的存在。Array1D 所象徵的一維陣列物件，觀念上對 Array2D 的使用者而言並不存在。使用者就好像是在使用真正的、活生生的、如假包換的二維陣列。

每個 `Array1D` 物件象徵一個一維陣列，觀念上它並不存在於 `Array2D` 的使用者心中。凡「用來代表（象徵）其他物件」的物件，常被稱爲 proxy objects（替身物件），而用以表現 proxy objects 者，我們稱爲 proxy classes。本例的 `Array1D` 便是個 proxy class，其實體代表一個觀念上並不存在的一維陣列。（proxy objects 和 proxy classes 等術語並非全球通用；proxy objects 有時候被稱爲 surrogates，「代用品」之意）

區分 operator[] 的讀寫動作

利用 proxies 的觀念來製作 classes，使其物件像是多維陣列，這種用途很常見，但 proxy classes 遠比此更具彈性。舉個例子，條款 5 顯示 proxy classes 如何能夠用來阻止「單引數 constructors」被用來執行不受歡迎的型別轉換。在 proxy classes 的各種用途之中，最先驅的當屬協助區分 `operator[]` 的讀寫動作了。

考慮一個 reference-counted（參用計數）字串型別，它支援 `operator[]`。此等型別曾在條款 29 有過詳細討論。如果你錯過 reference counting 的基礎觀念，現在是把它補起來的好時機。

一個支援 `operator[]` 的字串型別，允許使用者寫出以下程式碼：

```
String s1, s2;       // String 類似標準程式庫所提供的 string；
                     // 由於此處運用了 proxies，所以這個 class
                     // 並不相容於標準的 string 介面
...
cout << s1[5];            // 讀取 s1
s2[5] = 'x';              // 塗寫 s2
s1[3] = s2[8];            // 塗寫 s1, 讀取 s2
```

注意 `operator[]` 可以在兩種不同情境下被呼叫：用來讀取一個字元，或是用來塗寫一個字元。讀取動作是所謂的右值運用（rvalue usages）；塗寫動作是所謂的左值運用（lvalue usages）。這些術語來自編譯器開發團隊，lvalue 意指指派動作（assignment）的左手邊，rvalue 意指右手邊。一般而言，以一個物件做爲 lvalue，意思是它可以被修改，而以之做爲 rvalue 的意思是它不能被修改。

我們希望區分 `operator[]` 的左值運用和右值運用，因爲（特別是針對 reference-counted 資料結構）「讀取」作法可能比「塗寫」作法簡單快速得多。一如條款 29

所解釋，「塗寫」一個 reference-counted objects，可能需得複製一份完整的資料結構，而「讀取」則只是簡單傳回一個值。很不幸的是，operator[] 內沒有任何辦法可以決定它被呼叫當時的情境：是的，沒有人能夠在 operator[] 內區分它是左值運用或右值運用。

『但是等等』，你說，『不需要區分。我們可以利用常數性來對 operator[] 多載化，那使我們得以區分讀或寫』。換句話說，你建議我這樣解決問題：

```
class String {
public:
  const char& operator[](int index) const;        // 針對讀取
  char& operator[](int index);                     // 針對塗寫
  ...
};
```

哎呀呀，這是沒有用的。編譯器在 const 和 non-const member functions 之間的選擇，只以「喚起該函式的物件是否是 const」爲基準，並不考慮它們在什麼情境下被喚起。因此：

```
String s1, s2;
...
cout << s1[5];       // 呼叫 non-const operator[]，因爲 s1 不是 const

s2[5] = 'x';         // 也呼叫 non-const operator[]，因爲 s2 不是 const

s1[3] = s2[8];       // 左右都呼叫 non-const operator[]，因爲 s1 和 s2
                     // 都是 non-const 物件。
```

換句話說，將 operator[] 多載化，並不能因此區分其被讀或被寫狀態。

我們曾經在條款 29 中重新設計程式，解決這個無法滿足的狀態。我們保守地假設，對 operator[] 的所有呼叫動作都是爲了「塗寫」目的。這次我們不再那麼輕易棄守了。雖然仍然無法在 operator[] 內區分左值運用和右值運用，但問題仍然需要解決。我們必須找出新的方法。如果你讓自己被種種限制束縛住，生活將是多麼無趣。

我的想法基於一個事實：雖然或許不可能知道 operator[] 是在左值或右值情境下被喚起，我們還是可以區分讀和寫 — 只要將我們所要的處理動作延緩，直至知道

operator[] 的傳回結果將如何被使用爲止。我們需要知道的，就是如何延緩我們的決定（決定物件究竟是被讀或被寫），直到 operator[] 回返。這是條款 17 的緩式評估（lazy evaluation）例子之一。

Proxy class 允許我們購買我們所需要的時間，因爲我們可以修改 operator[]，令它傳回字串字元的一個 proxy，而不傳回字元本身。然後我們可以等待，看看這個 proxy 如何被運用。如果它被讀，我們可以（有點過時地）將 operator[] 的呼叫動作視爲一個讀取動作。如果它被寫，我們必須將 operator[] 的呼叫視爲一個塗寫動作。

稍後你會看到程式碼。首先，重要的是瞭解我們即將使用的 proxies。對於一個 proxy，你只有三件事情可做：

- 產生它，本例也就是指定它代表哪一個字串字元。
- 以它做爲指派動作（assignment）的標的（接受端），這種情況下你是對它所代表的字串字元做指派（設值）動作。如果這麼使用，proxy 代表的將是「喚起 operator[] 函式」的那個字串的左值運用。
- 以其他方式使用之。如果這麼使用，proxy 表現的是「喚起 operator[] 函式」的那個字串的右值運用。

下面是一個 reference-counted String class，其中利用 proxy class 來區分 operator[] 的左值運用和右值運用：

```
class String {                      // reference-counted strings;
public:                             // 條款 29 有其細節。

  class CharProxy {                 // proxies for string chars
  public:
    CharProxy(String& str, int index);            // 建構

    CharProxy& operator=(const CharProxy& rhs);   // 左值運用
    CharProxy& operator=(char c);

    operator char() const;                        // 右值運用

  private:
    String& theString;              // 這個 proxy 所附屬（相應）的字串。
    int charIndex;                  // 這個 proxy 所代表的字串字元。
  };

  const CharProxy
    operator[](int index) const;        // 針對 const Strings
```

```
  CharProxy operator[](int index);      // 針對 non-const Strings
  ...
friend class CharProxy;
private:
  RCPtr<StringValue> value;
};
```

除了增加 CharProxy class（我將於稍後討論）以外，這個 String class 和條款 29 最後的那個 String class 之間唯一的差別就是，此處的兩個 operator[] 都傳回 CharProxy 物件。然而 String 的使用者可以忽略此點，猶如 operator[] 傳回字元（或字元的 references — 見條款 1）一樣：

```
String s1, s2;           // reference-counted 字串（採用 proxies）
...
cout << s1[5];           // 仍然合法，仍然有效運作
s2[5] = 'x';             // 也合法，也有效運作
s1[3] = s2[8];           // 當然合法，當然有效運作
```

有趣的並不是它能夠有效運作，有趣的是它如何運作。

考慮第一個句子：

```
cout << s1[5];
```

算式 s1[5] 產生一個 CharProxy 物件。此物件不曾定義 output 運算子，所以你的編譯器趕緊尋找一個它能夠實施的隱式型別轉換，使 operator<< 呼叫動作能夠成功（見條款 5）。編譯器真的找到了一個：將 CharProxy 隱式轉型爲 char。此轉換函式宣告於 CharProxy class 內。於是編譯器自動喚起這個轉換運算子，於是 CharProxy 所表現的字串字元被列印出去。這是典型的 CharProxy-to-char 轉換，發生在所有「被用來做爲右值」的 CharProxy 物件身上。

左值運用的處理方式又不相同。看看這一行：

```
s2[5] = 'x';
```

和上例一樣，算式 s2[5] 導出一個 CharProxy 物件，但這次這個物件是 assignment 動作的標的物。會喚起哪個 assignment 運算子呢？由於標的物是個 CharProxy，所以被呼叫的 assignment 運算子會是 CharProxy class 所定義的那個。這很重要，因爲在 CharProxy 的 assignment 運算子內，我們確知「被指派的 CharProxy 物件係

被用來做爲一個左值」。我們因此知道 proxy 所代表的那個字串字元將被用來做爲一個左值，並因此必須採取所有必要行動，對該字元實施左值處理。

同樣道理，以下句子：

```
s1[3] = s2[8];
```

呼叫的是兩個 CharProxy 物件的 assignment 運算子，在該運算子中我們知道左端物件被用來做爲一個左值，右端物件被用來做爲一個右值。

『是，是，是』你喃喃抱怨說『秀給我看呀』。沒問題。下面是 String 的 operator[]：

```
const String::CharProxy String::operator[](int index) const
{
  return CharProxy(const_cast<String&>(*this), index);
}
String::CharProxy String::operator[](int index)
{
  return CharProxy(*this, index);
}
```

每個函式都只是產生並傳回「被請求之字元」的一個 proxy。沒有任何動作施加於此字元身上：我們延緩此等行爲，直到知道該行爲是「讀取」或是「塗寫」。

注意，const operator[] 傳回的是個 const proxy。由於 CharProxy::operator= 不是一個 const member function，如此的（const）proxies 不能被用來做爲指派（設值）動作的標的物。因此不論是 const operator[] 所傳回的 proxy，或是該 proxy 所代表的字元，都不能被用來做爲左值。太好了，這正是我們希望 const operator[] 所具備的行爲。

也請注意，當 const operator[] 產生一個 CharProxy 物件準備傳回時，作用於 *this 身上的 const_cast（見條款 2）。爲了滿足 CharProxy constructor 的限制，那是必要的，因爲 CharProxy constructor 只接受 non-const String。轉型（*cast*）常常讓人憂心，不過此例之中被 operator[] 傳回的 CharProxy 物件本身是 const，所以倒是不必擔心 String 內含的字元（被 proxy 指向者）會被修改。是的，沒有任何風險。

operator[]傳回的每一個 proxy 都會記住它所附屬的字串，以及它所在的索引位置：

```
String::CharProxy::CharProxy(String& str, int index)
: theString(str), charIndex(index) {}
```

將 proxy 轉換爲右值，是非常直接了當的事 — 我們只需傳回該 proxy 所表現的字元副本即可：

```
String::CharProxy::operator char() const
{
  return theString.value->data[charIndex];
}
```

如果你已經忘記 String 物件、value 成員，以及 value 所指之 data 成員彼此間的關係，請回到條款 29 重新建立你的記憶。由於這個函式以 by value 方式傳回一個字元，並且由於 C++ 限制只能在右值情境下使用這樣的 by value 傳回值，所以這個轉換函式只能夠用於右值合法之處。

接下來我們實作 CharProxy 的 assignment 運算子。其間必須面對一個事實：proxy 所代表的字元即將被做爲指派（設值）動作的標的，也就是成爲一個左值。我們可以實作出 CharProxy 的傳統 assignment 運算子如下：

```
String::CharProxy&
String::CharProxy::operator=(const CharProxy& rhs)
{
  // 如果本字串與其他 String 物件共享同一個實值，
  // 將實值複製一份，供本字串單獨使用。
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }

  // 現在進行指派（設值）動作：將 rhs 所代表的字元值
  // 指派給 *this 所代表的字元。
  theString.value->data[charIndex] =
    rhs.theString.value->data[rhs.charIndex];

  return *this;
}
```

把這份碼拿來和條款 29（p.207）的 non-const String::operator[] 比較，你會發現它們非常類似。此乃預料之中。我們在條款 29 悲觀地假設所有 non-const operator[] 喚起動作都是爲了塗寫，所以我們以近似上述的方式來完成它們。此處我們把完成「塗寫動作」的碼移到 CharProxy 的 assignment 運算子中，避免 non-const operator[] 在右值情境下竟爾付出塗寫的昂貴代價。順帶一提，請注意，此函式要求處理 String 的 private data member value。這就是爲什麼稍早出現的

String 定義式中將 CharProxy 宣告爲 friend 的原因。

第二個 CharProxy assignment 運算子與上述傳統版本幾乎雷同：

```
String::CharProxy& String::CharProxy::operator=(char c)
{
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }

  theString.value->data[charIndex] = c;

  return *this;
}
```

身爲一位合格的軟體工程師，你當然會將這兩個 assignment 運算子內重複的碼抽出來放進一個 private CharProxy member function，然後讓兩個運算子都去呼叫它，是吧！

限制

Proxy class 很適合用來區分 operator[] 的左值運用和右值運用，但是這項技術並非沒有缺點。我們希望 proxy object 能夠無間隙地取代它們所代表的物件，但是這樣的想法卻很難達成。因爲除了指派（設值，assignment）之外，物件亦有可能在其他情境下被當做左值使用，而那種情況下的 proxies 常常會有與真實物件不同的行爲。

再次考慮條款 29 的程式片段（譯註：p.191 下），正是因爲它才引發我們決定爲每個 StringValue 物件加上一個「可共享」旗標。如果 String::operator[] 傳回的是個 CharProxy 而非 char&，那段碼將不再能夠通過編譯：

```
String s1 = "Hello";
char *p = &s1[1];          // 錯誤！
```

算式 s1[1] 傳回一個 CharProxy，所以 "=" 右手邊的算式型別是 CharProxy*。由於不存在任何函式可以將 CharProxy* 轉換爲 char*，所以上述對 p 的初始化動作無法通過編譯。一般而言，「對 proxy 取址」所獲得的指標型別和「對真實物件取址」所獲得的指標型別不同。

爲了消弭這個難點，我們需要在 CharProxy class 內將取址（address-of）運算子加以多載化：

```
class String {
public:
  class CharProxy {
  public:
    ...
    char * operator&();
    const char * operator&() const;
    ...
  };
  ...
};
```

這些函式很容易實作出來。其中的 const 版只是傳回一個指標，指向「proxy 所表現之字元」的 const 版本：

```
const char * String::CharProxy::operator&() const
{
  return &(theString.value->data[charIndex]);  //譯註：ref p.219, p.186
}
```

non-const 版做的事就稍微多些，因爲它必須傳回一個指標，指向一個可被修改的字元。這很類似條款 29 的 String::operator[] non-const 版行爲（譯註：p.207），所以其實作碼也十分類似：

```
char * String::CharProxy::operator&()
{
  // 確定「標的字元」（本函式將傳回一個指標指向它）所屬的字串實值不爲
  // 任何其他 String 物件共享（譯註：如果共享，就做一份專屬副本出來）
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }

  // 我們不知道 clients 會將本函式傳回的指標保留多久，所以「標的字元」
  // 所屬的字串實值（一個 StringValue 物件）絕不可以被共享。
  theString.value->markUnshareable();

  return &(theString.value->data[charIndex]);
}
```

這段碼和其他 CharProxy member functions 有許多共同點，我知道你一定會將它們抽出來成爲一個 private member function，讓所有用到它的人呼叫之，是吧！

如果我們有一個 template，用來實現 reference-counted 陣列，其中利用 proxy classes 來區分 operator[] 被喚起時的左值運用和右值運用，那麼 chars 和其替身 CharProxys 的第二個不同點更是顯而易見：

```
template<class T>                          // reference-counted 陣列.
class Array {                              // 運用 proxies.
public:
  class Proxy {
  public:
    Proxy(Array<T>& array, int index);
    Proxy& operator=(const T& rhs);
    operator T() const;
    ...
  };
  const Proxy operator[](int index) const;
  Proxy operator[](int index);
  ...
};
```

考慮這些陣列的使用方式：

```
Array<int> intArray;
...
intArray[5] = 22;                  // 沒問題
intArray[5] += 5;                  // 錯誤！
intArray[5]++;                     // 錯誤！
```

一如預期，以 operator[] 做爲指派動作之標的物，可以成功，但是將 operator[] 用於 operator+= 或 operator++ 呼叫式的左手邊，會失敗。那是因爲 operator[] 傳回一個 proxy，而該 proxy objects 並沒有供應 operator+= 或 operator++。類似情況發生在其他需要左值的運算子身上，包括 operator*=, operator<<=, operator--等等。如果你希望這些運算子都能夠和「傳回 proxies」的 operator[] 合作，你必須爲 Array<T>::Proxy class 一一定義這些函式。這個工作量可不小，你或許不打算自己動手。不幸的是，要不你就乖乖動手，要不你就失去它們，沒有其他選擇。

另一個相關問題是「透過 proxies 喚起真實物件的 member functions」。如果你直率地那麼做，會失敗。例如，假設我們希望實作出一個 reference-counted 陣列，每個元素都是分數（rational numbers）。我們可以定義一個 Rational class，然後使用剛才見過的那個 Array template：

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const;
  int denominator() const;
  ...
};
Array<Rational> array;
```

下面便是我們希望的陣列運用方法，但是結果令人失望：

```
cout << array[4].numerator();                    // 錯誤！
int denom = array[22].denominator();             // 錯誤！
```

這個困難其實在意料之內；operator[] 傳回的是 Rational 物件的替身（proxy），而不是一個真正的 Rational 物件。但是 numerator 和 denominator 這兩個 member functions 只對 Rationals（而非其 proxies）存在。因此編譯器必然不讓上述的碼通過。爲了讓 proxies 模仿其所代表之物件的行爲，你必須將適用於真實物件的每一個函式加以多載化，使它們也適用於 proxies。

Proxies 無法取代真實物件的另一種情況是，使用者將它們傳遞給「接受 references to non-const objects」的函式：

```
void swap(char& a, char& b);    // 將 a 和 b 的值對調
String s = "+C+";    // 喔歐，我想把它改爲 "C++"
swap(s[0], s[1]);    // 按說這應能夠把 "+C+" 改爲 "C++"，但它卻無法編譯
```

String::operator[] 傳回一個 CharProxy，但是 swap 要求其引數必須是 char& 型別。CharProxy 可能會被隱喻轉換爲一個 char，但是沒有任何函式可以將它轉換爲一個 char&。此外，就算能夠轉換，轉換所得的那個 char 也無法繫結於 swap 的 char& 參數身上，因爲那個 char 是個暫時物件（operator char 的傳回值），而條款 19 有一些好理由使編譯器拒絕將暫時物件繫結至 non-const reference 參數身上。

proxies 難以完全取代真正物件的最後一個原因在於隱式型別轉換。當 proxy object 被隱喻轉換爲它所代表的真正物件，會有一個使用者自定的轉換函式被喚起。例如只要呼叫 operator char，便可將一個 CharProxy 轉換爲它所代表的 char。然而正如條款 5 所言，編譯器在「將呼叫端引數轉換爲對應的被呼叫端（函式）參數」過程中，運用「使用者自定轉換函式」的次數只限一次。於是就有可能發生這樣的情況：以真實物件傳給函式，成功；以 proxies 傳給函式，失敗。舉個實例，假設我們有一個 TVStation class 以及一個函式 watchTV：

```
class TVStation {
public:
  TVStation(int channel);
  ...
};
void watchTV(const TVStation& station, float hoursToWatch);
```

由於 int 至 TVStation 之間有隱式型別轉換（見條款 5），所以我們可以這麼做：

```
watchTV(10, 2.5);                    // 觀看頻道 10，共 2.5 小時。
```

但是如果使用前述用於 reference-counted 陣列的 template，並以 proxy classes 來區分 operator[] 的左值運用和右值運用，就不能這麼做了：

```
Array<int> intArray;
intArray[4] = 10;
watchTV(intArray[4], 2.5);     // 錯誤！沒有任何轉換動作可以將
                               // Proxy<int> 轉換爲 TVStation
```

如果你瞭解隱式型別轉換伴隨而來的問題，對上述情況就比較不會激動得說不出話來。事實上比較好的設計是，將 TVStation class 的 constructor 宣告爲 explicit，這麼一來即使先前第一次呼叫 watchTV，也無法通過編譯。隱式型別轉換的細節以及 explicit 對它產生的影響，請見條款 5。

評估

Proxy classes 允許我們完成某些十分困難或幾乎不可能完成的行爲。多維陣列是其中之一，左值/右值的區分是其中之二，壓抑隱式轉換（見條款 5）是其中之三。

然而，proxy classes 也有缺點。如果扮演函式傳回值的角色，那些 proxy objects 將是一種暫時物件（見條款 19），需要被產生和被摧毀。而建構和解構並非毫無成本，雖然這份成本比起 proxies 所帶來的好處（例如可以區分塗寫動作和讀取動作）可能微不足道。此外，proxy classes 的存在也增加了軟體系統的複雜度，因爲額外的 classes 使產品更難設計、實作、瞭解、維護。

最後，當 class 的身份從「與真實物件合作」移轉到「與替身物件（proxies）合作」，往往會造成 class 語意的改變，因爲 proxy objects 所展現的行爲常常和真正物件的行爲有些隱微差異。有時候這會造成 proxies 在系統設計上的一個弱勢，不過通常很少需要用到「proxies 和真實物件有異」的那些操作行爲，例如，很少有人會想要在

本條款起始處所舉的二維陣列例子中對一個 `Array1D` 物件取址，也很少有人會將 `ArrayIndex` 物件（條款 5）交給一個「參數型別並非 `ArrayIndex`」的函式。許多情況下，proxies 可完美取代其所代表之真正物件。一旦出現這種情況，意味兩者間的隱微差異不是重點。

條款 31：讓函式根據一個以上的物件型別來決定如何虛擬化

有時候，呃，套用一句厘語，『好事不應單行』。怎麼說呢？舉個例子，假設你積極尋找一份軟體開發工作，準備毛遂自薦給華盛頓雷蒙區的一家聲望好、評論佳、薪水高的著名軟體公司 — 喔，呵呵，我是指 Nintendo 啦。爲了讓 Nintendo 的管理階層注意到你，你可能決定寫一個視訊遊戲軟體，場景發生於外太空，涉及太空船、太空站、小行星等天體。

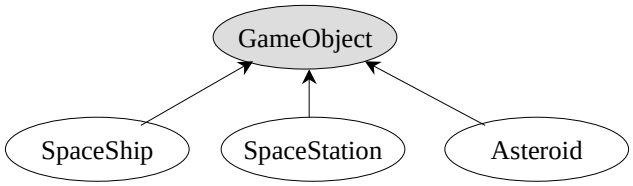
當太空船、太空站、小行星颼颼掠過你所構築的人工世界，它們自然也應該有互相碰撞的風險。假設碰撞規則如下：

- 如果太空船和太空站以低速碰撞，太空船會泊進太空站內。否則太空船和太空站受到的損害與其碰撞速度成正比。
- 如果太空船和太空船碰撞，或是太空站和太空站碰撞，碰撞雙方都遭受損害，損害程度與撞擊速度成正比。
- 如果小號的小行星碰撞到太空船或太空站，小行星會損毀。如果碰撞的是大號小行星，則太空船或太空站損毀。
- 如果小行星撞擊另一顆小行星，兩者都碎裂成更小的小行星，向四面八方散逸。

聽起來這或許是個無聊的遊戲，但是對此處目的而言已經足夠。此處的目的是思考如何架構出兩物碰撞的 C++ 碼。

一開始我們應該標示出太空船、太空站、小行星三者的共同特性。如果沒有意外，它們統統處於運動狀態，所以它們都有速度。有了這樣的共通性，我們很自然地會定義一個被三者繼承的 base class。這樣的 class 在實際設計時，幾乎必然成爲一個抽象

基礎類別（abstract base class）。如果你注意到我在條款 33 所給的警告，base classes 應該總是抽象的。整個繼承體系看起來如下所示：




```
class GameObject { ... };
class SpaceShip: public GameObject { ... };
class SpaceStation: public GameObject { ... };
class Asteroid: public GameObject { ... };
```

假設你已經深入程式的五臟六腑，準備寫出函式碼來檢驗並處理物件的碰撞。你可能會產生這樣一個函式：

```
void checkForCollision(GameObject& object1,
                       GameObject& object2)
{
  if (theyJustCollided(object1, object2)) {
    processCollision(object1, object2);
  }
  else {
    ...
  }
}
```

現在，挑戰浮現出來了。當你呼叫 `processCollision`，你知道 `object1` 和 `object2` 彼此碰撞，而且你也知道其碰撞結果視 `object1` 到底是什麼以及 `object2` 到底是什麼而定，但是你並不知道它們到底是什麼；你只知道它們都是 `GameObjects`。如果碰撞的處理程序只依 `object1` 的動態型別而定，你可以令 `processCollision` 成爲 `GameObject` 的虛擬函式，然後呼叫 `object1.processCollision(object2)`。如果碰撞處理程序依 `object2` 的動態型別而定，你的應對之道應該相同。然而事實上，碰撞結果同時視 `object1` 和 `object2` 兩者的動態型別而定。顯然，你看，上述的函式呼叫只因「單一」物件而虛擬化，是不夠的。

你需要的是某種函式，其行爲視一個以上的物件型別而定。C++ 並未提供這樣的函式。儘管如此你還是必須滿足出上述要求。問題是如何滿足？

可能性之一就是捨棄 C++，選擇另一種程式語言。例如改用 CLOS — Common Lisp Object System。CLOS 支援你所能夠想像的最一般化的物件導向函式喚起機制：所謂 multi-methods。Multi-method 是一種「根據你所希望的多個參數而虛擬化」的函式。CLOS 甚至走得更遠，給你堅實的控制權，讓你控制「多載化（overloaded）multi-methods」的呼叫行爲決議（*resolved*）程序。

然而，假設你必須以 C++ 完成任務，也就是你必須自行想辦法完成上述需求（常被稱爲 double-dispatching）。此名稱來自物件導向程式設計社群，在那個領域裡，人們把一個「虛擬函式呼叫動作」稱爲一個 "message dispatch"（訊息派送）。因此某個函式呼叫如果根據兩個參數而虛擬化，自然而然地就被稱爲 "double dispatch"。更廣泛的情況（函式根據多個參數而虛擬化）則被稱爲 multiple dispatch。有數種作法值得考慮，沒有一個完美無瑕，但這不應該令你感到驚訝。C++ 並無直接支援 double-dispatching，所以你必須自行完成「編譯器對虛擬函式的實作工作」（見條款 24）；如果那份工作輕而易舉，今天我們大概都停留在 C 語言並自己完成虛擬函式機制了。不，我們不是這樣，也不想這樣。所以，繫緊你的安全帶，迎面而來將是一段岐嶇不平的山路。

虛擬函式 + RTTI（執行時期型別辨識）

虛擬函式實作出 single dispatch；那是我們所需要的一半。編譯器有能力爲我們做出虛擬函式，所以讓我們從「在 `GameObject` 中宣告一個虛擬函式 `collide`」開始。這個函式會在 derived classes 中被改寫：

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  ...
};
```

此處我只顯示 derived class `SpaceShip`，至於 `SpaceStation` 和 `Asteroid` 的情況完全相同。

最一般化的 double-dispatching 實作法，把我們帶回蠻荒世界：利用一長串的 if-then-elses 來模擬虛擬函式。在如此粗暴而不優雅的世界裡，我們先找出 otherObject 的真實型別，然後對它測試所有可能性：

```
// 如果我們和一個未知型別的物件相撞，就丟出以下的 exception：
class CollisionWithUnknownObject {
public:
  CollisionWithUnknownObject(GameObject& whatWeHit);
  ...
};

void SpaceShip::collide(GameObject& otherObject)
{
  const type_info& objectType = typeid(otherObject);

  if (objectType == typeid(SpaceShip)) {
    SpaceShip& ss = static_cast<SpaceShip&>(otherObject);

    process a SpaceShip-SpaceShip collision;
  }
  else if (objectType == typeid(SpaceStation)) {
    SpaceStation& ss =
      static_cast<SpaceStation&>(otherObject);

    process a SpaceShip-SpaceStation collision;
  }
  else if (objectType == typeid(Asteroid)) {
    Asteroid& a = static_cast<Asteroid&>(otherObject);

    process a SpaceShip-Asteroid collision;
  }
  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

注意，我們只需決定碰撞雙方之一的型別。另一物件是 *this，其型別已經被虛擬函式機制決定下來了。我們正位於一個 SpaceShip member function 內，所以 *this 一定是個 SpaceShip 物件。因此我們只需設法找出 otherObject 的真正型別即可。

這段碼並不複雜。它很容易寫出，也可以有效運作。那正是 RTTI 令人煩惱的一個理由：它看起來無害。這段碼內可能出現的危機，竟只由最後一個 else 子句所發出的 exception 來警示。

此刻，我們必須對封裝（encapsulation）說再見，因爲每一個 `collide` 函式都必須知道其每一個手足類別 — 也就是所有繼承自 `GameObject` 的那些 classes。更明確地說，如果有個新型物件加入遊戲行列，我們必須修改上述程式中的每一個可能遭遇新物件的 RTTI-based `if-then-else` 串鏈，如果我們遺忘了其中任何一個，程式就會出錯，而此錯誤並不明顯。此外，編譯器無法幫助我們偵測出這樣的疏失，因爲它對我們的行爲一無所悉（見條款 E39）。

這種「以型別爲行事基準」的程式方法在 C 語言中有一段很長的歷史，而我們對它的一個固有印象就是：它會造成程式難以維護。欲對這種程式再擴充（譯註：指加入新的型別），基本上是很麻煩的。這便是虛擬函式當初被發明的主要原因：把生產及維護「以型別爲行事基準之函式」的負荷，從程式員肩上移轉給編譯器。如果我們使用 RTTI 來實作 double-dispatching，等於回到了那個老舊而糟糕的年代。

老舊糟糕年代下的技術會導至 C 語言發生錯誤，也會導至 C++ 語言發生錯誤。爲了辨識出人爲過失，我們爲 `collide` 函式加上最後一個 `else` 子句，該子句用來處理不明撞擊物。這樣的情況原則上是不可能的，但是當我們決定使用 RTTI，我們的道理又在哪裡呢？有許多種方法可以處理此等預料之外的撞擊，但是沒有一個令人滿意。在這個例子中我們選擇丟出 exception，但是呼叫者是否能夠比我們處理得更好，實在不無疑問，因爲我們面對的是未知的東西。

只使用虛擬函式

有一個辦法可以將「以 RTTI 作法實現 double-dispatching」的風險降至最低，但是在我們討論那個作法之前，先看看如何能夠只以虛擬函式來解決這個問題。此策略一開始的基本結構和先前的 RTTI 法相同，`collide` 被宣告爲 `GameObject` 內的虛擬函式，並在每一個 derived class 內重新獲得定義。此外，`collide` 亦在每一個 class 內被多載化，每一個多載化版本對應繼承體系中的一個 derived class：

```
class SpaceShip;          // 前置宣告（forward declarations）
class SpaceStation;
class Asteroid;
```

```
class GameObject {
public:
  // 譯註：collide 被多載化，每一個版本對應繼承體系中的一個 derived class
  virtual void collide(GameObject&   otherObject) = 0;
  virtual void collide(SpaceShip&    otherObject) = 0;
  virtual void collide(SpaceStation& otherObject) = 0;
  virtual void collide(Asteroid&     otherobject) = 0;
  ...
};
class SpaceShip: public GameObject {
public:
  // 譯註：collide 被多載化，每一個版本對應繼承體系中的一個 derived class
  virtual void collide(GameObject&   otherObject);
  virtual void collide(SpaceShip&    otherObject);
  virtual void collide(SpaceStation& otherObject);
  virtual void collide(Asteroid&     otherobject);
  ...
};
```

基本想法是，將 double-dispatching 以兩個 single dispatches（也就是兩個分離的虛擬函式呼叫）實作出來：其一用來決定第一物件的動態型別，其二用來決定第二物件的動態型別。和先前一樣，第一個虛擬函式呼叫動作針對的是「接獲 `GameObject&` 參數」的 `collide` 函式。該函式現在簡單得有點讓人嚇一跳：

```
void SpaceShip::collide(GameObject& otherObject)
{
  otherObject.collide(*this);
}
```

乍見之下這似乎是個 `collide` 遞迴呼叫，只是把參數次序顛倒，也就是 `otherObject` 搖身一變爲呼叫者，而 `*this` 則變成參數。然而，睜大眼睛再看一遍，這並不是遞迴呼叫。如你所知，編譯器必須根據「此函式所獲得之引數」的靜態型別，才能解析出哪一組函式被呼叫。本情況下，四個不同的 `collide` 函式都可能被呼叫，但是雀屏中選者實乃根據 `*this` 的靜態型別。該型別是什麼？在 class `SpaceShip` 的 member function 內，`*this` 的型別一定是 `SpaceShip`。因此上述程式碼將喚起「參數型別爲 `SpaceShip&`（而非 `GameObject&`）」的 `collide` 函式。

所有的 `collide` 都是虛擬函式，所以 `SpaceShip::collide` 內的呼叫動作會被決議爲「對應於 `otherObject` 真實型別」的那個 `collide` 函式；而在其中，兩個物件的真實型別都知道了：左端物件是 `*this`（其型別也就是實作出此 `collide` 函式之 class），右端物件的真實型別是 `SpaceShip`，亦即參數型別。

看過 SpaceShip 的其他 collide 函式內容後，情況可能會更清楚：

```
void SpaceShip::collide(SpaceShip& otherObject)
{
  process a SpaceShip-SpaceShip collision;
}
void SpaceShip::collide(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}
void SpaceShip::collide(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

如你所見，既不會亂七八糟，也不令人吃驚；不需要 RTTI，也不需要爲了未知的物件型別丟出 exceptions。不可能會有未知的物件型別 — 那正是使用虛擬函式的關鍵點。事實上，如果不是因爲有致命的瑕疵，這幾乎可以說是 double-dispatching 問題的完美解答。

我所謂致命的瑕疵，和稍早我們看到的 RTTI 解法一樣：每個類別都必須知道其手足類別。一旦有新的 classes 加入，程式碼就必須修改。然而，程式碼的修改在本解法的情況有些不同。這裡並沒有 if-then-elses 需要修改，卻有某些常常容易出錯的東西：每一個 class 的定義式都必須修正，含入一個新的虛擬函式。舉個例子，如果你決定爲遊戲軟體加上一個新的 class Satellite（衛星，繼承自 GameObject），你必須爲程式中每一個既有的 classes 增加一個新的 collide 函式。

「修改既有的 classes」往往不是你可以單獨（或有權力）決定的事。如果你並非完全自力寫出這個遊戲軟體，而是以一個市售商用程式庫爲起點 — 該程式庫有個視訊遊戲軟體的應用程式框構（application framework），你或許就不需要寫 GameObject class 或是其 derived classes。此情況下，增加新的 member functions（不論是否爲 virtual）是不可能的。有時候，你可能真的可以碰觸那些有待修改的 classes，但你沒有權力那麼做。假設你被 Nintendo 錄用，而你被指派的專案運用了某個程式庫，內含 GameObject 及其他 classes。當然你不會是唯一使用這個程式庫的人，而 Nintendo 或許並不願意爲了「你個人需要增加一個新的物件型別」而重新編譯每一個運用此程式庫的軟體。實際情況是，程式庫愈被廣泛採用，愈是難得修改，因爲重新編譯程式

庫的每一個使用者（應用程式），成本實在太大了。（如何設計程式庫使其編譯相依程度得以最小化，細節請見條款 E34）

簡單地說就是，如果你需要在你的程式中實作 double-dispatching，最好的方向就是修改設計，消除此項需求。如果不能，那麼，虛擬函式法比 RTTI 法安全一些，但是如果你對表頭檔（譯註：內含 classes 的宣告和定義）的政治權力不夠，這種作法會束縛你的系統擴充性。至於 RTTI 法，雖不需要重新編譯，卻往往導至軟體難以維護。你總是得付出代價，才能獲得機會。

自行模擬虛擬函式表格（Virtual Function Tables）

有一個方法可以強化那些機會。回憶條款 24，編譯器通常藉由函式指標陣列（vtbl）來實作虛擬函式；當某個虛擬函式被呼叫，編譯器便索引至該陣列內，取得一個函式指標。有了 vtbl，編譯器便可以不必執行一大串 `if-then-else` 運算，並得以在所有的虛擬函式呼叫端產生相同的碼，用以：(1) 決定正確的 vtbl 索引，(2) 呼叫 vtbl 中的索引位置內所指的函式。

沒有理由你不能夠自行完成這一切。如果這麼做，不只你的 RTTI-based 碼會更有效率（因爲索引至一個陣列內並喚起其中的函式指標，幾乎總是比執行一連串 `if-then-else` 更有效率，所需的碼也更少），你也可以將 RTTI 的使用隔離至單一地點：vtbl 初始化之處。我必須提醒你，如果你不夠堅強，往下閱讀之前請先大口深呼吸。

讓我們從修改 `GameObject` 繼承體系內的函式開始：

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};

class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  virtual void hitSpaceShip(SpaceShip& otherObject);
  virtual void hitSpaceStation(SpaceStation& otherObject);
  virtual void hitAsteroid(Asteroid& otherobject);
  ...
};
```

```
void SpaceShip::hitSpaceShip(SpaceShip& otherObject)
{
  process a SpaceShip-SpaceShip collision;
}

void SpaceShip::hitSpaceStation(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}

void SpaceShip::hitAsteroid(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

就像一開始所說的 RTTI 解法一樣，`GameObject` class 只含一個碰撞處理函式，此函式執行「兩個必要的 dispatches」中的第一個。而和後來出現的虛擬函式解法一樣，每一種天體互動都被封裝於一個函式之中，不過現在這些函式有不同的名字，不再共享相同的 `collide` 名稱。這裡放棄了多載化（overloading），我們很快就會看到理由（譯註：p.243）。上述設計涵蓋我們需要的每一樣東西，獨缺 `SpaceShip::collide` 實作碼；那是喚起各個撞擊函式的地方。一如先前，一旦我們成功地實作出 `SpaceShip` class，`SpaceStation` 和 `Asteroid` classes 可以蕭規曹隨。

在 `SpaceShip::collide` 函式內，我們需要一種方法，將參數 `otherObject` 的動態型別對映至某個 member function 指標，指向適當的碰撞處理函式。一種簡單的作法就是產生一個關聯式（associative）陣列，只要獲得 class 名稱，便能導出適當的 member function 指標。直接使用這個陣列來實作 `collide` 是有可能的，但如果加上一個中介函式 `lookup`，會更容易些。`lookup` 接獲一個 `GameObject` 並傳回適當的 member function 指標。也就是說，你交給 `lookup` 一個 `GameObject`，它會傳回一個指標，指向「當你和 `GameObject` 物件碰撞時」必須喚起的 member function。

下面是 `lookup` 的宣告：

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);
  static HitFunctionPtr lookup(const GameObject& whatWeHit);
  ...
};
```

函式指標的語法從來就不曾讓人覺得可愛有趣，member function 的函式指標，情況尤有過之。所以我以 typedef 定義出 `HitFunctionPtr`，代表一個「指向 `SpaceShip` member function」的指標 — 該函式之參數爲一個 `GameObject&`，不傳回任何東西。

一旦有了 `lookup` 函式，`collide` 的情況就像「黑暗已經過去，黎明還會遠嗎」：

```
void SpaceShip::collide(GameObject& otherObject)
{
  HitFunctionPtr hfp =
    lookup(otherObject);                  // 找出呼叫的對象（一個函式）

  if (hfp) {                              // 如果找到目標
    (this->*hfp)(otherObject);            // 就呼叫之
  }
  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

如果關聯式陣列的內容能夠與 `GameObject` 繼承體系保持一致，`lookup` 就一定能夠針對我們傳進去的物件，找出一個有效的函式指標。但人類畢竟不是上帝，即使是最小心謹慎並且技巧高超的軟體系統，還是可能到處蔓延錯誤。這就是爲什麼我們還是需要檢驗，確定 `lookup` 傳回一個有效指標。這也是爲什麼我們還是在 `lookup` 失敗時（雖然應該不可能發生）丟出一個 exception。

現在剩下的唯一工作就是將 `lookup` 實作出來。如果已有一個關聯式陣列，可將物件型別映射到某個 member function 指標，那麼 `lookup` 本身很簡單。但是產生、初始化、摧毀該關聯式陣列，倒成爲一個有趣的問題。

此一陣列應該在被使用之前先被產生並初始化，並在它不再被使用時加以摧毀。我們可以利用 `new` 和 `delete` 親手產生和摧毀陣列，但是那樣容易發生錯誤：如何能夠保證陣列不會在初始化之前先被使用？比較好的解決辦法就是讓編譯器將此程序自動化，亦即讓關聯式陣列成爲 `lookup` 內的 static 物件。那麼，只有在 `lookup` 第一次被呼叫時它才會被產生，而在 `main` 結束之後它才會被摧毀（見條款 E47）。

此外，我們可以運用 Standard Template Library（見條款 35）所提供的 `map` template 做爲關聯式陣列，因爲 `map` 的功能正是如此：

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);
  typedef map<string, HitFunctionPtr> HitMap;
  ...
};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;
  ...
}
```

此處 `collisionMap` 便是我們的關聯陣列，用來將一個以 `string` 呈現的 class 名稱，對應到一個 `SpaceShip` member function 指標。由於 `map<string, HitFunctionPtr>` 相當冗長，我以一個 typedef 簡化之。（你不妨試試，重寫 `collisionMap` 宣告式而不利用 `HitMap` 和 `HitFunctionPtr` 這兩個 typedefs。大部份人試一次後就手軟了）

有了 `collisionMap`，`lookup` 的難度陡降，因爲資料的搜尋原本就是 `map` class 直接支援的行爲之一，而我們又總是能夠對著 `typeid` 傳回的物件呼叫 `name` 函式（它將傳回標的物的動態型別名稱[3]）。只要找出 `collisionMap` 內「與 `lookup` 之引數相應」的那個動態型別，`lookup` 便算完成了。

`lookup` 函式碼十分直接易懂，但如果你不熟悉 Standard Template Library（見條款 35），可能又不是那麼易懂。別擔心。函式內的註解會說明進行的動作。

```
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;           // 稍後我們將看到如何
                                        // 初始化這玩意兒。

  // 爲 whatWeHit 尋找碰撞處理函式。傳回值是一個類似指標的物件，
  // 我們稱之爲 "iterator" (見條款 35).
  HitMap::iterator mapEntry=
    collisionMap.find(typeid(whatWeHit).name());
```

[3] 事實並非總是如此。C++ standard 並未明定 `type_info::name` 的傳回值。不同編譯器的行爲不盡相同。例如，已知一個 class `SpaceShip`，我知道有個編譯器的 `type_info::name` 會爲之傳回 "class SpaceShip"。較好的設計是以「class 相應之 `type_info` 物件」的位址來鑑識 class，因爲它絕對是獨一無二的。那麼，`HitMap` 的型別應該宣告爲 `map<const type_info*, HitFunctionPtr>` 才是。

```
  // 如果尋找失敗，mapEntry == collisionMap.end()；
  // 這是標準的 map 行爲。見條款 35。
  if (mapEntry == collisionMap.end()) return 0;

  // 如果到達這裡，表示搜尋成功。此時 mapEntry 指向一個完整的 map 條目，
  // 那是一個 (string, HitFunctionPtr) pair。我們只需其中的第二個成份，
  // 當做傳回值。
  return (*mapEntry).second;
}
```

函式的最後一個述句傳回的是 `(*mapEntry).second`，而非使用較爲傳統的 `mapEntry->second` 型式，這是爲了滿足 STL 的奇特行爲。詳情見 p.96 註解。

將自行模擬的虛擬函式表格（Virtual Function Tables）初始化

現在我們面臨了 `collisionMap` 的初始化問題。我們或許希望這麼做：

```
// 一個不正確的作法
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;

  collisionMap["SpaceShip"] = &hitSpaceShip;
  collisionMap["SpaceStation"] = &hitSpaceStation;
  collisionMap["Asteroid"] = &hitAsteroid;
  ...
}
```

但是這會在每次 `lookup` 被呼叫時將「member function 指標」安插到 `collisionMap` 內，這是不必要的。此外上述作法也無法編譯，不過那是次要問題，很快就可以解決。

此刻我們所需要的是一個方法，可以將「member function 指標」放進 `collisionMap` 內一次就好 — 在 `collisionMap` 誕生時刻。這很容易達成，只需寫一個 private static member function，名爲 `initializeCollisionMap`，用以產生並初始化我們的 `map`，然後以 `initializeCollisionMap` 的傳回值做爲 `collisionMap` 的初值即可：

```
class SpaceShip: public GameObject {
private:
  static HitMap initializeCollisionMap();
  ...
};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap = initializeCollisionMap();
  ...
}
```

但這意味我們可能必須爲複製行爲（亦即「將 initializeCollisionMap 傳回之 map 物件複製給 collisionMap」）付出成本（見條款 19 和 20）。我們希望最好避免此事。如果 initializeCollisionMap 傳回的是一個指標，就不需要付出成本，但那麼一來我們又得操心「指標所指的 map 物件應該在適當時候被刪除」這檔事兒。

幸運的是，有個方法讓我們魚與熊掌兼得。我們可以把 collisionMap 放進一個 smart pointer（見條款 28）中，於是當指標本身被摧毀，其所指物亦會自動被刪除。事實上 C++ 標準程式庫就有一個名爲 auto_ptr 的 template，正是一個做爲此等用途的 smart pointer（見條款 9）。只要讓 collisionMap 成爲 lookup 中的一個 static auto_ptr，我們就可以令 initializeCollisionMap 傳回指標，指向一個已經初始化的 map 物件，而不必擔心資源遺失問題；當 collisionMap 被摧毀，其所指的那個 map 亦會被自動摧毀。於是：

```
class SpaceShip: public GameObject {
private:
  static HitMap * initializeCollisionMap();
  ...
};

SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static auto_ptr<HitMap>
    collisionMap(initializeCollisionMap());
  ...
}
```

最直接而明顯的 initializeCollisionMap 實作法似乎是這樣：

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;

  return phm;
}
```

但正如稍早我所提示，這無法通過編譯。因爲 HitMap 被宣告爲用來存放「member functions 指標」，這些指標都需有相同型別（亦即 GameObject）的參數。但是

hitSpaceShip 的參數是一個 SpaceShip，hitSpaceStation 的參數是一個 SpaceStation，而 hitAsteroid 的參數是一個 Asteroid。雖然 SpaceShip, SpaceStation 和 Asteroid 都可以被隱式轉換爲 GameObject，但是「以這些型別爲參數」之各個函式指標之間，卻不存在隱式轉換。

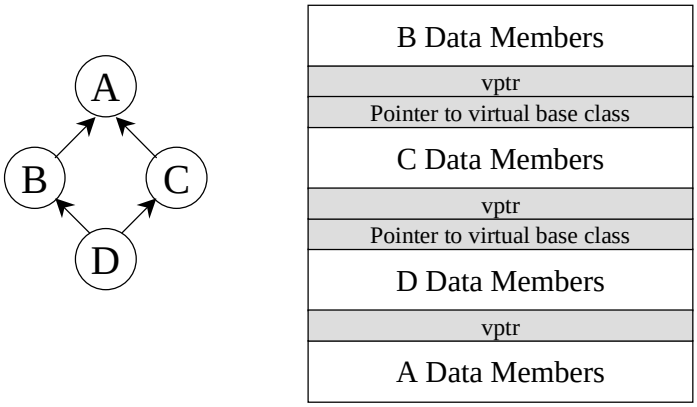
爲了安撫編譯器，你可能會企圖使用 reinterpret_casts（見條款 2），通常那是在兩個函式指標之間做轉型時的一個選擇：

```
// 一個壞主意…
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceShip);

  (*phm)["SpaceStation"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceStation);

  (*phm)["Asteroid"] =
    reinterpret_cast<HitFunctionPtr>(&hitAsteroid);

  return phm;
}
```

這雖然可通過編譯，卻是個壞主意。它伴隨了某些你應該絕對禁止的行爲：欺騙編譯器。這樣的轉型是告訴編譯器說，hitSpaceShip, hitSpaceStation 和 hitAsteroid 都是「期望獲得一個 GameObject 引數」的函式，但此並非事實。hitSpaceShip 期望獲得一個 SpaceShip，hitSpaceStation 期望獲得一個 SpaceStation，而 hitAsteroid 期望獲得一個 Asteroid。上述的轉型語法對編譯器說了謊話。

比道德問題更嚴重的是，編譯器不喜歡被欺騙。當它們發現它們被辜負，它們會找出報仇雪恨的方法。在這個例子中，它們可能會在你「透過 *phm 呼叫某些函式」時，爲你產生不良的碼 — 如果在那些函式中 GameObject's derived classes 運用了多重繼承或是擁有虛擬基礎類別（virtual base classes）。換句話說，如果 SpaceStation, SpaceShip 或 Asteroid 有 GameObject 之外的其他 base classes，你可能會發現，你在 collide 中對碰撞處理函式的呼叫，會導至相當粗魯的行爲（譯註：因爲非自然多型 unnatural polymorphism 之故，見下說明）。

爲了說明其前因後果，讓我們再次考慮條款 24 所描述的 A-B-C-D 繼承體系以及 D 物件可能的物件佈局：



D 物件內的四個「class 成份」，每一個都有不同位址。這很重要，因爲雖然指標和 references 的行爲不同（見條款 1），編譯器通常是以指標來實作 references。因此，pass-by-reference 通常是利用「傳遞一個指向物件的指標」完成。當物件（例如 D 物件）擁有多個 base classes，並以 by reference 方式傳遞給函式，編譯器是否傳遞了正確的位址（此位址相應於被呼叫函式之參數宣告型別），將是非常重要的關鍵。

如果你欺騙編譯器，告訴它你的函式期望獲得一個 GameObject，而其實它真正期望獲得的是個 SpaceShip 或 SpaceStation 呢？那麼，當你呼叫那個函式，編譯器就會傳遞錯誤的位址，導至執行時期可怕的大屠殺。這種問題很難找出原因。轉型令人沮喪，原因有許多個，這是其中之一。

那麼就把「轉型法」三振出局吧。很好，但這麼一來先前所說的「函式指標型別」不吻合的情況便依然存在。只有一個辦法可以解決衝突：改變函式的型別，使它們統統接納 GameObject 引數：

```
class GameObject {                  // 這個並未改變
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};
```

```
class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);

  // 這些函式現在統統接受一個 GameObject 參數
  virtual void hitSpaceShip(GameObject& spaceShip);
  virtual void hitSpaceStation(GameObject& spaceStation);
  virtual void hitAsteroid(GameObject& asteroid);
  ...
};
```

我們對 double-dispatching 的解法原本是以「多載化的虛擬函式 `collide`」爲基礎。現在終於瞭解爲什麼這裡不能再使用多載化了吧（譯註：p.236 曾說會給一個理由，就是這裡）。這便是爲什麼我們決定以「由 member function 指標所組成的關聯式陣列」取而代之的緣故。所有撞擊函式有著完全相同的參數型別，所以我必須給它們不同的函式名稱。

現在我們可以寫出我們一直希望的 `initializeCollisionMap` 型式了：

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{ // 譯註：同 p.240
  HitMap *phm = new HitMap;

  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;

  return phm;
}
```

很可惜，撞擊函式如今獲得的是個一般性的 `GameObject` 參數，而非它們所期望的精確的 derived class 參數。爲了在預期行爲中夾帶真實性，我們必須在每個函式頂端運用 `dynamic_cast`（見條款 2）：

```
// 譯註：舊版本在 p236
void SpaceShip::hitSpaceShip(GameObject& spaceShip)
{
  SpaceShip& otherShip=
    dynamic_cast<SpaceShip&>(spaceShip);

  process a SpaceShip-SpaceShip collision;
}

void SpaceShip::hitSpaceStation(GameObject& spaceStation)
{
  SpaceStation& station=
    dynamic_cast<SpaceStation&>(spaceStation);

  process a SpaceShip-SpaceStation collision;
}
```

```
void SpaceShip::hitAsteroid(GameObject& asteroid)
{
  Asteroid& theAsteroid =
    dynamic_cast<Asteroid&>(asteroid);
  process a SpaceShip-Asteroid collision;
}
```

dynamic_casts 如果轉型失敗，會丟出一個 bad_cast exception。當然啦，它們應該絕對不會失敗才是，因爲上述各個 hit_ 函式應該絕不會獲得不正確的參數型別。不過不怕一萬只怕萬一，總比滿口抱歉的好。

使用「非成員（Non-Member）函式」之碰撞處理函式

現在我們知道如何建立一個類似 vtbl 的關聯式陣列，使我們得以完成 double-dispatch 的第二半部，我們也知道如何將關聯式陣列的細節封裝於一個 lookup 函式內。然而由於這個陣列內含指標，指向 member functions，所以如果有新的 GameObject 型別加入這個遊戲軟體，我們仍然需要修改 class 的定義。這也意味所有客戶都必須重新編譯，甚至那些並不在乎新型物件的人。舉個例子，如果 Satellite 加進這場遊戲，我們必須膨脹 SpaceShip class，多寫一個用來處理「衛星和太空船之間的碰撞」的函式宣告。SpaceShip 的所有使用者都因而必須重新編譯，即使他們並不在乎衛星存在與否。這不正是我們拒絕「純以虛擬函式來解決 double-dispatching」的原因嗎？先前的解法甚至比眼前所討論的解法單純得多呢。

如果關聯式陣列內含的指標所指的向 non-member functions，重新編譯的問題便可消除。猶有進者，改用「non-member 碰撞處理函式」，可以解決一個截至目前一直被我們忽略的設計問題，那就是：如果兩個不同型別的物體發生碰撞，到底哪一個 class 應該負責處理？以先前發展的情況來看，如果物件 1 和物件 2 碰撞，而物件 1 碰巧是 processCollision 的左端引數，那麼此一碰撞事件將在物件 1 所屬的 class 中被處理。然而如果物件 2 碰巧是 processCollision 的左端引數，那麼此一碰撞事件就在物件 2 所屬的 class 中被處理。這合理嗎？難道沒有更好的設計，使型別分別爲 A 和 B 的兩物發生碰撞時，既不由 A 也不由 B 處理，而是由某個中立第三者處理嗎？

如果將碰撞處理函式移出 classes 之外，我們就可以給予使用者一些「不含任何碰撞處理函式」的 class 定義式（位於表頭檔中），然後便可構築我們的 processCollision 函式如下：

```
#include "SpaceShip.h"
#include "SpaceStation.h"
#include "Asteroid.h"

namespace {                    // 未具名的 namespace — 見稍後說明

  // 主要的碰撞處理函式
  void shipAsteroid(GameObject& spaceShip,
                    GameObject& asteroid);

  void shipStation(GameObject& spaceShip,
                   GameObject& spaceStation);

  void asteroidStation(GameObject& asteroid,
                       GameObject& spaceStation);
  ...
  // 次要的碰撞處理函式，只是爲了實現對稱性：
  // 對調參數位置，然後呼叫主要的碰撞處理函式。
  void asteroidShip(GameObject& asteroid,
                    GameObject& spaceShip)
  { shipAsteroid(spaceShip, asteroid); }

  void stationShip(GameObject& spaceStation,
                   GameObject& spaceShip)
  { shipStation(spaceShip, spaceStation); }

  void stationAsteroid(GameObject& spaceStation,
                       GameObject& asteroid)
  { asteroidStation(asteroid, spaceStation); }
  ...

  // 稍後我會對這些 types/functions 有所描述和說明
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);
  typedef map< pair<string,string>, HitFunctionPtr > HitMap;

  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2);
  HitMap * initializeCollisionMap();

  HitFunctionPtr lookup(const string& class1,
                        const string& class2);
} // namespace 結束

void processCollision(GameObject& object1,
                      GameObject& object2)
{
  HitFunctionPtr phf = lookup(typeid(object1).name(),
                              typeid(object2).name());

  if (phf) phf(object1, object2);
  else throw UnknownCollision(object1, object2);
}
```

注意此處使用了一個未具名的 namespace，內含用以實作 `processCollision` 的各個函式。未具名之 namespace 內的每樣東西對其所駐在的編譯單元（檔案）而言都是私有的，其效果就好像在檔案裡頭將函式宣告爲 static 一樣。由於 namespaces 的出現，「檔案生存空間（file scope）內的 statics」已經不再繼續被鼓勵使用，所以你應該儘快讓自己習慣使用無具名的 namespaces — 如果你的編譯器支援它的話。

這份實作碼的觀念和先前的 member functions 版相同，但其中有些極小差異。第一，`HitFunctionPtr` 如今是個 typedef，表示一個指標，指向一個 non-member function。第二，exception class `CollisionWithUnknownObject` 已經被重新命名爲 `UnknownCollision` 並改爲取得兩個（而不再是一個）物件。最後一點，`lookup` 現在必須接獲兩個型別名稱，並執行 double-dispatch 的完整兩半部。這也意味我們的 collision map 如今必須持有三份資訊：兩個型別名稱和一個 `HitFunctionPtr`。

但是標準的 `map` class 只能持有兩份資訊。我們可以輕易解決這個問題：標準的 `pair` template 讓我們將兩個型別名稱綑綁在一塊兒，成爲單一物件。於是，`initializeCollisionMap` 加上其輔助函式 `makeStringPair`，看起來像這樣：

```
// 我們藉此函式，以兩個 char* 字面常數產生一個 pair<string,string> 物件。
// 此函式被用於 initializeCollisionMap（稍後出現）內。請注意此函式如何
// 形成傳回值最佳化（return value optimization，見條款 20）。
namespace {         // 又是無具名的 namespace — 見稍後說明
  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2)
  { return pair<string,string>(s1, s2);  }
} // namespace 結束

namespace {         // 又是無具名的 namespace — 見稍後說明
  HitMap * initializeCollisionMap()
  {
    HitMap *phm = new HitMap;

    (*phm)[makeStringPair("SpaceShip","Asteroid")] =
      &shipAsteroid;

    (*phm)[makeStringPair("SpaceShip", "SpaceStation")] =
      &shipStation;
    ...
    return phm;
  }
} // namespace 結束
```

lookup 也必須修改，以便接納 pair<string, string> 物件，此物構成 collision map 的第一成份：

```
namespace {          // 又是無具名的 namespace，稍後我會解釋。真的，相信我
  HitFunctionPtr lookup(const string& class1,
                        const string& class2)
  {
    static auto_ptr<HitMap>
      collisionMap(initializeCollisionMap());
    // 稍後對於 make_pair 有一些說明
    HitMap::iterator mapEntry=
      collisionMap->find(make_pair(class1, class2));
    if (mapEntry == collisionMap->end()) return 0;
    return (*mapEntry).second;
  }
} // namespace 結束
```

這幾乎和先前版本完全相同。唯一真正的差異是在以下述句中使用 make_pair 函式：

```
HitMap::iterator mapEntry=
  collisionMap->find(make_pair(class1, class2));
```

make_pair 是標準程式庫（見條款 E49 和條款 35）提供的一個十分便利的 function template，可免除我們在建構一個 pair 物件時「必須指定型別」的麻煩。以上述句也可以改寫爲：

```
HitMap::iterator mapEntry=
  collisionMap->find(pair<string,string>(class1, class2));
```

這段碼需要多打幾個字，而爲 pair 指定型別又顯多餘（它們一定和 class1 及 class2 的型別相同，不是嗎），所以大部份人比較喜歡採用剛才那個 make_pair 型式。

由於 makeStringPair, initializeCollisionMap 和 lookup 都被宣告於一個未具名的 namespace 內，所以它們都必須實作於相同的 namespace 內。這也就是爲什麼上述函式的實作碼都被我放在一個未具名的 namespace 內（和它們的宣告駐在同一編譯單元）：這麼一來聯結器才能夠正確地將其定義（亦即其實作碼）和稍早出現的宣告關聯在一起。

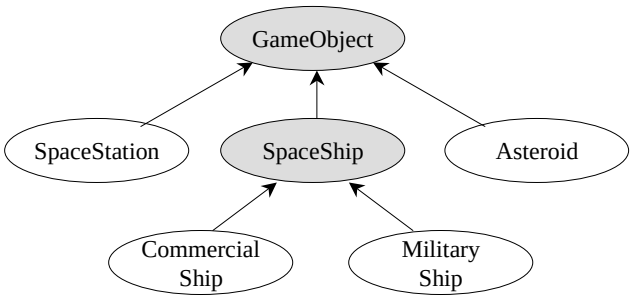
我們終於達成了目標。一旦有新的 GameObject subclasses 加進這個繼承體系中，原有的 classes 不再需要重新編譯（除非它們想要使用新的 classes）。我們不需維護糾

葛混亂的「以 RTTI 爲基礎的」`switch` 或 `if-then-else`。如果新的 classes 欲加入 `GameObject` 繼承體系，只需其本身定義良好；我們的系統亦只需要局部改變：在 `initializeCollisionMap` 內爲 `map` 增加一筆（或更多）項目，並在「與 `processCollision` 相應的那個無具名 namespace」內增加新碰撞處理函式的宣告。經過了許多努力，終於到達現在這一點，但至少漫長的旅程是值得的。是嗎？是嗎？

也許！

「繼承」+「自行模擬的虛擬函式表格」

我們必須面對最後一個問題（如果此刻你開始心生奇怪，怎麼總是有最後一個問題需要面對，我想你終於真正認識了「爲虛擬函式設計一個實作機制」的困難度）。目前我們所做的每一件事都可以有效運作 — 只要在呼叫碰撞處理函式時不發生 inheritance-based 型別轉換。但假設我們開發了一個遊戲軟體，該軟體有時候必須區分商業太空船和軍事太空船。我們可以修改繼承體系如下，其中已注意條款 33 的忠告，令具象類別 `CommercialShip` 和 `MilitaryShip` 繼承自新的抽象類別 `SpaceShip`：



假設商業太空船和軍事太空船的碰撞行爲完全相同，因此我們希望使用原本已開發出來的碰撞處理函式。更明確地說，如果一個 `MilitaryShip` 和一個 `Asteroid` 碰撞，我們希望被喚起的是：

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
```

但結果並非如此，而是丟出一個 `UnknownCollision` exception。那是因爲 `lookup` 被要求針對型別名稱 "MilitaryShip" 和 "Asteroid" 找出一個對應函式，而 `collisionMap` 中其實並沒有如此函式。雖然一個 `MilitaryShip` 物件可被視爲一

個 SpaceShip 物件，但 lookup 函式並不知道。

猶有甚者，根本沒有簡單的辦法來告知此事。如果你需要實作出 double-dispatching 而且你需要支援 inheritance-based 參數轉換（像上個例子那樣），你唯一可用的資源就是回到我們稍早驗證過的「雙虛擬函式呼叫」機制。那也就意味當你擴大你的繼承體系，你必須要求每個人都重新編譯。有時候情況就是如此，生活只好牽就。

將自行模擬的虛擬函式表格初始化（再度討論）

以上就是關於 double-dispatching 的所有處理技術，但是以如此憾人的音符做爲奏鳴曲的結束，實在讓人覺得不舒服。而且這種不舒適感恐怕會「繞樑三日」。所以，讓我們大略描述 collisionMap 初始化的另一個作法，做爲結論。

截至目前，整個設計完全是靜態的。一旦我們登錄了一個用來處理兩物撞擊的函式，它就永遠杵在那裡了。如果我們希望對撞擊處理函式做新增、移除、修改等動作，抱歉，門兒都沒有。

其實門道還是有的。我們可利用一個 map 來儲存撞擊處理函式，放進某 class 內，該 class 提供一些 member functions，讓我們得以動態修改 map 的內容。例如：

```
class CollisionMap {
public:
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);

  void addEntry(const string& type1,
                const string& type2,
                HitFunctionPtr collisionFunction,
                bool symmetric = true);            // 見稍後說明

  void removeEntry(const string& type1,
                   const string& type2);

  HitFunctionPtr lookup(const string& type1,
                        const string& type2);

  // 這個函式傳回一個 reference，代表僅有的一個 map — 見條款 26
  static CollisionMap& theCollisionMap();

private:
  // 這些函式都是 private，用以防止產生多個 maps — 見條款 26
  CollisionMap();
  CollisionMap(const CollisionMap&);
};
```

這個 class 使我們得以爲 map 增加條目（entries）、移除條目，並尋找與「某一對型別名稱」相應的碰撞處理函式。它也運用條款 26 的技術，將 CollisionMap 物件的個數限制爲 1，因爲我們的系統只需一個 map（帶有多個 maps 的複雜遊戲也很容易想像）。最後，它允許我們將「對稱撞擊造成 map 條目增加」的情況予以簡化（也就是說，T1 物件撞擊 T2 物件的效果和 T2 物件撞擊 T1 物件的效果一樣）：在 addEntry 被呼叫且其 symmetric 參數被指定爲 true 時，自動爲我們加上 map 條目。

有了這個 CollisionMap class，使用者就可以這樣直接地爲 map 加上一個條目：

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "Asteroid",
                                         &shipAsteroid);
void shipStation(GameObject& spaceShip,
                 GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "SpaceStation",
                                         &shipStation);
void asteroidStation(GameObject& asteroid,
                     GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("Asteroid",
                                         "SpaceStation",
                                         &asteroidStation);
...
```

請小心確保這些 map 條目在其對應的任何撞擊發生之前就被加入 map 之中。辦法之一是令 GameObject's subclasses 的 constructors 加以檢查，看看是否在物件產生之際已有適當的 map 條目加入。這需要付出一些時間成本。另一種作法就是產生一個 RegisterCollisionFunction class:

```
class RegisterCollisionFunction {
public:
  RegisterCollisionFunction(
         const string& type1,
         const string& type2,
         CollisionMap::HitFunctionPtr collisionFunction,
         bool symmetric = true)
  {
    CollisionMap::theCollisionMap().addEntry(type1, type2,
                                             collisionFunction,
                                             symmetric);
  }
};
```

client 於是可以利用這種型別的全域物件來自動註冊它們需要的函式：

```
RegisterCollisionFunction cf1("SpaceShip", "Asteroid",
                              &shipAsteroid);

RegisterCollisionFunction cf2("SpaceShip", "SpaceStation",
                              &shipStation);

RegisterCollisionFunction cf3("Asteroid", "SpaceStation",
                              &asteroidStation);
...
int main(int argc, char * argv[])
{
  ...
}
```

由於這些全域物件都是在 `main` 被喚起前產生的，它們的 constructors 所註冊的函式也會在 `main` 被呼叫之前加入 `map` 內。稍後如果我們加入一個新的 derived class：

```
class Satellite: public GameObject { ... };
```

並寫出一個或多個新的碰撞處理函式：

```
void satelliteShip(GameObject& satellite,
                   GameObject& spaceShip);

void satelliteAsteroid(GameObject& satellite,
                       GameObject& asteroid);
```

這些新函式可以類似方法加入 `map` 之中，不需擾動原有的碼：

```
RegisterCollisionFunction cf4("Satellite", "SpaceShip",
                              &satelliteShip);

RegisterCollisionFunction cf5("Satellite", "Asteroid",
                              &satelliteAsteroid);
```

還是沒有完美辦法可以實作出 double dispatch，但此法讓我們可以輕鬆完成一個以 `map` 爲基礎的實作品 — 如果此種作法最吻合我們需要的話。

雜項討論

我們終於抵達了最後一站。本章內含難以歸類的準則。一開始的兩個條款討論 C++ 軟體開發過程如何設計出能夠容納日後變化的系統。是的，物件導向方法應用於系統建構的一個強大力量就是，它支援日後的變化。這些條款描述了一些特定步驟，你可以用來強化你的軟體工事，抵抗這個拒絕停滯的世界帶來的刀戟箭弩。

接下來我將驗證如何在同一個程式中結合 C 和 C++。這個需求導至語言上的額外考量，不過 C++ 畢竟生存於真實世界之中，有時候我們必須面對這樣的問題。

最後，我把「C++ 標準規格」公開之後的各項語言變化做一番摘要整理。我特別涵蓋標準程式庫中翻天覆地的大變化（亦請參考條款 E49）。如果你未曾密切跟隨標準化腳步，對於這些變化可能會有很大的驚喜。是的，標準程式庫中有許多讓人愉悅的東西。

條款 32：在未來時態下發展程式

世事永遠在變。

身爲軟體開發人員，我們可能不是知道的很多，但我們確切知道世事永遠在變。我們不一定知道改變的是什麼，改變如何到來，改變何時發生，或爲什麼會發生，但我們真的知道：事情會改變。

好的軟體對於變化有良好的適應能力。好的軟體可以容納新的性質，可以移植到新的平台，可以適應新的需求，可以掌握新的輸入。軟體具備如此的彈性、穩健、可信賴度，並非是天上掉下來的禮物，而是那些「即使面對今天的束縛，仍然對明天可能的需求念茲在茲」的設計者和實作者共同努力的結果。把目光擺在未來

時態的程式員，才寫得出這種軟體 — 可優雅接受變化的軟體。

所謂在未來時態下設計程式，就是接受「事情總會改變」的事實，並準備應因之道。也許程式庫會加入新的函式，導至新的多載化（overloadings）發生，於是導至潛伏的模稜兩可函式發作（見條款 E26）。也許繼承體系會加入新的 classes，致使今天的 derived classes 成爲明天的 base classes。也許新的應用軟體會出現，函式會在新的環境下被呼叫，而我們必須考慮那種情況下仍能正確執行任務。記住，程式的維護者通常都不是當初的開發者，所以設計和實作時應該注意到如何幫助其他人理解、修改、強化你的程式。

要做到這件事情，辦法之一就是以 C++「本身」（而非只是註解或說明文件）來表現各種規範。舉個例子，如果某個 class 在設計時絕不打算成爲 derived classes，那麼就不應該只是在表頭檔的 class 上端擺一行註解就好，而是應該以 C++ 語法來阻止衍化的發生：條款 26 告訴你怎麼做。如果一個 class 要求其所有物件實體都必須於 heap 內產生，那麼請不要只是告訴 clients 那麼做，應該以條款 27 厲行這項約束。如果 copying 和 assignment 對某個 class 沒有意義，我們應該將其 copy constructor 和 assignment operator 宣告爲 private（見條款 E27）。C++ 提供了很大的威力、彈性、表現力。使用這些語言特徵來厲行你的設計吧。

既然知道事情總會改變，那麼就請在這演化速度有如浪滔沙的軟體世界中寫一些抗變性比較高的 classes。是的，請避免 "demand-paged" 式的虛擬函式，那會使你習慣於「不讓任何函式成爲 virtual，除非有人需要」。你應該決定函式的意義，並決定它是否適合在 derived classes 內被重新定義。如果是，就把它宣告爲 virtual，即使眼前並沒有人重新定義之。如果不是，就把它宣告爲 nonvirtual，並且不要只爲了圖某人的方便就改變其定義。請確定你所做的改變對於整個 class 的上下關係乃至於它所表現的抽象性是合理的（見條款 E36）。

請爲每一個 class 處理 assignment 和 copy construction 動作，即使沒有人使用那樣的動作。現在沒有人使用，並不意味將來都沒有人使用（見條款 E18）。如果這些函式不易完成，請將它們宣告爲 `private`（見條款 E27），那就不會有任何人不經意喚起「編譯器自動產生，行爲卻錯誤」的版本（這常常發生於 default assignment operators 和 copy constructors 身上 — 見條款 E11）。

請不要做出令人大吃一驚的怪異行爲：請努力讓 classes 的運算子和函式擁有自然的語法和直觀的語意。請和內建型別的行爲保持一致：如果有疑惑，不妨看看 `ints` 有怎樣的表現。

記住，任何事情只要有人能夠做，就會有人做。他們會丟出 exceptions、他們會「將物件自己派給自己」、他們會在尚未獲得初值前就使用物件、他們會給物件初值卻從不使用它、他們會給物件過大的值、他們會給物件過小的值、他們會給物件 null 值。通常，只要編譯沒問題，就會有人做。所以，請讓你的 classes 容易被正確地使用，不容易被誤用。請接受「客戶會犯錯」的事實，並設計你的 classes 有預防、偵測、或甚至更正的能力（見條款 33 和條款 E46）。

請努力寫出可移植的碼。這並不會比寫出不可移植的碼更困難，只有某些罕見情況會顯著影響效率，使我們不得不調整爲不具移植性的架構（見條款 16）。即使程式被設計用於訂製型硬體，通常也可能有移植的需求，因爲硬體庫存量往往數年內便會到達不動如山的窘境。撰寫具移植性的碼，你便能夠輕易轉換平台，放大客戶群，並誇耀說自己支援開放系統。如果你對作業系統押錯寶，具移植性的碼也可以使你不至於全盤皆輸。

請設計你的碼，使「系統改變所帶來的衝擊」得以區域化。儘可能採用封裝特性質、儘可能讓實作細目成爲 private（見條款 E20）。如果可用，就儘量用無具名的 namespaces 或檔案內的 `static` 物件和 `static` 函式（見條款 31）。儘量避免設計出 virtual base classes，因爲這種 classes 必須被其每一個 derived class（即使是間接衍生者）初始化（見條款 4 和條款 E43）。請避免以 RTTI 做爲設計基礎並因而導至一層一層的 `if-then-else` 述句（見條款 31；條款 E39 對此有良好措施）：因爲每當 class 繼承體系一有改變，每一組這樣的述句都得更新，如果你忘了其中一個，編譯器不會給你任何警告。

這些都是廣爲人知且常被提起的忠告，但大部份程式員還是掉進「現在式」的泥淖中。不幸的是許多書籍作者也缺乏高瞻遠囑。看看這位 C++ 專家提出的忠告：

> 只要有人刪除 `B*` 而它實際上指向 `D`，便表示你需要一個 virtual destructor。

這裡的 `B` 是指 base class，`D` 是指 derived class。換句話說這位作者建議如果你的程式看起來像這樣，你的 `B` 就不需要一個 virtual destructor 囉：

```
class B { ... };          // 不需要 virtual destructor
class D: public B { ... };
B *pb = new D;
```

然而如果你加上這一行，情況就改變了：

```
delete pb;                // 現在，你的 B 需要一個 virtual destructor
```

client 端的微小改變 — 增加一個 `delete` 述句而已 — 竟導至需要改變 B 的定義，更進而導至 B 的所有客戶都必須重新編譯。如果恪遵這位作者的忠告，那麼單單增加一行敘句便導至程式庫的所有客戶大規模地重新編譯和聯結。這絕非高效率的軟體設計。

針對同一議題，另一位作者寫道：

> 如果一個 public base class 沒有 virtual destructor，那麼其 derived class 以及「該 derived class 的 data members」都不應該有 destructor。

換言之，下面是好的：

```
class string {                // 來自 C++ 標準程式庫
public:
  ~string();
};
class B { ... };              // destructor 內沒有 data members。
                              // 沒有 virtual destructor。
```

但如果有個新的 class 繼承自 B，事情便有了變化：

```
class D: public B {
  string name;                // 現在 ~B 必須是 virtual
};
```

再一次，B 用途上的小小改變（此處是增加一個 derived class，其中內含一個「帶有 destructor」的成員）可能造成 client 端大規模的重新編譯和聯結。但是軟體的小改變應該只造成小小的衝擊才是。所以這樣的設計並不好。

同一位作者又寫道：

> 如果多重繼承（multiple inheritance）體系中有任何 destructors，那麼每一個 base class 都應該有一個 virtual destructor。

請注意我引用的這些例子中的「現在式思維」：使用者此刻如何運用指標？哪些 class members 此刻擁有 destructors？繼承體系中的哪些 classes 此刻擁有 destructors？

未來式思維就不一樣。我們不再問自己，class 此刻如何被使用，我們問的是這個 class 被設計做爲什麼用途。未來式思維說，如果 class 被設計用來做爲一個 base class（即使它目前不是），它就應該擁有一個 virtual destructor（見條款 E14）。如此的 classes 才會在現在和未來都有正確的行爲，而且不會在誕生新的 derived classes 時，影響程式庫的其他使用者（至少在 destructor 這個主題上它們不受影響。如果 class 有必要做其他改變，使用者就有可能受到影響）。

我手上有一個市售的類別程式庫（發表於 C++ 標準程式庫的 `string` 規格公開之前），內含一個沒有 virtual destructor 的 `String` class。廠商的解釋是：

> 我們不讓 destructor 成爲 virtual，因爲我們不希望 `String` 有個 vtbl。我們並不意圖擁有 `String*`，所以不會造成問題。我們清楚知道這可能造成的困難。

這是現在式思維？還是未來式思維？

當然，vtbl 是正當的技術性考量（見條款 24 和條款 E14）。大部份 `String` classes 的設計都只在每一個 `String` 物件內放置一個孤零零的 `char*` 指標，所以在每個 `String` 物件身上增加一個 vptr，會使其大小倍增。這很容易讓人瞭解爲什麼廠商沒有意願那麼做，特別是身爲一個被高度運用的 `String` class — 此等 class 的效率很可能輕易掉進程式那 20% 的部份（見條款 16），帶來巨大的影響。

但是，一個字串物件所消耗的記憶體總量 — 包括物件本身所需以及用來放置字串實值之 heap 記憶體 — 往往遠大於一個 `char*` 指標所需。由此觀之，一個 vptr 所帶來的額外負擔其實是微不足道的。儘管如此，這畢竟是個正當的技術考量。（當然 ISO/ANSI 標準委員會可能是這麼想：標準 `string` 型別已經有一個 nonvirtual destructor 了）

令人困惑的是廠商的註解：「我們並不意圖擁有 `String*`，所以這不會造成問題」。那可能是真的，但 `String` class 是程式庫的一部份，而程式庫可能給成千上萬的開發人員使用。「成千上萬」是很可怕的數字，其中每一個人對 C++ 有著不同程度的經驗，每一個人做不同的事情。是否那些開發人員都瞭解到 `String` 並沒有 virtual destructor 呢？是否他們都知道，由於 `String` 沒有 virtual destructor，當他們從 `String` 衍生出新的 classes，是一種高度冒險的行爲？是否廠商自信其客戶都能夠瞭解，由於缺乏 virtual destructor，「透過 `String*` 指標刪除一個物件」

將無法正確運作，而作用於 Strings 指標和 String reference 身上的 RTTI 運算子也可能傳回不正確的資訊？是否這個 class 很容易被正確地運用，很不容易被錯誤地運用？

廠商應該爲其 String class 提供說明文件，使大家更清楚知道這個 class 並非設計做爲衍化用途。但如果程式員漏看這項警告，或是根本就沒有閱讀這份文件呢？

最好的辦法就是以 C++ 本身性質來禁止衍化的發生。條款 26 對此有些描述：限制物件必須產生於 heap 之中，然後以 auto_ptr 物件來處置 heap 物件。這使得 String 的「生成介面」既不傳統也不便利，必須這樣才能產出一個物件來：

```
auto_ptr<String> ps(String::makeString("Future tense C++"));
...         // 將 ps 視爲一個指標，指向一個 String 物件，
            // 但不必操心其刪除事宜。
```

而不再是如此：

```
String s("Future tense C++");
```

或許，將「行爲不適當之 derived classes」所造成的風險降低，此利益值得我們忍受上述語法的不方便。（對 String 而言情況並非如此，不過對其他 classes 而言，這項交易還滿划算的）

「現在式思維」當然有其必要，畢竟你所開發的軟體必須和目前的編譯器合作；你無法等待最後一個語言特性實作出來。你的軟體必須在目前的硬體上執行，必須在客戶的環境下執行；你不能夠強迫你的客戶升級他們的系統，或是修改他們的作業環境。你的軟體現在就必須提供可接受的效率；承諾數年之後給一個更小、更快的版本，通常不會對潛在客戶帶來溫暖。你手上開發的軟體必須「很快」上市，那通常意味在即將到來的某個時刻。這些都是重要的壓迫，你無法忽略它們。

未來式思維只不過是加上一些額外的考量：

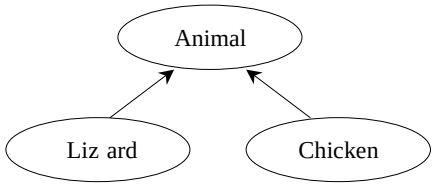
- 提供完整的 classes（見條款 E18）— 即使某些部份目前用不到。當新的需求進來，你比較不需要回頭去修改那些 classes。

- 設計你的介面，使有利於共同的操作行爲，阻止共同的錯誤（見條款 E46）。讓這些 classes 輕易地被正確運用，難以被錯誤運用。例如，面對那些「copying 和 assignment 並不合理」的 classes，請禁止那些動作的發生（見條款 E27）。請防止局部設值（partial assignments，見條款 33）的發生。
- 儘量使你的碼一般化（泛型化），除非有不良的巨大報應。舉個例子，如果你正在寫一個演算法，用於樹狀結構（tree）的來回巡訪，請考慮將它一般化，俾使能夠處理任何種類的 directed acyclic（非環狀的）graph。

「未來式思維」可增加你的程式碼的重用性、加強其可維護性、使它更穩健強固、並促使在一個「改變實乃必然」的環境中有著優雅的改變。它必須和「目前的規範」取得平衡。太多程式員專注於目前的需要，其他什麼都不管，因而犧牲了他們所設計並實作的軟體長期生存與發育的能力。做個不一樣的人，做個離經叛道者吧。請在未來時態下開發程式。

條款 33：將非葉端類別（non-leaf classes）設計爲抽象類別（abstract classes）

假設你正在進行一項專案，以軟體來處理動物。此軟體將大部份動物視爲十分類似，只有兩種動物需要特殊對待：蜥蜴和雞。在此情況下，很明顯我們可以將動物、蜥蜴、雞這三個 classes 組織如下：



`Animal` class 負責將你所處理的所有動物的共同特徵具體化，`Lizard` 和 `Chicken` classes 則分別將 `Animal` 特殊化爲蜥蜴和雞。

下面是這些 classes 定義式的一個梗概：

```
class Animal {
public:
  Animal& operator=(const Animal& rhs);
  ...
};
```

```
class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};

class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...
};
```

這裡只顯示出 assignment 運算子，不過那就夠我們忙上一陣子了。考慮這段碼：

```
Lizard liz1;
Lizard liz2;

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
...
*pAnimal1 = *pAnimal2;
```

其中有兩個問題。第一，最後一行喚起 `Animal` class 的 assignment 運算子 — 即使涉及的物件型別爲 `Lizard`。於是，只有「`liz1` 的 `Animal` 成份」會被修改。這便是所謂的局部指派（局部設值，partial assignments）。在此動作之後，`liz1` 的 `Animal` members 內容與 `liz2` 相同，但是 `Lizard` members 則保持不變。

第二個問題是，很多人真的這樣寫程式。是的，透過指標對物件進行指派（設值）動作，並不罕見，特別是有豐富 C 經驗而改用 C++ 的程式員。我們希望指派（assignment）動作的行爲更合理些。一如條款 32 指出，我們的 classes 應該輕易地被正確運用，難以被錯誤運用，而上述繼承體系內的 classes 卻很容易被錯用。

一個解決辦法就是讓 assignment 運算子成爲虛擬函式。如果 `Animal::operator=` 是虛擬函式，先前的指派行爲就會喚起 `Lizard` 的 assignment 運算子，那便是正確的呼叫。然而，如果我們真的將 assignment 運算子宣告爲 virtual，看看會發生什麼事：

```
class Animal {
public:
  virtual Animal& operator=(const Animal& rhs);
  ...
};
```

```
class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);
  ...
};

class Chicken: public Animal {
public:
  virtual Chicken& operator=(const Animal& rhs);
  ...
};
```

由於 C++ 語言後期的一個特性變化，我們得以訂製 assignment 運算子的傳回型別，使它們各傳回一個 reference，代表正確類別。但是此一規則強迫我們在每一個 class 中爲此虛擬函式宣告完全相同的參數型別。那就意味 `Lizard` 和 `Chicken` classes 的 assignment 運算子必須接受「任何種類之 `Animal` 物件出現在指派動作的右邊」。也就是說，我們必須面對一個事實 — 以下是合法的程式碼：

```
Lizard liz;
Chicken chick;

Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;
...
*pAnimal1 = *pAnimal2;              // 將一隻雞指派給一隻蜥蜴
```

這是一種異型指派：`Lizard` 在左邊而 `Chicken` 在右邊。異型指派在 C++ 中向來不會造成問題，因爲這個語言的強烈型別檢驗（strong typing）通常會將它們視爲不合法。然而如果讓 `Animal` 的 assignment 運算子成爲虛擬函式，我們便打開了「異型指派」的一扇門。

這使我們頓感爲難。我們希望允許「透過指標進行」的同型指派，但又希望禁止「透過同樣那些指標」而進行的異型指派。換言之，我們希望允許這麼做：

```
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
...
*pAnimal1 = *pAnimal2;              // 將一隻蜥蜴指派給另一隻蜥蜴
```

但禁止這麼做：

```
Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;
...
*pAnimal1 = *pAnimal2;              // 將一隻雞指派給一隻蜥蜴
```

要區分這些情況，恐怕得執行時期才有辦法，因爲將 `*pAnimal2` 指派給 `*pAnimal1`，有時候有效，有時候無效。我們因而進入了「因型別而造成的執行時期錯誤」的黑暗世界裡。明確地說，如果我們面對的是個異型指派，就應該在 `operator=` 內發出一個錯誤訊息。如果面對的是同型指派，我們就希望以正常方式執行指派動作。

`dynamic_cast`（見條款 2）可以協助我們實現上述願望。下面是 `Lizard` assignment 運算子的作法：

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  // 確定 rhs 真的是一隻蜥蜴
  const Lizard& rhs_liz = dynamic_cast<const Lizard&>(rhs);

  proceed with a normal assignment of rhs_liz to *this;
}
```

這個函式只有在 `rhs` 真的是一隻蜥蜴時，才將 `rhs` 指派給 `*this`。如果它不是，函式便向外傳播「因 `dynamic_cast` 轉型失敗而發出的 `bad_cast` exception」。（事實上該 exception 的型別是 `std::bad_cast`，因爲標準程式庫的各個組件以及被標準組件丟出的各種 exceptions 都位於 namespace `std` 內。關於標準程式庫，請見條款 E49 和條款 35）

即使不擔心 exceptions，這個函式在平常情況下（將一個 `Lizard` 物件指派給另一個 `Lizard` 物件）似乎沒有必要如此複雜和昂貴，因爲如果動用到 `dynamic_cast`，那就需要諮詢一個 `type_info` 結構（見條款 24）：

```
Lizard liz1, liz2;
...
liz1 = liz2;       // 不需執行 dynamic_cast，因爲這個指派動作一定有效
```

不需勞駕 `dynamic_cast` 的複雜度和成本，我們依然可以處理這種情況，作法是爲 `Lizard` 加上傳統的 assignment 運算子（譯註：形成兩個多載函式）：

```
class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);

  Lizard& operator=(const Lizard& rhs);          // 加上這行
  ...
};

Lizard liz1, liz2;
...
liz1 = liz2;              //呼叫「接受一個 const Lizard&」的 operator=

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
...
*pAnimal1 = *pAnimal2;    //呼叫「接受一個 const Animal&」的 operator=
```

事實上，有了第二個 operator= 之後，第一個 operator= 的實作亦得簡化爲：

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  return operator=(dynamic_cast<const Lizard&>(rhs));
}
```

這一次我們企圖將 rhs 轉型爲一個 Lizard。如果轉型成功，就喚起正常的 assignment 運算子。否則就會丟出一個 bad_cast exception。

坦白說，執行時期的所有型別檢驗動作，以及對 dynamic_casts 的各種運用，都讓我感到焦慮。就拿一件事來說，某些編譯器仍未支援 dynamic_cast，所以程式碼凡使用它者，雖然理論上具有移植性，實際上卻非如此。更重要的是，它竟然要求 Lizard 和 Chicken 的使用者每次執行一個指派動作時都待命捕捉 bad_cast exceptions，並做某些合理應對。在我的經驗中，許多程式員不喜歡把程式寫成這樣子。然而如果他們不這麼寫，就無法獲得當初試圖防堵「局部指派（partial assignments）」而發展下來的這種種利益。

既然 virtual assignment operators 有著十分難以滿足的情況，重新出發似乎是合理的。讓我們嘗試找出一個辦法，阻止 clients 一開始就做出有問題的指派（設值）動作。如果這樣的動作在編譯期就被拒絕，我們就不必擔心它們會做出什麼蠢事兒了。

阻止此等指派動作的最簡單辦法就是，讓 operator= 成爲 Animal 的 private 函式。如果這樣，蜥蜴就可以被指派給蜥蜴，而雞仔可以被指派給雞仔，局部指派和異型指派也得以禁止：

```
class Animal {
private:
  Animal& operator=(const Animal& rhs);    // 此函式如今成爲 private
  ...
};

class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};

class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...
};

Lizard liz1, liz2;
...
liz1 = liz2;                          // 很好

Chicken chick1, chick2;
...
chick1 = chick2;                      // 也很好

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &chick1;
...
*pAnimal1 = *pAnimal2;                // 錯誤！企圖呼叫
                                      // private Animal::operator=
```

不幸的是，Animal 是一個具象類別（concrete class），而上述方法卻使得 Animal 物件彼此間的指派動作也不合法：

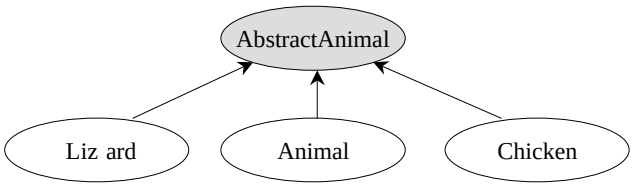
```
Animal animal1, animal2;
...
animal1 = animal2;                    // 錯誤！企圖呼叫
                                      // private Animal::operator=
```

此外，它也造成我們無法正確實作出 Lizard 和 Chicken 的 assignment 運算子，因爲 derived classes 的 assignment 運算子有義務呼叫 base classes 的 assignment 運算子（見條款 E16）：

```
Lizard& Lizard::operator=(const Lizard& rhs)
{
  if (this == &rhs) return *this;

  Animal::operator=(rhs);  // 錯誤！企圖呼叫 private 函式。但
                           // Lizard::operator= 確實必須呼叫該函式，
                           // 才能將「Animal 成份」指派給 *this。
  ...
}
```

將 Animal::operator= 宣告爲 protected，可以解決後一個問題，但是前一個難題仍然存在：阻止 Lizard 和 Chicken 物件之局部指派（藉由 Animal 指標）的同時，我們必須允許 Animal 物件相互指派。可憐的程式員怎麼做才好？

最簡單的辦法就是消除「允許 Animal 物件相互指派」的需要，而完成此事的最簡單作法就是讓 Animal 成爲一個抽象類別。身爲抽象類別，Animal 就無法被實體化，也就沒有必要「允許 Animal 物件彼此指派」了。當然啦，這會導至新的問題，因爲這個系統最初設計時曾假設 Animal 物件是必要的。一個很簡單的方法可以解決這個難點。不要讓 Animal 成爲抽象，改以一個新的 class — 比方說 AbstractAnimal（由 Animal, Lizard 和 Chicken 物件的共同特徵組成）— 成爲抽象類別。然後再令原先的每個具象類別繼承自 AbstractAnimal。修改後的繼承體系如下：



於是各個 classes 定義如下：

```
class AbstractAnimal {
protected:
  AbstractAnimal& operator=(const AbstractAnimal& rhs);

public:
  virtual ~AbstractAnimal() = 0;                    // 稍後說明
  ...
};
```

```
class Animal: public AbstractAnimal {
public:
  Animal& operator=(const Animal& rhs);
  ...
};
class Lizard: public AbstractAnimal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};
class Chicken: public AbstractAnimal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...
};
```

這個設計提供你所需要的每一樣東西。蜥蜴、雞、動物之間允許同型指派；局部指派和異型指派都在禁止之列；derived class 的 assignment 運算子可以呼叫 base class 的 assignment 運算子。猶有進者，沒有任何一行「涉及 `Animal`, `Lizard` 或 `Chicken` classes」的程式碼需要修改，因爲這些 classes 都依然存在而且行爲依舊（如同 `AbstractAnimal` 尚未加入之前）。當然啦，程式碼必須重新編譯，但那只是小小的代價。

爲了讓一切都能有效運作，`AbstractAnimal` 必須是抽象類別 — 它必須內含至少一個純虛擬函式。大部份時候，選出一個這樣的函式不成問題，但是偶爾你可能會發現你必須產生一個像 `AbstractAnimal` 這樣的 class，其中沒有任何 member functions 可以很自然地被你宣告爲純虛擬函式。這種情況下，傳統作法是讓 destructor 成爲純虛擬函式；如同以上所示。爲了正確支援透過指標而形成的多型（polymorphism）特性，base classes 無論如何需要一個 virtual destructors（見條款 E14），所以讓如此的 destructor 成爲純虛擬函式，唯一的成本就是必須在 class 定義式之外實作其內容，這有時候不太方便。（想看個例子嗎？p.195 有）

如果「純虛擬函式竟然需要實作碼」的觀念令你震驚，鎮定些！將函式宣告爲純虛擬，並非暗示它沒有實作碼，而是意味：

- 目前這個 class 是抽象的。
- 任何繼承此 class 之具象類別，都必須將該純虛擬函式重新宣告爲一個正常的虛擬函式（也就是說，不可以再令它 `"=0"`）。

的確，大部份純虛擬函式並沒有實作碼，但是 pure virtual destructors 是個例外。它們必須被實作出來，因爲只要有一個 derived class destructor 被喚起，它們便會被呼叫。此外，它們通常執行一些有用的工作，像是釋放資源（見條款 9）或記錄運轉訊息等等。純虛擬函式之實作或許並不常見，但對 pure virtual destructors 而言，實作之不僅是平常的事，甚且是必要的事。

你可能已經注意到了，上述對於「透過基礎類別之指標，進行指派動作」的討論，是以「具象基礎類別（如 `Animal`）含有 data members」的假設做爲基礎。如果具象基礎類別之中沒有任何 data members，你可能會說，那就沒問題了，令一個具象類別繼承另一個不含資料的具象類別，是安全的。

有兩種情況可能發生在你那「不含資料」的具象基礎類別身上：未來它可能擁有 data members，或者它可能一直都沒有 data member。如果是前者，你所做的一切便都只是逃避問題，直到 data members 加入才面對。你只不過是以短暫的方便暫替長期的苦惱（見條款 32）而已。至於第二種情況，如果你的基礎類別真的不該擁有任何 data members，那它豈不一開始就應該是個抽象類別嗎？一個具象基礎類別卻沒有資料，用途在哪裡呢？

將一個具象基礎類別如 `Animal` 者，以一個抽象基礎類別如 `AbstractAnimal` 者取代，好處不只在於讓 `operator=` 的行爲更容易被瞭解，也降低了「企圖以多型方式對待陣列」的機會 — 後者帶來的不愉快後果條款 3 曾有說明。這個技術最具意義的利益乃在於設計層面，因爲將具象基礎類別以抽象基礎類別取而代之，可強迫你明白認知有用之抽象性質的存在。也就是說，它讓你針對有用的觀念產生新的抽象類別，即使你並不知道存在著那些有用的觀念。

如果你有兩個具象類別 `C1` 和 `C2`，而你希望 `C2` 以 public 方式繼承 C1。你應該將原本的雙類別繼承體系改爲三類別繼承體系：產生一個新的抽象類別 `A`，並令 `C1` 和 `C2` 都以 public 方式繼承 `A`：

你最初的想法　　改變後的繼承體系

這種轉變的主要價值在於，它強迫你驗明抽象類別 A。很顯然，C1 和 C2 有某些共同的東西；這正是爲什麼它們要以 public inheritance（見條款 E35）形成彼此關係的緣故。如果採用上述轉變，你就必須鑑定出所謂「某些共同的東西」是什麼。此外，你必須將那些共同的東西形式化爲一個 C++ class，使它比一個模糊的概念更具體化些，進而成爲一個正式而條理分明的抽象性質，有著定義完好的 member functions，和定義完好的語意。

這些說法導至某種令人憂慮的思想。畢竟，每一個 class 都是用來表現某種抽象性，是否我們應該在繼承體系中爲每一個概念產生兩個 classes，一個是抽象的（用以具體化抽象性中的抽象成份），另一個是具象的（用以具體化抽象性中的物件生成部份）？不。如果你這麼做，你的繼承體系會有太多 classes。這樣的繼承體系難以瞭解、難以維護、編譯起來也費時。那不是物件導向設計的目標。

物件導向設計的目標是辨識出一些有用的抽象性，並強迫它們 — 也只有它們 — 成爲抽象類別。但如何辨識出有用的抽象性呢？誰知道什麼樣的抽象性可能在未來被證明是有用的？誰能夠預言將來誰會繼承什麼東西呢？

哦，我不知道如何預言一個繼承體系的未來用途或用法，但是我確知一件事情：某個環境下如果需要某種抽象性質，可能只是一種巧合，但如果多個環境下都需要某種抽象性質，那便通常饒富意義。因此，所謂「有用的抽象性」，就是在眾多環境下都被需要的那種。也就是說它們與以下性質的 classes 相符：其本身有用（亦即，爲該型別產生一些物件是有用的）、對一個或多個 derived classes 也有用。

所以，爲什麼要將「具象基礎類別」轉變爲「抽象基礎類別」，原因很明白了：只有在原有之具象類別被當做基礎類別使用（亦即當該類別被重用（reused）時），才強迫導入一個新的抽象類別。這樣的抽象性是有用的，因爲透過先前的闡述，它們證明了自己當之無愧。

當某個概念初次靈光乍現的時候，我們無法判斷是否應該爲它同時產生一個「針對概念而做」的抽象類別，和一個針對「該概念之相應物件」而做的具象類別。但是當這個概念第二次被需要的時候，我們就可以認爲，爲它同時產生抽象類別和具象類別是正當的。先前我所描述的繼承體系的轉變只是將這樣的過程機械化，因而強迫設計者和實作者明白表現出有用的抽象性 — 即使他們並不清楚什麼是

有用的概念。這也碰巧使我們得以輕鬆完成 assignment 運算子的穩健行爲。

簡單考慮一個例子。假設你正在開發一個應用軟體，該軟體處理網路上電腦之間的資訊搬移，作法是將資訊打破成爲一個個封包（packets），並以某種通訊協定來傳輸它們。這裡我們只考慮用來表現封包的所有 classes。我們假設這樣的 classes 對此應用軟體而言是有意義的。

假設你只處理單一種類的傳輸協定，以及單一種類的封包。或許你聽過其他通訊協定以及封包形式，但你不想支援它們，也沒有計劃在將來支援它們。你應該爲「封包」設計一個抽象類別（用以表現「封包」所呈現的概念），並再爲你真正使用的那種封包設計一個具象類別嗎？如果這麼做，你一定是希望將來增加新的封包形式時，不需改變封包的基礎類別。這可以使你在增加新的封包形式後，不必將封包的各個應用程式重新編譯一遍。但是此種設計需要兩個 classes，而你此刻真正只需要一個 class（用於你所使用的封包形式）。將設計搞得更複雜，爲的是允許未來說不定不會發生的擴充性，是否值得 ？

這裡無法明白告訴你正確的選擇是什麼，但是經驗顯示，我們不太可能爲自己並不十分瞭解的概念設計出好的 classes。如果你爲「封包」產生一個抽象類別，你要如何正確地設計它？尤其是當你的經驗只侷限於單一封包形式的時候？記住，只有當你有能力設計某種 class，使未來的 classes 可以繼承自它而它不需要任何改變，你才能夠從一個「封包抽象類別」中獲得利益。（如果它需要改變，你必須重新編譯封包的所有應用程式，那麼你就什麼好處都沒撈到）

不太可能設計得出一個令人滿意的「抽象」封包類別，除非你對於多種不同的封包格式造詣深厚，並且知道在不同的環境下如何使用它們。面對你那有限而薄弱的經驗，我的忠告是不需要爲封包定義一個抽象類別。日後當你發現有「從具象封包類別繼承下來」的需要時，才補上一個抽象類別就好（譯註：就像先前的 `Animal` 和 `AbstractAnimal` 那樣）。

這裡我所描述的類別變換，是鑑定「抽象類別是否必要」的一種方法，不是唯一方法。還有許多其他方法可以鑑定抽象類別的適當候選人；物件導向分析的相關書籍對此談了很多。我並沒有說「當你發現自己有需要讓一個具象類別繼承自另一個具象類別」便是導入抽象類別的唯一時機，然而一旦有需要將兩個具象類別

以 public inheritance 的方式產生關聯，通常的確就表示你需要一個新的抽象類別了。

不過，常常，不美好的現實會迫使平靜的理論思考激起陣陣波濤。協力廠商開發的各種 C++ 類別程式庫，數量持續激增，如果你發現你需要產生一個具象類別，繼承自程式庫中的一個具象類別，而你只能使用該程式庫，不能修改，怎麼辦？

你無法修改程式庫以安插一個新的抽象類別，所以你的選擇不但有限，而且也不吸引人：

- 將你的具象類別衍生自既存的（程式庫中的）具象類別，但需注意本條款一開始所驗證的 assignment 相關問題，並且小心條款 3 所描述的陣列相關陷阱。
- 試著在程式庫的繼承體系中找一個更高層的抽象類別，其中有你需要的大部份功能，然後繼承它。當然，這可能不是一個合適的類別，就算是，你也可能必須重複許多努力，這些努力其實已經存在於你希望爲之擴張機能的那個具象類別的實作碼身上。
- 以「你所希望繼承的那個程式庫類別」來實作你自己的新類別（見條款 E40 和 E42）。例如，你可以令隸屬於程式庫類別的一個物件成爲你的 data member，然後在你的新類別中重新實作該程式庫類別的介面：

```
class Window {                          // 這一個是程式庫內的類別
public:
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;

  int width() const;
  int height() const;
};

class SpecialWindow {                   // 這一個是你希望繼承自
public:                                 // Window 的類別
  ...
  // 放過那些非虛擬函式
  int width() const { return w.width(); }
  int height() const { return w.height(); }

  // 重新實作那些繼承而來的虛擬函式
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;

private:
  Window w;
};
```

如果採用這個策略，每當程式庫廠商修改你所依賴的類別時，你就必須也修改你自己的類別。此策略也要求你必須放棄重新定義「宣告於程式庫類別中的虛擬函式」的權力，因爲除非你繼承它們，否則不能夠重新定義虛擬函式。

- 做個乖寶寶，手上有什麼就用什麼。修改你的軟體，俾使程式庫中的那些具象類別夠用。寫一些 non-member functions 以供應你希望（但是沒辦法）加入 class 內部去的機能。這樣做出來的軟體可能不夠乾淨清爽，也不好維護，效率不夠好，擴充性不佳，但至少你可以如期交貨。

這些選擇沒有一項吸引人，所以你必須加上某些工程意見，並選擇其中最適合你的作法。這並不有趣，但是生活有時候就是這樣。爲了讓你自己（以及其他人）將來更輕鬆，不妨給程式庫廠商一些建議（與抱怨）。如果夠幸運（而且來自客戶的相同意見夠多），那些設計便有可能在某個時刻有所改善。

老樣子，一般性的法則是：繼承體系中的 non-leaf（非尾端）類別應該是抽象類別。如果使用外界供應的程式庫，你或許可以對此法則做點變通，但如果程式碼完全在你掌控之下，堅持這個法則，可以爲你帶來許多好處，並提昇整個軟體的可靠度、穩健度、精巧度、擴充度。

條款 34：如何在同一個程式中結合 C++ 和 C

以 C++ 組件搭配 C 組件一起發展程式，很類似以多個 C 編譯器所產生的目的檔（object files）組合出整個 C 程式；在許多方面，憂慮的事情是一樣的。其實沒有什麼辦法可以組合這些檔案，除非不同的編譯器在那些「與編譯器相依的特性」上（例如 `ints` 和 `doubles` 的大小、參數由呼叫端至被呼叫端的傳遞機制、誰（呼叫端或被呼叫端）負責傳遞動作…等等）取得一致。「軟體開發過程中混合運用不同的編譯器」這個主題，在語言標準化過程中完全被忽略，所以唯一可知道「來自編譯器 A 的目的檔」和「來自編譯器 B 的目的檔」是否可以安全結合於同一個程式的辦法，就是詢問廠商 A 和廠商 B 關於其產出碼的相容性。C 和 C++ 混合使用，形勢就像上述一樣，所以在你嘗試這麼做之前，請確定你的 C++ 和 C 編譯器產生相容的目的檔。

確定這個大前題之後，另有四件事情你需要考慮：name mangling（名稱重整）、statics

（靜態物件）初始化、動態記憶體配置、資料結構的相容性。

Name Mangling（名稱重整）

Name mangling，一如你所可能知道的，是一種程序；透過它，你的 C++ 編譯器爲程式內的每一個函式編出獨一無二的名稱。在 C 語言中，此程序並無必要，因爲你不能夠將函式名稱多載化（overload）；但是幾乎所有的 C++ 程式都至少有一些函式擁有相同的名稱。例如，考慮 iostream 程式庫，其中宣告有數個版本的 `operator<<` 和 `operator>>`。多載化（overloading）並不相容於大部份聯結器，因爲聯結器往往將多個同名函式視爲不正常。Name mangling 是對聯結器的一個退讓；更明確地說是對「聯結器往往堅持所有函式名稱必須獨一無二」的一個讓步。

只要你一直在 C++ 封閉環境中，name mangling 就不應該是你關心的焦點。如果你有一個函式名爲 `drawLine`，被某個編譯器重整爲 `xyzzy`，你還是可以（並應該）使用 `drawLine` 這個名稱，沒有理由在乎底層目的檔使用的名稱是 `xyzzy`。

但如果 `drawLine` 處於 C 函式庫中，故事就不一樣了。這種情況下你的 C++ 原始檔或許會含入一個表頭檔，內含宣告如下：

```
void drawLine(int x1, int y1, int x2, int y2);
```

而你對 `drawLine` 的呼叫動作以一般形式出現，每一個如此的呼叫都被編譯器翻譯爲一個重整後的函式名稱。所以當你這麼寫：

```
drawLine(a, b, c, d);          // 呼叫未經重整的函式名稱
```

你的目的檔內含的是一個像這樣的函式呼叫碼：

```
xyzzy(a, b, c, d);             // 呼叫重整後的函式名稱
```

但如果 `drawLine` 是個 C 函式，那麼「`drawLine` 編譯碼」所在的那個目的檔（或動態聯結函式庫）內將會有一個名爲 `drawLine` 的函式，其名稱並未重整。當你企圖將那個目的檔聯結進來，你會獲得一個錯誤訊息，因爲聯結器企圖尋找名爲 `xyzzy` 的函式，而那並不存在。

爲解決這個問題，你需要某種方法告訴你的 C++ 編譯器，叫它不要重整某些函式名稱。是的，絕對不要重整以其他語言撰寫的函式的名稱 — 不論是以 C, assembler, FORTRAN, Lisp, Forth, COBOL, 或任何其他語言。畢竟，如果你呼叫一個名爲 `drawLine` 的 C 函式，它的真正名稱就是叫做 `drawLine`；你的目的碼（object

code）應該內含一份參考，指向那個名稱，而非一個經過重整的名稱。

要壓抑 name mangling，必須使用 C++ 的 `extern "C"` 指令：

```
// 宣告一個函式，名爲 drawLine；不要重整其名稱。
extern "C"
void drawLine(int x1, int y1, int x2, int y2);
```

不要掉進陷阱之中，以爲既然有 `extern "C"`，必然也會有 `extern "Pascal"` 和 `extern "FORTRAN"`。不，那不是真的，至少在 C++ 標準規格中不是。對待 `extern "C"` 的最佳態度並非是將它視爲一種主張，主張相關函式以 C 寫成；而是視之爲一個敘述，說那個函式應該以 C 語言的方式來呼叫。（技術上來說，`extern "C"` 意味這個函式有 C linkage，但這又是什麼意思呢？不過，至少它總是意味了一個意思，那就是 name mangling 會被壓抑不發）

舉個例子，如果你非常不幸地必須以 assembler 寫一個函式，你也可以將它宣告爲 `extern "C"`：

```
// 此函式以 assembler 完成，所以不要重整其名稱。
extern "C" void twiddleBits(unsigned char bits);
```

你甚至可以將 C++ 函式宣告爲 `extern "C"`。如果你正以 C++ 語言開發一個程式庫，而你希望供應給其他語言的客戶使用，那麼這個技巧也許有用。只要壓抑你的 C++ 函式名稱的 name mangling 程序，你的客戶就可以使用你所選擇的那個自然而直觀的名稱，而非經過編譯器重整後的名稱：

```
// 以下 C++ 函式被設計用於 C++ 語言之外，所以其名稱不應該被重整。
extern "C" void simulate(int iterations);
```

通常你會有一麻袋函式，其名稱不需要重整，如果每個函式之前都得加上 `extern "C"`，頗令人痛苦。幸運的是不必如此。`extern "C"` 可以施行於一整組函式身上，只要以大括號封住頭尾範圍即可：

```
extern "C" {    // 解除以下所有函式的 name mangling
  void drawLine(int x1, int y1, int x2, int y2);

  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...
}
```

`extern "C"` 的運用可以簡化「必須同時被 C 和 C++ 使用」的表頭檔維護工作。當這個檔案用於 C++ 時，你希望含有 `extern "C"`；用於 C 時，你不希望如此。由於前處理器（preprocessor）符號 `__cplusplus` 只針對 C++ 才有定義，所以這種「通用於數種語言」的表頭檔可以架構如下：

```
#ifdef __cplusplus
extern "C" {
#endif
  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...
#ifdef __cplusplus
}
#endif
```

順帶一提，沒有所謂的「name mangling 標準演算法」。不同的編譯器可自由地以不同的方法來重整名稱，而各家編譯器也的確各行其道。這是一件好事情。如果所有編譯器都以相同方法來重整名稱，你可能會誤以爲它們統統產生相容的碼。目前的情況是，如果你嘗試混用不相容的 C++ 編譯器所產生的目的碼（object code），聯結時就有機會因「被重整的名稱彼此不吻合」而獲得錯誤訊息。這暗示你或許還有其他的相容性問題，早點找出這種不相容問題，當然比晚找出的好。

Statics 的初始化

一旦掌握了 name mangling，接下來你需要面對一個事實：許多碼會在 `main` 之前和之後執行起來。更明確地說，static class 物件、全域物件、namespace 內的物件、以及檔案範圍（file scope）內的物件，其 constructors 總是在 `main` 之前就獲得執行。這個過程稱爲 static initialization（見條款 E47）。這對於我們一般以爲的 C++ 程式或 C 程式，是一種直接的反抗，因爲我們通常把 `main` 視爲程式的進入點。同樣道理，透過 static initialization 產生出來的物件，其 destructors 必須在所謂的 static destruction 過程中被呼叫。那個程序發生在 `main` 結束之後。

啊呀，`main` 被認爲是程式的起點，但卻有些物件必須建構於 `main` 之前，這真令人進退維谷。爲了解決這個問題，許多編譯器在 `main` 一開始處安插了一個函式

呼叫，呼叫一個由編譯器提供的特殊函式。正是這個特殊函式完成了 static initialization。同樣道理，編譯器往往在 main 的最尾端安插一個函式呼叫，呼叫另一個特殊函式，其中完成 static 物件的解構。經過編譯的 main，看起來像這樣：

```
int main(int argc, char *argv[])
{
  performStaticInitialization();    // 此行由編譯器加入

  the statements you put in main go here;

  performStaticDestruction();       // 此行由編譯器加入
}
```

不要太過嚴苛地看待它們。函式 performStaticInitialization 和 performStaticDestruction 通常有更隱秘的名稱，它們甚至是 inline 函式，這麼一來你就不會在你的目的檔（object files）中看到任何這類函式。重點是：如果一個 C++ 編譯器採用這種方法來建構及解構 static 物件，那麼除非程式中有 main，否則此等物件既不會被建構也不會被解構。由於此種 static initialization 和 static destruction 作法十分普遍，只要你負責撰寫某個 C++ 軟體的任何一部份，你都應該嘗試寫 main。

有時候，在 C 成份中撰寫 main 似乎比較合理 — 如果程式主要以 C 完成而 C++ 只是個支援程式庫的話。儘管如此，C++ 程式庫中內含 static 物件仍是極有可能的（如果目前不是這樣，未來也可能這樣）— 見條款 32），所以如果能夠，還是儘量在 C++ 中撰寫 main 的好。然而這並非意味你需要重寫你的 C 程式碼。只要將你的 C main 重新命名爲 realMain，然後讓 C++ main 呼叫 realMain 即可：

```
extern "C"                              // 以 C 語言
int realMain(int argc, char *argv[]);   // 完成此函式

int main(int argc, char *argv[])        // 以 C++ 語言完成此函式
{
  return realMain(argc, argv);
}
```

如果你這麼做，最好是在 main 的上方放一些註解，解釋爲什麼要這麼做。

如果你無法在專案中的 C++ 這一部份撰寫 main，你會遭遇一個問題，因爲沒有其他具移植性的辦法可以確保 static 物件的 constructors 和 destructors 會被呼

叫。這並非意味一切都輸掉了，只是說你必須更辛苦些。編譯器廠商都會被告知這個問題，所以幾乎每一家廠商都會提供某種語言以外的機制，用來啓動 static initialization 和 static destruction 程序。至於如何啓動，細節請看你的編譯器文件，或是逕洽製造商。

動態記憶體配置

動態記憶體配置的一般規則很簡單：程式的 C++ 部份使用 `new` 和 `delete`（見條款 8），程式的 C 部份則使用 `malloc`（及其變種）和 `free`。只要記憶體是以 `new` 配置而得，就以 `delete` 刪除之。只要記憶體是以 `malloc` 配置而得，就以 `free` 釋放之。對著一個「`new` 傳回的指標」呼叫 `free`，會導至未定義的行爲，對著一個「`malloc` 傳回的指標」呼叫 `delete`，情況也一樣。所以，唯一需要記憶的事情就是，嚴密地將你的 `new`s/`delete`s 與你的 `malloc`s/`free`s 分隔開來。

有時候說比做容易得多。考慮粗糙（但好用）的 `strdup` 函式，它雖然並非 C 或 C++ 標準的一份子，卻被廣泛使用：

```
char * strdup(const char *ps);        // 傳回一個 ps 所指字串的副本
```

如果要避免發生記憶體遺失問題，`strdup` 配置的記憶體必須由 `strdup` 的呼叫者負責釋放。但是這塊記憶體如何釋放呢？使用 `delete`？或是呼叫 `free`？如果你呼叫的 `strdup` 來自 C 函式庫，那麼應該是後者。如果它來自一個 C++ 程式庫，那麼應該是前者。因此，呼叫了 `strdup` 之後，你應該做的事情不只隨系統的不同而不同，也隨編譯器的不同而不同。爲了降低這種頭痛的移植問題，請儘量避免呼叫標準程式庫（見條款 E49 和條款 35）以外的函式、或是大部份電腦平台上尚未穩定的函式。

資料結構的相容性

在 C++ 程式和 C 程式之間傳遞資料，可能嗎？想要讓 C 函式瞭解 C++ 的特性，是不可能的，所以兩個語言之間的對話層次必須限制於 C 能夠接受的範圍。因此，沒有任何具移植性的作法，可以將物件或是「member functions 指標」傳給 C 函式。然而由於 C 確實瞭解一般指標，所以如果你的 C++ 和 C 編譯器有著相容的輸出，兩個語言的函式便可以安全地交換物件指標、non-member 函式指標，或是 static 函式指標。很自然地，structs 以及內建型別之變數（例如 `int`s, `char`s）

也可以安全跨越 C++/C 邊界。

由於 C++「掌管 `struct` 記憶體佈局」的規則，與 C 語言的相關規則一致，所以同一個 struct 定義式在兩種語言編譯器中被編譯出來後，應該有相同的佈局。如此的 structs 可安全地在 C++ 和 C 之間往返。如果你爲 C++ 版的 struct 加上一些非虛擬函式，其記憶體佈局應該不會改變。所以 struct（或 class）之中如果只含非虛擬函式，其物件應相容於 C structs（譯註：因爲 C++ member functions 並不在物件佈局中留下任何蛛絲馬跡）。如果加上虛擬函式，這場遊戲就玩不下去了，因爲在 class 內加入虛擬函式，會造成其物件採用不同的記憶體佈局（見條款 24）。令一個 struct 繼承另一個 struct（或 class）通常也會改變其佈局，所以一個 struct 如果帶有 base structs（或 classes），無法和 C 函式交換。

從資料結構的觀點來看，我們可以說：在 C 和 C++ 之間對資料結構做雙向交流，應該是安全的 — 前提是那些結構的定義式在 C 和 C++ 中都可編譯。爲 C++ struct 加上非虛擬函式，雖然不相容於 C，但可能不影響其相容性；其他任何改變則幾乎都會影響。

摘要

如果你打算在同一個程式中混用 C++ 和 C，請記住以下幾個簡單守則：

- 確定你的 C++ 和 C 編譯器產出相容的目的檔（object files）。
- 將雙方都使用的函式宣告爲 `extern "C"`。
- 如果可能，儘量在 C++ 中撰寫 `main`。
- 總是以 `delete` 刪除 `new` 傳回的記憶體；總是以 `free` 釋放 `malloc` 傳回的記憶體。
- 將兩個語言間的「資料結構傳遞」限制於 C 所能瞭解之形式；C++ structs 如果內含非虛擬函式，倒是不受此限。

條款 35：讓自己習慣使用標準的 C++ 語言

自從 1990 發行之後，*The Annotated C++ Reference Manual*（暱稱 *ARM*；見 p.285）就成爲那些有需要知道 C++ 裡頭有什麼、沒有什麼的實戰級程式員最後也最可靠的參考憑藉。然而在 *ARM* 出版後的這些年，ISO/ANSI 委員會對這個語言的標準化工作，在巨觀和微觀上都改變（主要是擴充）了這個語言。以「最後參考憑藉」的標準而言，*ARM* 不再適任。

後-*ARM* 時代對 C++ 的改變，巨幅影響了 C++ 優良程式的寫法。所以，對 C++ 程式員而言一件很重要的事情便是，趕快熟悉 C++ 標準規格與 *ARM* 之間的主要差異。

C++ ISO/ANSI 標準規格，是編譯器廠商的諮詢對象，也是 C++ 書籍作者寫作時的手邊依據，同時也是 C++ 程式員遇到爭議問題時的最後仲裁。在 *ARM* 出版後的這些年，C++ 最重要的幾項改變是：

- 增加了一些新的語言特性：RTTI、namespaces、`bool`、關鍵字 `mutable` 和 `explicit`、enums 做爲多載化函式之引數所引發的型別晉升轉換、以及「在 class 定義區內直接爲整數型（integral）`const` static class members 設定初值」的能力。
- 擴充了 **Templates** 的彈性：允許 member templates 存在、接納「明白指示 template 當場具現化」的標準語法、允許 function templates 接受「非型別引數（non-type arguments）」、可以 class templates 做爲其他 template 的引數。
- 強化了異常處理機制（**Exception handling**）：編譯期間更嚴密地檢驗 exception specifications（譯註：見條款 14）、允許 `unexpected` 函式丟出 `bad_exception` 物件（譯註：見 p.76）。
- 修改了記憶體配置常式：加入 `operator new[]` 和 `operator delete[]`、記憶體未能配置成功時由 `operators new/new[]` 丟出一個 exception、提供一個 `operators new/new[]` 新版本，在記憶體配置失敗時傳回 0（見條款 E7）。

- **增加了新的轉型形式**：`static_cast`, `dynamic_cast`, `const_cast` 和 `reinterpret_cast`。
- **語言規則更爲優雅精鍊**：重新定義虛擬函式時，其傳回型別不再一定得與原定義完全吻合。此外暫時物件的壽命也有了明白的規範。

幾乎所有這些改變都描述於 *The Design and Evolution of C++*（見 p.285）。目前各 C++ 教科書（寫於 1994 之後者）應該也都涵蓋了它們（如果你發現哪本書沒這麼做，丟了它）。此外，*More Effective C++*（本書）有一些例子，示範如何使用這些新特性中的大部份。如果你對其中某個特性感到好奇，試著從書後索引去尋找它們。

然而，C++ 語言結構的改變，與標準程式庫所經歷的天翻地覆相比，可說是小巫見大巫。標準程式庫的演化過程不像語言那樣有良好的公開，例如 *The Design and Evolution of C++* 就幾乎沒有提到標準程式庫。市場上討論標準程式庫的書籍，有些恐怕亦已過氣，因爲標準程式庫在 1994 年有相當大的變化。

標準程式庫的能力可區分爲以下幾個大項（亦見條款 E49）：

- **支援 C 標準函式庫**。別擔心，C++ 還記得它的根源。某些微小的變化，使「C++ 版本的 C 函式庫」與「C++ 的強烈型別檢驗性質」得以一致。但是，你對 C 函式庫所知道的一切，以及對它的愛恨情仇，在 C++ 中都依然存在。
- **支援 `string`s**。就像 C++ 標準程式庫小組主席 Mike Vilot 所說，『如果沒有提供一個標準的 `string` 型別，恐怕會出現街頭流血事件』（有些人就是這麼感情用事）。冷靜點，放下那些磚頭棍棒 — C++ 標準程式庫提供有 `string`。
- **支援國別（本土化，localization）**。不同的文化使用不同的字元集，並在顯示日期、時間、排序事物、貨幣值的時候，有著不同的習俗。標準程式庫對於國別的支援，使程式開發得以輕鬆容納多種文化差異。
- **支援 I/O**。iostream 程式庫仍舊是標準 C++ 的一部份，但是委員會對它做了一些修補。雖然某些 classes 被剔除了（特別值得注意的是 `iostream` 和 `fstream`），某些 classes 被取代了（例如 `string`-based `stringstreams` 取

代了 char*-based strstreams，後者不再被標準委員會認同），不過 iostream 內的各個標準 classes 仍可忠實反應那些早已存在多年的基本功能。

- **支援數值應用**。複數（complex numbers）長久以來是許多 C++ 教科書的示範對象，如今終於被奉祀於標準程式庫的殿堂上。此外，標準程式庫還包含特殊的陣列類別（valarrays），可以制止別名（aliasing）的發生。這些陣列比傳統的內建陣列有更進取的最佳化傾向，特別是在多工架構（multiprocessing architectures）下。標準程式庫也提供一些常用的數值函式，包括「部份和（partial sum）」以及「相鄰差值（adjacent difference）」。

- **支援廣泛用途的 containers（容器）和 algorithms（演算法）**。C++ 標準程式庫內含一組 class templates 和 function templates，統稱爲 Standard Template Library（STL）。STL 是 C++ 標準程式庫中最具革命性的部份。稍後我會摘要說明其特徵。

描述 STL 之前，我必須先介紹兩個你有必要知道的 C++ 標準程式庫特質。

第一，標準程式庫中的每一樣東西幾乎都是 template。本書之中我或許說過「標準的 string class」這樣的話，但事實上並沒有這樣的 class。實際上是一個名爲 basic_string 的 class template，用來表現字元序列，此 template 接受一個參數，正是架構出該序列的字元型別。這使得 strings 可由 chars 構成，也可由 wide chars、Unicode chars 或其他什麼東西構成。

我們一般所想的 string class，其實是 basic_string<char>。由於其使用極爲頻繁，標準程式庫特別提供了一個 typedef：

```
typedef basic_string<char> string;
```

即使如此，還是虛飾了許多細節，因爲 basic_string template 接受三個引數；除了第一個之外都有預設值。爲了真正瞭解 string 型別，你必須面對完整的、清晰的 basic_string 宣告：

```
template<class charT,
         class traits = string_char_traits<charT>,
         class Allocator = allocator>
  class basic_string;
```

不需要瞭解這等官樣文章才能使用 string 型別，因爲雖然 string 是上述那個「簡直來自地獄」的 template 的具現體的一個 typedef，其行爲卻像是個 non-

template class。只要在心裡埋藏一顆種子：如果需要自行指定字串所容納的字元型別，或是想要微調那些字元的行爲，或是想要篡奪字串的記憶體配置控制權，`basic_string` template 允許你那麼做。

`string` 型別的設計方法 — 泛化爲一個 template — 在標準程式庫中一再出現。IOstreams 嗎？喔，它們是 templates，它們有個型別參數（type parameter）用來定義資料流（streams）的字元型別。`complex` 嗎？喔，也是 templates，它有一個型別參數用來定義複數的實部和虛部的數值型態。`valarrays`？也是 templates，它有一個型別參數用來定義每一個陣列中的內容是什麼型態。至於 STL 的所有組成當然完全都是 templates。如果你不熟悉 templates，現在是開始學習的絕佳時機。

關於標準程式庫，其他需要知道的就是，它的所有成份都位於 namespace `std` 內。爲了在不需寫出完整資格修飾名稱（那可能會很長）的情況下使用標準程式庫的零組件，你可以利用一個 `using` directive 或（最好）使用多個 `using` declarations（見條款 E28）。幸運的是，當你 `#include` 某些表頭檔，編譯器會自動注意到「資格修飾名稱」的相關語法。

Standard Template Library（STL）

C++ 標準程式庫中最大的組成份子就是 STL：Standard Template Library。（由於 C++ 程式庫的幾乎每一樣東西都是 template，STL 這個名稱似乎不再那麼適當。不過這畢竟是標準程式庫中 containers（容器）和 algorithms（演算法）這一部份的名稱，所以不論如何，我們還是這麼用吧）

STL 影響了 C++ 標準程式庫的許多（甚至可說大部份）結構，所以熟悉其一般原則，對你而言非常重要。STL 並不難理解，它係以三個基本概念爲基礎：containers, iterators 和 algorithms。**Containers** 持有一系列物件。**Iterators** 是一種類似指標的物件，讓你可以巡訪 STL containers，就像以指標來巡訪內建陣列一樣。**Algorithms** 是可作用於 STL containers 身上的函式，以 iterators 來協助工作。

瞭解 STL 的最簡單方法就是，時時拿 C++（和 C）在陣列上的運作法則做例子。其實只有一個規則是我們需要知道的：一個指向某陣列的指標，可以合法地指向陣列中的任何元素，抑或超越陣列尾端的任何位置。如果指標指向陣列尾端以外的位置，那麼它就只能用來和其他指向陣列的指標相比較；對它取值

（dereferencing）是沒有意義的。

我們可以利用這個規則來寫個函式，在陣列中尋找某特定值。面對整數陣列，我們的函式如下：

```
int * find(int *begin, int *end, int value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

這個函式在 begin 和 end 範圍（不含 end — end 係指向陣列最後一個元素的下一個位置）之間搜尋 value，並傳回一個指標，指向它所找到的第一個目標；如果沒有找到，就傳回 end。

「傳回 end」似乎是「搜尋無結果」的一個可笑表示法。傳回 0（null 指標）難道不比較好嗎？當然 null 似乎比較自然，但不見得比較好。find 函式必須傳回某個特殊指標，用以表示搜尋失敗；就此目的而言，end 指標和 null 指標一樣好。此外，一如我們即將見到的，end 指標可以對其他型別的 containers 帶來泛型效果，這是 null 指標做不到的。

坦白說，你或許根本不是以這樣的寫法完成你的 find 函式，但這麼寫不是沒有道理的，而且其泛型程度出人意料地好。如果你能遵循這個簡單的例子，你便能夠精通 STL 的大部份觀念。

你可以這樣使用 find 函式：

```
int values[50];
...
int *firstFive = find(values,      // 搜尋範圍是：
                      values+50,   // values[0] ~ values[49]
                      5);          // 搜尋目標是 5
if (firstFive != values+50) {      // 搜尋成功了嗎？
  ...                              // 是的
}
else {
  ...                              // 不，搜尋失敗
}
```

也可以使用 find 來搜尋陣列的子範圍：

```
int *firstFive = find(values,      // 搜尋範圍是：
                      values+10,   // values[0] ~ values[9]
                      5);          // 搜尋目標是 5
int age = 36;
...
int *firstValue = find(values+10,  // 搜尋範圍是：
                       values+20,  // values[10] ~ values[19]
                       age);       // 搜尋目標擺在 age 內。
```

find 函式中並沒有什麼動作只限對整數陣列運作，所以可將它改成一個 template:

```
template<class T>
T * find(T *begin, T *end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

在轉換爲 template 的過程中，請注意我們將數值的傳遞由 pass-by-value 轉換爲 pass-by-reference-to-const。那是因爲現在我們傳遞的是任意型別，我們必須關心 pass-by-value 的成本。「by-value 參數」的成本包括：每次函式被喚起時，會連帶呼叫該參數的 constructor 和 destructor。使用 pass-by-reference 可避免這些成本，因爲它不需要喚起物件的 constructor 和 destructor（見條款 E22）。

上述的 template 很好，但還可以進一步泛型化。請看施加於 begin 和 end 身上的動作，用到的不外乎是「不等於」（inequality）判斷式、取值（dereferencing）、前序遞增（prefix increment，見條款 6）、複製（以便產生函式傳回值 — 見條款 19）。這些都可以多載化，那麼何必將 find 限制爲只能使用指標呢？爲什麼不讓凡是支援了這些操作行爲的物件都可以被使用呢？這麼做的話，便可使 find 函式從內建指標的意義中跳脫出來。舉個例子，我們可以爲串列（linked list）資料結構定義一個類似指標的物件，其前序遞增運算子（prefix increment operator）可將該「泛型指標」移到串列的下一元素。

這便是隱藏於 STL 背後的 iterators 觀念。Iterators 是一種行爲類似指標的物件，針對 STL containers 而定義。它們是條款 28 所說的 smart pointers 的表兄妹，只不過 smart pointers 的規模宏大得多。從技術的眼光來看，它們是以相同的技術實作出來。

「iterators 是行爲類似指標的一種物件」。是的，擁抱了觀念之後，我們便可以將 find 函式內的指標以 iterators 取代，於是 find 重新寫過：

```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

恭喜！你完成了 Standard Template Library 的一小部份。STL 內含幾十個 algorithms，可與 containers 和 iterators 搭配使用，`find` 便是其中之一。

STL containers 包括 `bitset`, `vector`, `list`, `deque`, `queue`, `priority_queue`, `stack`, `set`, 和 `map`，你可以將 `find` 應用於任何一種 container 身上：

```
list<char> charList;     // 產生 STL list 物件，用來存放 chars
...
// 搜尋 charList 中的第一個 'x'
list<char>::iterator it = find(charList.begin(),
                               charList.end(),
                               'x');
```

『哇喔！』，我聽到你大叫，『這看起來完全不像先前那個陣列例子』。它其實是；只不過你得先知道關鍵。

爲了將 `find` 應用於一個 `list` 物件身上，你需要一對 iterators，分別指向 `list` 的第一個元素和「最後一個元素的下一位置」。如果沒有 `list` class 的幫助，這是件困難的工作，因爲你完全不知道 `list` 是怎麼實作出來的。幸運的是 `list` 及其他所有 STL containers 都提供有一對 member functions `begin` 和 `end`。這些 member functions 傳回你需要的 iterators，於是你將它們當做 `find` 的前兩個參數，像上面那樣。

`find` 傳回一個 iterator 物件，指向被找到的元素（如果有的話），或是傳回 `charList.end()`（如果沒找到任何元素的話）。由於你對 `list` 如何實作一無所知，所以你對 iterators 如何指入 `lists` 之內也一無所知。那麼，你如何知道 `find` 傳回什麼樣的東西呢？再一次，`list` class（及所有其他 STL containers）有強制解法：它提供一個 typedef `iterator`，這便是「指向 `list`」之 iterators 型別。由於 `charList` 是一個由 `chars` 組成的 `list`，所以一個「指向此等 `list`」之 iterator，其型別便是 `list<char>::iterator`，這正是上述例子所用的東西。（每個 STL container class 實際上定義有兩個 iterator 型別：`iterator` 和 `const_iterator`。前者的行爲像一般指標，後者的行爲像個 pointer-to-const）

相同的作法亦可施行於其他 STL containers 身上。此外，C++ 指標也是一種 STL iterators，所以先前的陣列也可以用於 STL 的 `find` 函式：

```
int values[50];
...
int *firstFive = find(values, values+50, 5);   // 沒問題，呼叫 STL find
```

STL 的核心十分單純。它只是一堆「固守著一組公約」的 class templates 和 function templates。STL container classes 提供有 `begin` 和 `end` 函式，傳回「該 class 所定義之 iterator 型別」的物件。STL algorithm 利用 STL containers 的 iterator 物件，在 containes 內部元素之間移動，以利元素的處理。STL iterators 則是行爲類似指標的一種 class templates。整個故事就是這樣，沒有巨大的繼承體系，也沒有虛擬函式，只有一些 class templates 和 function templates，以及一組由它們共同承諾的公約。

這導出另一個發展：STL 是可擴充的。你可以將自己的 containers, algorithms, 和 iterators 加入 STL 家族內。只要你遵循 STL 的規矩，標準的 STL containers 便能夠和你的 algorithms 合作，而你的 containers 也將能夠與標準的 STL algorithms 合作。當然，你的這些 templates 不可能成爲 C++ 標準程式庫的一員，但它們都建立在相同的原理基礎上，同樣有高度的可重用性。

C++ 標準程式庫的組成比我在這裡所描述的多得多。在能夠有效運用這個程式庫之前，你必須付出更多的學習 — 比我這份摘要多得多。在你能夠寫出與 STL 相容的 templates 之前，你必須對 STL 核心所奉行的那套公約做更多的學習。C++ 標準程式庫遠比 C 函式庫豐富許多，但花費在它身上的學習時間絕對值得（見條款 E49）。此外，這個程式庫的設計原理 — 一般性、可擴充性、可訂製性、高效率、可重複運用性 — 都值得你學習。學習 C++ 標準程式庫，不僅可以增加你的知識，知道如何將包裝完整的組件運用於自己的軟體上面，也可以使你學習如何更有效率地運用 C++ 特性，並對如何設計更好的程式庫有所體會。

推薦讀物

Recommended Reading

可能你對 C++ 相關資訊的慾望還未飽足。別擔心，還有更多 — 多得是。以下列出我對 C++ 進階讀物的推薦。我想不必特別聲明，推薦當然是主觀的。不過我願意再說一次：推薦是主觀的，你有選擇。

書籍

C++ 書籍成百成千，新的競爭者以極大的頻率加入這場騷動之中。我並未看過所有這些書籍，仔細閱讀過的也不是非常多，但我的經驗是，其中有的非常好，有的並不理想。

下面開出來的書單是我自己在 C++ 軟體開發過程中遇到問題時的諮詢對象。其他好書當然也有，但這些是我自己使用過的，所以我可以放心推薦。

從描述「語言本身」的書籍開始，應該是個好起點。除非你非常在乎這些書籍與官方標準文件之間的一些細微差異，否則我建議你閱讀它們：

> *The Annotated C++ Reference Manual*, Margaret A. Ellis andBjarne Stroustrup, Addison-Wesley, 1990, ISBN 0-201-51459-1.
>
> *The Design and Evolution of C++*, Bjarne Stroustrup, Addison-Wesley, 1994, ISBN 0-201-54330-3.

這兩本書涵蓋的不只是語言本身的描述，也解釋了隱藏在設計之下的基本原理 — 那是你無法從官方標準文件中獲得的東西。*The Annotated C++ Reference Manual* 如

今已經不夠完整（它出版之後，C++ 又新增了一些語言特性 — 見條款 35），從某方面說已經過氣，但它仍然是語言核心方面（包括 templates 和 exceptions）的最佳參考書籍。*The Annotated C++ Reference Manual* 遺漏的主題，大部份已涵蓋於 *The Design and Evolution of C++* 之中，唯獨欠缺 Standard Template Library（見條款 35）的討論。這些書籍都不是學習指南或自修書，它們是參考書籍，但如果你不瞭解這些書籍所談的東西，實在不能說是真正瞭解 C++。

至於 C++ 語言及標準程式庫方面的一般性參考書籍，沒有誰比得上 C++ 語言創造者所寫的書：

> *The C++ Programming Language* (Third Edition), BjarneStroustrup, Addison-Wesley, 1997, ISBN 0-201-88954-4.

從 C++ 語言誕生伊始，Stroustrup 便涉及其設計、實作、應用、以及標準化。他知道的恐怕比任何其他人都多。他對 C++ 語言特性的描述，形成了這本密度極高的讀物。密度高主要是因爲，資訊就是那麼多。書中與 C++ 標準程式庫相關的各章篇幅，提供了 C++ 重要部份的一個良好導入。

如果你即將跨越語言本身，努力思考如何有效運用 C++，可以考慮我的另一本書：

> *Effective C++, Second Edition: 50 Specific Ways to Improve YourPrograms and Designs*, Scott Meyers, Addison-Wesley, 1998, ISBN0-201-92488-9.

此書組織風格類似本書，但涵蓋不同（可說是較爲基礎）的題材。

有一本書的基調和我的 *Effective C++* 差不多，但涵蓋主題不同：

> *C++ Strategies and Tactics*, Robert Murray, Addison-Wesley, 1993,ISBN 0-201-56382-7.

Murray 的書在 template 相關設計方面特別著重，他在這上面貢獻了兩章。另有一章專注於「從 C 的開發遷徙至 C++ 的開發」這個重要題目。我對 reference counting（參用計數，條款 29）的討論亦是以 *C++ Strategies and Tactics* 的觀念爲基礎。

如果你是那種喜歡從讀碼過程中學習程式技術的人，下面這本書適合你：

C++ Programming Style, Tom Cargill, Addison-Wesley, 1992,ISBN 0-201-56365-7.

此書的每一章，一開始都以某些公開發行的 C++ 軟體做爲例子，示範如何正確進行某些事情。然後 Cargill 開始解剖 — 活體解剖 — 每個程式，找出可能出現麻煩的地點、不良的設計選擇，短暫易碎的實作決定、以及根本就錯誤的動作。然後他重寫每一個例子，消除那些缺點。經過他的改造，程式碼更強固、更易維護、更有效率、亦更容易移植，並且仍能解決原先的問題。任何 C++ 程式員，特別是對那些需要檢閱別人程式的人，最好能夠留心此書帶給我們的課題。

C++ Programming Style 沒有討論的一個主題是 exceptions。Cargill 把他那對語言特性有特異功能的鷹眼擺在以下文章中，該文告訴我們，爲什麼寫出「面對 exception 依然安全（所謂 exception-safe）」的程式，其困難度比大部份程式員所瞭解的更困難：

"Exception Handling: A False Sense of Security,"*C++ Report*,Volume 6, Number 9, November-December 1994, pages 21-24.

如果你重視 exceptions 的使用，動手之前請先讀過上篇文章。

一旦你精通了 C++ 基本面，準備開始收成，你必須讓自己熟悉：

Advanced C++: Programming Styles and Idioms, James Coplien,Addison-Wesley, 1992, ISBN 0-201-54855-0.

我常稱此書爲 "the LSD book"（譯註：Lysergic Acid Diethylamide 是一種迷幻藥），因爲它有紫色的封面，而且它會使你的心神膨脹。Coplien 涵蓋某些直接而明確的素材，但他真正的焦點在於告訴你如何在 C++ 中做到你不以爲能夠做到的事情。想要將物件建構於另一個物件之上嗎？他告訴你怎麼做。想要迴避強烈型別檢驗（strong typing）嗎？他給你一個方法。想要在程式執行時爲 classes 加上資料和函式嗎？他爲你解釋如何進行。大部份時候，你大概只是循著他所敘述的技術前進，盡情地瀏覽，但有時候這些技術剛好可以提供你手上某個棘手問題的解答。猶有進者，它揭露了 C++ 可以做到什麼樣的事情。這本書可能令你駭怕，可能使你茫然，但是當你讀完它，你再也不會像以前一樣地看待 C++ 了。

如果你打算設計和實作 C++ 程式庫，漏看以下書籍，只能說是有勇無謀：

Designing and Coding Reusable C++, Martin D. Carroll andMargaret A. Ellis, Addison-Wesley, 1995, ISBN 0-201-51284-X.

Carroll 和 Ellis 討論了程式庫設計和實作上的許多實用方向，這些方向常被每個人忽略。好的程式庫的特色是小、快、可擴充、容易升級、在 template 具現化時有優雅的表現、威力強大、而且穩健。沒有一個人能夠達到上述每一項要求，所以你必須有所取捨。*Designing and Coding Reusable C++* 驗證這些取捨，並提供現實的忠告，告訴你如何達成你所選擇的目標。

不論你寫的軟體是否應用於科學或工程，你都應該看看這本書：

Scientific and Engineering C++, John J. Barton and Lee R.Nackman, Addison-Wesley, 1994, ISBN 0-201-53393-6.

此書第一部份爲 FORTRAN 程式員解釋 C++（這可不是件值得羨慕的工作），但是稍後所涵蓋的技術幾乎適用於任何領域。書中對 templates 的廣泛運用，簡直像一場革命；這或許是目前爲止最先進的應用，我猜一旦你看過這些作者以 templates 完成的奇蹟，你再也不會以爲所謂 templates 只不過是比 macros 好一點的玩意兒。

壓軸的是物件導向軟體開發過程的 patterns 訓練（見 p123）。這個主題描述於：

Design Patterns: Elements of Reusable Object-Oriented Software,Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides,Addison-Wesley, 1995, ISBN 0-201-63361-2.

這本書對於 patterns 背後的概念，提供了一份導覽。其主要貢獻在於對可應用於眾多領域之 23 個基本 patterns，加以分類整理。任何人翻閱這份分類目錄，幾乎一定可以找到一個你曾經於過去某時候自行發明的 pattern。而同時你也幾乎一定會發現此書的設計優於你的設計。書中所提的 patterns 名稱，幾乎已經成爲物件導向設計領域的標準辭彙。如果不知道這些名稱，可能會造成你和你的同事之間溝通上的危機。此書特別重視軟體應該如何設計和實作，才能將未來的演化優雅地納入（見條款 32 和 33）。

Design Patterns 也以 CD-ROM 的形式呈現：

> *Design Patterns CD: Elements of Reusable Object-OrientedSoftware*, Erich Gamma, Richard Helm, Ralph Johnson, and JohnVlissides, Addison-Wesley, 1998, ISBN 0-201-63498-8.

雜誌刊物

喜歡 C++ 硬梆梆素材的傢伙們，這本刊物大概是你生活中唯一的樂趣：

> *C++ Report*, SIGS Publications, New York, NY.

這本刊物做了一個神志清醒的決定，放棄 "C++ only" 的老根。不過由於持續增加與特定領域和特定系統有關的程式設計主題，並且做得很好，所以即使 C++ 方面的題材偶爾「程度不夠深」，整體而言仍然很有價值。

如果你喜歡 C 甚於 C++，或是如果你發現 *C++ Report* 的題材太極端，或許可以在這本刊物中找到一些符合自己口味的文章：

> *C/C++ Users Journal*, Miller Freeman, Inc., Lawrence, KS.

如其名稱所示，這本刊物兼含 C 和 C++。其中 C++ 文章對讀者程度的要求，比 *C++ Report* 所要求的低。此外，編輯群對於作者的約束很緊，所以刊出的文章相對比較主流。這有助於過濾掉狂暴邊緣的某些想法，不過這也限制了你真正看到「銳利」技術的機會。

網路社群（Usenet Newsgroups）

有三個 Usenet 討論群專注於 C++ 這個大主題。一般性的、什麼都可以上去的討論群是 `comp.lang.c++`。這個場所進行全方位交易，從高階程式技術的細微討論，到胡說八道似的瞎擾和（誰誰誰喜歡 C++ 啦，誰誰誰討厭 C++ 啦），再到全世界大學生在此火燒屁股地尋求作業協助，都有！這個討論群上的卷籍實在是太多了。除非你有數個小時的自由時間，我想你會運用過濾器來幫你篩出粗糠中的小麥。選一個好點兒的過濾軟體吧，粗糠很多呢！

1995 年 11 月，一個有主持人駐守的 `comp.lang.c++` 誕生了。這個名爲 `comp.lang.c++.moderated` 的討論群，也是用於 C++ 及其相關主題的一般討論，但主持人會刪除特定平台上的問題和見解、線上常見問答集（on-line FAQ，

Frequently Asked Questions）已涵蓋的問題、口水戰（不管是爲了什麼）、以及大部份 C++ 程式開發者比較不感興趣的話題。

一個範圍更窄的討論群是 `comp.std.c++`，它專注於 C++ 標準規格的討論。語言方面的專家多出沒於此討論群中。如果你那吹毛求疵的 C++ 問題在其他討論群上一直沒有人回答，或回答得不令你滿意，這裡是發問的好地方。這個討論群有主持人，所以「良訊/雜訊」比相當令人滿意；你不會在這裡看到任何作業請託。

auto_ptr 實作碼

條款 9, 10, 26, 31 和 32 都證明了 auto_ptr template 具有非常值得注意的功能。不幸的是，極少編譯器搭配有正確的 auto_ptr 產品[1]。條款 9 和條款 28 大致描述了自行撰寫 auto_ptr 的方法，但當我們著手真實世界中的專案時，最好是能夠擁有更實際的東西。

下面列出兩份 auto_ptr 作品。第一份作品對 class 的介面有註解說明，並將所有 member functions 實作於 class 定義區外。第二份作品則是在 class 定義區內實作每一個 member function。以風格而言，第二份作品比第一份拙劣，因爲它沒有將 class 的介面和實作分離開來。不過，由於 auto_ptr 只是生產出簡單的 classes，所以第二份作品或許反而讓人看起來一目瞭然。

下面是 auto_ptr 及其介面說明：

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0);      // 關於 "explicit"，見條款 5

  template<class U>              // copy constructor member template
  auto_ptr(auto_ptr<U>& rhs);    //（見條款 28）：以任何相容的 auto_ptr
                                 // 做爲一個新的 auto_ptr 的初值。
  ~auto_ptr();

  template<class U>       // assignment operator member template
  auto_ptr<T>&            // （見條款 28）：以任何相容的 auto_ptr
  operator=(auto_ptr<U>& rhs);    // 做爲指派動作的右端。
```

[1] 這主要是因爲 auto_ptr 的規格一直變來變去。1997 年 11 月才終於確認了最後一份規格。細節請參考本書 WWW 網站的 auto_ptr 相關資訊。注意，本處所列的 auto_ptr 未含官方版本中的某些細節，例如 auto_ptr 係被置於 std namespace（見條款 35）內、其 member functions 保證不丟出 exceptions 等等。

```
  T& operator*() const;              // 見條款 28
  T* operator->() const;             // 見條款 28

  T* get() const;                    // 傳回 dumb pointer

  T* release();                      // 撤回 dumb pointer 的擁有權
                                     // 並傳回其值。

  void reset(T *p = 0);              // 將擁有的指標刪除；
                                     // 並承擔 p 的擁有權。
private:
  T *pointee;
template<class U>                    // 讓所有的 auto_ptr classes 都
friend class auto_ptr<U>;            //成爲另一個 auto_ptr的 friends
};

template<class T>
inline auto_ptr<T>::auto_ptr(T *p)
: pointee(p)
{}

template<class T>
  template <class U>                 // 譯註：本書 CD 版少此行。
  inline auto_ptr<T>::auto_ptr(auto_ptr<U>& rhs)
  : pointee(rhs.release())
  {}

template<class T>
inline auto_ptr<T>::~auto_ptr()
{ delete pointee; }

template<class T>
  template<class U>
  inline auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<U>& rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }

template<class T>
inline T& auto_ptr<T>::operator*() const
{ return *pointee; }

template<class T>
inline T* auto_ptr<T>::operator->() const
{ return pointee; }

template<class T>
inline T* auto_ptr<T>::get() const
{ return pointee; }
```

```
template<class T>
inline T* auto_ptr<T>::release()
{
  T *oldPointee = pointee;
  pointee = 0;
  return oldPointee;
}

template<class T>
inline void auto_ptr<T>::reset(T *p)
{
  if (pointee != p) {
    delete pointee;
    pointee = p;
  }
}
```

下面這個 auto_ptr 把所有的函式都定義在 class 定義區內：

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0): pointee(p) {}

  template<class U>
  auto_ptr(auto_ptr<U>& rhs): pointee(rhs.release()) {}

  ~auto_ptr() { delete pointee; }

  template<class U>
  auto_ptr<T>& operator=(auto_ptr<U>& rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }

  T& operator*() const { return *pointee; }
  T* operator->() const { return pointee; }

  T* get() const { return pointee; }
  T* release()
  {
    T *oldPointee = pointee;
    pointee = 0;
    return oldPointee;
  }

  void reset(T *p = 0)
  {
    if (pointee != p) {
      delete pointee;
      pointee = p;
    }
  }
```

```
  private:
    T *pointee;
  template<class U> friend class auto_ptr<U>;
};
```

如果你的編譯器尚未支援關鍵字 `explicit`，你可以利用 `#define` 將它自以上程式碼中移除（尚稱安全）：

```
#define explicit
```

這並不會使我們的 `auto_ptr` 失去任何功能，但是會流失一點點安全性。細節請見條款 5。

如果你的編譯器尚未支援 member templates，你可以使用條款 28 所描述的 non-template `auto_ptr` copy constructor 和 assignment operator。這會使你的 `auto_ptr`s 使用上比較沒那麼方便，但是再沒有其他方法可以近似 member templates 的行爲了。如果 member templates（或其他語言性質）對你非常重要，請讓你的編譯器廠商知道。愈多客戶要求新的語言特性，廠商們就會愈快提供它們。

索引（一）

General Index

這份索引範圍涵蓋本書所有內容，但不包括我用來做爲範例的 classes, functions, 和 templates。如果你想參考任何一個特定的 classes, functions, 或 templates 範例，請查閱 313 頁的索引（二）。至於 C++ 標準程式庫中的 classes, functions, 和 templates（例如 `string`, `auto_ptr`, `list`, `vector` 等等），請查閱本份索引。

大部份時候，運算子（operators）都條列於 *operator* 欄。也就是說 `operator<<` 列於 `operator<<` 欄下而非 `<<` 欄下，依此類推。不過如果運算子的名稱是字詞（words）或類字詞（word-like），例如 `new`, `delete`, `sizeof`, `const_cast` 等等，它們會被我列在適當的字詞欄下，如 `new`, `delete`, `sizeof`, `const_cast` 等等。

範例中凡用到較新或較少爲人所知之語言特性者，皆被我列於 *example uses* 欄下。

譯註：本索引完全依照英文版製作，未加任何增減修改。由於中文版頁次與英文版頁次相同，且專用術語時時中英並陳，故直接採用英文版索引，效果尚稱良好。事實上由於英文索引之語言結構關係，欲在保持原索引結構精準度的原則下譯爲中文，亦不可得。竊以爲本書之讀者應能夠（並樂意）接受英文索引。

請注意：
(1) 雖說中文版頁次與英文版頁次相同，但每頁頭尾若非恰爲分段點，便可能與英文版之文字排列佈局稍有差異（中英語文結構差異之故）。使用本份索引時請注意此點。
(2) 英文版致謝（Acknowledgments）部分，中文版略而未譯。本索引出現之 xi~xv 頁碼，即爲原致謝文內之人名、書名或相關字詞。
(3) 中文版之「目錄」，中英並陳，篇幅多於原書。因恰在書籍最前端，故不影響本份索引。

Before A

A

B

C

D

E

F

G

K

L

M

N

O

P

Q

R

S

T

W

Y

索引（二）

Index of Example Classes, Functions, and Templates

Classes and Class Templates

Functions and Function Templates

More Effective C++ 中文版

Effective C++（Scott Meyers 的第一本書）的榮耀：
『對於任何渴望在中階或高階層面精通 C++ 的人，我慎重推薦 *Effective C++*。』
-- *The C/C++ User's Journal*

必備讀物 *Effective C++* 的作者再一次出發，列出 35 個改善程式技術與設計思維的新方法。以多年經驗爲基礎，Meyers 解釋如何更有效率地撰寫軟體：使效率更高、更穩健強固、更一致化、更具移植性、更富重用性。簡單地說就是撰寫更好的 C++ 軟體。

More Effective C++ 的重點包括：

- 經過驗證的一些用來改善程式效率的方法，包括尖銳而犀利地檢驗 C++ 語言特性所帶來的時間和空間上的成本。
- 廣泛描述 C++ 專家所使用的高階技術，包括 placement new, virtual constructors, smart pointers, reference counting, proxy classes, double-dispatching.
- 以實例說明 exception handling 帶給 C++ classes 和 functions 意義深長的衝擊 — 包括其結構和行爲。
- 務實面對新的語言特性，包括 `bool`, `mutable`, `explicit`, `namespaces`, member templates, C++ 標準程式庫，以及更多東西。如果你的編譯器尚未支援這些特性，Meyers 告訴你如何在沒有它們的情況下完成該完成的事情。

More Effective C++ 一書充滿了實用性高且擲地鏗鏘的忠告，爲你每天可能面對的問題帶來幫助。和其前一本兄弟書籍 *Effective C++* 一樣，*More Effective C++* 對每一位以 C++ 爲開發工具的程式員而言，都是必備讀物。

Scott Meyers 是 C++ 領域公認的權威，並對全球客戶提供諮詢服務。他是 *Effective C++* 的作者，*C++ Report* 的知名專欄作家，全球各技術研討會上極具號召力的講師。他於 1993 年拿到布朗大學（Brown University）的計算機科學博士學位。

侯捷　集電腦技術性讀物之著、譯、評於一身，在臺灣軟體界廣受尊敬。他是《多型與虛擬》和《*C++* 泛型技術》的作者，《*C++ Primer* 中文版》和《*COM* 本質論》等書的譯者，*Run!PC* 雜誌的知名專欄作家，也是極具號召力的大學教師與研討會講師。

侯捷網站 http://www.jjhou.com