GOTOP

深度探索

C++ 物件模型

Inside The C++ Object Model

Stanley B. Lippman 著

侯捷 譯

- Object Lessons
- The Semantics of Constructors
- The Semantics of Data
- The Semantics of Function
- Semantics of Construction, Destruction, and Copy
- Runtime Semantics
- On the Cusp of the Object Model

深度探索

C++ 物件模型

Inside The C++ Object Model

Stanley B. Lippman 著

侯捷 譯

碁峰資訊股份有限公司

本立道生

（侯捷 譯序）

對於傳統的循序性（sequential）語言，我們向來沒有太多的疑惑，雖然在函式呼叫的背後，也有著堆疊建制、參數排列、回返位址、堆疊清除等等幕後機制，但函式呼叫是那麼地自然而明顯，好像只是夾帶著一個包裹，從程式的某一個地點跳到另一個地點去執行。

但是對於物件導向（Object Oriented）語言，我們的疑惑就多了。究其因，這種語言的編譯器爲我們（程式員）做了太多的服務：建構式、解構式、虛擬函式、繼承、多型...。有時候它爲我們合成出一些額外的函式（或運算子），有時候它又擴張我們所寫的函式內容，放進更多的動作。有時候它還會爲我們的 objects 加油添醋，放進一些奇妙的東西，使你面對 *sizeof* 的結果大驚失色。

存在我心裡頭一直有個疑惑：電腦程式最基礎的形式，總是脫離不了一行一行的循序執行模式，爲什麼 OO（物件導向）語言卻能夠「自動完成」這麼多事情呢？另一個疑惑是，威力強大的 polymorphism（多型），其底層機制究竟如何？

如果不瞭解編譯器對我們所寫的 C++ 碼做了什麼手腳，這些困惑永遠解不開。

這本書解決了過去令我百思不解的諸多疑惑。我要向所有已具備 C++ 多年程式設計經驗的同好們大力推薦這本書。

這本書同時也是躍向元件軟體（component-ware）基本精神的跳板。不管你想學習 COM（Component Object Model）或 CORBA（Common Object Request Broker Architecture）或是 SOM（System Object Model），瞭解 C++ Object Model，將使你更清楚軟體元件（components）設計上的難點與運應之道。不但我自己在學習 COM 的道路上有此強烈的感受，***Essential COM***（**COM 本質論**，侯捷譯，碁峰 1998）的作者 Don Box 也在他的書中推崇 Lippman 的這一本卓越的書籍。

是的，這當然不會是一本輕鬆的書籍。某些章節（例如３、４兩章）可能給你立即的享受 -- 享受於面對底層機制有所體會與掌控的快樂；某些章節（例如５、６、７三章）可能帶給你短暫的痛苦 -- 痛苦於艱難深澀難以吞嚥的內容。這些快樂與痛苦，其實就是我翻譯此書時的心情寫照。無論如何，我希望透過我的譯筆，把這本難得的好書帶到更多人面前，引領大家見識 C++ 底層建設的技術之美。

侯捷 1998.05.20 于新竹
jjhou@ccca.nctu.edu.tw

請注意：本書屬性，作者 Lippman 在其前言中有很詳細的描述，我不再多言。翻譯用詞與心得，記錄在第０章（譯者的話）之中，對您或有導讀之功。

請注意：原文本有大大小小約 80~90 個筆誤。有的無傷大雅，有的影響閱讀順暢甚鉅（如前後文所用符號不一致、內文與圖形所用符號不一致 -- 甚至因而導至圖片的文字解釋不正確）。我已在第０章（譯者的話）列出所有我找到的錯誤 。此外，某些場合我還會在錯誤出現之處再加註，表示原文內容爲何。這麼做不是畫蛇添足，也不爲彰顯什麼。我知道有些讀者拿著原文書和中譯書對照著看，我把原書錯誤加註出來，可免讀者懷疑是否我打錯字或是譯錯了。另一方面也是爲了文責自負...唔...萬一 Lippman 是對的而 J.J.Hou 錯了呢 ?! 我雖有相當把握，還是希望明白攤開來讓讀者檢驗。

深度探索 C++ 物件模型

Inside The C++ Object Model

目 錄

前 言

（Stanley B. Lippman）

差不多有 10 年之久，我在貝爾實驗室（Bell Laboratories）埋首於 C++ 的實作任務。最初的工作是在 cfront 上面（Bjarne Stroustrup 的第一個 C++ 編譯器），從 1986 年的 1.1 版到 1991 年九月的 3.0 版。然後移轉到 Simplifier（這是我們內部的命名），也就是 Foundation 專案中的 C++ 物件模型部份。在 Simplifier 設計期間，我開始醞釀這本書。

Foundation 專案是什麼？在 Bjarne 的領導下，貝爾實驗室中的一個小組探索著以 C++ 完成大規模程式設計時的種種問題的解決之道。Foundation 專案是我們爲了建構大系統而努力定義的一個新的開發模型；我們只使用 C++，並不提供多重語言的解決方案。這是個令人興奮的工作，一方面是因爲工作本身，一方面是因爲工作夥伴：Bjarne、Andy Koenig、Rob Murray、Martin Carroll、Judy Ward、Steve Buroff、Peter Juhl、以及我自己。Barbara Moo 管理我們這一群人（Bjarne 和 Andy 除外）。Barbara Moo 常說管理一個軟體團隊，就像放牧一群驕傲的貓。

我們把 Foundation 想像是一個核心，在那上面，其他人可以爲使用者鋪設一層真正的開發環境，把它整修爲他們所期望的 UNIX 或 Smalltalk 模型。私底下我們把它稱爲 Grail（傳說中耶穌最後晚餐所用的聖杯），人人都想要，但是從來沒人找到過！

Grail 使用一個由 Rob Murray 發展出來並命名爲 ALF 的物件導向階層架構，提供一個永久的、以語意爲基礎的表現法。在 Grail 中，傳統編譯器被分解爲數個各自分離的可執行檔。parser 負責建立程式的 ALF 表現法。其他每一個元件（像是 type checking、simplification、code generation）以及工具（像是 browser）都在程式的一個 ALF 表現體上操作（並可能加以膨脹）。Simplifier 是編譯器的一部份，處於 type checking 和 code generation 之間。Simplifier 這個名稱是由 Bjarne 所倡議，它原本是 cfront 的一個階段（phase）。

在 type checking 和 code generation 之間，Simplifier 做什麼事呢？它用來轉換內部的程式表現。有三種轉換風味是任何物件模型都需要的：

1. 與編譯器息息相關的轉換（Implementation-dependent transformations）

這是與特定編譯器有關的轉換。在 ALF 之下，這意味著我們所謂的 "tentative" nodes。例如，當 parser 看到這個運算式：

```
fct();
```

它並不知道是否 **(a)** 這是一個函式喚起動作，或者 **(b)** 這是 overloaded call operator 在 class object *fct* 上的一種應用。預設情況下，這個式子所代表的是一個函式呼叫，但是當 **(b)** 的情況出現，Simplifier 就要重寫並調換 call subtree。

2. 語言語意轉換（Language semantics transformations）

這包括 constructor/destructor 的合成和擴張、memberwise 初始化、對於 memberwise copy 的支援、在程式碼中安插 conversion operators、暫時性物件、以及對 constructor/destructor 的呼叫。

3. 程式碼和物件模型的轉換（Code and object model transformations）

這包括對 virtual functions、virtual base class 和 inheritance 的一般支援、*new* 和 *delete* 運算子、class objects 所組成的陣列、local static class instances、帶有非常數運算式（nonconstant expression）之 global object 的靜態初始化動作。我對 Simplifier 所規劃的一個目標是：提供一個物件模型體系，在其中，物件的實作是一個虛擬介面，支援各種物件模型。

最後兩種類型的轉換構成了本書的基礎。這意味本書是爲編譯器設計者而寫的嗎？不是，絕對不是！這本書是由一位編譯器設計者針對中高階 C++ 程式員所寫。隱藏在這本書背後的假設是，程式員如果瞭解 C++ 物件模型，就可以寫出比較沒有錯誤傾向而且比較有效率的碼。

什麼是 C++ 物件模型

有兩個概念可以解釋 C++ 物件模型：

1. 語言中直接支援物件導向程式設計的部份。

2. 對於各種支援的底層實作機制。

語言層面的支援，涵蓋於我的 ***C++ Primer*** 一書以及其他許多 C++ 書籍當中。至於第二個概念，則幾乎不能夠於目前任何讀物中發現，只有 [ELLIS90] 和 [STROUP94] 勉強有一些蛛絲馬跡。本書主要專注於 C++ 物件模型的第二個概念。本書語言遵循 C++ 委員會於 1995 冬季會議中通過的 Standard C++ 草案（除了某些細節，這份草案應該能夠反映出此語言的最終版本）。

C++ 物件模型的第一個概念是一種「不變量」。例如，C++ class 的完整 virtual functions 在編譯時期就固定下來了，程式員沒有辦法在執行時期動態增加或取代其中某一個。這使得虛擬喚起動作得有快速的派送（dispatch）結果，付出的成本則是執行時期的彈性。

物件模型的底層實作機制，在語言層面上是看不出來的 -- 雖然物件模型的語意本身可以使得某些實作品（編譯器）比其他實作品更接近自然。例如，virtual function calls，一般而言是藉由一個表格（內含 virtual functions 位址）的索引而決議得知。一定要使用如此的 virtual table 嗎？不，編譯器可以自由引進其他任何變通作法。如果使用 virtual table，那麼其佈局、存取方法、產生時機、以及數百個細節也都必須決定下來，而所有決定也都由每一個實作品（編譯器）自行取捨。不過，既然說到這裡，我也必須明白告訴你，目前所有編譯器對於 virtual function 的實作法都是使用各個 class 專屬的 virtual table，大小固定，並且在程式執行前就建構好了。

如果 C++ 物件模型的底層機制並未標準化，那麼你可能會問：何必探討它呢？主要的理由是，我的經驗告訴我，如果一個程式員瞭解底層實作模型，他就能夠寫出效率較高的碼，自信心也比較高。一個人不應該用猜的，或是等待某大師的宣判，才確定「何時提供一個 copy constructor 而何時不需要」。這類問題的解答應該來自於我們自身對物件模型的瞭解。

寫這本書的第二個理由是爲了消除我們對於 C++ 語言（及其對物件導向的支援）的各種錯誤認知。下面一段話節錄自我收到的一封信，來信者希望將 C++ 引進其程式環境中：

> 我和一群人工作，他們過去不曾寫過（或完全不熟悉）C++ 和 OO。其中一位工程師從 1985 就開始寫 C 了，他非常強烈地認爲 C++ 只對那些 user-type 程式才好用，對 server 程式卻不理想。他說如果要寫一個快速而有效率的資料庫引擎，應該使用 C 而非 C++。他認爲 C++ 龐大又遲緩。

C++ 當然並不是天生地龐大又遲緩，但我發現這似乎成爲 C 程式員的一個共識。然而，光是這麼說並不足以使人信服，何況我又被認爲是 C++ 的「代言人」。這本書就是企圖極盡可能地將各式各樣的 Object facilities（如 inheritance、virtual functions、指向 class members 的指標…）所帶來的額外負荷說個清楚。

除了我個人回答這封信，我也把此信轉寄給 HP 的 Steve Vinoski；先前我曾與他討論過 C++ 的效率問題。以下節錄自他的回應：

> 過去數年我聽過太多與你的同事類似的看法。許多情況下，這些看法是源於對 C++ 事實真象的缺乏瞭解。就在上週，我才和一位朋友閒聊，他在一家 IC 製造廠服務，他說他們不使用 C++，因爲「它在你的背後做事情」。我緊迫盯人，於是他說根據他的瞭解，C++ 呼叫 *malloc()* 和 *free()* 而不讓程式員知道。這當然不是真的。這是一種所謂的迷思與傳說，引導出類似於你的同事的看法...
>
> 在抽象性和實際性之間找出平衡點，需要知識、經驗、以及許多思考。C++ 的使用需要付出許多心力，但是我的經驗告訴我，這項投資的報酬率相當高。

我喜歡把這本書想像是我對那一封讀者來信的回答。是的，這本書是一個知識陳列庫，幫助大家去除圍繞在 C++ 四週的迷思與傳說。

如果 C++ 物件模型的底層機制會因爲實作品（編譯器）和時間的變動而不同，我如何能夠對於任何特定主題提供一般化的討論呢？靜態初始化（Static initialization）可爲此提供一個有趣的例子。

已知一個 class *X* 有著 constructor，像這樣：

```
class X
{
  friend istream&
    operator>>( istream&, X& );
public:
  X( int sz = 1024 ) { ptr = new char[ sz ]; }
  ...
private:
  char *ptr;
};
```

而一個 class *X* 的 global object 的宣告，像這樣：

```
X buf;

main()
{
  // buf 必須在這個時候建構起來
  cin >> setw( 1024 ) >> buf;
  ...
}
```

C++ 物件模型保證，*X* constructor 將在 *main()* 之前便把 *buf* 初始化。然而它並沒有說明這是如何辦到的。答案是所謂的靜態初始化（static initialization），實際作法則有賴開發環境對此的支援屬於哪一層級。

原始的 cfront 實作品不單只是假想沒有環境支援，它也假想沒有明白的目標平台。唯一能夠假想的平台就是 UNIX 及其衍化的一些變體。我們的解決之道也因此只專注在 UNIX 身上：我們使用 nm 命令。CC 命令（一個 UNIX shell script）產生出一個可執行檔，然後我們把 nm 施行於其上，產生出一個新的 .c 檔案。然後編譯此一新的 .c 檔，再重新聯結出一個可執行檔（這就是所謂的 munch solution）。這種作法是以編譯器時間來交換移植性。

接下來是提供一個「平台特定」解決之道：直接驗證並穿越 COFF-based 程式的可執行檔（此即所謂的 patch solution），不再需要 nm、compile、relink。COFF 是 Common Object File Format 的縮寫，是 System V pre-Release 4 UNIX 系統所發展出來的格式。這兩種解決方案都屬於程式層面，也就是說，針對每一個需要靜態初始化的 .c 檔，cfront 會產生出一個 *sti* 函式，執行必要的初始化動作。不論是 patch solution 或是 munch solution，都會去尋找以 *sti* 開頭的函式，並且安排它們以一種未被定義的次序執行起來（藉由安插在 *main()* 之後第一行的一個 library function *_main()* 執行之）（譯註：本書第 6 章對此有詳細說明）。

System V COFF-specific C++ 編譯器與 cfront 的各個版本平行發展。由於瞄準了一個特定平台和特定作業系統，此編譯器因而能夠影響聯結器特別爲它修改：產生出一個新的 .ini section，用以收集需要靜態初始化的 objects。聯結器的這種擴

充方式，提供了所謂的 environment-based solution，那當然更在 program-based solution 層次之上。

至此，任何以 cfront program-based solution 爲基礎的一般化（泛型）動作將令人迷惑。爲什麼？因爲 C++ 已經成爲主流語言，它已經接收了更多更多的 environment-based solutions。這本書如何維護其間的平衡呢？我的策略如下：如果在不同的 C++ 編譯器上有重大的實作技術差異，我就討論至少兩家作法。但如果 cfront 之後的編譯器實作模型只是解決 cfront 原本就已理解的問題，例如對虛擬繼承的支援，那麼我就闡述歷史的演化。當我說到「傳統模型」，我的意思是 Stroustrup 的原始構想（反應在 cfront 身上），它提供一種實作模範，在今天所有的商業化實作品上仍然可見。

本書組織

第 1 章，**關於物件（Object Lessons）**，提供以物件爲基礎的觀念背景，以及由 C++ 提供的物件導向程式設計典範（paradigm。譯註：關於 paradigm 這個字，請參閱本書 #22 頁的譯註）。本章包括對物件模型的一個大略遊覽，說明目前普及的工業產品，但沒有對於多重繼承和虛擬繼承有太靠近的觀察（那是第 3 章和第 4 章的重頭戲）。

第 2 章，**建構式語意學（The Semantics of Constructors）**，詳細討論 constructor 如何工作。本章談到 constructors 何時被編譯器合成，以及對你的程式效率帶來什麼樣的意義。

第 3 章至第 5 章是本書的重要題材。在這裡，我詳細地討論了 C++ 物件模型的細節。第 3 章，**Data 語意學（The Semantics of Data）**，討論 data members 的處理。第 4 章，**Function 語意學（The Semantics of Function）**，專注於各式各樣的 member functions，並特別詳細地討論如何支援 virtual functions。第 5 章，**建構、解構、拷貝語意學（Semantics of Construction, Destruction, and Copy）**，討論如何支援 class 模型，也討論到 object 的生

命期。每一章都有測試程式以及測試數據。我們對效率的預測，將拿來和實際結果做一比較。

第 6 章，執行時期語意學（**Runtime Semantics**），檢視執行時期的某些物件模型行爲。包括暫時物件的生命及其死亡，以及對 new 運算子和 delete 運算子的支援。

第 7 章，在物件模型的尖端（**On the Cusp of the Object Model**），專注於 exception handling、template support、runtime type identification。

預定的讀者

這本書可以扮演家庭教師的角色，不過它定位在中階以上的 C++ 程式員，而非 C++ 新手。我嘗試提供足夠的內容，使它能夠被任何有點 C++ 基礎（例如讀過我的 ***C++ Primer*** 並有一些實際程式經驗）的人接受。理想的讀者是，曾經有過數年的 C++ 程式經驗，希望更瞭解「底層做些什麼事」的人。書中某些部份甚至對於 C++ 高手也具吸引力，像是暫時性物件的產生，以及 named return value（NRV）最佳化的細節等等。在與本書相同素材的各個公開演講場合中，我已經證實了這些材料的吸引力。

程式範例及其執行

本書的程式範例主要有兩個目的：

1. 爲了提供書中所談之 C++ 物件模型各種概念之具體說明。

2. 提供測試，以量測各種語言性質之相對成本。

無論哪一種意圖，都只是爲了展現物件模型。舉例而言，雖然我在書中有大量的舉例，但我並非建議一個真實的 3D graphic library 必須以虛擬繼承的方式來表現一個 3D 點（不過你可以在 [POKOR94] 中發現作者 Pokorny 的確這麼做）。

書中所有測試程式都在一部 SGI Indigo2xL 上編譯執行，使用 SGI 5.2 UNIX 作業系統中的 CC 和 NCC 編譯器。CC 是 cfront 3.0.1 版（它會產生出 C 碼，再由一個 C 編譯器重新編譯爲可執行檔）。NCC 是 Edison Design Group 的 C++ front-end 2.19 版，內含一個由 SGI 供應的程式碼產生器。至於時間量測，是採用 UNIX 的 timex 命令針對一千萬次迭代測試所得的平均值。

雖然在 xL 機器上使用這兩個編譯器，對讀者而言可能覺得有些神秘，我卻覺得對此書的目的而言，很好。不論是 cfront 或現在的 Edison Design Group's C++ front-end（Bjarne 稱其爲「cfront 的兒子」），都與平台無關。它們是一種一般化的編譯器，被授權給 34 家以上的電腦製造商（其中包括 Gray、SGI、Intel）和軟體開發環境廠商（包括 Centerline 和 Novell，後者是原先的 UNIX 軟體實驗室）。效率的量測並非爲了對目前市面上各家編譯系統做評比，而只是爲了提供 C++ 物件模型之各種特性的一個相對成本量測。至於商業評比的效率數據，你可以在幾乎任何一本電腦雜誌的電腦產品檢驗報告中獲得。

致謝

略

參考書目

[BALL92] Ball, Michael, "Inside Templates", C++ Report (September 1992)
[BALL93a] Ball, Michael, "What Are These Things Called Templates", C++ Report (February 1993)
[BALL93b] Ball, Michael, "Implementing Class Templates", C++ Report (September 1993)
[BOOCH93] Booch, Grady and Michael Vilot, "Simplifying the Booch Components", C++ Report (June 1993)
[BORL91] Borland Language Open Architecture Handbook, Borland International Inc., Scotts Valley, CA
[BOX95] Box, Don, "Building C++ Components Using OLE2", C++ Report (March/April 1995)
[BUDD91] Budd, Timothy, An Introduction to Object-Oriented Programming, Addison-Wesley Publishing Company, Reading, MA(1991)

[BUDGE92] Budge, Kent G., James S. Peery, and Allen C. Robinson, "High Performance Scientific Computing Using C++", Usenix C++ Conference Proceedings, Portland, OR(1992)
[BUDGE94] Budge, Kent G., James S. Peery, Allen C. Robinson, and Michael K. Wong, "Management of Class Temporaries in C++ Translation Systems", The Journal of C Language Translation (December 1994)
[CARROLL93] Carroll, Martin, "Design of the USL Standard Components", C++ Report (June 1993)
[CARROLL95] Carroll, Martin, and Margaret A. Ellis, "Designing and Coding Reusable C++, Addison-Wesley Publishing Company, Reading, MA(1995)
[CHASE94] Chase, David, "Implementation of Exception Handling, Part 1", The Journal of C Language Translation (June 1994)
[CLAM93a] Clamage, Stephen D., "Implementing New & Delete", C++ Report (May 1993)
[CLAM93b] Clamage, Stephen D., "Beginnings & Endings", C++ Report (September 1993)
[ELLIS90] Ellis, Margaret A. and Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley Publishing Company, Reading, MA(1990)
[GOLD94] Goldstein, Theodore C. and Alan D. Sloane, "The Object Binary Interface - C++ Objects for Evolvable Shared Class Libraries", Usenix C++ Conference Proceedings, Cambridge, MA(1994)
[HAM95] Hamilton, Jennifer, Robert Klarer, Mark Mendell, and Brian Thomson, "Using SOM with C++", C++ Report (July/August 1995)
[HORST95] Horstmann, Cay S., "C++ Compiler Shootout", C++ Report (July/August 1995)
[KOENIG90a] Koenig, Andrew and Stanley Lippman, "Optimizing Virtual Tables in C++ Release 2.0", C++ Report (March 1990)
[KOENIG90b] Koenig, Andrew and Bjarne Stroustrup, "Exception Handling for C++ (Revised)", Usenix C++ Conference Proceedings (April 1990)
[KOENIG93] Koenig, Andrew, "Combining C and C++", C++ Report (July/August 1993)
[ISO-C++95] C++ International Standard, Draft (April 28, 1995)
[LAJOIE94a] Lajoie, Josee, "Exception Handling: Supporting the Runtime Mechanism", C++ Report (March/April 1994)
[LAJOIE94b] Lajoie, Josee, "Exception Handling: Behind the Scenes", C++ Report (June 1994)
[LENKOV92] Lenkov, Dmitry, Don Cameron, Paul Faust, and Michey Mehta, "A Portable Implementation of C++ Exception Handling", Usenix C++ Conference Proceeding, Portland, OR(1992)
[LEA93] Lea, Doug, "The GNU C++ Library", C++ Report (June 1993)
[LIPP88] Lippman, Stanley and Bjarne Stroustrup, "Pointers to Class Members in C++", Implementor's Workshop, Usenix C++ Conference Proceedings (October 1988)
[LIPP91a] Lippman, Stanley, "Touring Cfront", C++ Journal, Vol.1, No.3 (1991)
[LIPP91b] Lippman, Stanley, "Touring Cfront: From Minutiae to Migraine", C++ Journal, Vol.1, No.4 (1991)

[LIPP91c] Lippman, Stanley, C++ Primer, Addison-Wesley Publishing Company, Reading, MA(1991)
[LIPP94a] Lippman, Stanley, "Default Constructor Synthesis", C++ Report (January 1994)
[LIPP94b] Lippman, Stanley, "Applying The Copy Constructor, Part1: Synthesis", C++ Report (February 1994)
[LIPP94c] Lippman, Stanley, "Applying The Copy Constructor, Part2", C++ Report (March/April 1994)
[LIPP94d] Lippman, Stanley, "Objects and Datum", C++ Report (June 1994)
[METAW94] MetaWare High C/C++ Language Reference Manual, Metaware Inc., Santa Crus, CA(1994)
[MACRO92] Jones, David and Martin J. O'Riordan, The Microsoft Object Mapping, Microsoft Corporation, 1992
[MOWBRAY95] Mowbray, Thomas J. and Ron Zahavi, The Essential Corba, John Wiley & Sons, Inc. (1995)
[NACK94] Nackman, Lee R., and John J. Barton Scientific and Engineering C++, An Introduction with Advanced Techniques and Examples, Addison-Wesley Publishing Company, Reading, MA(1994)
[PALAY92] Palay, Andrew J., "C++ in a Changing Environment", Usenix C++ Conference Proceedings, Portland, OR(1992)
[POKOR94] Pokorny, Cornel, Computer Graphics, Franklin, Beedle & Associates, Inc. (1994)
[PUGH90] Pugh, William and Grant Weddell, "Two-directional Record Layout for Multiple Inheritance", ACM SIGPLAN '90 Conference, White Plains, New York(1990)
[SCHMIDT94a] Schmidt, Douglas C., "A Domain Analysis of Network Daemon Design Dimensions", C++ Report (March/April 1994)
[SCHMIDT94b] Schmidt, Douglas C., "A Case Study of C++ Design Evolution", C++ Report (July/August 1994)
[SCHWARZ89] Schwarz, Jerry, "Initializing Static Variables in C++ Libraries", C++ Report (February 1989)
[STROUP82] Stroustrup, Bjarne, "Adding Classes to C: An Exercise in Language Evolution", Software: Practices & Experience, Vol.13 (1983)
[STROUP94] Stroustrup, Bjarne, "The Design and Evolution of C++", Addison-Wesley Publishing Company, Reading, MA(1994)
[SUN94a] The C++ Application Binary Interface, SunPro, Sun Microsystems, Inc.
[SUN94b] The C++ Application Binary Interface Rationale, SunPro, Sun Microsystems, Inc.
[VELD95] Veldhuizen, Todd, "Using C++ Template Metaprograms", C++ Report (May 1995)
[VINOS93] Vinoski, Steve, "Distributed Object Computing with CORBA", C++ Report (July/August 1993)
[VINOS94] Vinoski, Steve, "Mapping CORBA IDL into C++", C++ Report (September 1994)
[YOUNG95] Young, Douglas, Object-Oriented Programming with C++ and OSF/Motif, 2d ed., Prentice-Hall(1995)

第　章

導　讀
（譯者的話）

合適的讀者

很不容易三言兩語就說明此書的適當讀者。作者 Lippman 參與設計了全世界第一套 C++ 編譯器 cfront，這本書就是一位偉大的 C++ 編譯器設計者向你闡述他如何處理各種 explicit（明白出現於 C++ 程式碼）和 implicit（隱藏於程式碼背後）的 C++ 語意。

對於 C++ 程式老手，這必然是一本讓你大呼過癮的絕妙好書。

C++ 老手分兩類。一種人把語言用得爛熟，OO 觀念也有。另一種人不但如此，還對於檯面下的機制如編譯器合成的 default constructor 啦、object 的記憶體佈局啦...有莫大的興趣。本書對於第二類老手的吸引力自不待言，至於第一類老手，或許你沒那麼大的刨根究底的興趣，不過我還是非常推薦你閱讀此書。瞭解 C++ 物件模型，絕對有助於你在語言本身以及物件導向觀念兩方面的層次提昇。

你需要細細推敲每一個句子，每一個例子，囫圇吞棗是完全沒有用的。作者是 C++ 大師級人物，並且參與開發了第一套 C++ 編譯器，他的解說以及詮釋非常鞭辟入裏，你務必在看過每一小段之後，融會貫通，把思想觀念化爲己有，再接續另一小節。但閱讀次序並不需要按照書中的章節排列。

閱讀次序

我個人認爲，第 1, 3, 4 章最能帶給讀者立即而最大的幫助，這些都是經常引起程式員困惑的主題。作者在這些章節中有不少示意圖（我自己也加了不少）。你或許可以從這三章挑著看起。

其他章節比較晦澀一些（我的感覺），不妨「視可而擇之」。

當然，這都是十分主觀的認定。客觀的意見只有一個：你可以隨你的興趣與需求，從任一章開始看起。各章之間沒有必然關聯性。

翻譯風格

太多朋友告訴我，他們閱讀中文電腦書籍，不論是著作或譯作，最大的閱讀困難在於一大堆沒有標準譯名的技術名詞或習慣用語（至於那些誤謬不知所云的奇怪作品當然本就不在考慮之列）。其實，就算國立編譯館有統一譯名（或曾有過，誰知道？），流通於工業界與學術界之間的還是原文名詞與術語。

對於工程師，我希望我所寫的書和我所譯的書能夠讓各位讀來通體順暢；對於學生，我還希望多發揮一點引導的力量，引導各位多使用、多認識原文術語和專有名詞，不要說出像「無模式對話盒（modeless dialog）」這種奇怪的話。

由於本書讀者定位之故，我決定保留大量的原文技術名詞與術語。我清楚地知道，在我們的技術領域裡，研究人員或工程師如何使用這些語彙。

當然，有些中文譯名夠普遍，也夠有意義，我並不排除使用。其間的挑選與決定，不可避免地帶了點個人色彩。

下面是本書出現的原文名詞（按字母排序）及其意義：

英文名詞	中文名詞或（及）其意義
access level	存取層級。就是 C++ 的 public、private、protected 三種等級。
access section	存取區段。就是 class 中的 public、private、protected 三種段落。
alignment	邊界調整，調整至某些 bytes 的倍數。其結果視不同的機器而定。例如 32 位元機器通常調整至 4 的倍數。
bind	繫結，將程式中的某個符號真正附著（決議）至一塊實體上。
chain	串鏈
class	類別
class hierarchy	class 體系，class 階層架構
composition	組合。通常與繼承（inheriance）並同討論。
concrete inheritance	具體繼承（相對於抽象繼承）
constructor	建構式
data member	資料成員（亦或被稱爲 member variable）
declaration, declare	宣告
definition, define	定義（通常附帶「在記憶體中挖一塊空間」的行爲）
derived	衍生
destructor	解構式
encapsulation	封裝
explicit	明白的（通常指 C++ 程式碼中有出現的）
hierarchy	體系，階層架構
implement	實作（動詞）
implementation	實作品、實作物。本書有時候指 C++ 編譯器。大部份時候

英文名詞	中文名詞或（及）其意義
	是指 class member function 的內容。
implicit	隱含的、暗喻的（通常指未出現在 C++ 程式碼中的）
inheritance	繼承
inline	行內（C++ 的一個關鍵字）
instance	實體（有些書籍譯爲「案例」，極不妥當）
layout	佈局。本書常常出現這個字，意指 object 在記憶體中的資料分佈情況。
mangle	名稱切割重組（C++ 對於函式名稱的一種處理方式）
member function	成員函式。亦或被稱爲 function member。
members	成員，泛指 data members 和 member functions
object	物件（根據 class 的宣告而完成的一份佔有記憶體的實體）
offset	偏移位置
operand	運算元
operator	運算子
overhead	額外負擔（因某種設計，而導至的額外成本）
overload	多載
overloaded function	多載函式
override	改寫（對 virtual function 的重新設計）
paradigm	典範（請參考 #22 頁）
pointer	指標
polymorphism	多型（「物件導向」最重要的一個性質）
programming	程式設計、程式化
reference	參考、參用（動詞）。
reference	C++ 的 & 運算子所代表的東西。當名詞解。
resolve	決議。函式呼叫時聯結器所做的一種動作，將符號與函式實體產生關係。如果你呼叫 *func()* 而聯結時找不到 *func()* 實體，就會出現 "unresolved externals" 聯結錯誤。
slot	表格中的一格（一個元素）；條孔；條目；條格。

英文名詞	中文名詞或（及）其意義
subtype	子型別
type	型態，型別（指的是 int、float 等內建型別，或 C++ classes 等自定型別）
virtual	虛擬
virtual function	虛擬函式
virtual inheritance	虛擬繼承
virtual table	虛擬表格（爲實現虛擬機制而設計的一種表格，內放 virtual functions 的位址）

有時候考量到上下文的因素，面對同一個名詞，在譯與不譯之間，我可能會有不同的選擇。例如，面對 "pointer"，我會譯爲「指標」，但由於我並未將 reference 譯爲「參考」（實在不對味），所以如果原文是 "the manipulation of a pointer or reference in C++ ..."，爲了中英對等或平衡的緣故，我不會把它譯爲「C++ 中對於指標和 reference 的操作行爲...」，我會譯爲「C++ 中對於 pointer 和 reference 的操作行爲...」。

譯註

書中有一些譯註。大部份譯註，如果夠短的話，會被我直接放在括弧之中，接續本文。較長的譯註，則被我安排在被註文字的段落下面（緊臨，並加標示）。

勘誤錯誤

這本書雖說質地極佳，製作的嚴謹度卻不及格！有損 Lippman 的大師地位。

屬於「作者筆誤」之類的錯誤，比較無傷大雅，例如少了一個 ; 符號，或是多了一個 ; 符號，或是少了一個 } 符號，或是多了一個) 符號等等。比較嚴重的錯誤，是程式碼變數名稱或函式名稱或 class 名稱與文字敘述不一致，甚或是圖片中對於 object 佈局的畫法，與程式碼中的宣告不一致。這兩種錯誤都會嚴重耗費

讀者的心神。

只要是被我發現的錯誤，都已被我修正。以下是錯誤更正列表。

◆ 示例：L5 表示第 5 行，L-9 表示倒數第 9 行。頁碼所示爲原書頁碼。

頁碼	原文位置	原文內容	應修改爲
p.35	最後一行	Bashful(),	Bashful();
p.57	表格第二行	1.32.36	1:32.36
p.61	L1	memcpy... 程式碼最後少一個)	
p.61	L10	Shape()...程式碼最後少了一個 }	
p.64	L-9	程式碼最後多了一個 ;	
p.78	最後四行碼	==	似乎應爲 =
p.84	圖 3.1b 說明	struct Point3d	class Point3d
p.87	L-2	virtual... 程式碼最後少了一個 ;	
p.87	全頁多處	pc2_2（不符合命名意義）	pc1_2（符合命名意義）
p.90	圖 3.2a 說明	Vptr placement and end of class	Vptr placement at end of class
p.91	圖 3.2(b)	__vptr__has_vrts	__vptr__has_virts
p.92	碼 L-7	class Vertex2d	class Vertex3d
p.92	碼 L-6	public Point2d	public Point3d
p.93	圖 3.4 說明	Vertex2d 的物件佈局	Vertex3d 的物件佈局
p.92~ p.94		符號名稱混亂，前後誤謬不符	已全部更改過
p.97	碼 L2	public Point3d, public Vertex	配合圖 3.5ab，應調整次序爲 public Vertex, public Point3d
p.99	圖 3.5(a)	符號與書中程式碼多處不符	已全部更改過
p.100	圖 3.5(b)	符號與書中程式碼多處不符	已全部更改過
p.100	L-2	? pv3d + ... 最後多了一個)	
p.106	L16	pt1d::y	pt2d::_y
p.107	L10	& 3d_point::z;	&Point3d::z;

頁碼	原文位置	原文內容	應修改爲
p.108	L6	& 3d_point::z;	&Point3d::z;
p.108	L-6	int d::*dmp, d *pd	int Derived::*dmp, Derived *pd
p.109	L1	d *pd	Derived *pd
p.109	L4	int b2::*bmp = &b2::val2;	int Base2::*bmp = &Base2::val2;
p.110	L2	不符合稍早出現的程式碼	把 pt3d 改爲 Point3d
p.115	L1	magnitude()	magnitude3d()
p.126	L12	Point2d pt2d = new Point2d;	ptr = new Point2d;
p.136	圖 4.2 右下	Derived::~close()	Derived::close()
p.138	L-12	class Point3d... 最後少一個 {	
p.140	程式碼	沒有與文字中的 class 命名一致	所有的 pt3d 改爲 Point3d
p.142	L-7	if (this ... 程式碼最右邊少一個)	
p.143	程式碼	沒有與文字中的 class 命名一致	所有的 pt3d 改爲 Point3d
p.145	L-6	pointer::z()	Point::z()
p.147	L1	pointer::*pmf	Point::*pmf
p.147	L5	point::x()	Point::x()
p.147	L6	point::z()	Point::z()
p.147	中段碼 L-1	程式碼最後缺少一個)	
p.148	中段碼 L1	(ptr->*pmf) 函式最後少一個 ;	
p.148	中段碼 L-1	(*ptr->vptr[.. 函式最後少一個)	
p.150	程式碼	沒有與文字中的 class 命名一致	所有的 pt3d 改爲 Point3d
p.150	L-7	pA.__vptr__pt3d... 最後少一個 ;	
p.152	L4	point new_pt;	Point new_pt;
p.156	L7	{	}
p.160	L11, L12	Abstract_Base	Abstract_base
p.162	L-3	Abstract_base 函式最後少一個 ;	
p.166	中，碼 L3	Point1 local1 = ...	Point local1 = ...
p.166	中，碼 L4	Point2 local2;	Point local2;

頁碼	原文位置	原文內容	應修改爲
p.174	中，碼 L-1	Line::Line() 函式最後多了一個 ；	
p.174	中下，碼 L-1	Line::Line() 函式最後多了一個 ；	
p.175	中上，碼 L-1	Line::~Line() 函式最後多一個 ；	
p.182	中下，碼 L6	Point3d::Point3d()	PVertex::PVertex()
p.183	上，碼 L9	Point3d::Point3d()	PVertex::PVertex()
p.185	上，碼 L3	y = 0.0 之前缺少 float	
p.186	中下，碼 L6	缺少一個 return	
p.187	中，碼 L3	const Point3d &p	const Point3d &p3d
p.204	下，碼 L3	缺少一個 return 1;	
p.208	中下，碼 L2	new Pvertex;	new PVertex;
p.219	上，碼 L1	__nw(5*sizeof(int));	__new(5*sizeof(int));
p.222	上，碼 L8	// new (ptr_array... 程式碼少個 ；	
p.224	中，碼 L1	Point2w ptw = ...	Point2w *ptw = ...
p.224	下，碼 L5	operator new() 函式定義多一個 ；	
p.225	上，碼 L2	Point2w ptw = ...	Point2w *ptw = ...
p.226	下，碼 L1	Point2w p2w = ...	Point2w *p2w = ...
p.229	中，碼 L1	c.operator==(a + b);	c.operator=(a + b);
p.232	中下，碼 L2	x xx;	X xx;
p.232	中下，碼 L3	x yy;	X yy;
p.232	下，碼 L2	struct x _1xx;	struct X _1xx;
p.232	下，碼 L3	struct x _1yy;	struct X _1yy;
p.233	碼 L2	struct x __0__Q1;	struct X __0__Q1;
p.233	碼 L3	struct x __0__Q2;	struct X __0__Q2;
p.233	中，碼	if 條件句的最後多了一個 ；	
p.253	碼 L-1	foo() 函式碼最後多了一個 ；	

推薦

我個人翻譯過不少書籍，每一本都精挑細選後才動手（品質不夠的原文書，譯它做啥?!）在這麼些譯本當中，我從來不做直接而露骨地推薦。好的書籍自然而然會得到識者的欣賞。過去我譯的那些明顯具有實用價值的書籍，總有相當數量的讀者有強烈的需求，所以我從不擔心沒有足夠的人來爲好書散播口碑。但 Lippman 的這本書不一樣，它可能不會爲你帶來明顯而立即的實用性，它可能因此在書肆中蒙上一層灰（其原文書我就沒聽說多少人讀過），枉費我從眾多原文書中挑出這本好書。我擔心聽到這樣的話：

物件模型？呃，我會寫 C++ 程式，寫得一 級棒，這些屬於編譯器層面的東西，於我何干哉！

物件模型是深層結構的知識，關係到「與語言無關、與平台無關、跨網路可執行」軟體元件（software component）的基礎原理。也因此，瞭解 C++ 物件模型，是學習目前軟體元件三大規格（COM、CORBA、SOM）的技術基礎。

如果你對軟體元件（software component）沒有興趣，C++ 物件模型也能夠使你對虛擬函式、虛擬繼承、虛擬介面有脫胎換骨的新體認，或是對於各種 C++ 寫法所帶來的效率利益有通盤的認識。

我因此要大聲地說：有經驗的 C++ programmer 都應該看看這本書。

如果您對 COM 有興趣，我也要同時推薦你看另一本書：***Essential COM***，Don Box 著，Addison Wesley 1998 出版（COM 本質論，侯捷譯，碁峰 1998）。這也是一本論述非常清楚的書籍，把 COM 的由來（爲什麼需要 COM、如何使用 COM）以循序漸進的方式闡述得非常深刻，是我所看過最理想的一本 COM 基礎書籍。

看 ***Essential COM*** 之前，你最好有這本 ***Inside The C++ Object Model*** 的基礎。

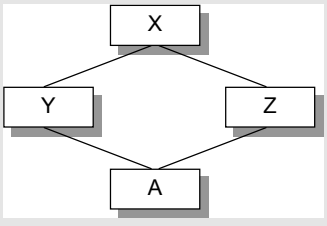
第　章

Data 語意學
（The Semantics of Data）

前些時候我收到一封來自法國的電子郵件，發信人似乎有些迷惘也有些煩亂。他志願（要不就是被選派）爲他的專案團隊提供一個「永恆的」library。在做準備工作的時候，他寫出以下的碼並列印出它們的 *sizeof* 結果：

```
class X { };
class Y : public virtual X { };
class Z : public virtual X { };
class A : public Y, public Z { };
```

譯註：X, Y, Z, A 的繼承關係如下圖所示：



上述 *X, Y, Z, A* 中沒有任何一個 class 內含明顯的資料，其間只表示了繼承關係。所以發信者認爲每一個 class 的大小都應該是 0。當然不對！即使是 class *X* 的大小也不爲 0：

```
sizeof X 的結果爲 1
sizeof Y 的結果爲 8
sizeof Z 的結果爲 8
sizeof A 的結果爲 12
```

譯註：以下是我在 Visual C++ 5.0 上的執行結果

```
sizeof X 的結果爲 1
sizeof Y 的結果爲 4
sizeof Z 的結果爲 4
sizeof A 的結果爲 8
```

原因將在 p.86 的譯註和 p.87 的正文中解釋

讓我們依序看看每一個 class 的宣告，並看看它們爲什麼獲得上述結果。

一個空的 class 如：

```
// sizeof X == 1
class X { };
```

事實上並不是空的，它有一個隱藏的 1 byte 大小，那是被編譯器安插進去的一個 char。這使得此一 class 的兩個 objects 得以在記憶體中配置獨一無二的位址：

```
X a, b;
if (&a == &b) cerr << "yipes!" << endl;
```

令來信讀者感到驚訝和沮喪的，我懷疑是 *Y* 和 *Z* 的 *sizeof* 結果：

```
// sizeof Y == sizeof Z == 8
class Y : public virtual X { };
class Z : public virtual X { };
```

在來信者的機器上，*Y* 和 *Z* 的大小都是 8。這個大小和機器有關，也和編譯器有關。事實上 *Y* 和 *Z* 的大小受到三個因素的影響：

1. **語言本身所造成的額外負擔（overhead）** 當語言支援 virtual base classes，就會導至一些額外負擔。在 derived class 中，這個額外負擔反映在某種型式的指標身上，它或者指向 virtual base class subobject，或者指向一個相關表格；表格中存放的若不是 virtual base class subobject 的位址，就是其偏移位置（offset）。在來信者的機器上，指標是 4 bytes

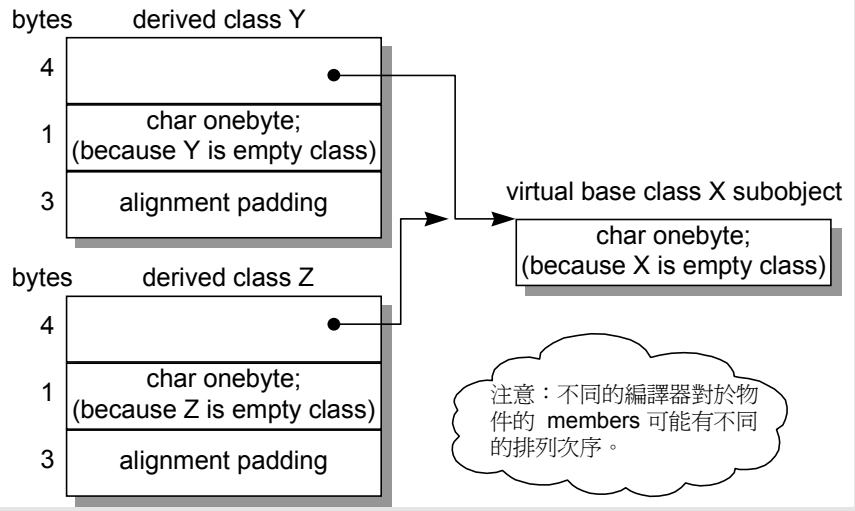
（我將在 3.4 節討論 virtual base class）。

2. **編譯器對於特殊情況所提供的最佳化處理**　Virtual base class *X* subobject 的 1 bytes 大小也出現在 class *Y* 和 *Z* 身上。傳統上它被放在 derived class 的固定（不變動）部份的尾端。某些編譯器會對 empty virtual base class 提供特殊支援（以下第三點之後的一段文字對此有比較詳細的討論）。來信讀者所使用的編譯器，顯然並未提供這項特殊處理。

3. **Alignment 的限制**　class *Y* 和 *Z* 的大小截至目前爲 5 bytes。在大部份機器上，群聚的結構體大小會受到 alignment 的限制，使它們能夠更有效率地在記憶體中被存取。在來信讀者的機器上，alignment 是 4 bytes，所以 class *Y* 和 *Z* 必須填補 3 bytes。最終得到的結果就是 8 bytes。

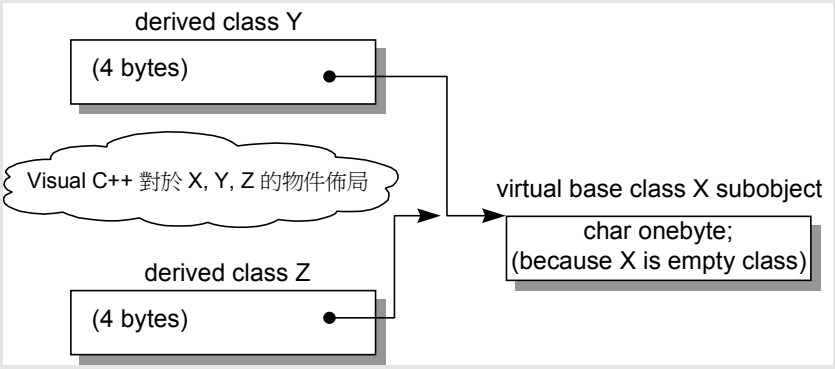
譯註：alignment 就是將數值調整到某數的整數倍。在 32 位元電腦上，通常 alignment 爲 4 bytes（32 位元），以使 bus 的「運輸量」達到最高效率。

譯註：我以下圖表現上述的 *X, Y, Z* 物件佈局



Empty virtual base class 已經成爲 C++ OO 設計的一個特有術語了。它提供一個 virtual interface，沒有定義任何資料。某些新近的編譯器（譯註）對此提供了特殊處理（請看 [SUN94a]）。在這個策略之下，一個 empty virtual base class 被視爲 derived class object 最開頭的一部份，也就是說它並沒有花費任何的額外空間。這就節省了上述第 2 點的 1 bytes （譯註：因爲既然有了 members，就不需要原本爲了 empty class 而安插的一個 char），也就不再需要第 3 點所說的 3 bytes 的填補。只剩下第 1 點所說的額外負擔。在此模型下，*Y* 和 *Z* 的大小都是 4 而不是 8。

譯註：Visual C++ 就是上述這一類型的編譯器。我以下圖來表現 Visual C++ 對於 class *X*, *Y*, *Z* 的物件佈局：



編譯器之間的潛在差異正說明了 C++ 物件模型的演化。這個模型爲一般情況提供了解決之道。當特殊情況逐漸被挖掘出來，種種啓發（嘗試錯誤）法於是被引入，提供最佳化的處理。如果成功，啓發法於是就提昇爲普遍的策略，並跨越各種實作品而合併。它被視爲標準（雖然它並不被規範爲標準），久而久之也就成了語言的一部份。Virtual function table 就是一個好例子，另一個例子是第 2 章討論過的「named return value（NRV）最佳化」。

那麼，你期望 class *A* 的大小是什麼呢？很明顯，某種程度上必須視你所使用的編譯器而定。首先，請你考慮那種並未特別處理 empty virtual base class 的編譯器。如果我們忘記 *Y* 和 *Z* 都是「虛擬衍生」自 class *X*，我們可能會回答 16，畢竟 *Y* 和 *Z* 的大小都是 8。然而當我們對 class *A* 施以 *sizeof* 運算子，得到的答案竟然是 12。到底怎麼回事？

記住，一個 virtual base class subobject 只會在 derived class 中存在一份實體，不管它在 class 繼承體系中出現了多少次！class *A* 的大小由下列數點決定：

- 被大家共享的唯一一個 class *X* 實體，大小爲 1 byte。
- Base class *Y* 的大小，減去「因 virtual base class *X* 而配置」的大小，結果是 4 bytes。Base class *Z* 的算法亦同。加起來是 8 bytes。
- class *A* 自己的大小：0 byte。
- class *A* 的 alignment 數量（如果有的話）。前述三項總合，表示調整前的大小是 9 bytes。class *A* 必須調整至 4 bytes 邊界，所以需要填補 3 bytes。結果是 12 bytes。

現在如果我們考慮那種「特別對 empty virtual base class 做了處理」的編譯器呢？一如前述，class *X* 實體的那 1 byte 將被拿掉，於是額外的 3 bytes 填補額也不必了，因此 class *A* 的大小將是 8 bytes。注意，如果我們在 virtual base class *X* 中放置一個（以上）的 data members，兩種編譯器（「有特殊處理」者和「沒有特殊處理」者）就會產生出完全相同的物件佈局。

C++ Standard 並不強制規定如「base class subobjects 的排列次序」或「不同存取層級的 data members 的排列次序」這種瑣碎細節。它也不規定 virtual functions 或 virtual base classes 的實作細節。C++ Standard 只說：那些細節由各家廠商自定。我在本章以及全書中，都會區分「C++ Standard」和「目前的 C++ 實作標準」兩種討論。

在這一章中，class 的 data members 以及 class hierarchy 是中心議題。一個 class 的 data members，一般而言，可以表現這個 class 在程式執行時的某種狀態。Nonstatic data members 放置的是「個別的 class object」感興趣的資料，static data members 則放置的是「整個 class」感興趣的資料。

C++ 物件模型儘量以空間最佳化和存取速度最佳化的考量來表現 nonstatic data members，並且保持和 C 語言 struct 資料配置的相容性。它把資料直接存放在每一個 class object 之中。對於繼承而來的 nonstatic data members（不管是 virtual 或 nonvirtual base class）也是如此。不過並沒有強制定義其間的排列順序。至於 static data members 則被放置在程式的一個 global data segment 中，不會影響個別的 class object 的大小。在程式之中，不管該 class 被產生出多少個 objects（經由直接產生或間接衍生），static data members 永遠只存在一份實體（譯註：甚至即使該 class 沒有任何 object 實體，其 static data members 也已存在）。但是一個 template class 的 static data members 的行爲稍有不同，7.1 節有詳細的討論。

每一個 class object 因此必須有足夠的大小以容納它所有的 nonstatic data members。有時候其值可能令你吃驚（正如那位法國來信者），因爲它可能比你想像的還大，原因是：

1. 由編譯器自動加上的額外 data members，用以支援某些語言特性（主要是各種 virtual 特性）。
2. 因爲 alignment（邊界調整）的需要。

3.1 Data Member 的繫結（The Binding of a Data Member）

考慮下面這段程式碼：

```
// 某個 foo.h 表頭檔，從某處含入
extern float x;
```

```
// 程式員的 Point3d.h 檔案
class Point3d
{
public:
    Point3d( float, float, float );
    // 問題：被傳回和被設定的 x 是哪一個 x 呢？
    float X() const { return x; }
    void X( float new_x ) const { x =  new_x; }
    // ...
private:
    float x, y, z;
};
```

如果我問你 *Point3d::X()* 傳回哪一個 *x*？是 class 內部那個 *x*，還是外部（extern）那個 *x*？今天每個人都會回答我是內部那一個。這個答案是正確的，但並不是從過去以來一直都正確！

在 C++ 最早的編譯器上，如果在 *Point3d::X()* 的兩個函式實體中對 *x* 做出參閱（取用）動作，這動作將會指向 global *x* object！這樣的繫結結果幾乎普遍地不在大家的預期之中，並因此導出早期 C++ 的兩種防禦性程式設計風格：

1. 把所有的 data members 放在 class 宣告起頭處，以確保正確的繫結：

```
class Point3d
{
    // 防禦性程式設計風格 #1
    // 在 class 宣告起頭處先放置所有 data member
    float x, y, z;
public:
    float X() const { return x; }
    // ... etc. ...
};
```

2. 把所有的 inline functions，不管大小都放在 class 宣告之外：

```
class Point3d
{
public:
    // 防禦性程式設計風格 #2
    // 把所有的 inlines 都移到 class 之外
    Point3d();
```

```
    float X() const;
    void X( float ) const;
    // ... etc. ...
};

inline float
Point3d::
X() const
{
    return x;
}

// ... etc. ...
```

這些程式設計風格事實上到今天還存在，雖然它們的必要性已經自從 C++ 2.0 之後（伴隨著 ***C++ Reference Manual*** 的修訂）就消失了。這個古早的語言規則被稱爲 "member rewriting rule"，大意是「一個 inline 函式實體，在整個 class 宣告未被完全看見之前，是不會被評估求值（evaluated）的」。C++ Standard 以 "member scope resolution rules" 來精鍊這個 "rewriting rule"，其效果是，如果一個 inline 函式在 class 宣告之後立刻被定義的話，那麼就還是對其評估求值（evaluate）。也就是說，當一個人寫下這樣的碼：

```
extern int x;

class Point3d
{
public:
    // 對於函式本體的分析將延遲直至
    //   class 宣告的右大括弧出現才開始。
    float X() const { return x; }
    // ...
private:
    float x;
};

// 事實上，分析在這裡進行
```

對 member functions 本體的分析，會直到整個 class 的宣告都出現了才開始。因此在一個 inline member function 軀體之內的一個 data member 繫結動作，會在整

個 class 宣告完成之後才發生。

然而，這對於 member function 的 argument list 並不爲真。Argument list 中的名稱還是會在它們第一次遭遇時被適當地決議（resolved）完成。因此在 extern 和 nested type names 之間的非直覺繫結動作還是會發生。例如在下面的程式片段中，*length* 的型別在兩個 member function signatures 中都決議（resolve）爲 global typedef，也就是 int。當後續再有 *length* 的 nested typedef 宣告出現，C++ Standard 就把稍早的繫結標示爲非法：

```
typedef int length;

class Point3d
{
public:
    // 喔歐: length 被決議（resolved）爲 global
    // 沒問題: _val 被決議（resolved）爲 Point3d::_val
    void mumble( length val ) { _val = val; }
    length mumble() { return _val; }
    // ...

private:
    // length 必須在「本 class 對它的第一個參考動作」之前被看見。
    // 這樣的宣告將使先前的參考動作不合法。
    typedef float length;
    length _val;
    // ...
};
```

上述這個語言狀況，仍然需要某種防禦性程式風格：請總是把「nested type 宣告」放在 class 的起始處。在上述例子中，如果把 *length* 的 nested typedef 定義於「在 class 中被參考」之前，就可以確保非直覺繫結的正確性。

3.2 Data Member 的佈局（Data Member Layout）

已知下面一組 data members：

```
class Point3d {
public:
    // ...
private:
    float x;
    static List<Point3d*> *freeList;
    float y;
    static const int chunkSize = 250;
    float z;
};
```

Nonstatic data members 在 class object 中的排列順序將和其被宣告的順序一樣，任何中間介入的 static data members 如 *freeList* 和 *chunkSize* 都不會被放進物件佈局之中。在上述例子裡，每一個 *Point3d* 物件是由三個 float 組成，次序是 *x*, *y*, *z*。static data members 存放在程式的 data segment 中，和個別的 class objects 無關。

C++ Standard 要求，在同一個 access section（也就是 private、public、protected 等區段）中，members 的排列只需符合「較晚出現的 members 在 class object 中有較高的位址」這一條件即可（請看 C++ Standard 9.2 節）。也就是說各個 members 並不一定得連續排列。什麼東西可能會介於被宣告的 members 之間呢？members 的邊界調整（alignment）可能就需要填補一些 bytes。對於 C 和 C++ 而言這的確是真的，對目前的 C++ 編譯器實作情況而言，這也是真的。

編譯器還可能會合成一些內部使用的 data members，以支援整個物件模型。vptr 就是這樣的東西，目前所有的編譯器都把它安插在每一個「內含 virtual function 之 class」的 object 內。vptr 會被放在什麼位置呢？傳統上它被放在所有明白宣告的 members 的最後頭。不過如今也有一些編譯器把 vptr 放在一個 class object 的最

前端。C++ Standard 秉持先前所說的「對於佈局所持的放任態度」，允許編譯器把那些內部產生出來的 members 自由放在任何位置上，甚至放在那些被程式員宣告出來的 members 之間。

C++ Standard 也允許編譯器將多個 access sections 之中的 data members 自由排列，不必在乎它們出現在 class 宣告中的次序。也就是說，下面這樣的宣告：

```
class Point3d {
public:
    // ...
private:
    float x;
    static List<Point3d*> *freeList;
private:
    float y;
    static const int chunkSize = 250;
private:
    float z;
};
```

其 class object 的大小和組成都和我們先前宣告的那個相同，但是 members 的排列次序則視編譯器而定。編譯器可以隨意把 *y* 或 *z* 或什麼東西放第一個，不過就我所知道，目前沒有任何編譯器會這麼做。

目前各家編譯器都是把一個以上的 access sections 連鎖在一起，依照宣告的次序，成爲一個連續區塊。Access sections 的多寡並不會招來額外負擔。例如在一個 section 中宣告 8 個 members，或是在 8 個 sections 中總共宣告 8 個 members，得到的 object 大小是一樣的。

下面這個 template function，接受兩個 data members，然後判斷誰先出現在 class object 之中。如果兩個 members 都是不同的 access sections 中的第一個被宣告者，此函式就可以用來判斷哪一個 section 先出現（如果你對 class member 的指標並不熟悉，請參考 3.6 節）：

```
template< class class_type,
          class data_type1,
          class data_type2 >
char*
access_order(
   data_type1 class_type::*mem1,
   data_type2 class_type::*mem2 )
{
   assert (mem1 != mem2 );
   return
       mem1 < mem2
          ? "member 1 occurs first"
          : "member 2 occurs first";
}
```

上述函式可以這樣被喚起：

```
access_order( &Point3d::z, &Point3d::y);
```

於是 *class_type* 會被繫結爲 *Point3d*，而 *data_type1* 和 *data_type2* 會被繫結爲 float。

3.3 Data Member 的存取

已知下面這段程式碼：

```
Point3d origin;
origin.x = 0.0;
```

你可能會問 *x* 的存取成本是什麼？答案視 *x* 和 *Point3d* 如何宣告而定。*x* 可能是個 static member，也可能是個 nonstatic member。*Point3d* 可能是個獨立（非衍生）的 class，也可能是從另一個單一的 base class 衍生而來；雖然可能性不高，但它甚至可能是多重繼承或虛擬繼承而來。下面數節將依次檢驗每一種可能性。

在開始之前，讓我先丟出一個問題。如果我們有兩個定義，*origin* 和 *pt*：

```
Point3d origin, *pt = &origin;
```

我用它們來存取 data members，像這樣：

```
origin.x = 0.0;
pt->x = 0.0;
```

透過 *origin* 存取，和透過 *pt* 存取，有什麼重大差異嗎？如果你的回答是 yes，請你從 class *Point3d* 和 data member *x* 的角度來說明差異的發生因素。我會在這一節結束前重返這個問題並提出我的答案。

Static Data Members

Static data members，按其字面意義，被編譯器提出於 class 之外，一如我在 1.1 節所說，並被視爲一個 global 變數（但只在 class 生命範圍之內可見）。每一個 member 的存取許可（譯註：private 或 protected 或 public），以及與 class 的關聯，並不會招致任何空間上或執行時間上的額外負擔 -- 不論是在個別的 class objects 或是在 static data member 本身。

每一個 static data member 只有一個實體，存放在程式的 data segment 之中。每次程式參閱（取用）static member，就會被內部轉化爲對該唯一之 extern 實體的直接參考動作。例如：

```
// origin.chunkSize == 250;
Point3d::chunkSize == 250;       // 譯註：我想作者的意思可能是要說
                                 //  Point3d::chunkSize = 250;
// pt->chunkSize == 250;
Point3d::chunkSize == 250;       // 譯註：我想作者的意思可能是要說
                                 //  Point3d::chunkSize = 250;
```

從指令執行的觀點來看，這是 C++ 語言中「透過一個指標和透過一個物件來存取 member，結論完全相同」的唯一一種情況。這是因爲「經由 member selection operators（譯註：也就是 '.' 運算子）對一個 static data member 做存取動作」只是文法上的一種便宜行事而已。member 其實並不在 class object 之中，因此存取 static members 並不需要透過 class object。

但如果 *chunkSize* 是一個從複雜繼承關係中繼承而來的 member，又當如何？或許它是一個「virtual base class 的 virtual base class」（或其他同等複雜的繼承架構）的 member 也說不定。哦，那無關緊要，程式之中對於 static members 還是只有唯一一個實體，而其存取路徑仍然是那麼直接。

如果 static data member 的存取是經由函式呼叫，或其他某些語法呢？舉個例子，如果我們寫：

```
foobar().chunkSize == 250;        // 譯註：我想作者的意思可能是要說
                                  //  foobar().chunkSize = 250;
```

喚起 *foobar()* 會發生什麼事情？在 C++ 的準標準（pre-Standard）規格中，沒有人知道會發生什麼事，因爲 ARM 並未指定 *foobar()* 是否必須被求值（evaluated）。cfront 的作法是簡單地把它丟掉。但 C++ Standard 明白要求 *foobar()* 必須被求值（evaluated），雖然其結果並無用處。下面是一種可能的轉化：

```
// foobar().chunkSize == 250;  // 譯註：我想作者的意思是要說
                               // foobar().chunkSize = 250;

// evaluate expression, discarding result
(void) foobar();
Point3d.chunkSize == 250;      // 譯註：我想作者的意思是要說
                               // Point3d.chunkSize = 250;
```

若取一個 static data member 的位址，會得到一個指向其資料型別的指標，而不是一個指向其 class member 的指標，因爲 static member 並不內含在一個 class object 之中。例如：

```
&Point3d::chunkSize;
```

會獲得型態如下的記憶體位址：

```
const int*
```

如果有兩個 classes，每一個都宣告了一個 static member *freeList*，那麼當它們都被放在程式的 data segment，就會導至名稱衝突。編譯器的解決方法是暗中對每一個 static data member 編碼（這種手法有個很美的名稱：name-mangling），以獲得一個獨一無二的程式識別代碼。有多少個編譯器，就有多少種 name-mangling 作法！通常不外乎是表格啦、文法措辭啦等等。任何 name-mangling 作法都有兩個重點：

1. 一個演算法，推導出獨一無二的名稱。
2. 萬一編譯系統（或環境工具）必須和使用者交談，那些獨一無二的名稱可以輕易被推導回到原來的名稱。

Nonstatic Data Members

Nonstatic data members 直接存放在每一個 class object 之中。除非經由明白的（explicit）或暗喻的（implicit）class object，沒有辦法直接存取它們。只要程式員在一個 member function 中直接處理一個 nonstatic data member，所謂"implicit class object" 就會發生。例如下面這段碼：

```
Point3d
Point3d::translate( const Point3d &pt) {
    x += pt.x;
    y += pt.y;
    z += pt.z;
}
```

表面上所看到的對於 *x, y, z* 的直接存取，事實上是經由一個 "implicit class object"（由 *this* 指標表達）完成。事實上這個函式的參數是：

```
// member function 的內部轉化
Point3d
Point3d::translate( Point3d *const this, const Point3d &pt) {
    this->x += pt.x;
    this->y += pt.y;
    this->z += pt.z;
}
```

Member functions 在本書第 4 章有比較詳細的討論。

欲對一個 nonstatic data member 做存取動作，編譯器需要把 class object 的起始位址加上 data member 的偏移位置（offset）。舉個例子，如果：

```
origin._y = 0.0;
```

那麼位址 *&origin._y* 將等於：

```
&origin + (&Point3d::_y - 1);
```

請注意其中的 -1 動作。指向 data member 的指標，其 offset 值總是被加上 1，這樣可以使編譯系統區分出「一個指向 data member 的指標，用以指出 class 的第一個 member」和「一個指向 data member 的指標，沒有指出任何 member」兩種情況。「指向 data members 的指標」將在 3.6 節有比較詳細的討論。

每一個 nonstatic data member 的偏移位置（offset）在編譯時期即可獲知，甚至如果 member 屬於一個 base class subobject（衍生自單一或多重繼承串鏈）也是一樣。因此，存取一個 nonstatic data member，其效率和存取一個 C struct member 或一個 nonderived class 的 member 是一樣的。

現在讓我們看看虛擬繼承。虛擬繼承將爲「經由 base class subobject 存取 class members」導入一層新的間接性，譬如：

```
Point3d *pt3d;
pt3d->_x = 0.0
```

其執行效率在 *_x* 是一個 struct member、一個 class member、單一繼承、多重繼承的情況下都完全相同。但如果 *_x* 是一個 virtual base class 的 member，存取速度會比較慢一點。下一節我會驗證「繼承對於 member 佈局的影響」。在我們尚未進行到那裡之前，請回憶本節一開始的一個問題：以兩種方法存取 *x* 座標，像這樣：

```
origin.x = 0.0;
pt->x = 0.0;
```

「從 *origin* 存取」和「從 *pt* 存取」有什麼重大的差異？答案是「當 *Point3d* 是一個 derived class，而其繼承架構中有一個 virtual base class，並且被存取的 member（如本例的 *x*）是一個從該 virtual base class 繼承而來的 member」時，就會有重大的差異。這時候我們不能夠說 *pt* 必然指向哪一種 class type（因此我們也就不知道編譯時期這個 member 真正的 offset 位置），所以這個存取動作必須延遲至執行時期，經由一個額外的間接導引，才能夠解決。但如果使用 *origin*，就不會有這些問題，其型態無疑是 *Point3d* class，而即使它繼承自 virtual base class，members 的 offset 位置也在編譯時期就固定了。一個積極進取的編譯器甚至可以靜態地經由 *origin* 就解決掉對 *x* 的存取。

3.4「繼承」與 Data Member

在 C++ 繼承模型中，一個 derived class object 所表現出來的東西，是其自己的 members 加上其 base class(es) members 的總合。至於 derived class members 和 base class(es) members 的排列次序並未在 C++ Standard 中強制指定；理論上編譯器可以自由安排之。在大部份編譯器上頭，base class members 總是先出現，但屬於 virtual base class 的除外（一般而言，任何一條通則一旦碰上 virtual base class 就沒輒兒，這裡亦不例外）。

瞭解這種繼承模型之後，你可能會問，如果我爲 2D（二維）或 3D（三維）座標點提供兩個抽象資料型態如下：

```
// supporting abstract data types
class Point2d {
public:
    // constructor(s)
    // operations
    // access functions
private:
    float x, y;
};
```

Point2d

Point3d

```
class Point3d {
public:
    // constructor(s)
    // operations
    // access functions
private:
    float x, y, z;
};
```

這和「提供兩層或三層繼承架構，每一層（代表一個維度）是一個 class，衍生自較低維層次」有什麼不同？下面各小節的討論將涵蓋「單一繼承且不含 virtual functions」、「單一繼承並含 virtual functions」、「多重繼承」、「虛擬繼承」等四種情況。**圖 3.1a** 就是 *Point2d* 和 *Point3d* 的物件佈局圖，在沒有 virtual functions 的情況下（如本例），它們和 C struct 完全一樣。

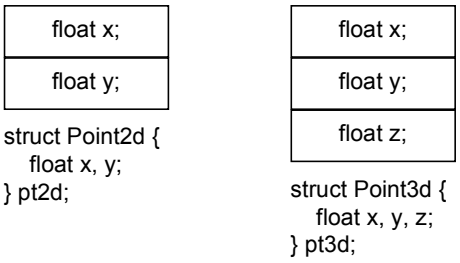
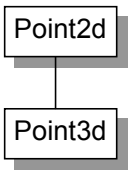


圖 3.1a 個別 structs 的資料佈局

無繼承不無多型（Inheritance without Polymorphism）

想像一下，程式員或許希望，不論是 2D 或 3D 座標點，都能夠共享同一個實體，但又能夠繼續使用「與型別性質相關（所謂 type-specific）」的實體。我們有一個設計策略，就是從 *Point2d* 衍生出一個 *Point3d*，於是 *Point3d* 將繼承 *x* 和 *y* 座標的一切（包括資料實體和操作方法）。帶來的影響則是可以共享「資料本身」以及「資料的處理方法」，並將之區域化。一般而言，具體繼承（concrete inheritance，譯註：相對於虛擬繼承 virtual inheritance）並不會增加空間或存取時間上的額外負擔。

```
class Point2d {
public:
    Point2d( float x = 0.0, float y = 0.0 )
        : _x( x ), _y( y ) { };

    float x() { return _x; }
    float y() { return _y; }

    void x( float newX ) { _x = newX; }
    void y( float newY ) { _y = newY; }

    void operator+=( const Point2d& rhs ) {
        _x += rhs.x();
        _y += rhs.y();
    }
    // ... more members

protected:
    float _x, _y;
};
```




```
// inheritance from concrete class
class Point3d : public Point2d {
public:
    Point3d( float x = 0.0, float y = 0.0, float z = 0.0 )
        : Point2d( x, y ), _z( z ) { };

    float z() { return _z; }
    void z( float newZ ) { _z = newZ; }

    void operator+=( const Point3d& rhs ) {
        Point2d::operator+=( rhs );
        _z += rhs.z();
    }
    // ... more members

protected:
    float _z;
};
```

這樣子設計的好處就是可以把管理 x 和 y 座標的程式碼區域化。此外這個設計可以明顯表現出兩個抽象類別之間的緊密關係。當這兩個 classes 獨立的時候，

Point2d object 和 *Point3d* object 的宣告和使用都不會有所改變。所以這兩個抽象類別的使用者不需要知道 objects 是否爲獨立的 classes 型態，或是彼此之間有繼承的關係。圖 **3.1b** 顯示 *Point2d* 和 *Point3d* 繼承關係的實物佈局，其間並沒有宣告 virtual 介面。

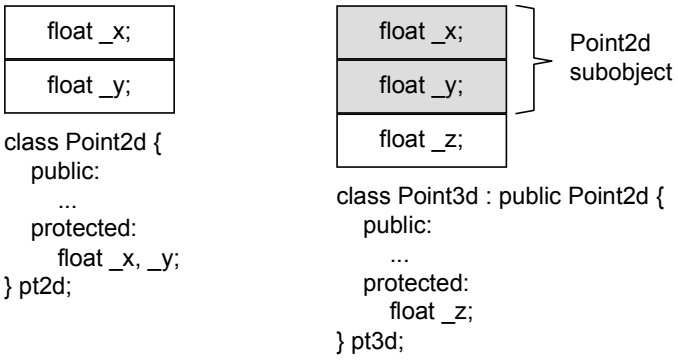
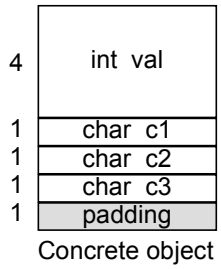


圖 3.1b　單一繼承而且沒有 virtual function 時的資料佈局

把兩個原本獨立不相干的 classes 湊成一對 "type/subtype"，並帶有繼承關係，會有什麼易犯的錯誤呢？經驗不足的人可能會重複設計一些相同動作的函式。以我們例子中的 constructor 和 operator+= 爲例，它們並沒有被做成 inline 函式（也可能是編譯器爲了某些理由沒有支援 inline member functions）。*Point3d* object 的初始化動作或加法動作，將需要部份的 *Point2d* object 和部份的 *Point3d* object 做爲成本。一般而言，選擇某些函式做成 inline 函式，是設計 class 時的一個重要課題。

第二個易犯的錯誤是，把一個 class 分解爲兩層或更多層，有可能會爲了「表現 class 體系之抽象化」而膨脹所需空間。C++ 語言保證「出現在 derived class 中的 base class subobject 有其完整原樣性」，正是重點所在。這似乎有點難以理解！最好的解釋方法就是徹底瞭解一個實例，讓我們從一個具體的 class 開始：




```
class Concrete {
public:
    // ...
private:
    int val;
    char c1;
    char c2;
    char c3;
};
```

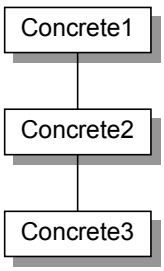
在一部 32 位元機器中，每一個 *Concrete* class object 的大小都是 8 bytes，細分如下：

1. *val* 佔用 4 bytes

2. *c1* 和 *c2* 和 *c3* 各佔用 1 bytes

3. alignment（調整到 word 邊界）需要 1 bytes

現在假設，經過某些分析之後，我們決定了一個更邏輯的表達方式，把 *Concrete* 分裂爲三層架構：




```
class Concrete1 {
public:
    // ...
private:
    int val;
    char bit1;
};

class Concrete2 : public Concrete1 {
public:
    // ...
private:
    char bit2;
};

class Concrete3 : public Concrete2 {
public:
    // ...
private:
    char bit3;
};
```

從設計的觀點來看，這個架構可能比較合理。但從實務的觀點來看，我們可能會受困於一個事實：現在 *Concrete3* object 的大小是 16 bytes，比原先的設計多了一倍。

怎麼回事，還記得「base class subobject 在 derived class 中的原樣性」嗎？讓我們踏遍這一繼承架構的記憶體佈局，看看到底發生了什麼事。

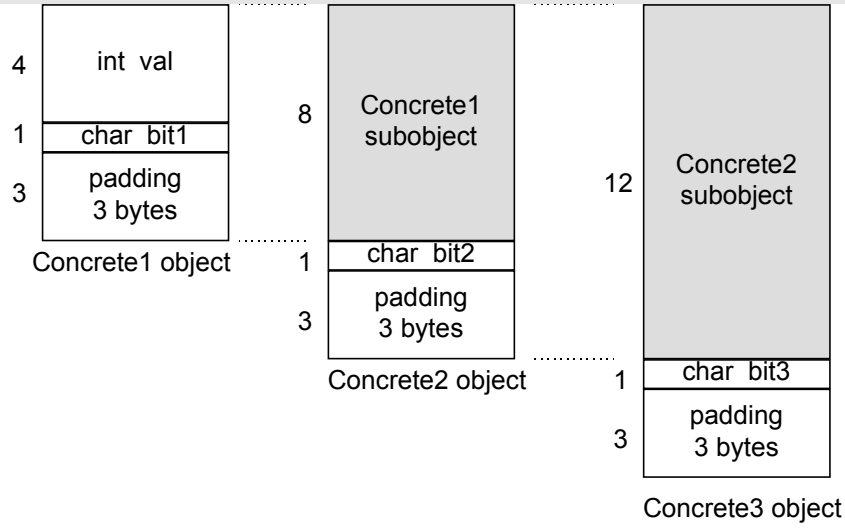
Concrete1 內含兩個 members：*val* 和 *bit1*，加起來是 5 bytes。而一個 *Concrete1* object 實際用掉 8 bytes，包括填補用的 3 bytes，以使 object 能夠符合一部機器的 word 邊界。不論是 C 或 C++ 都是這樣。一般而言，邊界調整（alignment）是由處理器（processor）來決定。

到目前爲止沒什麼需要抱怨。但這種典型的佈局會導至輕率的程式員犯下錯誤。*Concrete2* 加了唯一一個 nonstatic data member *bit2*，資料型態爲 char。輕率的程式員以爲它會和 *Concreate1* 包綑在一起，佔用原本用來填補空間的 1 bytes；於是 *Concreate2* object 的大小爲 8 bytes，其中 2 bytes 用於填補空間。

然而 *Concrete2* 的 *bit2* 實際上卻是被放在填補空間所用的 3 bytes 之後。於是其大小變成 12 bytes，不是 8 bytes。其中有 6 bytes 浪費在填補空間上。相同道理使得 *Concrete3* object 的大小是 16 bytes，其中 9 bytes 用於填補空間。

『真是愚蠢』，我們那位純真小甜甜這麼說。許多讀者以電子郵件、電話、或是嘴巴對我也這麼說。你可瞭解爲什麼這個語言有這樣的行爲？

譯註：下圖可說明 *Concreate1*、*Concreate2*、*Concreate3* 的物件佈局：



讓我們宣告以下一組指標：

```
Concrete2 *pc2;
Concrete1 *pc1_1, *pc1_2;
```

其中 *pc1_1* 和 *pc1_2* 兩者都可以指向前述三種 classes objects。下面這個指定動作：

```
*pc1_2 = *pc1_1;
```

應該執行一個預設的 "memberwise" 複製動作（複製一個個的 members），對象是被指之 object 的 *Concrete1* 那一部份。如果 *pc1_1* 實際指向一個 *Concrete2* object 或 *Concrete3* object，則上述動作應該將複製內容指定給其 *Concrete1* subobject。

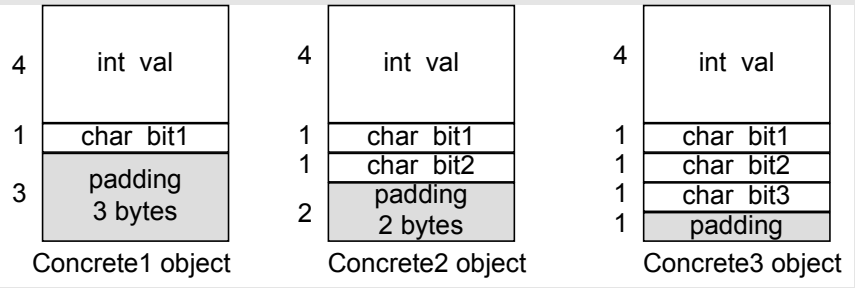
然而，如果 C++ 語言把 derived class members（也就是 *Concrete2*::*bit2* 或

Concrete3::*bit3*）和 *Concrete1* subobject 包裹在一起，去除填補空間，上述那些語意就無法保留了，那麼下面的指定動作：

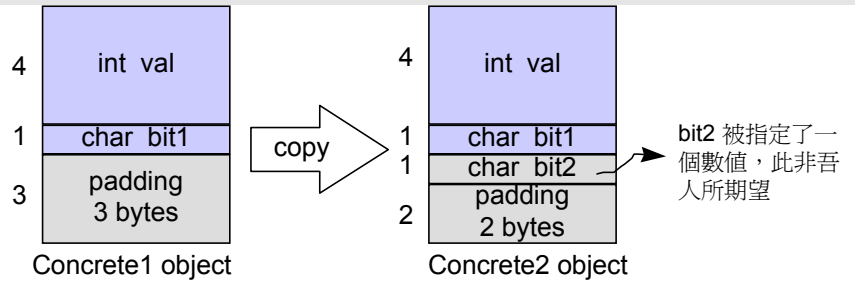
```
pc1_1 = pc2;  // 譯註：令 pc1_1 指向 Concrete2 物件

// 喔歐: derived class subobject 被覆寫掉，
// 於是其 bit2 member 現在有了一個並非預期的數值
*pc1_2 = *pc1_1;
```

就會將「被包裹在一起、繼承而得的」members 內容覆寫掉。程式員必須花費極大的心力才能找出這個臭蟲！

譯註：讓我以圖形解釋。如果「base class subobject 在 derived class 中的原樣性」受到破壞，也就是說，編譯器把 base class object 原本的填補空間讓出來給 derived class members 使用，像這樣：



那麼當發生 *Concrete1* subobject 的複製動作時，就會破壞 *Concrete2* members。



加上多型（Adding Polymorphism）

如果我要處理一個座標點，而不打算在乎它是一個 *Point2d* 或 *Point3d* 實體，那麼我需要在繼承關係中提供一個 virtual function 介面。讓我們看看如果這麼做，情況會有什麼改變：

```
// 譯註：以下的 Point2d 宣告請與 #101 頁的宣告做比較
class Point2d {
public:
    Point2d( float x = 0.0, float y = 0.0 )
        : _x( x ), _y( y ) { };

    // x 和 y 的存取函式與前一版相同。
    // 由於對不同維度的點，這些函式動作固定不變，所以不必設計爲 virtual

    // 加上 z 的保留空間（目前什麼也沒做）
    virtual float z() { return 0.0; }  // 譯註：2d 點的 z 爲 0.0 是合理的
    virtual void z( float ) { }

    // 設定以下的運算子爲 virtual
    virtual void
    operator+=( const Point2d& rhs ) {
        _x += rhs.x();
        _y += rhs.y();
    }
    // ... more members

protected:
    float _x, _y;
};
```

只有當我們企圖以多型的方式（polymorphically）處理 2d 或 3d 座標點，在設計之中導入一個 virtual 介面才顯合理。也就是說，寫下這樣的碼：

```
void foo( Point2d &p1, Point2d &p2 ) {
    // ...
    p1 += p2;
    // ...
}
```

其中 *p1* 和 *p2* 可能是 2d 也可能是 3d 座標點。這並不是先前任何設計所能支

援的。這樣的彈性，當然正是物件導向程式設計的中心。支援這樣的彈性，勢必對我們的 *Point2d* class 帶來空間和存取時間的額外負擔：

- 導入一個和 *Point2d* 有關的 virtual table，用來存放它所宣告的每一個 virtual functions 的位址。這個 table 的元素個數一般而言是被宣告之 virtual functions 的個數，再加上一個或兩個 slots （用以支援 runtime type identification）。
- 在每一個 class object 中導入一個 vptr，提供執行時期的聯結，使一個 object 能夠找到相應的 virtual table。
- 加強 constructor，使它能夠爲 vptr 設定初值，讓它指向 class 所對應的 virtual table。這可能意味在 derived class 和每一個 base class 的 constructor 中，重新設定 vptr 的值。其情況視編譯器的最佳化的積極性而定。第 5 章對此有比較詳細的討論。
- 加強 destructor，使它能夠抹消「指向 class 之相關 virtual table」的 vptr。要知道，vptr 很可能已經在 derived class destructor 中被設定爲 derived class 的 virtual table 位址。記住，destructor 的呼叫次序是反向的：從 derived class 到 base class。一個積極的最佳化編譯器可以壓抑大量的那些指定動作。

這些額外負擔帶來的衝擊程度視「被處理的 *Point2d* objects 的個數和生命期」而定，也視「對這些 objects 做多型程式設計所得的利益」而定。如果一個應用程式知道它所能使用的 point objects 只限於二維座標點或三維座標點，這種設計所帶來的額外負擔可能變得令人無法接受[1]。

以下是新的 *Point3d* 宣告：

```
// 譯註：以下的 Point3d 宣告請與 #101 頁的宣告做比較
class Point3d : public Point2d {
public:
```

[1] 我不知道是否有哪個產品系統真正使用了一個多型的 *Point* 類別體系。

```
    Point3d( float x = 0.0, float y = 0.0, float z = 0.0 )
        : Point2d( x, y ), _z( z ) { };

    float z() { return _z; }
    void z( float newZ ) { _z = newZ; }

    void operator+=( const Point2d& rhs )
        // 譯註：注意上行是 Point2d& 而非 Point3d&
        Point2d::operator+=( rhs );
        _z += rhs.z();
    }
    // ... more members
protected:
    float _z;
};
```

譯註：上述新的（與 p.101 比較）*Point2d* 和 *Point3d* 宣告，最大一個好處是，你可以把 operator+= 運用在一個 *Point3d* 物件和一個 *Point2d* 物件身上：

```
Point2d p2d(2.1, 2.2);
Point3d p3d(3.1, 3.2, 3.3);
p3d += p2d;
```

得到的 *p3d* 新值將是 (5.2, 5.4, 3.3);

雖然 class 的宣告語法沒有改變，但每一件事情都不一樣了：兩個 *z()* member functions 以及 operator+=() 運算子都成了虛擬函式；每一個 *Point3d* class object 內含一個額外的 vptr member（繼承自 *Point2d*）；多了一個 *Point3d* virtual table；此外每一個 virtual member function 的喚起也比以前複雜了（第 4 章對此有詳細說明）。

目前在 C++ 編譯器那個領域裡有一個主要的討論題目：把 vptr 放置在 class object 的哪裡會最好？在 cfront 編譯器中，它被放在 class object 的尾端，用以支援下面的繼承類型，如**圖 3.2a** 所示：

```
struct no_virts {
   int d1, d2;
};
```

```
class has_virts : public no_virts {
public:
   virtual void foo();
   // ...
private:
   int d3;
};

no_virts *p = new has_virts;
```

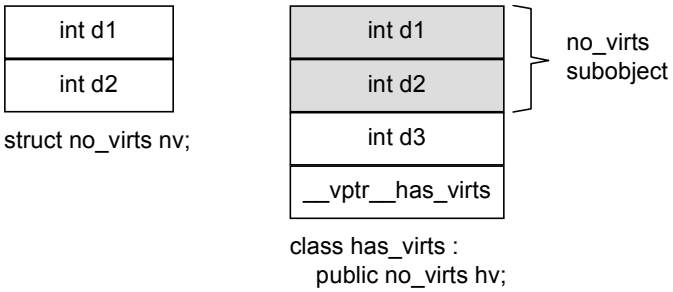


圖 3.2a　Vptr 被放在 class 的尾端

把 vptr 放在 class object 的尾端，可以保留 base class C struct 的物件佈局，因而允許在 C 程式碼中也能使用。這種作法在 C++ 最初問世時，被許多人採用。

到了 C++ 2.0，開始支援虛擬繼承以及抽象基礎類別，並且由於物件導向典範（OO paradigm）的出頭，某些編譯器開始把 vptr 放到 class object 的起頭處（例如 Martin O'Riordan，他領導 Microsoft 的第一個 C++ 編譯器產品，就十分主張這種作法）。請看**圖 3.2b** 的圖解說明。

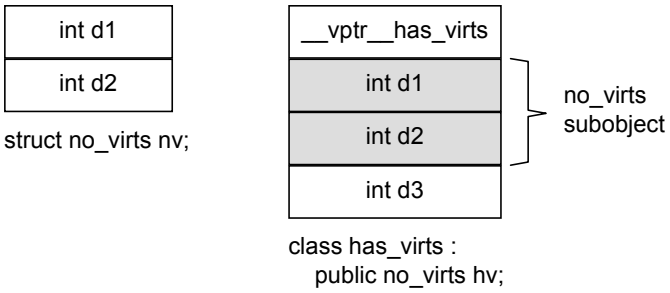


圖 3.2b　Vptr 被放在 class 的前端

把 vptr 放在 class object 的前端，對於「在多重繼承之下，透過指向 class members 的指標，喚起 virtual function」，會帶來一些幫助（請參考 4.4 節）。否則，不僅「從 class object 起始點開始量起」的 offset 必須在執行時期備妥，甚至與 class vptr 之間的 offset 也必須備妥。當然，vptr 放在前端，代價就是喪失了 C 語言相容性。這種喪失有多少意義？有多少程式會從一個 C struct 衍生出一個具多型性質的 class 呢？目前我手上並沒有什麼統計數據可以告訴我這一點。

圖 3.3 顯示 *Point2d* 和 *Point3d* 加上了 virtual function 之後的繼承佈局。注意此圖是把 vptr 放在 base class 的尾端。

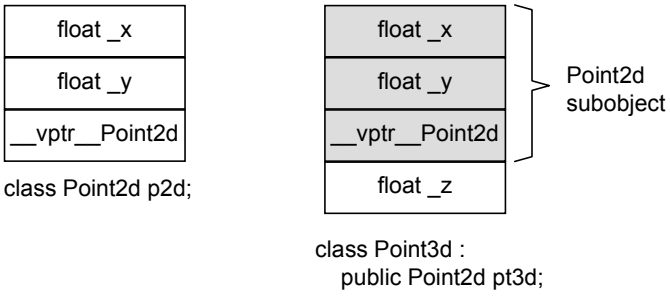


圖 3.3　單一繼承並含虛擬函式之情況下的資料佈局

多重繼承（Multiple Inheritance）

單一繼承提供了一種「自然多型（natural polymorphism）」形式，是關於 classes 體系中的 base type 和 derived type 之間的轉換。請看**圖 3.1b**、**圖 3.2a** 或**圖 3.3**，你會看到 base class 和 derived class 的 objects 都是從相同的位址開始，其間差異只在於 derived object 比較大，用以多容納它自己的 nonstatic data members。下面這樣的指定動作：

```
Point3d p3d;
Point2d *p = &p3d;
```

把一個 derived class object 指定給 base class（不管繼承深度有多深）的指標或 reference。這動作並不需要編譯器去調停或修改位址。它很自然地可以發生，而且提供了最佳執行效率。

圖 3.2b 把 vptr 放在 class object 的起始處。如果 base class 沒有 virtual function 而 derived class 有（譯註：正如**圖 3-2b**），那麼單一繼承的自然多型（natural polymorphism）就會被打破。這種情況下，把一個 derived object 轉換爲其 base 型態，就需要編譯器的介入，用以調整位址（因 vptr 插入之故）。在既是多重繼承又是虛擬繼承的情況下，編譯器的介入更有必要。

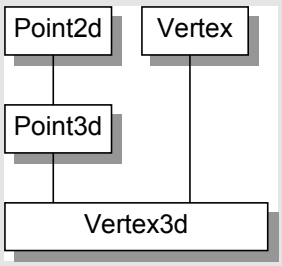
多重繼承既不像單一繼承，也不容易模塑出其模型。多重繼承的複雜度在於 derived class 和其上一個 base class 乃至於上上一個 base class...之間的「非自然」關係。例如，考慮下面這個多重繼承所獲得的 class *Vertex3d*：

> 譯註：原書的 p92~p94 有很多前後不一致的地方，以及很多「本身雖沒有錯誤卻可能誤導讀者思想」的敘述。程式碼和圖片說明也不相符，簡直一團亂！我已將之全部更正。如果您拿著原文書對照此中譯本看，請不要乍見之下對我產生誤會。

```
class Point2d {
public:
```

```
   // ... （譯註：擁有 virtual 介面。所以 Point2d 物件之中會有 vptr）
protected:
   float _x, _y;
};

class Point3d : public Point2d {
public:
    // ...
protected:
    float _z;
};

class Vertex {
public:
   // ... （譯註：擁有 virtual 介面。所以 Vertex 物件之中會有 vptr）
protected:
   Vertex *next;
};

class Vertex3d :  // 譯註：原書誤把 Vertex3d 寫爲 Vertex2d
   public Point3d, public Vertex {  //譯註：原書誤把 Point3d寫爲 Point2d
public:
   // ...
protected:
   float mumble;
};
```

譯註：至此，*Point2d*、*Point3d*、*Vertex*、*Vertex3d* 的繼承關係如下：



多重繼承的問題主要發生於 derived class objects 和其第二或後繼的 base class objects 之間的轉換；不論是直接轉換如下：

```
extern void mumble( const Vertex& );
Vertex3d v;
...
// 將一個 Vertex3d 轉換爲一個 Vertex。這是「不自然的」。
mumble( v );
```

或是經由其所支援的 virtual function 機制做轉換。因支援「virtual function 之喚起動作」而引發的問題將在 4.2 節討論。

對一個多重衍生物件，將其位址指定給「最左端（也就是第一個）base class 的指標」，情況將和單一繼承時相同，因爲二者都指向相同的起始位址。需付出的成本只有位址的指定動作而已（圖 **3.4** 顯示出多重繼承的佈局）。至於第二個或後繼的 base class 的位址指定動作，則需要將位址修改過：加上（或減去，如果 downcast 的話）介於中間的 base class subobject(s) 大小，例如：

```
Vertex3d v3d;
Vertex  *pv;
Point2d *p2d;   // 譯註：原書命名爲 *pp，不符合命名原則。改爲 *p2d 較佳。
Point3d *p3d;  // 譯註：Point3d 的定義請看圖 3.3 和 #108 頁
```

那麼下面這個指定動作：

```
pv = &v3d;
```

需要這樣的內部轉化：

```
// 虛擬 C++ 碼
pv = (Vertex*)(((char*)&v3d) + sizeof( Point3d ));
```

而下面的指定動作：

```
p2d = &v3d;
p3d = &v3d;
```

都只需要簡單地拷貝其位址就好。如果有兩個指標如下：

```
Vertex3d *pv3d;  // 譯註：原書命名爲 *p3d，不符命名通則。改爲 *pv3d 較佳。
Vertex   *pv;
```

那麼下面的指定動作：

```
pv = pv3d;
```

不能夠只是簡單地被轉換爲：

```
// 虛擬 C++ 碼
pv = (Vertex*)((char*)pv3d) + sizeof( Point3d );
```

因爲如果 *pv3d* 爲 0，*pv* 將獲得 *sizeof(Point3d)* 的值。這是錯誤的！所以，對於指標，內部轉換動作需要有一個條件測試：

```
// 虛擬 C++ 碼
pv = pv3d
   ? (Vertex*)((char*)pv3d) + sizeof( Point3d )
   : 0;
```

至於 reference，則不需要針對可能的 0 值做防衛，因爲 reference 不可能參考到「無物」（no object）。

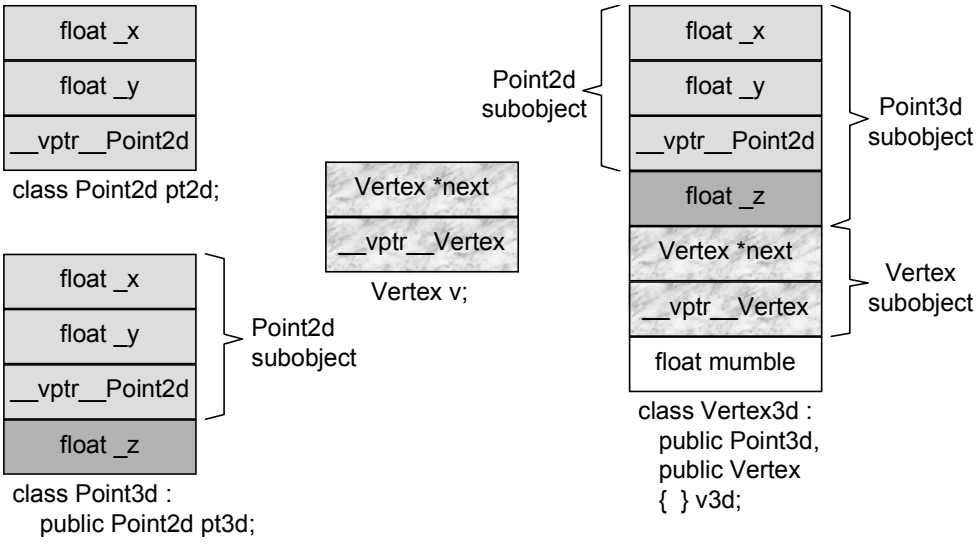


圖 3.4　資料佈局：多重繼承（Multiple Inheritance）

譯註：原書的圖 **3.4** 只畫出 *Vertex2d*，沒有畫出 *Vertex3d*。雖然其中的 *Vertex2d* 物件佈局圖「可能」是正確的（我們並沒有在書中看到其宣告碼），但我相信這其實是 Lippman 的筆誤，因爲它與書中的許多討論沒有關係。所以我把真正與書中討論有關的 *Vertex3d* 的物件佈局畫於譯本的圖 **3.4**，如上。

C++ Standard 並未要求 *Vertex3d* 中的 base classes *Point3d* 和 *Vertex* 有特定的排列次序。原始的 cfront 編譯器是把它們根據宣告次序來排列。因此 cfront 編譯器製作出來的 *Vertex3d* 物件，將可被視爲是一個 *Point3d* subobject（其中又有一個 *Point2d* subobject）加上一個 *Vertex* subobject，最後再加上 *Vertex3d* 自己的部份。目前各編譯器仍然是以此方式完成多重 base classes 的佈局（但如果加上虛擬繼承，就不一樣）。

某些編譯器（例如 MetaWare）設計有一種最佳化技術，只要第二個（或後繼）base class 宣告了一個 virtual function，而第一個 base class 沒有，就把多個 base classes 的次序調換。這樣可以在 derived class object 中少產生一個 vptr。這項最佳化技術並未得到全球各廠商的認可，因此並不普及。

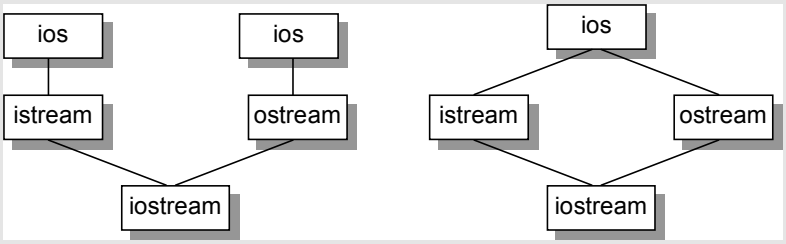
如果要存取第二個（或後繼）base class 中的一個 data member，將會是怎樣的情況？需要付出額外的成本嗎？不，members 的位置在編譯時就固定了，因此存取 members 只是一個簡單的 offset 運算，就像單一繼承一樣簡單 -- 不管是經由一個指標或是一個 reference 或是一個 object 來存取。

虛擬繼承（Virtual Inheritance）

多重繼承的一個語意上的副作用就是，它必須支援某種型式的「shared subobject 繼承」。典型的一個例子是最早的 *iostream* library：

```
// pre-standard iostream implementation
class ios { ... };
class istream : public ios { ... };
class ostream : public ios { ... };
class iostream :
  public istream, public ostream { ... };
```

譯註：下圖可表現 iostream 的繼承體系圖。左爲多重繼承，右爲虛擬多重繼承。



不論是 *istream* 或 *ostream* 都內含一個 *ios* subobject。然而在 *iostream* 的物件佈局中，我們只需要單一一份 *ios* subobject 就好。語言層面的解決辦法是導入所謂的虛擬繼承：

```
class ios { ... };
class istream : public virtual ios { ... };
class ostream : public virtual ios { ... };
class iostream :
  public istream, public ostream { ... };
```

一如其語意所呈現的複雜度，要在編譯器中支援虛擬繼承，實在是困難度頗高。在上述 *iostream* 例子中，實作技術的挑戰在於，要找到一個足夠有效的方法，將 *istream* 和 *ostream* 各自維護的一個 *ios* subobject，摺疊成爲一個由 *iostream* 維護的單一 *ios* subobject，並且還可以保存 base class 和 derived class 的指標（以及 references）之間的多型指定動作（polymorphism assignments）。

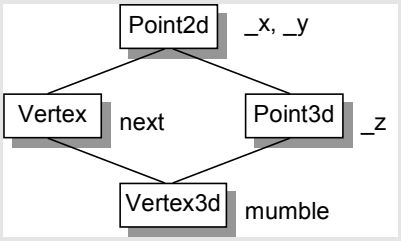
一般的實作法如下所述。Class 如果內含一個或多個 virtual base class subobjects，像 *istream* 那樣，將被分割爲兩部份：一個不變區域和一個共享區域。不變區域

中的資料，不管後繼如何衍化，總是擁有固定的 offset（從 object 的起頭算起），所以這一部份資料可以被直接存取。至於共享區域，所表現的就是 virtual base class subobject。這一部份的資料，其位置會因爲每次衍生動作而有變化，所以它們只可以被間接存取。各家編譯器實作技術之間的差異就在於間接存取的方法不同。以下說明三種主流策略。下面是 *Vertex3d* 虛擬繼承的階層架構[2]：

```
class Point2d {
public:
   ...
protected:
   float _x, _y;
};

class Vertex : public virtual Point2d {
public:
   ...
protected:
   Vertex *next;
};

class Point3d : public virtual Point2d {
public:
   ...
protected:
   float _z;
};

class Vertex3d :
   public Vertex, public Point3d
   // 譯註：原書上一行的兩個 classes 次序相反。爲與圖 3.5ab 配合，故改之。
{
public:
   ...
protected:
   float mumble;
};
```

[2] 這個階層架構是 [POKOR94] 所倡議的，那是一本很好的 3D Graphics 教科書，使用 C++ 語言。

譯註：下圖可表現 *Point2d*、*Point3d*、*Vertex*、*Vertex3d* 的繼承體系：



一般的佈局策略是先安排好 derived class 的不變部份，然後再建立其共享部份。

然而，這中間存在著一個問題：如何能夠存取 class 的共享部份呢？cfront 編譯器會在每一個 derived class object 中安插一些指標，每個指標指向一個 virtual base class。要存取繼承得來的 virtual base class members，可以藉由相關指標間接完成。舉個例，如果我們有以下的 *Point3d* 運算子：

```
void
Point3d::
operator+=( const Point3d &rhs )
{
   _x += rhs._x;
   _y += rhs._y;
   _z += rhs._z;
};
```

在 cfront 策略之下，這個運算子會被內部轉換爲：

```
// 虛擬 C++ 碼
__vbcPoint2d->_x += rhs.__vbcPoint2d->_x; // 譯註：vbc 意爲：
__vbcPoint2d->_y += rhs.__vbcPoint2d->_y;  //   virtual base class
_z += rhs._z;
```

而一個 derived class 和一個 base class 的實體之間的轉換，像這樣：

```
Point2d *p2d = pv3d; // 譯註：原書爲 Vertex *pv = pv3d; 恐爲筆誤
```

在 cfront 實作模型之下，會變成：

```
// 虛擬 C++ 碼
Point2d *p2d = pv3d ? pv3d->__vbcPoint2d : 0;
// 譯註：原書爲  Vertex *pv = pv3d ? pv3d->__vbcPoint2d : 0;
//       恐爲筆誤（感謝黃俊達先生與劉東岳先生來信指導）
```

這樣的實作模型有兩個主要的缺點：

1. 每一個物件必須針對其每一個 virtual base class 背負一個額外的指標。然而理想上我們卻希望 class object 有固定的負擔，不因爲其 virtual base classes 的個數而有所變化。想想看這該如何解決？
2. 由於虛擬繼承串鏈的加長，導至間接存取層次的增加。這意思是，如果我有三層虛擬衍化，我就需要三次間接存取（經由三個 virtual base class 指標）。然而理想上我們卻希望有固定的存取時間，不因爲虛擬衍化的深度而改變。

MetaWare 和其他編譯器到今天仍然使用 cfront 的原始實作模型來解決第二個問題，它們經由拷貝動作取得所有的 nested virtual base class 指標，放到 derived class object 之中。這就解決了「固定存取時間」的問題，雖然付出了一些空間上的代價。MetaWare 提供一個編譯時期的選項，允許程式員選擇是否要產生雙重指標。**圖 3.5a** 說明這種「以指標指向 base class」的實作模型。

至於第一個問題，一般而言有兩個解決方法。Microsoft 編譯器引入所謂的 virtual base class table。每一個 class object 如果有一個或多個 virtual base classes，就會由編譯器安插一個指標，指向 virtual base class table。至於真正的 virtual base class 指標，當然是被放在該表格中。雖然此法已行之有年，但我並不知道是否有其他任何編譯器使用此法。說不定 Microsoft 對此法提出專利，以至別人不能使用它。

第二個解決方法，同時也是 Bjarne 比較喜歡的方法（至少當我還和他共事於 Foundation 專案時），是在 virtual function table 中放置 virtual base class 的 offset（而不是位址）。**圖 3.5b** 顯示這種 base class offset 實作模型。我在 Foundation

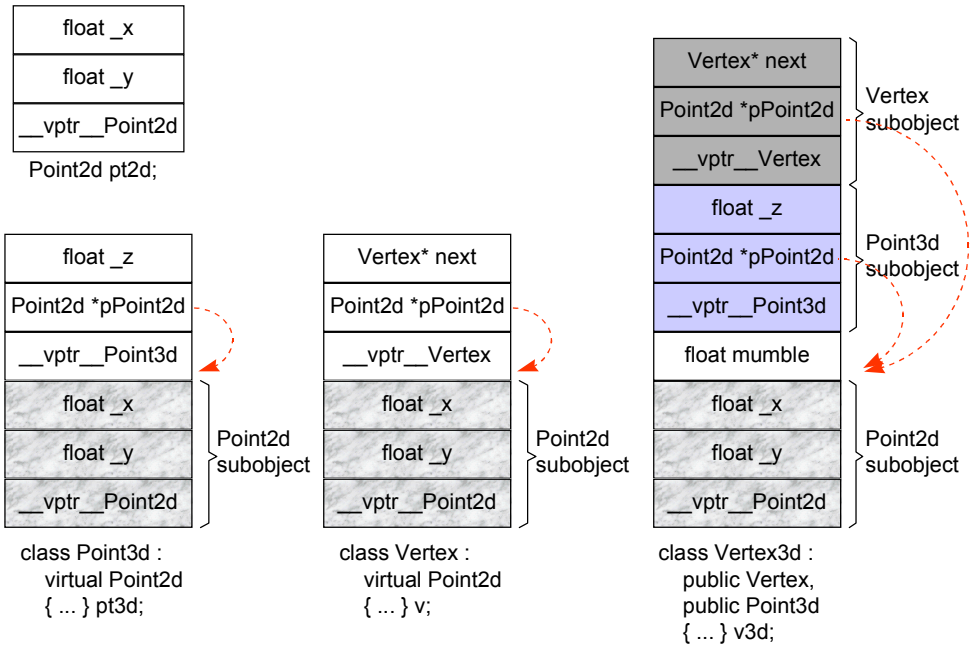


圖 3.5a　虛擬繼承，使用 Pointer Strategy 所產生的資料佈局（譯註：原書圖 3.5a 把 Vertex 寫爲 Vertex2d，與書中程式碼不符，所以我全部改爲 Vertex）

專案中實作出此法，將 virtual base class offset 和 virtual function entries 混雜在一起。在新近的 Sun 編譯器中，virtual function table 可經由正值或負值來索引。如果是正值，很顯然就是索引到 virtual functions；如果是負值，則是索引到 virtual base class offsets。在這樣的策略之下，*Point3d* 的 operator+= 運算子必須被轉換爲以下形式（爲了可讀性，我沒有做型別轉換，同時我也沒有先執行對效率有幫助的位址預先計算動作）：

```
// 虛擬 C++ 碼
(this + __vptr__Point3d[-1])->_x +=
   (&rhs + rhs.__vptr__Point3d[-1])->_x;
(this + __vptr__Point3d[-1])->_y +=
   (&rhs + rhs.__vptr__Point3d[-1])->_y;
_z += rhs._z;
```

雖然在此策略之下，對於繼承而來的 members 做存取動作，成本會比較昂貴，不過此成本已經被分散至「對 member 的使用」上，屬於區域性成本。Derived class 實體和 base class 實體之間的轉換動作，例如：

```
Point2d *p2d = pv3d; // 譯註：原書爲 Vertex *pv = pv3d; 恐爲筆誤
```

在上述實作模型下將變成：

```
// 虛擬 C++ 碼
Point2d *p2d = pv3d ? pv3d + pv3d->__vptr__Point3d[-1] : 0;
// 譯註：上一行原書爲：
//   Vertex *pv = pv3d ? pv3d + pv3d->__vptr__Point3d[-1] : 0;
//   恐爲筆誤（感謝黃俊達先生與劉東岳先生來信指導）
```

上述每一種方法都是一種實作模型，而不是一種標準。每一種模型都是用來解決「存取 shared subobject 內的資料（其位置會因每次衍生動作而有變化）」所引發的問題。由於對 virtual base class 的支援帶來額外的負擔以及高度的複雜性，每一種實作模型多少有點不同，而且我想還會隨著時間而進化。

經由一個非多型的 class object 來存取一個繼承而來的 virtual base class 的 member，像這樣：

```
Point3d origin;
...
origin._x;
```

可以被最佳化爲一個直接存取動作，就好像一個經由物件喚起的 virtual function 呼叫動作，可以在編譯時期被決議（resolved）完成一樣。在這次存取以及下一次存取之間，物件的型別不可以改變，所以「virtual base class subobjects 的位置會變化」的問題在此情況下就不再存在了。

一般而言，virtual base class 最有效的一種運用形式就是：一個抽象的 virtual base class，沒有任何 data members。

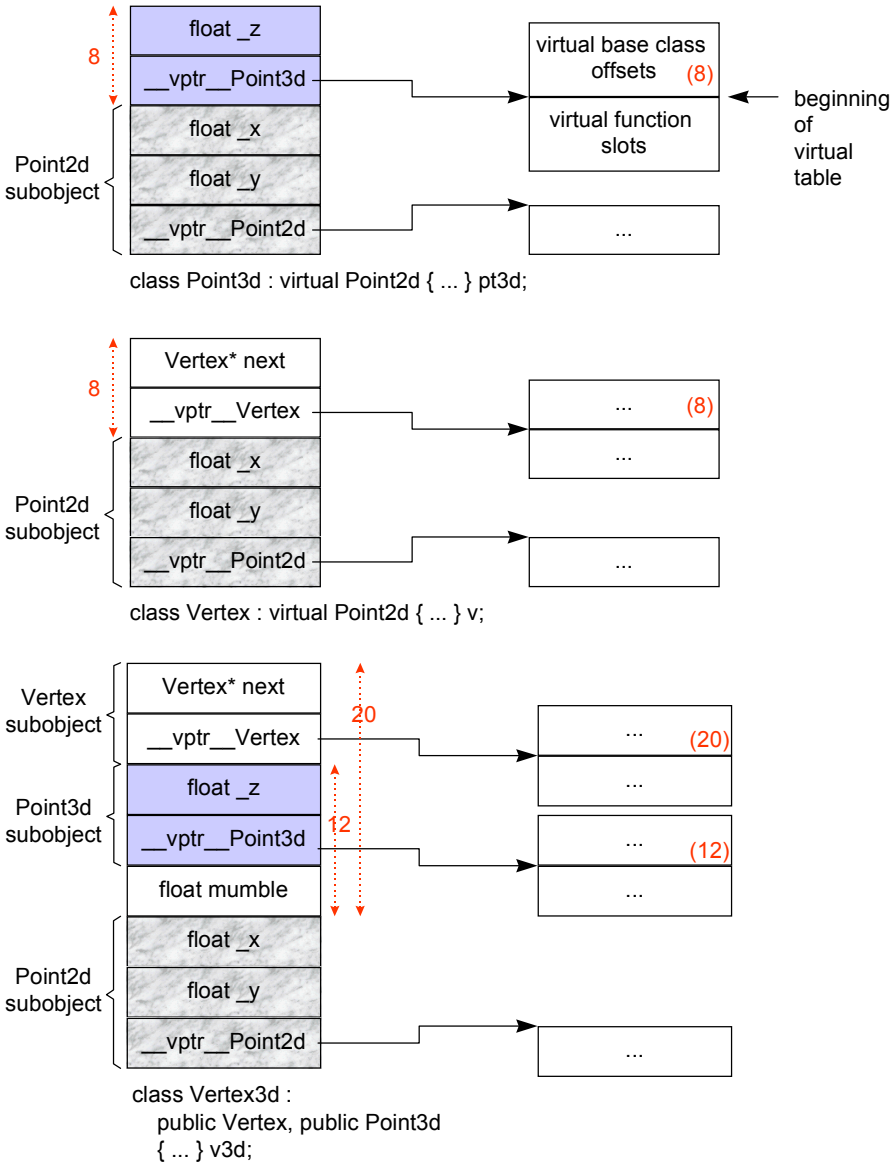


圖 3.5b　虛擬繼承，使用 Virtual Table Offset Strategy 所產生的資料佈局（譯註：原書圖 3.5b 把 Vertex 寫爲 Vertex2d，與書中程式碼不符，所以我全部改爲 Vertex）

3.5 物件成員的效率（Object Member Efficiency）

下面數個測試，旨在量測聚合（aggregation）、封裝（encapsulation）、以及繼承（inheritance）所引發的額外負荷的程度。所有量測都是以個別區域變數的加法、減法、指派（assign）等動作的存取成本爲依據：下面就是個別的區域變數：

```
float pA_x = 1.725,  pA_y = 0.875,  pA_z = 0.478;
float pB_x = 0.315,  pB_y = 0.317,  pB_z = 0.838;
```

每個運算式需執行一千萬次，如下所示（當然啦，一旦座標點的表現方式有變化，運算語法也就得隨之變化）：

```
for ( int iters = 0; iters < 10000000; iters++ )
{
  pB_x = pA_x - pB_z;
  pB_y = pA_y + pB_x;
  pB_z = pA_z + pB_y;
}
```

我們首先針對三個 float 元素所組成的區域陣列進行測試：

```
enum fussy { x, y, z };

for ( int iters = 0; iters < 10000000; iters++ )
{
  pB[ x ] = pA[ x ] - pB[ z ];
  pB[ y ] = pA[ y ] + pB[ x ];
  pB[ z ] = pA[ z ] + pB[ y ];
}
```

第二個測試是把同質的陣列元素轉換爲一個 C struct 資料抽象型別，其中的成員皆爲 float，成員名稱是 x, y, z：

```
for ( int iters = 0; iters < 10000000; iters++ )
{
  pB.x = pA.x - pB.z;
  pB.y = pA.y + pB.x;
  pB.z = pA.z + pB.y;
}
```

更深一層的抽象化，是做出資料封裝，並使用 inline 函式。座標點現在以一個獨立的 *Point3d* class 來表示。我嘗試兩種不同型式的存取函式，第一，我定義一個 inline 函式，傳回一個 reference，允許它出現在 assignment 運算子的兩端：

```
class Point3d {
public:
   Point3d( float xx = 0.0, float yy = 0.0, float zz = 0.0 )
      : _x( xx ), _y( yy ), _z( zz ) {  }

   float& x()  { return _x; }
   float& y()  { return _y; }
   float& z()  { return _z; }

private:
   float _x, _y, _z;
};
```

那麼真正對每一個座標元素的存取動作應該是像這樣：

```
for ( int iters = 0; iters < 10000000; iters++ )
{
   pB.x() = pA.x() - pB.z();
   pB.y() = pA.y() + pB.x();
   pB.z() = pA.z() + pB.y();
}
```

我所定義的第二種存取函式型式是，提供一對 get/set 函式：

```
float x() { return _x; }    // 譯註：此即 get 函式
void x( float newX )        // 譯註：此即 set 函式
         { _x = newX; }
```

於是對每一個座標值的存取動作應該像這樣：

```
pB.x( pA.x() - pB.z() );
```

表格 3.1 列出兩種編譯器對於上述各種測試的結果。只有當兩個編譯器的效率有明顯差異時，我才會把兩者分別列出。

表格 3.1 不斷加強抽象化程度之後，資料的存取效率

	最佳化	未最佳化
個別的區域變數	0.80	1.42
區域陣列		
CC	0.80	2.55
NCC	0.80	1.42
struct 之中有 public 成員	0.80	1.42
class 之中有 inline Get 函式		
CC	0.80	2.56
NCC	0.80	3.10
class 之中有 inline Get & Set 函式		
CC	0.80	1.74
NCC	0.80	2.87

這裡所顯示的重點在於，如果把最佳化開關打開，「封裝」就不會帶來執行時期的效率成本。使用 inline 存取函式亦然。

我很奇怪爲什麼在 CC 之下存取陣列，幾乎比 NCC 慢兩倍，尤其是陣列存取所牽扯的只是 C 陣列，並沒有用到任何複雜的 C++ 特性。一位程式碼產生（code generation）專家將這種反常現象解釋爲「一種奇行怪癖...與特定的編譯器有關」。或許是真的，但它發生在我正用來開發軟體的編譯器身上耶！我決定挖掘其中秘密。叫我「愛挑毛病的喬治」吧，如果你喜歡的話！如果你對此一題目不感興趣，請直接跳往下一個主題。

在下面的 assembly 語言輸出片段中，l.s 表示載入（load）一個單精度浮點數，s.s. 表示儲存（store）一個單精度浮點數，sub.s 表示將兩個單精度浮點數相減。下面是兩種編譯器的 assembly 語言輸出結果，它們都載入兩個值，將某一個減去另一個，然後儲存其結果。在效率較差的 CC 編譯器中，每一個區域變數的位址都被計算並放進一個暫存器之中（addu 表示無正負號的加法）：

```
// CC assembler output
#  13  pB[ x ] = pA[ x ] - pB[ z ];
   addu  $25, $sp, 20
   l.s   $f4, 0($25)
   addu  $24, $sp, 8
   l.s   $f6, 8($24)
   sub.s $f8, $f4, $f6
   s.s   $f8, 0($24)
```

而在 NCC 編譯器的 assembly 輸出中，載入（load）步驟直接計算位址：

```
// NCC assembler output
#  13  pB[ x ] = pA[ x ] - pB[ z ];
   l.s   $f4, 20($sp)
   l.s   $f6, 16($sp)
   sub.s $f8, $f4, $f6
   s.s   $f8, 8($sp)
```

如果區域變數被存取多次，CC 策略或許比較有效率。然而對於單一存取動作，把變數位址放到一個暫存器中很明顯地增加了運算式的成本。不論哪一種編譯器，只要把最佳化開關打開，兩段碼都會變得相同，在其中，迴路內的所有運算都會以暫存器內的數值來執行。

讓我下一個結論：如果沒有把最佳化開關打開，就很難猜測一個程式的效率表現，因爲程式碼潛在性地受到專家所謂的「一種奇行怪癖...與特定編譯器有關」的魔咒影響。在你開始「原始碼層面的最佳化動作」以加速程式的運作之前，你應該先確實地量測效率，而不是靠著推論與常識判斷。

在下一個測試中，我首先要介紹 *Point* 抽象化的一個三層單一繼承表達法，然後再介紹 *Point* 抽象化的一個虛擬繼承表達法。我要測試直接存取和 inline 存取（多重繼承並不適用於此一模型，所以我決定放棄它）。三層單一繼承表達法如下：

```
class Point1d { ... };                    // 維護 x
class Point2d : public Point1d { ... };   // 維護 y
class Point3d : public Point2d { ... };   // 維護 z
```

「單層虛擬繼承」是從 *Point1d* 中虛擬衍生出 *Point2d*；「雙層虛擬繼承」則又從 *Point2d* 中虛擬衍生出 *Point3d*。表格 **3.2** 列出兩種編譯器的測試結果。同樣地，只有當兩種編譯器的效率有明顯不同時，我才會把兩者分別列出。

表格 3.2 在繼承模型之下的資料存取

	最佳化	未最佳化
單一繼承		
直接存取	0.80	1.42
使用 inline 函式		
CC	0.80	2.55
NCC	0.80	3.10
虛擬繼承（單層）		
直接存取	1.60	1.94
使用 inline 函式		
CC	1.60	2.75
NCC	1.60	3.30
虛擬繼承（雙層）		
直接存取		
CC	2.25	2.74
NCC	3.04	3.68
使用 inline 函式		
CC	2.25	3.22
NCC	2.50	3.81

單一繼承應該不會影響測試的效率，因爲 members 被連續儲存於 derived class object 中，並且其 offset 在編譯時期就已知了。測試結果一如預期，和表格 **3.1** 中的抽象資料型別結果相同。這結果在多重繼承的情況下應該也是相同的，但我不能確定。

再一次，值得注意的是，如果把最佳化關閉，以常識來判斷，我們說效率應該相同（對於「直接存取」和「inline 存取」兩種作法）。然而實際上卻是 inline 存取比較慢。我們再次得到教訓：程式員如果關心其程式效率，應該實際量測，不要光憑推論或常識判斷或假設。另一個需要注意的是，最佳化動作並不一定總是

能夠有效運作，我不只一次以最佳化方式來編譯一個已通過編譯的正常程式，卻以失敗收場。

虛擬繼承的效率令人失望！兩種編譯器都沒能夠辨識出對「繼承而來的 data member *pt1d::_x*」的存取係透過一個非多型物件（因而不需要執行時期的間接存取）。兩個編譯器都會對 *pt1d::_x*（及雙層虛擬繼承中的 *pt2d::_y*）產生間接存取動作，雖然其在 *Point3d* 物件中的位置早在編譯時期就固定了。「間接性」壓抑了「把所有運算都移往暫存器執行」的最佳化能力。但是間接性並不會嚴重影響非最佳化程式的執行效率。

3.6 指向 Data Members 的指標（Pointer to Data Members）

指向 data members 的指標，是一個有點神秘但頗有用處的語言特性，特別是如果你需要詳細調查 class members 的底層佈局的話。這樣的調查可用以決定 vptr 是放在 class 的起始處或是尾端。另一個用途，展現於 3.2 節，可用來決定 class 中的 access sections 的次序。一如我曾說過，那是一個神秘但有時候有用的語言特性。

考慮下面的 *Point3d* 宣告。其中有一個 virtual function，一個 static data member，以及三個座標值：

```
class Point3d {
public:
    virtual ~Point3d();
    // ...
protected:
    static Point3d origin;
    float x, y, z;
};
```

每一個 *Point3d* class object 含有三個座標值，依序爲 *x, y, z*，以及一個 vptr。至於 static data member *origin* 將被放在 class object 之外。唯一可能因編譯器不同

而不同的是 vptr 的位置。C++ Standard 允許 vptr 被放在物件中的任何位置：在起始處，在尾端，或是在各個 members 之間。然而實際上，所有編譯器不是把 vptr 放在物件的頭，就是放在物件的尾。

那麼，取某個座標成員的位址，代表什麼意思？例如，以下動作所得到的值代表什麼：

```
& Point3d::z;  // 譯註：原書的 & 3d_point::z; 應爲筆誤
```

上述動作將得到 *z* 座標在 class object 中的偏移位置（offset）。最低限度其值將是 *x* 和 *y* 的大小總和，因爲 C++ 語言要求同一個 access level 中的 members 的排列次序應該和其宣告次序相同。

然而 vptr 的位置就沒有限制。不過容我再說一次，實際上 vptr 不是放在物件的頭，就是放在物件的尾。在一部 32 位元機器上，每一個 float 是 4 bytes，所以我們應該期望剛才獲得的值要不是 8，就是 12（在 32 位元機器上一個 vptr 是 4 bytes）。

然而，這樣的期望卻還少 1 bytes。對於 C 和 C++ 程式員而言，這多少算是個有點年代的錯誤了。

如果 vptr 放在物件的尾巴，三個座標值在物件佈局中的 offset 分別是 0, 4, 8。如果 vptr 放在物件的起頭，三個座標值在物件佈局中的 offset 分別是 4, 8, 12。然而你若去取 data members 的位址，傳回的值總是多 1，也就是 1, 5, 9 或 5, 9, 13 等等。你知道爲什麼 Bjarne 決定要這麼做嗎？

譯註：如何取 *& Point3d::z* 的值並列印出來？以下是示範作法：

```
printf("&Point3d::x = %p\n", &Point3d::x);  // 結果 VC5：4，BCB3：5
printf("&Point3d::y = %p\n", &Point3d::y);  // 結果 VC5：8，BCB3：9
printf("&Point3d::z = %p\n", &Point3d::z);  // 結果 VC5：C，BCB3：D
```

注意，不可以這麼做：

```
cout << "&Point3d::x = " << &Point3d::x << endl;
cout << "&Point3d::y = " << &Point3d::y << endl;
cout << "&Point3d::z = " << &Point3d::z << endl;
```

否則會得到錯誤訊息：

error C2679: binary '<<' : no operator defined which takes a right-hand operand of type 'float Point3d::*' (or there is no acceptable conversion) (new behavior; please see help)

我使用的編譯器是 Microsoft Visual C++ 5.0。爲什麼執行結果並不如書中所說增加 1 呢？原因可能是 Visual C++ 做了特殊處理，其道理與本章一開始對於 empty virtual base class 的討論相近！

問題在於，如何區分一個「沒有指向任何 data member」的指標，和一個指向「第一個 data member」的指標？考慮這樣的例子：

```
float Point3d::*p1 = 0;
float Point3d::*p2 = &Point3d::x;
// 譯註：Point3d::* 的意思是：「指向 Point3d data member」之指標型別。

// 喔歐：如何區分？
if ( p1 == p2 ) {
   cout << " p1 & p2 contain the same value -- ";
   cout << " they must address the same member!" << endl;
}
```

爲了區分 *p1* 和 *p2*，每一個真正的 member offset 值都被加上 1。因此，不論編譯器或使用者都必須記住，在真正使用該值以指出一個 member 之前，請先減掉 1。

認識「指向 data members 的指標」之後，我們發現，要解釋：

```
& Point3d::z;   // 譯註：原書的 & 3d_point::z; 應爲筆誤
```

和

```
& origin.z
```

之間的差異，就非常明確了。鑒於「取一個 nonstatic data member 的位址，將會得到它在 class 中的 offset」，取一個「繫結於真正 class object 身上的 data member」的位址，將會得到該 member 在記憶體中的真正位址。把

```
& origin.z
```

所得結果減（譯註：原文爲加，錯誤）*z* 的偏移值（相對於 *origin* 起始位址），並加 1（譯註：原文爲減，錯誤），就會得到 *origin* 起始位址。上一行的傳回值型別應該是：

```
float*
```

而不是

```
float Point3d::*
```

由於上述動作所參考的是一個特定單一實體，所以取一個 static data member 的位址，意義也相同。

在多重繼承之下，若要將第二個（或後繼）base class 的指標，和一個「與 derived class object 繫結」之 member 結合起來，那麼將會因爲「需要加入 offset 值」而變得相當複雜。例如，假設我們有：

```
struct Base1 { int val1; };
struct Base2 { int val2; };
struct Derived : Base1, Base2 { ... };

void func1( int Derived::*dmp, Derived *pd )
{
   // 期望第一個參數得到的是個「指向 derived class 之 member」的指標。
   // 如果傳進來的卻是一個「指向 base class 之 member」的指標，會怎樣？
   pd->*dmp;
}

void func2( Derived *pd )
{
   // bmp 將成爲 1
   int Base2::*bmp = &Base2::val2;
```

```
    // 喔歐，bmp == 1，
    // 但是在 Derived 中，val2 == 5
    func1( bmp, pd );
}
```

當 *bmp* 被做爲 *func1()* 的第一個參數，它的值就必須因介入的 *Base1* class 大小而調整，否則 *func1()* 中這樣的動作：

```
pd->*dmp;
```

將存取到 *Base1*::*val1*，而非程式員所以爲的 *Base2*::*val2*。要解決這個問題，必須：

```
// 經由編譯器內部轉換
func1( bmp + sizeof( Base1 ), pd );
```

然而，一般而言，我們不能夠保證 *bmp* 不是 0，因此必須特別護衛之：

```
// 內部轉換
// 防範 bmp == 0
func1( bmp ? bmp + sizeof( Base1 ) : 0, pd );
```

譯註：我實際寫了一個小程式，列印上述各個 member 的 offset 值：

```
printf("&Base1::val1 = %p \n", &Base1::val1);          // (1)
printf("&Base2::val2 = %p \n", &Base2::val2);          // (2)
printf("&Derived::val1 = %p \n", &Derived::val1);      // (3)
printf("&Derived::val2 = %p \n", &Derived::val2);      // (4)
```

經過 Visual C++ 5.0 編譯後，執行結果竟然都是 0。(1)(2)(3)都是 0 是可以理解的（爲什麼不是 1？可能是因爲 Visual C++ 有特殊處理；稍早 p.131 中我的另一個譯註曾有說明）。但爲什麼 (4) 也是 0，而不是 4？是否編譯器已經內部處理過了呢？很可能（我只能如此猜測）。

如果我把 *Derived* 的宣告改爲：

```
struct Derived : Base1, Base2 { int vald; };
```

那麼：

```
printf("&Derived::vald = %p \n", &Derived::vald);
```

將得到 8，表示 *vald* 的前面的確有 *val1* 和 *val2*。

「指向 Members 的指標」的效率問題

下面的測試企圖獲得一些量測數據，讓我們瞭解，在 3D 座標點的各種 class 表現法之下，使用「指向 members 的指標」所帶來的影響。一開始的兩個案例並沒有繼承關係，第一個案例是要取得一個「已繫結之 member」的位址：

```
float *ax = &pA.x;
```

然後施以指派（assignment）、加法、減法動作如下：

```
*bx = *ax - *bz;
*by = *ay + *bx;
*bz = *az + *by;
```

第二個案例則是針對三個 members，取得「指向 data member 之指標」的位址：

```
float Point3d::*ax = &Point3d::x;
```

而指派（assignment）、加法和減法等動作，都是使用「指向 data member 之指標」語法，把數值繫結到物件 *pA* 和 *pB* 中：

```
pB.*bx = pA.*ax - pB.*bz;
pB.*by = pA.*ay + pB.*bx;
pB.*bz = pA.*az + pB.*by;
```

回憶 3.5 節中的直接存取動作，平均時間是 0.8 秒（當最佳化開啓）或 1.42 秒（當最佳化關閉）。現在再執行這兩個測試，結果列於表格 **3.3** 中。

表格 3.3　存取 Nonstatic Data Member

	最佳化	未最佳化
直接存取（請參考 3.5 節）	0.80	1.42
指標指向已繫結之 Member	0.80	3.04
指標指向 Data Member		
CC	0.80	5.34
NCC	4.04	5.34

未最佳化的結果正如預期。也就是說，爲每一個「member 存取動作」加上一層間接性（經由已繫結之指標），會使執行時間多出一倍不止。以「指向 member 的指標」來存取資料，再一次幾乎用掉了雙倍時間。要把「指向 member 的指標」繫結到 class object 身上，需要額外地把 offset 減 1。更重要的是，當然，最佳化可以使所有三種存取策略的效率變得一致，唯 NCC 編譯器除外。你不妨注意一下，在這裡，NCC 編譯器所產生的碼在最佳化情況下有著令人震驚的可憐效率，這反映出它所產生出來的 assembly 碼有著可憐的最佳化動作，這和 C++ 原始碼如何表現並無直接關係 -- 要知道，我曾檢驗過 CC 和 NCC 產生出來的未最佳化 assembly 碼，兩者完全一樣！

下一組測試要看看「繼承」對於「指向 data member 的指標」所帶來的效率衝擊。在第一個案例中，獨立的 *Point* class 被重新設計爲一個三層單一繼承體系，每一個 class 有一個 member：

```
class Point { ... };                      // float x;
class Point2d : public Point   { ... };   // float y;
class Point3d : public Point2d { ... };   // float z;
```

第二個案例仍然是三層單一繼承體系，但導入一層虛擬繼承：*Point2d* 虛擬衍生自 *Point*。結果，每次對於 *Point::x* 的存取，將是對一個 virtual base class data member 的存取。最後一個案例，實用性很低，幾乎純粹是好奇心的驅使：我加上第二層虛擬繼承，使 *Point3d* 虛擬衍生自 *Point2d*。表格 **3.4** 顯示測試結果。

注意：由於 NCC 最佳化的效率在各項測試中都是一致的，我已經把它從表格中剔除了。

表格 3.4 「指向 Data Member 的指標」存取方式

	最佳化	未最佳化
沒有繼承	0.80	5.34
單一繼承（三層）	0.80	5.34
虛擬繼承（單層）	1.60	5.44
虛擬繼承（雙層）	2.14	5.51

由於被繼承的 data members 是直接存放在 class object 之中，所以繼承的引入一點也不會影響這些碼的效率。虛擬繼承所帶來的主要衝擊是，它妨礙了最佳化的有效性。爲什麼？在兩個編譯器中，每一層虛擬繼承都導入一個額外層次的間接性。在兩個編譯器中，每次存取 *Point::x*，像這樣：

```
pB.*bx
```

會被轉換爲：

```
&pB->__vbcPoint + ( bx - 1 )
```

而不是轉換最直接的：

```
&pB + ( bx - 1 )
```

額外的間接性會降低「把所有的處理都搬移到暫存器中執行」的最佳化能力。