GOTOP

深度探索

C++ 对象模型

Inside The C++ Object Model

Stanley B. Lippman 着

侯捷 译

- 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++ 对象模型

Inside The C++ Object Model

Stanley B. Lippman 着

侯捷 译

碁峰信息股份有限公司

本立道生

（侯捷 译序）

对于传统的循序性（sequential）语言，我们向来没有太多的疑惑，虽然在函式呼叫的背后，也有着堆栈建制、参数排列、回返地址、堆栈清除等等幕后机制，但函式呼叫是那么地自然而明显，好像只是夹带着一个包裹，从程序的某一个地点跳到另一个地点去执行。

但是对于对象导向（Object Oriented）语言，我们的疑惑就多了。究其因，这种语言的编译器为我们（程序员）做了太多的服务：建构式、解构式、虚拟函式、继承、多型...。有时候它为我们合成出一些额外的函式（或运算子），有时候它又扩张我们所写的函式内容，放进更多的动作。有时候它还会为我们的 objects 加油添醋，放进一些奇妙的东西，使你面对 sizeof 的结果大惊失色。

存在我心里头一直有个疑惑：计算机程序最基础的形式，总是脱离不了一行一行的循序执行模式，为什么 OO（对象导向）语言却能够「自动完成」这么多事情呢？另一个疑惑是，威力强大的 polymorphism（多型），其底层机制究竟如何？

如果不了解编译器对我们所写的 C++ 码做了什么手脚，这些困惑永远解不开。

这本书解决了过去令我百思不解的诸多疑惑。我要向所有已具备 C++ 多年程序设计经验的同好们大力推荐这本书。

这本书同时也是跃向组件软件（component-ware）基本精神的跳板。不管你想学习 COM（Component Object Model）或 CORBA（Common Object Request Broker Architecture）或是 SOM（System Object Model），了解 C++ Object Model，将使你更清楚软件组件（components）设计上的难点与运应之道。不但我自己在学习 COM 的道路上有此强烈的感受，Essential COM（COM 本质论，侯捷译，碁峰 1998）的作者 Don Box 也在他的书中推崇 Lippman 的这一本卓越的书籍。

是的，这当然不会是一本轻松的书籍。某些章节（例如 3、4 两章）可能给你立即的享受 -- 享受于面对底层机制有所体会与掌控的快乐；某些章节（例如 5、6、7 三章）可能带给你短暂的痛苦 -- 痛苦于艰难深涩难以吞咽的内容。这些快乐与痛苦，其实就是我翻译此书时的心情写照。无论如何，我希望透过我的译笔，把这本难得的好书带到更多人面前，引领大家见识 C++ 底层建设的技术之美。

侯捷 1998.05.20 于新竹

jjhou@ccca.nctu.edu.tw

请注意：本书属性，作者 Lippman 在其前言中 有很详细的描述，我不再多言。翻译用词与心得，记录在第 0 章（译者的话）之中 ，对您或有导读之功。

请注意：原文本有大大小小约 80~90 个笔误。有的无伤大雅，有的影响阅读顺畅甚巨（如前后文所用符号不一 致、内文与图形所用符号不一 致 -- 甚至因而导至图片的文字解释不正确）。我已在第 0 章（译者的话）列出所有我找到的错误 。此外，某些场合我还会在错误出现之处再加注，表示原文内容为何。这么做不是画蛇添足，也不为彰显什么。我知道有些读者拿着原文书和中 译书对照着看，我把原书错误加注出来，可免读者怀疑是否我打错字或是译错了。另一 方面也是为了文责自负...唔...万一 Lippman 是对的而 J.J.Hou 错了呢 ?! 我虽有相当把握，还是希望明白摊开来让读者检验。

深度探索 C++ 对象模型

Inside The C++ Object Model

目录

前言

（Stanley B. Lippman）

差不多有 10 年之久，我在贝尔实验室（Bell Laboratories）埋首于 C++ 的实作任务。最初的工作是在 cfront 上面（Bjarne Stroustrup 的第一个 C++ 编译器），从 1986 年的 1.1 版到 1991 年九月的 3.0 版。然后移转到 Simplifier（这是我们内部的命名），也就是 Foundation 项目中的 C++ 对象模型部份。在 Simplifier 设计期间，我开始酝酿这本书。

Foundation 项目是什么？在 Bjarne 的领导下，贝尔实验室中的一个小组探索着以 C++ 完成大规模程序设计时的种种问题的解决之道。Foundation 项目是我们为了建构大系统而努力定义的一个新的开发模型；我们只使用 C++，并不提供多重语言的解决方案。这是个令人兴奋的工作，一方面是因为工作本身，一方面是因为工作伙伴：Bjarne、Andy Koenig、Rob Murray、Martin Carroll、Judy Ward、Steve Buroff、Peter Juhl、以及我自己。Barbara Moo 管理我们这一群人（Bjarne 和 Andy 除外）。Barbara Moo 常说管理一个软件团队，就像放牧一群骄傲的猫。

我们把 Foundation 想象是一个核心，在那上面，其它人可以为使用者铺设一层真正的开发环境，把它整修为他们所期望的 UNIX 或 Smalltalk 模型。私底下我们把它称为 Grail（传说中耶稣最后晚餐所用的圣杯），人人都想要，但是从来没人找到过！

Grail 使用一个由 Rob Murray 发展出来并命名为 ALF 的对象导向阶层架构，提供一个永久的、以语意为基础的表现法。在 Grail 中，传统编译器被分解为数个各自分离的可执行档。parser 负责建立程序的 ALF 表现法。其它每一个组件（像是 type checking、simplification、code generation）以及工具（像是 browser）都在程序的一个 ALF 表现体上操作（并可能加以膨胀）。Simplifier 是编译器的一部份，处于 type checking 和 code generation 之间。Simplifier 这个名称是由 Bjarne 所倡议，它原本是 cfront 的一个阶段（phase）。

在 type checking 和 code generation 之间，Simplifier 做什么事呢？它用来转换内部的程序表现。有三种转换风味是任何对象模型都需要的：

1. 与编译器息息相关的转换（Implementation-dependent transformations）

这是与特定编译器有关的转换。在 ALF 之下，这意味着我们所谓的 "tentative" nodes。例如，当 parser 看到这个表达式：

```
fct();
```

它并不知道是否 (a) 这是一个函式唤起动作，或者 (b) 这是 overloaded call operator 在 class object fct 上的一种应用。预设情况下，这个式子所代表的是一个函式呼叫，但是当 (b) 的情况出现，Simplifier 就要重写并调换 call subtree。

2. 语言语意转换（Language semantics transformations）

这包括 constructor/destructor 的合成和扩张、memberwise 初始化、对于 memberwise copy 的支持、在程序代码中安插 conversion operators、暂时性对象、以及对 constructor/destructor 的呼叫。

3. 程序代码和对象模型的转换（Code and object model transformations）

这包括对 virtual functions、virtual base class 和 inheritance 的一般支持、new 和 delete 运算子、class objects 所组成的数组、local static class instances、带有非常数表达式（nonconstant expression）之global object 的静态初始化动作 我对。Simplifier 所规划的一个目标是：提供一个对象模型体系，在其中，对象的实作是一个虚拟接口，支持各种对象模型。

最后两种类型的转换构成了本书的基础。这意味本书是为编译器设计者而写的吗？不是，绝对不是！这本书是由一位编译器设计者针对中高阶 C++ 程序员所写。隐藏在这本书背后的假设是，程序员如果了解 C++ 对象模型，就可以写出比较没有错误倾向而且比较有效率的码。

什么是 C++ 对象模型

有两个概念可以解释 C++ 对象模型：

1. 语言中直接支持对象导向程序设计的部份。

2. 对于各种支持的底层实作机制。

语言层面的支持，涵盖于我的 C++ Primer 一书以及其它许多 C++ 书籍当中。至于第二个概念，则几乎不能够于目前任何读物中发现，只有 [ELLIS90] 和 [STROUP94] 勉强有一些蛛丝马迹。本书主要专注于 C++ 对象模型的第二个概念。本书语言遵循 C++ 委员会于 1995 冬季会议中通过的 Standard C++ 草案（除了某些细节，这份草案应该能够反映出此语言的最终版本）。

C++ 对象模型的第一个概念是一种「不变量」。例如，C++ class 的完整 virtual functions 在编译时期就固定下来了，程序员没有办法在执行时期动态增加或取代其中某一个。这使得虚拟唤起动作得有快速的派送（dispatch）结果，付出的成本则是执行时期的弹性。

对象模型的底层实作机制，在语言层面上是看不出来的 -- 虽然对象模型的语意本身可以使得某些实作品（编译器）比其它实作品更接近自然。例如，virtual function calls，一般而言是藉由一个表格（内含 virtual functions 地址）的索引而决议得知。一定要使用如此的 virtual table 吗？不，编译器可以自由引进其它任何变通作法。如果使用 virtual table，那么其布局、存取方法、产生时机、以及数百个细节也都必须决定下来，而所有决定也都由每一个实作品（编译器）自行取舍。不过，既然说到这里，我也必须明白告诉你，目前所有编译器对于 virtual function 的实作法都是使用各个 class 专属的 virtual table，大小固定，并且在程式执行前就建构好了。

如果 C++ 对象模型的底层机制并未标准化，那么你可能会问：何必探讨它呢？主要的理由是，我的经验告诉我，如果一个程序员了解底层实作模型，他就能够写出效率较高的码，自信心也比较高。一个人不应该用猜的，或是等待某大师的宣判，才确定「何时提供一个 copy constructor 而何时不需要」。这类问题的解答应该来自于我们自身对对象模型的了解。

写这本书的第三个理由是为了消除我们对于 C++ 语言（及其对对象导向的支援）的各种错误认知。下面一段话节录自我收到的一封信，来信者希望将 C++ 引进其程序环境中：

> 我和一群人工作，他们过去不曾写过（或完全不熟悉）C++ 和 OO。其中一位工程师从 1985 就开始写 C 了，他非常强烈地认为 C++ 只对那些 user-type 程序才好用，对 server 程序却不理想。他说如果要写一个快速而有效率的数据库引擎，应该使用 C 而非 C++。他认为 C++ 庞大又迟缓。

C++ 当然并不是天生地庞大又迟缓，但我发现这似乎成为 C 程序员的一个共识。然而，光是这么说并不足以使人信服，何况我又被认为是 C++ 的「代言人」。这本书就是企图极尽可能地将各式各样的 Object facilities（如 inheritance、virtual functions、指向 class members 的指标…）所带来的额外负荷说个清楚。

除了我个人回答这封信，我也把此信转寄给 HP 的 Steve Vinoski；先前我曾与他讨论过 C++ 的效率问题。以下节录自他的回应：

> 过去数年我听过太多与你的同事类似的看法。许多情况下，这些看法是源于对 C++ 事实真象的缺乏了解。就在上周，我才和一位朋友闲聊，他在一家 IC 制造厂服务，他说他们不使用 C++，因为「它在你的背后做事情」。我紧迫盯人，于是他说根据他的了解，C++ 呼叫 malloc() 和 free() 而不让程序员知道。这当然不是真的。这是一种所谓的迷思与传说，引导出类似于你的同事的看法...
>
> 在抽象性和实际性之间找出平衡点，需要知识、经验、以及许多思考。C++ 的使用需要付出许多心力，但是我的经验告诉我，这项投资的报酬率相当高。

我喜欢把这本书想象是我对那一封读者来信的回答。是的，这本书是一个知识陈列库，帮助大家去除围绕在 C++ 四周的迷思与传说。

如果 C++ 对象模型的底层机制会因为实作品（编译器）和时间的变动而不同，我如何能够对于任何特定主题提供一般化的讨论呢？静态初始化（Static initialization）可为此提供一个有趣的例子。

已知一个 class X 有着 constructor，像这样：

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

而一个 class X 的 global object 的宣告，像这样：

```
X buf;

main()
{
  // buf 必须在这个时候建构起来
  cin >> setw( 1024 ) >> buf;
  ...
}
```

C++ 对象模型保证，X constructor 将在 main() 之前便把 buf 初始化。然而它并没有说明这是如何办到的。答案是所谓的静态初始化（static initialization），实际作法则有赖开发环境对此的支持属于哪一层级。

原始的 cfront 实作品不单只是假想没有环境支持，它也假想没有明白的目标平台。唯一能够假想的平台就是 UNIX 及其衍化的一些变体。我们的解决之道也因此只专注在 UNIX 身上：我们使用 nm 命令。CC 命令（一个 UNIX shell script）产生出一个可执行档，然后我们把 nm 施行于其上，产生出一个新的 .c 档案。然后编译此一新的 .c 档，再重新联结出一个可执行档（这就是所谓的 munch solution）。这种作法是以编译器时间来交换移植性。

接下来是提供一个「平台特定」解决之道：直接验证并穿越 COFF-based 程序的可执行档（此即所谓的 patch solution），不再需要 nm、compile、relink。COFF 是 Common Object File Format 的缩写，是 System V pre-Release 4 UNIX 系统所发展出来的格式。这两种解决方案都属于程序层面，也就是说，针对每一个需要静态初始化的 .c 檔，cfront 会产生出一个 sti 函式，执行必要的初始化动作。不论是 patch solution 或是 munch solution，都会去寻找以 sti 开头的函式，并且安排它们以一种未被定义的次序执行起来（藉由安插在 main() 之后第一行的一个 library function _main() 执行之）（译注：本书第 6 章对此有详细说明）。

System V COFF-specific C++ 编译器与 cfront 的各个版本平行发展。由于瞄准了一个特定平台和特定操作系统，此编译器因而能够影响联结器特别为它修改：产生出一个新的 .ini section，用以收集需要静态初始化的 objects。联结器的这种扩

充方式，提供了所谓的 environment-based solution，那当然更在 program-based solution 层次之上。

至此，任何以 cfront program-based solution 为基础的一般化（泛型）动作将令人迷惑。为什么？因为 C++ 已经成为主流语言，它已经接收了更多更多的 environment-based solutions。这本书如何维护其间的平衡呢？我的策略如下：如果在不同的 C++ 编译器上有重大的实作技术差异，我就讨论至少两家作法。但如果 cfront 之后的编译器实作模型只是解决 cfront 原本就已理解的问题，例如对虚拟继承的支持，那么我就阐述历史的演化。当我说到「传统模型」，我的意思是 Stroustrup 的原始构想（反应在 cfront 身上），它提供一种实作模范，在今天所有的商业化实作品上仍然可见。

本书组织

第 1 章，关于对象（Object **Lessons**），提供以对象为基础的观念背景，以及由 C++ 提供的对象导向程序设计典范 （paradigm。译注：关于 paradigm 这个字，请参阅本书 #22 页的译注）。本章包括对对象模型的一个大略游览，说明目前普及的工业产品，但没有对于多重继承和虚拟继承有太靠近的观察（那是第 3 章和第 4 章的重头戏）。

第 2 章，建构式语意学（The **Semantics of Constructors**），详细讨论 constructor 如何工作。本章谈到 constructors 何时被编译器合成，以及对你的程式效率带来什么样的意义。

第 3 章至第 5 章是本书的重要题材。在这里，我详细地讨论了 C++ 对象模型的细节。第 3 章，Data 语意学（The **Semantics of Data**），讨论 data members 的处理。第 4 章，Function 语意学（The **Semantics of Function**），专注于各式各样的 member functions，并特别详细地讨论如何支援 virtual functions。第 5 章，建构、解构、拷贝语意学（Semantics **of Construction, Destruction, and Copy**），讨论如何支持 class 模型，也讨论到 object 的生

命期。每一章都有测试程序以及测试数据。我们对效率的预测，将拿来和实际结果做一比较。

第 6 章，执行时期语意学（Runtime **Semantics**），检视执行时期的某些对象模型行为。包括暂时对象的生命及其死亡，以及对 new 运算子和 delete 运算子的支援。

第 7 章，在对象模型的尖端（On **the Cusp of the Object Model**），专注于 exception handling、template support、runtime type identification。

预定的读者

这本书可以扮演家庭教师的角色，不过它定位在中阶以上的 C++ 程序员，而非 C++ 新手。我尝试提供足够的内容，使它能够被任何有点 C++ 基础（例如读过我的 C++ Primer 并有一些实际程序经验）的人接受。理想的读者是，曾经有过数年的 C++ 程序经验，希望更了解「底层做些什么事」的人。书中某些部份甚至对于 C++ 高手也具吸引力，像是暂时性对象的产生，以及 named return value（NRV）最佳化的细节等等。在与本书相同素材的各个公开演讲场合中，我已经证实了这些材料的吸引力。

程序范例及其执行

本书的程序范例主要有两个目的：

1. 为了提供书中所谈之 C++ 对象模型各种概念之具体说明。

2. 提供测试，以量测各种语言性质之相对成本。

无论哪一种意图，都只是为了展现对象模型。举例而言，虽然我在书中有大量的举例，但我并非建议一个真实的 3D graphic library 必须以虚拟继承的方式来表现一个 3D 点（不过你可以在 [POKOR94] 中发现作者 Pokorny 的确这么做）。

书中所有测试程序都在一部 SGI Indigo2xL 上编译执行，使用 SGI 5.2 UNIX 作业系统中的 CC 和 NCC 编译器。CC 是 cfront 3.0.1 版（它会产生出 C 码，再由一个 C 编译器重新编译为可执行档）。NCC 是 Edison Design Group 的 C++ front-end 2.19 版，内含一个由 SGI 供应的程序代码产生器。至于时间量测，是采用 UNIX 的 timex 命令针对一千万次迭代测试所得的平均值。

虽然在 xL 机器上使用这两个编译器，对读者而言可能觉得有些神秘，我却觉得对此书的目的而言，很好。不论是 cfront 或现在的 Edison Design Group's C++ front-end（Bjarne 称其为「cfront 的儿子」），都与平台无关。它们是一种一般化的编译器，被授权给 34 家以上的计算机制造商（其中包括 Gray、SGI、Intel）和软件开发环境厂商（包括 Centerline 和 Novell，后者是原先的 UNIX 软件实验室）。效率的量测并非为了对目前市面上各家编译系统做评比，而只是为了提供 C++ 对象模型之各种特性的一个相对成本量测。至于商业评比的效率数据，你可以在几乎任何一本计算机杂志的计算机产品检验报告中获得。

致谢

略

参考书目

[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 0 章

导读
（译者的话）

合适的读者

很不容易三言两语就说明此书的适当读者。　　作者 Lippman 参与设计了全世界第一套 C++ 编译器 cfront，这本书就是一位伟大的 C++ 编译器设计者向你阐述他如何处理各种 explicit（明白出现于 C++ 程序代码）和 implicit（隐藏于程序代码背后）的 C++ 语意。

对于 C++ 程序老手，这必然是一本让你大呼过瘾的绝妙好书。

C++ 老手分两类。一种人把语言用得烂熟，OO 观念也有。另一种人不但如此，还对于台面下的机制如编译器合成的 default constructor 啦、object 的内存布局啦...有莫大的兴趣。本书对于第三类老手的吸引力自不待言，至于第一类老手，或许你没那么大的刨根究底的兴趣，不过我还是非常推荐你阅读此书。了解 C++ 对象模型，绝对有助于你在语言本身以及对象导向观念两方面的层次提升。

你需要细细推敲每一个句子，每一个例子，囫囵吞枣是完全没有用的。作者是 C++ 大师级人物，并且参与开发了第一套 C++ 编译器，他的解说以及诠释非常鞭辟入里，你务必在看过每一小段之后，融会贯通，把思想观念化为己有，再接续另一小节。但阅读次序并不需要按照书中的章节排列。

阅读次序

我个人认为，第 1, 3, 4 章最能带给读者立即而最大的帮助，这些都是经常引起程式员困惑的主题。作者在这些章节中有不少示意图（我自己也加了不少）。你或许可以从这三章挑着看起。

其它章节比较晦涩一些（我的感觉），不妨「视可而择之」。

当然，这都是十分主观的认定。客观的意见只有一个：你可以随你的兴趣与需求，从任一章开始看起。各章之间没有必然关联性。

翻译风格

太多朋友告诉我，他们阅读中文计算机书籍，不论是著作或译作，最大的阅读困难在于一大堆没有标准译名的技术名词或习惯用语（至于那些误谬不知所云的奇怪作品当然本就不在考虑之列）。其实，就算国立编译馆有统一译名（或曾有过，谁知道？），流通于工业界与学术界之间的还是原文名词与术语。

对于工程师，我希望我所写的书和我所译的书能够让各位读来通体顺畅；对于学生，我还希望多发挥一点引导的力量，引导各位多使用、多认识原文术语和专有名词，不要说出像「无模式对话盒（modeless dialog）」这种奇怪的话。

由于本书读者定位之故，我决定保留大量的原文技术名词与术语。我清楚地知道，在我们的技术领域里，研究人员或工程师如何使用这些语汇。

当然，有些中文译名够普遍，也够有意义，我并不排除使用。其间的挑选与决定，不可避免地带了点个人色彩。

下面是本书出现的原文名词（按字母排序）及其意义：

英文名词	中文名词或（及）其意义
access level	存取层级。就是 C++ 的 public、private、protected 三种等级。
access section	存取区段。就是 class 中的 public、private、protected 三种段落。
alignment	边界调整，调整至某些 bytes 的倍数。其结果视不同的机器而定。例如 32 位机器通常调整至 4 的倍数。
bind	系结，将程序中的某个符号真正附着（决议）至一块实体上。
chain	串链
class	类别
class hierarchy	class 体系，class 阶层架构
composition	组合。通常与继承（inheriance）并同讨论。
concrete inheritance	具体继承（相对于抽象继承）
constructor	建构式
data member	资料成员（亦或被称为 member variable）
declaration, declare	宣告
definition, define	定义（通常附带「在内存中挖一块空间」的行为）
derived	衍生
destructor	解构式
encapsulation	封装
explicit	明白的（通常指 C++ 程序代码中有出现的）
hierarchy	体系，阶层架构
implement	实作（动词）
implementation	实作品、实作物。本书有时候指 C++ 编译器。大部份时候

英文名词	中文名词或（及）其意义
	是指 class member function 的内容。
implicit	隐含的、暗喻的（通常指未出现在 C++ 程序代码中的）
inheritance	继承
inline	行内（C++ 的一个关键词）
instance	实体（有些书籍译为「案例」，极不妥当）
layout	布局。本书常常出现这个字，意指 object 在内存中的数据分布情况。
mangle	名称切割重组（C++ 对于函式名称的一种处理方式）
member function	成员函式。亦或被称为 function member。
members	成员，泛指 data members 和 member functions
object	对象（根据 class 的宣告而完成的一份占有内存的实体）
offset	偏移位置
operand	操作数
operator	运算子
overhead	额外负担（因某种设计，而导至的额外成本）
overload	多载
overloaded function	多载函式
override	改写（对 virtual function 的重新设计）
paradigm	典范（请参考 #22 页）
pointer	指标
polymorphism	多型（「对象导向」最重要的一个性质）
programming	程序设计、程序化
reference	参考、参用（动词）。
reference	C++ 的 & 运算子所代表的东西。当名词解。
resolve	决议。函式呼叫时联结器所做的一种动作，将符号与函式实体产生关系。如果你呼叫 func() 而联结时找不到 func() 实体，就会出现 "unresolved externals" 联结错误。
slot	表格中的一格（一个元素）；条孔；条目；条格。

英文名词	中文名词或（及）其意义
subtype	子型别
type	型态，型别（指的是 int、float 等内建型别，或 C++ classes 等自定型别）
virtual	虚拟
virtual function	虚拟函式
virtual inheritance	虚拟继承
virtual table	虚拟表格（为实现虚拟机制而设计的一种表格，内放 virtual functions 的地址）

有时候考虑到上下文的因素，面对同一个名词，在译与不译之间，我可能会有不同的选择。例如，面对 "pointer"，我会译为「指标」，但由于我并未将 reference 译为「参考」（实在不对味），所以如果原文是 "the manipulation of a pointer or reference in C++ ..."，为了中英对等或平衡的缘故，我不会把它译为「C++ 中对于指标和 reference 的操作行为...」，我会译为「C++ 中对于 pointer 和 reference 的操作行为...」。

译注

书中有一些译注。大部份译注，如果够短的话，会被我直接放在括号之中，接续本文。较长的译注，则被我安排在被注文字的段落下面（紧临，并加标示）。

原书错误

这本书虽说质地极佳，制作的严谨度却不及格！有损 Lippman 的大师地位。

属于「作者笔误」之类的错误，比较无伤大雅，例如少了一个；符号，或是多了一个；符号，或是少了一个｝符号，或是多了一个）符号等等。比较严重的错误，是程序代码变量名称或函式名称或 class 名称与文字叙述不一致，甚或是图片中对于 object 布局的画法，与程序代码中的宣告不一致。这两种错误都会严重耗费

读者的心神。

只要是被我发现的错误，都已被我修正。以下是错误更正列表。

示例：L5 表示第 5 行，L-9 表示倒数第 9 行。页码所示为原书页码。

页码	原文位置	原文内容	应修改为
p.35	最后一行	Bashful(),	Bashful();
p.57	表格第三行	1.32.36	1:32.36
p.61	L1	memcpy... 程序代码最后少一个)	
p.61	L10	Shape()...程序代码最后少了一个 }	
p.64	L-9	程序代码最后多了一个 ;	
p.78	最后四行码	==	似乎应为 =
p.84	图 3.1b 说明	struct Point3d	class Point3d
p.87	L-2	virtual... 程序代码最后少了一个 ;	
p.87	全页多处	pc2_2（不符合命名意义）	pc1_2（符合命名意义）
p.90	图 3.2a 说明	Vptr placement and end of class	Vptr placement at end of class
p.91	图 3.2(b)	__vptr__has_vrts	__vptr__has_virts
p.92	码 L-7	class Vertex2d	class Vertex3d
p.92	码 L-6	public Point2d	public Point3d
p.93	图 3.4 说明	Vertex2d 的物件布局	Vertex3d 的物件布局
p.92~ p.94		符号名称混乱，前后误谬不符	已全部更改过
p.97	码 L2	public Point3d, public Vertex	配合图 3.5ab，应调整次序为 public Vertex, public Point3d
p.99	图 3.5(a)	符号与书中程序代码多处不符	已全部更改过
p.100	图 3.5(b)	符号与书中程序代码多处不符	已全部更改过
p.100	L-2	? pv3d + ... 最后多了一个)	
p.106	L16	pt1d::y	pt2d::_y
p.107	L10	& 3d_point::z;	&Point3d::z;

页码 原文位置	原文内容	应修改为
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	不符合稍早出现的程序代码	把 pt3d 改为 Point3d
p.115 L1	magnitude()	magnitude3d()
p.126 L12	Point2d pt2d = new Point2d;	ptr = new Point2d;
p.136 图 4.2 右下	Derived::~close()	Derived::close()
p.138 L-12	class Point3d... 最后少一个 {	
p.140 程序代码	没有与文字中的 class 命名一致 所有的 pt3d 改为 Point3d	
p.142 L-7	if (this ... 程序代码最右边少一个)	
p.143 程序代码	没有与文字中的 class 命名一致 所有的 pt3d 改为 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 中段码 L-1	程序代码最后缺少一个)	
p.148 中段码 L1	(ptr->*pmf) 函式最后少一个 ；	
p.148 中段码 L-1	(*ptr->vptr[.. 函式最后少一个)	
p.150 程序代码	没有与文字中的 class 命名一致 所有的 pt3d 改为 Point3d	
p.150 L-7	pA.__vptr__pt3d... 最后少一个 ；	
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 函式最后少一个 ；	
p.166 中，码 L3	Point1 local1 = ...	Point local1 = ...
p.166 中，码 L4	Point2 local2;	Point local2;

页码	原文位置	原文内容	应修改为
p.174	中，码 L-1	Line::Line() 函式最后多了一个；	
p.174	中下，码 L-1	Line::Line() 函式最后多了一个；	
p.175	中上，码 L-1	Line::~Line() 函式最后多一个；	
p.182	中下，码 L6	Point3d::Point3d()	PVertex::PVertex()
p.183	上，码 L9	Point3d::Point3d()	PVertex::PVertex()
p.185	上，码 L3	y = 0.0 之前缺少 float	
p.186	中下，码 L6	缺少一个 return	
p.187	中，码 L3	const Point3d &p	const Point3d &p3d
p.204	下，码 L3	缺少一个 return 1;	
p.208	中下，码 L2	new Pvertex;	new PVertex;
p.219	上，码 L1	__nw(5*sizeof(int));	__new(5*sizeof(int));
p.222	上，码 L8	// new (ptr_array... 程序代码少个；	
p.224	中，码 L1	Point2w ptw = ...	Point2w *ptw = ...
p.224	下，码 L5	operator new() 函式定义多一个；	
p.225	上，码 L2	Point2w ptw = ...	Point2w *ptw = ...
p.226	下，码 L1	Point2w p2w = ...	Point2w *p2w = ...
p.229	中，码 L1	c.operator==(a + b);	c.operator=(a + b);
p.232	中下，码 L2	x xx;	X xx;
p.232	中下，码 L3	x yy;	X yy;
p.232	下，码 L2	struct x _1xx;	struct X _1xx;
p.232	下，码 L3	struct x _1yy;	struct X _1yy;
p.233	码 L2	struct x __0__Q1;	struct X __0__Q1;
p.233	码 L3	struct x __0__Q2;	struct X __0__Q2;
p.233	中，码	if 条件句的最后多了一个；	
p.253	码 L-1	foo() 函式码最后多了一个；	

推荐

我个人翻译过不少书籍，每一本都精挑细选后才动手（品质不够的原文书，译它做啥?!）在这么些译本当中，我从来不做直接而露骨地推荐。好的书籍自然而然会得到识者的欣赏。过去我译的那些明显具有实用价值的书籍，总有相当数量的读者有强烈的需求，所以我从不担心没有足够的人来为好书散播口碑。但 Lippman 的这本书不一样，它可能不会为你带来明显而立即的实用性，它可能因此在书肆中蒙上一层灰（其原文书我就没听说多少人读过），枉费我从众多原文书中挑出这本好书。我担心听到这样的话：

对象模型？呵，我会写 **C++** 程序，写得一级棒，这些属于编译器层面的东西，于我何有哉！

对象模型是深层结构的知识，关系到「与语言无关、与平台无关、跨网络可执行」软件组件（software component）的基础原理。也因此，了解 C++ 对象模型，是学习目前软件组件三大规格（COM、CORBA、SOM）的技术基础。

如果你对软件组件（software component）没有兴趣，C++ 对象模型也能够使你对虚拟函式、虚拟继承、虚拟接口有脱胎换骨的新体认，或是对于各种 C++ 写法所带来的效率利益有通盘的认识。

我因此要大声地说：有经验的 C++ programmer 都应该看看这本书。

如果您对 COM 有兴趣，我也要同时推荐你看另一本书：Essential COM，Don Box 着，Addison Wesley 1998 出版（COM 本质论，侯捷译，碁峰 1998）。这也是一本论述非常清楚的书籍，把 COM 的由来（为什么需要 COM、如何使用 COM）以循序渐进的方式阐述得非常深刻，是我所看过最理想的一本 COM 基础书籍。

看 Essential COM 之前，你最好有这本 Inside The C++ Object Model 的基础。

第 3 章

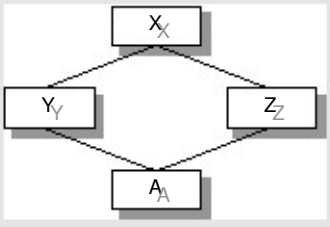
Data 语意学

（The Semantics of Data）

前些时候我收到一封来自法国的电子邮件，发信人似乎有些迷惘也有些烦乱。他志愿（要不就是被选派）为他的项目团队提供一个「永恒的」library。在做准备工作的时候，他写出以下的码并打印出它们的 sizeof 结果：

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

译注：X, Y, Z, A 的继承关系如下图所示：



上述 X, Y, Z, A 中没有任何一个 class 内含明显的资料，其间只表示了继承关系。所以发信者认为每一个 class 的大小都应该是 0。当然不对！即使是 class X 的大小也不为 0：

```
sizeof  X 的结果为  1
sizeof  Y 的结果为  8
sizeof  Z 的结果为  8
sizeof  A 的结果为  12
```

译注：以下是我在 Visual C++ 5.0 上的执行结果

```
sizeof X 的结果为 1
sizeof Y 的结果为 4
sizeof Z 的结果为 4
sizeof A 的结果为 8
```

原因将在 p.86 的译注和 p.87 的正文中解释

让我们依序看看每一个 class 的宣告，并看看它们为什么获得上述结果。

一个空的 class 如：

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

事实上并不是空的，它有一个隐藏的 1 byte 大小，那是被编译器安插进去的一个 char。这使得此一 class 的两个 objects 得以在内存中配置独一无二的地址：

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

令来信读者感到惊讶和沮丧的，我怀疑是 Y 和 Z 的 sizeof 结果：

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

在来信者的机器上，Y 和 Z 的大小都是 8。这个大小和机器有关，也和编译器有关。事实上 Y 和 Z 的大小受到三个因素的影响：

1. 语言本身所造成的额外负担（overhead）　当语言支持 virtual base classes，就会导至一些额外负担。在 derived class 中，这个额外负担反映在某种型式的指针身上，它或者指向 virtual base class subobject，或者指向一个相关表格；表格中存放的若不是 virtual base class subobject 的地址，就是其偏移位置（offset）。在来信者的机器上，指标是 4 bytes

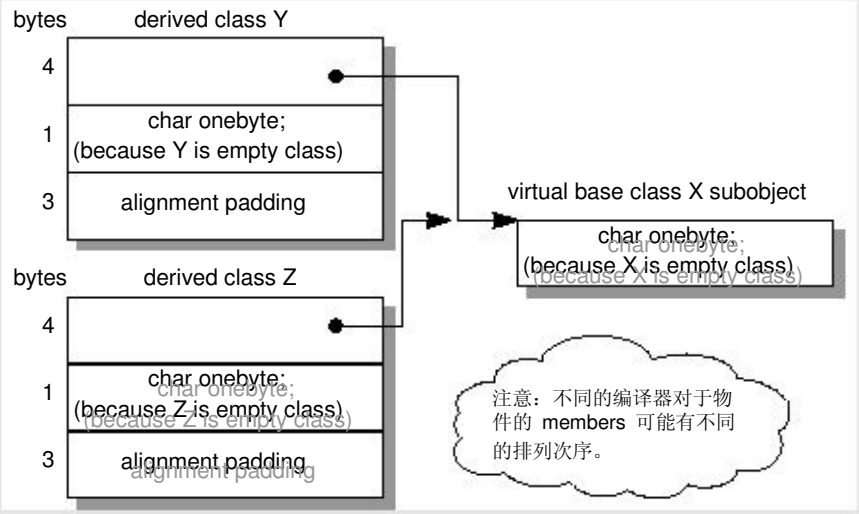
（我将在 3.4 节讨论 virtual base class）。

2. 编译器对于特殊情况所提供的最佳化处理　　Virtual base class X subobject 的 1 bytes 大小也出现在 class Y 和 Z 身上。传统上它被放在 derived class 的固定（不变动）部份的尾端。某些编译器会对 empty virtual base class 提供特殊支援（以下第三点之后的一段文字对此有比较详细的讨论）。来信读者所使用的编译器，显然并未提供这项特殊处理。

3. **Alignment** 的限制　　class Y 和 Z 的大小截至目前为 5 bytes。在大部份机器上，群聚的结构体大小会受到 alignment 的限制，使它们能够更有效率地在记忆体中被存取。在来信读者的机器上，alignment 是 4 bytes，所以 class Y 和 Z 必须填补 3 bytes。最终得到的结果就是 8 bytes。

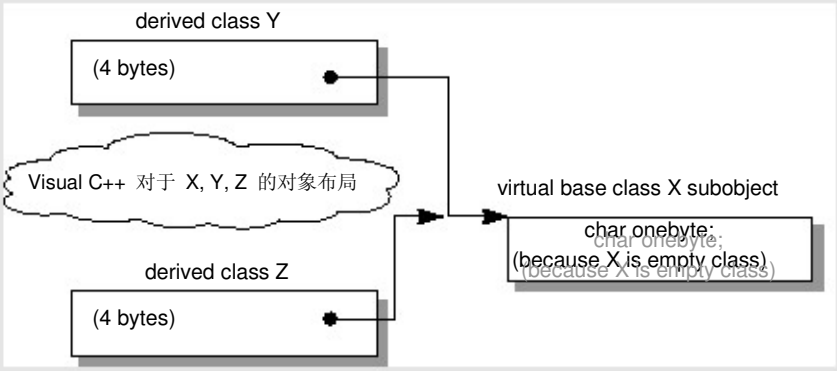
> 译注：alignment 就是将数值调整到某数的整数倍。在 32 位计算机上，通常 alignment 为 4 bytes（32 位），以使 bus 的「运输量」达到最高效率。

译注：我以下图表现上述的 X, Y, Z 对象布局



Empty virtual base class 已经成为 C++ OO 设计的一个特有术语了。它提供一个 virtual interface，没有定义任何数据。某些新近的编译器（译注）对此提供了特殊处理（请看 [SUN94a]）。在这个策略之下，一个 empty virtual base class 被视为 derived class object 最开头的一部份，也就是说它并没有花费任何的额外空间。这就节省了上述第 2 点的 1 bytes （译注：因为既然有了 members，就不需要原本为了 empty class 而安插的一个 char），也就不再需要第 3 点所说的 3 bytes 的填补。只剩下第 1 点所说的额外负担。在此模型下，Y 和 Z 的大小都是 4 而不是 8。

译注：Visual C++ 就是上述这一类型的编译器。我以下图来表现 Visual C++ 对于 class X, Y, Z 的物件布局：



编译器之间的潜在差异正说明了 C++ 对象模型的演化。这个模型为一般情况提供了解决之道。当特殊情况逐渐被挖掘出来，种种启发（尝试错误）法于是被引入，提供最佳化的处理。如果成功，启发法于是就提升为普遍的策略，并跨越各种实作品而合并。它被视为标准（虽然它并不被规范为标准），久而久之也就成了语言的一部份。Virtual function table 就是一个好例子，另一个例子是第 2 章讨论过的「named return value（NRV）最佳化」。

那么，你期望 class A 的大小是什么呢？很明显，某种程度上必须视你所使用的编译器而定。首先，请你考虑那种并未特别处理 empty virtual base class 的编译器。如果我们忘记 Y 和 Z 都是「虚拟衍生」自 class X，我们可能会回答 16，毕竟 Y 和 Z 的大小都是 8。然而当我们对 class A 施以 sizeof 运算子，得到的答案竟然是 12。到底怎么回事？

记住，一个 virtual base class subobject 只会在 derived class 中存在一份实体，不管它在 class 继承体系中出现了多少次！class A 的大小由下列数点决定：

被大家共享的唯一一个 class X 实体，大小为 1 byte。

Base class Y 的大小，减去「因 virtual base class X 而配置」的大小，结果是 4 bytes。Base class Z 的算法亦同。加起来是 8 bytes。

class A 自己的大小：0 byte。

class A 的 alignment 数量（如果有的话）。前述三项总合，表示调整前的大小是 9 bytes。class A 必须调整至 4 bytes 边界，所以需要填补 3 bytes。结果是 12 bytes。

现在如果我们考虑那种「特别对 empty virtual base class 做了处理」的编译器呢？一如前述，class X 实体的那 1 byte 将被拿掉，于是额外的 3 bytes 填补额也不必了，因此 class A 的大小将是 8 bytes。注意，如果我们在 virtual base class X 中放置一个（以上）的 data members，两种编译器（「有特殊处理」者和「没有特殊处理」者）就会产生出完全相同的对象布局。

C++ Standard 并不强制规定如「base class subobjects 的排列次序」或「不同存取层级的 data members 的排列次序」这种琐碎细节。它也不规定 virtual functions 或 virtual base classes 的实作细节。C++ Standard 只说：那些细节由各家厂商自定。我在本章以及全书中，都会区分「C++ Standard」和「目前的 C++ 实作标准」两种讨论。

在这一章中，class 的 data members 以及 class hierarchy 是中心议题。一个 class 的 data members，一般而言，可以表现这个 class 在程序执行时的某种状态。Nonstatic data members 放置的是「个别的 class object」感兴趣的数据，static data members 则放置的是「整个 class」感兴趣的资料。

C++ 对象模型尽量以空间最佳化和存取速度最佳化的考虑来表现 nonstatic data members，并且保持和 C 语言 struct 数据配置的兼容性。它把数据直接存放在每一个 class object 之中。对于继承而来的 nonstatic data members 不管是 virtual 或 nonvirtual base class） 也是如此。不过并没有强制定义其间的排列顺序。至于 static data members 则被放置在程序的一个 global data segment 中，不会影响个别的 class object 的大小。在程序之中，不管该 class 被产生出多少个 objects（经由直接产生或间接衍生），static data members 永远只存在一份实体（译注：甚至即使该 class 没有任何 object 实体，其 static data members 也已存在）。但是一个 template class 的 static data members 的行为稍有不同，7.1 节有详细的讨论。

每一个 class object 因此必须有足够的大小以容纳它所有的 nonstatic data members。有时候其值可能令你吃惊（正如那位法国来信者），因为它可能比你想像的还大，原因是：

1. 由编译器自动加上的额外 data members，用以支持某些语言特性（主要是各种 virtual 特性）。
2. 因为 alignment（边界调整）的需要。

3.1 Data Member 的系结（The Binding of a Data Member）

考虑下面这段程序代码：

```
// 某个 foo.h 表头档，从某处含入
extern float x;
```

```
// 程序员的 Point3d.h 档案
class Point3d
{
public:
    Point3d( float, float, float );
    // 问题：被传回和被设定的 x 是哪一个 x 呢?
    float X() const { return x; }
    void X( float new_x ) const { x = new_x; }
    // ...
private:
    float x, y, z;
};
```

如果我问你 Point3d::X() 传回哪一个 x？是 class 内部那个 x，还是外部（extern）那个 x？今天每个人都会回答我是内部那一个。这个答案是正确的，但并不是从过去以来一直都正确！

在 C++ 最早的编译器上，如果在 Point3d::X() 的两个函式实体中对 x 做出参阅（取用）动作，这动作将会指向 global x object！这样的系结结果几乎普遍地不在大家的预期之中，并因此导出早期 C++ 的两种防御性程序设计风格：

1. 把所有的 data members 放在 class 宣告起头处，以确保正确的系结：

```
class Point3d
{
    // 防御性程序设计风格 #1
    // 在 class 宣告起头处先放置所有 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:
    // 防御性程序设计风格 #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. ...
```

这些程序设计风格事实上到今天还存在，虽然它们的必要性已经自从 C++ 2.0 之后（伴随着 C++ Reference Manual 的修订）就消失了。这个古早的语言规则被称为 "member rewriting rule"，大意是「一个 inline 函式实体，在整个 class 宣告未被完全看见之前，是不会被评估求值 （evaluated）的」。C++ Standard 以 "member scope resolution rules" 来精炼这个 "rewriting rule"，其效果是，如果一个 inline 函式在 class 宣告之后立刻被定义的话，那么就还是对其评估求值（evaluate）。也就是说，当一个人写下这样的码：

```
extern int x;

class Point3d
{
public:
    // 对于函式本体的分析将延迟直至
    //   class 宣告的右大括号出现才开始。
    float X() const { return x; }
    // ...
private:
    float x;
};

// 事实上，分析在这里进行
```

对 member functions 本体的分析，会直到整个 class 的宣告都出现了才开始。因此在一个 inline member function 躯体之内的一个 data member 系结动作，会在整

个 class 宣告完成之后才发生。

然而，这对于 member function 的 argument list 并不为真。Argument list 中的名称还是会在它们第一次遭遇时被适当地决议（resolved）完成。因此在 extern 和 nested type names 之间的非直觉系结动作还是会发生。例如在下面的程序片段中，length 的型别在两个 member function signatures 中都决议（resolve）为global typedef，也就是 int。当后续再有 length 的 nested typedef 宣告出现，C++ Standard 就把稍早的系结标示为非法：

```
typedef int length;

class Point3d
{
public:
    // 喔欧: length 被决议（resolved）为 global
    // 没问题: _val 被决议（resolved）为 Point3d::_val
    void mumble( length val ) { _val = val; }
    length mumble() { return _val; }
    // ...

private:
    // length 必须在「本 class 对它的第一个参考动作」之前被看见。
    // 这样的宣告将使先前的参考动作不合法。
    typedef float length;
    length _val;
    // ...
};
```

上述这个语言状况，仍然需要某种防御性程序风格：请总是把「nested type 宣告」放在 class 的起始处。在上述例子中，如果把 length 的 nested typedef 定义于「在 class 中被参考」之前，就可以确保非直觉系结的正确性。

3.2 Data Member 的布局（Data Member Layout）

已知下面一组 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 中的排列顺序将和其被宣告的顺序一样，任何中间介入的 static data members 如 freeList 和 chunkSize 都不会被放进对象布局之中。在上述例子里，每一个 Point3d 对象是由三个 float 组成，次序是 x, y, z。static data members 存放在程序的 data segment 中，和个别的 class objects 无关。

C++ Standard 要求，在同一个 access section （也就是 private、public、protected 等区段）中，members 的排列只需符合「较晚出现的 members 在 class object 中有较高的地址」这一条件即可（请看 C++ Standard 9.2 节）。也就是说各个 members 并不一定得连续排列。什么东西可能会介于被宣告的 members 之间呢？members 的边界调整（alignment）可能就需要填补一些 bytes。对于 C 和 C++ 而言这的确是真的，对目前的 C++ 编译器实作情况而言，这也是真的。

编译器还可能会合成一些内部使用的 data members 以支持整个对象模型 vptr 就是这样的东西，目前所有的编译器都把它安插在每一个「内含 virtual function 之 class」的 object 内。vptr 会被放在什么位置呢？传统上它被放在所有明白宣告的 members 的最后头。不过如今也有一些编译器把 vptr 放在一个 class object 的最

前端。C++ Standard 秉持先前所说的「对于布局所持的放任态度」，允许编译器把那些内部产生出来的 members 自由放在任何位置，甚至放在那些被程序员宣告出来的 members 之间。

C++ Standard 也允许编译器将多个 access sections 之中的 data members 自由排列，不必在乎它们出现在 class 宣告中的次序。也就是说，下面这样的宣告：

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

其 class object 的大小和组成都和我们先前宣告的那个相同，但是 members 的排列次序则视编译器而定。编译器可以随意把 y 或 z 或什么东西放第一个，不过就我所知道，目前没有任何编译器会这么做。

目前各家编译器都是把一个以上的 access sections 连锁在一起，依照宣告的次序，成为一个连续区块。Access sections 的多寡并不会招来额外负担。例如在一个 section 中宣告 8 个 members，或是在 8 个 sections 中总共宣告 8 个 members，得到的 object 大小是一样的。

下面这个 template function，接受两个 data members，然后判断谁先出现在 class object 之中。如果两个 members 都是不同的 access sections 中的第一个被宣告者，此函式就可以用来判断哪一个 section 先出现（如果你对 class member 的指标并不熟悉，请参考 3.6 节）：

```
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";
}
```

上述函式可以这样被唤起：

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

于是 class_type 会被系结为 Point3d，而 data_type1 和 data_type2 会被系结为 float。

3.3 Data Member 的存取

已知下面这段程序代码：

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

你可能会问 x 的存取成本是什么？答案视 x 和 Point3d 如何宣告而定。x 可能是个 static member，也可能是个 nonstatic member。Point3d 可能是个独立（非衍生）的 class，也可能是从另一个单一的 base class 衍生而来；虽然可能性不高，但它甚至可能是多重继承或虚拟继承而来。下面数节将依次检验每一种可能性。

在开始之前，让我先丢出一个问题。如果我们有两个定义，origin 和 pt：

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

我用它们来存取 data members，像这样：

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

透过 origin 存取，和透过 pt 存取，有什么重大差异吗？如果你的回答是 yes，请你从 class Point3d 和 data member x 的角度来说明差异的发生因素。我会在这一节结束前重返这个问题并提出我的答案。

Static Data Members

Static data members，按其字面意义，被编译器提出于 class 之外，一如我在 1.1 节所说，并被视为一个 global 变量（但只在 class 生命范围之内可见）。每一个 member 的存取许可（译注：private 或 protected 或 public），以及与 class 的关联，并不会招致任何空间上 或执行时间上 的额外负担 -- 不论是在个别的 class objects 或是在 static data member 本身。

每一个 static data member 只有一个实体，存放在程序的 data segment 之中。每次程序参阅（取用）static member，就会被内部转化为对该唯一之 extern 实体的直接参考动作。例如：

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

从指令执行的观点来看，这是 C++ 语言中「透过一个指针和透过一个对象来存取 member，结论完全相同」的唯一一种情况。这是因为「经由 member selection operators（译注：也就是 '.' 运算子）对一个 static data member 做存取动作」只是文法上的一种便宜行事而已。member 其实并不在 class object 之中，因此存取 static members 并不需要透过 class object。

但如果 chunkSize 是一个从复杂继承关系中继承而来的 member，又当如何？或许它是一个「virtual base class 的 virtual base class」（或其它同等复杂的继承架构）的 member 也说不定。哦，那无关紧要，程序之中对于 static members 还是只有唯一一个实体，而其存取路径仍然是那么直接。

如果 static data member 的存取是经由函式呼叫，或其它某些语法呢？举个例子，如果我们写：

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

唤起 foobar() 会发生什么事情？在 C++ 的准标准（pre-Standard）规格中，没有人知道会发生什么事，因为 ARM 并未指定 foobar() 是否必须被求值（evaluated）。cfront 的作法是简单地把它丢掉。但 C++ Standard 明白要求 foobar() 必须被求值（evaluated），虽然其结果并无用处。下面是一种可能的转化：

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

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

若取一个 static data member 的地址，会得到一个指向其数据型别的指针，而不是一个指向其 class member 的指标，因为 static member 并不内含在一个 class object 之中。例如：

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

会获得型态如下的内存地址：

```
const int*
```

如果有两个 classes，每一个都宣告了一个 static member freeList，那么当它们都被放在程序的 data segment，就会导至名称冲突。编译器的解决方法是暗中对每一个 static data member 编码（这种手法有个很美的名称：name-mangling），以获得一个独一无二的程序识别代码。有多少个编译器，就有多少种 name-mangling 作法！通常不外乎是表格啦、文法措辞啦等等。任何 name-mangling 作法都有两个重点：

1. 一个算法，推导出独一无二的名称。
2. 万一编译系统（或环境工具）必须和使用者交谈，那些独一无二的名称可以轻易被推导回到原来的名称。

Nonstatic Data Members

Nonstatic data members 直接存放在每一个 class object 之中。除非经由明白的（explicit）或暗喻的（implicit）class object，没有办法直接存取它们。只要程序员在一个 member function 中直接处理一个 nonstatic data member，所谓"implicit class object" 就会发生。例如下面这段码：

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

表面上所看到的对于 x, y, z 的直接存取，事实上是经由一个 "implicit class object"（由 this 指标表达）完成。事实上这个函式的参数是：

```
// member function 的内部转化
Point3d
Point3d::translate( Point3d *const this, const Point3d &pt) {
    this->x += pt.x;
    this->y += pt.y;
    this->z += pt.z;
}
```

Member functions 在本书第 4 章有比较详细的讨论。

欲对一个 nonstatic data member 做存取动作，编译器需要把 class object 的起始位址加上 data member 的偏移位置（offset）。举个例子，如果：

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

那么地址 &origin._y 将等于：

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

请注意其中的 -1 动作。指向 data member 的指标，其 offset 值总是被加上 1，这样可以使编译系统区分出「一个指向 data member 的指标，用以指出 class 的第一个 member」和「一个指向 data member 的指标，没有指出任何 member」两种情况。「指向 data members 的指标」将在 3.6 节有比较详细的讨论。

每一个 nonstatic data member 的偏移位置（offset）在编译时期即可获知，甚至如果 member 属于一个 base class subobject（衍生自单一或多重继承串链）也是一样。因此，存取一个 nonstatic data member，其效率和存取一个 C struct member 或一个 nonderived class 的 member 是一样的。

现在让我们看看虚拟继承。虚拟继承将为「经由 base class subobject 存取 class members」导入一层新的间接性，譬如：

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

其执行效率在 _x 是一个 struct member、一个 class member、单一继承、多重继承的情况下都完全相同。但如果 _x 是一个 virtual base class 的 member，存取速度会比较慢一点。下一节我会验证「继承对于 member 布局的影响」。在我们尚未进行到那里之前，请回忆本节一开始的一个问题：以两种方法存取 x 坐标，像这样：

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

「从 origin 存取」和「从 pt 存取」有什么重大的差异？答案是「当 Point3d 是一个 derived class，而其继承架构中有一个 virtual base class，并且被存取的 member（如本例的 x）是一个从该 virtual base class 继承而来的 member」时，就会有重大的差异。这时候我们不能够说 pt 必然指向哪一种 class type（因此我们也就不知道编译时期这个 member 真正的 offset 位置），所以这个存取动作必须延迟至执行时期，经由一个额外的间接导引，才能够解决。但如果使用 origin，就不会有这些问题，其型态无疑是 Point3d class，而即使它继承自 virtual base class，members 的 offset 位置也在编译时期就固定了。一个积极进取的编译器甚至可以静态地经由 origin 就解决掉对 x 的存取。

3.4「继承」与 Data Member

在 C++ 继承模型中，一个 derived class object 所表现出来的东西，是其自己的 members 加上其 base class(es) members 的总合。至于 derived class members 和 base class(es) members 的排列次序并未在 C++ Standard 中强制指定；理论上编译器可以自由安排之。在大部份编译器上头，base class members 总是先出现，但属于 virtual base class 的除外（一般而言，任何一条通则一旦碰上 virtual base class 就没辄儿，这里亦不例外）。

了解这种继承模型之后，你可能会问，如果我为 2D（二维）或 3D（三维）坐标点提供两个抽象数据型态如下：

```
// 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;
};
```

这和「提供两层或三层继承架构，每一层（代表一个维度）是一个 class，衍生自较低维层次」有什么不同？下面各小节的讨论将涵盖「单一继承且不含 virtual functions」、「单一继承并含 virtual functions」、「多重继承」、「虚拟继承」等四种情况。图 **3.1a** 就是 Point2d 和 Point3d 的对象布局图，在没有 virtual functions 的情况下（如本例），它们和 C struct 完全一样。

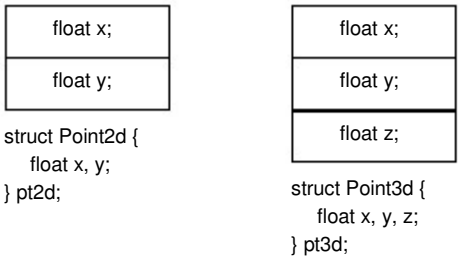
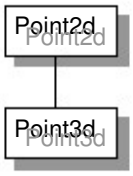


图 3.1a 个别 structs 的资料布局

只要继承不要多型（Inheritance without Polymorphism）

想象一下，程序员或许希望，不论是 2D 或 3D 坐标点，都能够共享同一个实体，但又能够继续使用「与型别性质相关（所谓 type-specific）」的实体。我们有一个设计策略，就是从 Point2d 衍生出一个 Point3d，于是 Point3d 将继承 x 和 y 座标的一切（包括数据实体和操作方法）。带来的影响则是可以共享「数据本身」以及「数据的处理方法」，并将之区域化。一般而言，具体继承（concrete inheritance，译注：相对于虚拟继承 virtual inheritance）并不会增加空间或存取时间上的额外负担。

```
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;
};
```

这样子设计的好处就是可以把管理 x 和 y 坐标的程序代码区域化。此外这个设计可以明显表现出两个抽象类别之间的紧密关系。当这两个 classes 独立的时候，

Point2d object 和 Point3d object 的宣告和使用都不会有所改变。所以这两个抽象类别的使用者不需要知道 objects 是否为独立的 classes 型态，或是彼此之间有继承的关系。图 **3.1b** 显示 Point2d 和 Point3d 继承关系的实物布局，其间并没有宣告 virtual 界面。

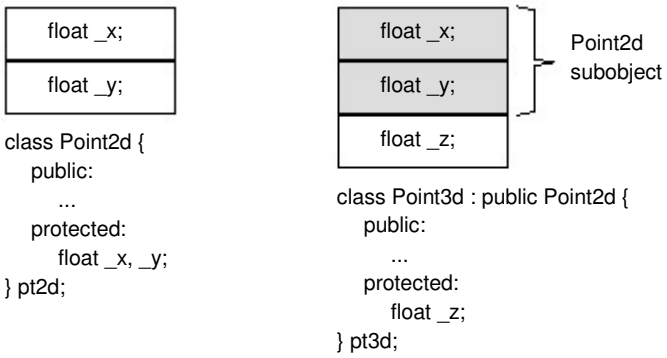
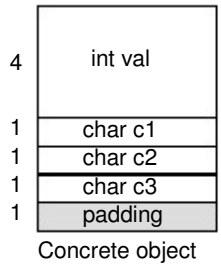


图 3.1b 单一继承而且没有 virtual function 时的资料布局

把两个原本独立不相干的 classes 凑成一对 "type/subtype"，并带有继承关系，会有什么易犯的错误呢？经验不足的人可能会重复设计一些相同动作的函式。以我们例子中的 constructor 和 operator+= 为例，它们并没有被做成 inline 函式（也可能是编译器为了某些理由没有支持 inline member functions）。Point3d object 的初始化动作或加法动作，将需要部份的 Point2d object 和部份的 Point3d object 做为成本。一般而言，选择某些函式做成 inline 函式，是设计 class 时的一个重要课题。

第二个易犯的错误是，把一个 class 分解为两层或更多层，有可能会为了「表现 class 体系之抽象化」而膨胀所需空间。C++ 语言保证「出现在 derived class 中的 base class subobject 有其完整原样性」，正是重点所在。这似乎有点难以理解！最好的解释方法就是彻底了解一个实例，让我们从一个具体的 class 开始：




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

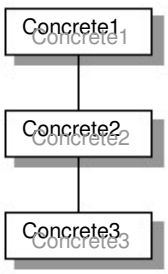
在一部 32 位机器中，每一个 Concrete class object 的大小都是 8 bytes，细分如下：

1. val 占用 4 bytes

2. c1 和 c2 和 c3 各占用 1 bytes

3. alignment（调整到 word 边界）需要 1 bytes

现在假设，经过某些分析之后，我们决定了一个更逻辑的表达方式，把 Concrete 分裂为三层架构：




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

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

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

从设计的观点来看，这个架构可能比较合理。但从实务的观点来看，我们可能会受困于一个事实：现在 Concrete3 object 的大小是 16 bytes，比原先的设计多了一倍。

怎么回事，还记得「base class subobject 在 derived class 中的原样性」吗？让我们踏遍这一继承架构的内存布局，看看到底发生了什么事。

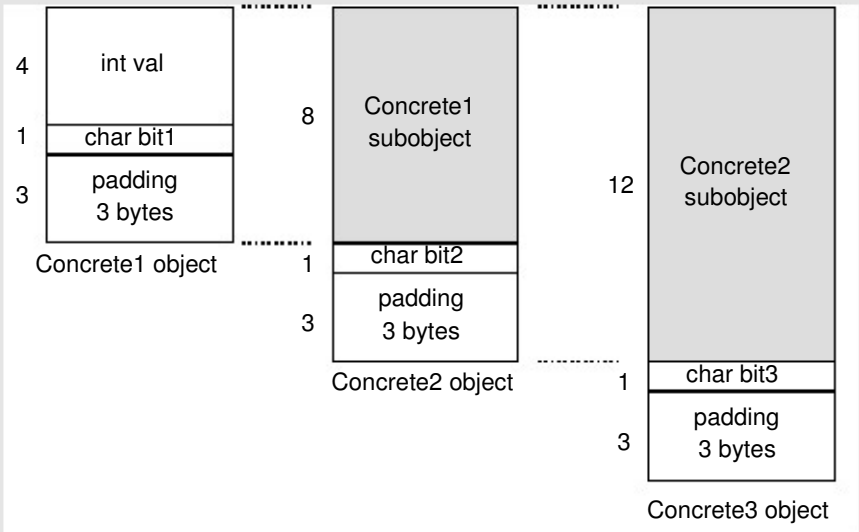
Concrete1 内含两个 members：val 和 bit1，加起来是 5 bytes。而一个 Concrete1 object 实际用掉 8 bytes，包括填补用的 3 bytes，以使 object 能够符合一部机器的 word 边界。不论是 C 或 C++ 都是这样。一般而言，边界调整（alignment）是由处理器（processor）来决定。

到目前为止没什么需要抱怨。但这种典型的布局会导至轻率的程序员犯下错误。Concrete2 加了唯一一个 nonstatic data member bit2，数据型态为 char。轻率的程式员以为它会和 Concreate1 包捆在一起，占用原本用来填补空间的 1 bytes；于是 Concreate2 object 的大小为 8 bytes，其中 2 bytes 用于填补空间。

然而 Concrete2 的 bit2 实际上却是被放在填补空间所用的 3 bytes 之后。于是其大小变成 12 bytes，不是 8 bytes。其中有 6 bytes 浪费在填补空间上。相同道理使得 Concrete3 object 的大小是 16 bytes，其中 9 bytes 用于填补空间。

『真是愚蠢』，我们那位纯真小甜甜这么说。许多读者以电子邮件、电话、或是嘴巴对我也这么说。你可了解为什么这个语言有这样的行为？

译注：下图可说明 Concreate1、Concreate2、Concreate3 的物件布局：



让我们宣告以下一组指标：

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

其中 pc1_1 和 pc1_2 两者都可以指向前述三种 classes objects。下面这个指定动作：

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

应该执行一个预设的 "memberwise" 复制动作（复制一个个的 members），对象是被指之 object 的 Concrete1 那一部份。如果 pc1_1 实际指向一个 Concrete2 object 或 Concrete3 object，则上述动作应该将复制内容指定给其 Concrete1 subobject。

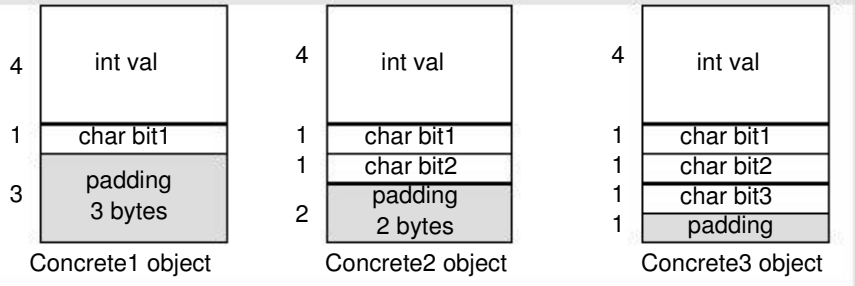
然而，如果 C++ 语言把 derived class members （也就是 Concrete2::bit2 或

Concrete3::bit3）和 Concrete1 subobject 包围在一起，去除填补空间，上述那些语意就无法保留了，那么下面的指定动作：

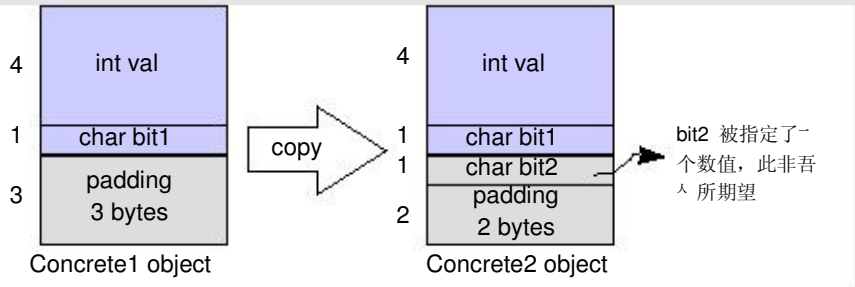
```
pc1_1 = pc2;    // 译注：令 pc1_1 指向 Concrete2 物件

// 喔欧：derived class subobject 被覆写掉，
// 于是其 bit2 member 现在有了一个并非预期的数值
*pc1_2 = *pc1_1;
```

就会将「被包围在一起、继承而得的」members 内容覆写掉。程序员必须花费极大的心力才能找出这个臭虫！

译注：让我以图形解释。如果「base class subobject 在 derived class 中的原样性」受到破坏，也就是说 编译器把 base class object 原本的填补空间让出来给 derived class members 使用，像这样：



那么当发生 Concrete1 subobject 的复制动作时，就会破坏 Concrete2 members。



加上多型（Adding Polymorphism）

如果我要处理一个坐标点，而不打算在乎它是一个 Point2d 或 Point3d 实体，那么我需要在继承关系中提供一个 virtual function 接口。让我们看看如果这么做，情况会有什么改变：

```
// 译注：以下的 Point2d 宣告请与 #101 页的宣告做比较
class Point2d {
public:
    Point2d( float x = 0.0, float y = 0.0 )
        : _x( x ), _y( y ) { };

    // x 和 y 的存取函式与前一版相同。
    // 由于对不同维度的点，这些函式动作固定不变，所以不必设计为 virtual

    // 加上 z 的保留空间（目前什么也没做）
    virtual float z() { return 0.0; }       // 译注：2d 点的 z 为 0.0 是合理的
    virtual void z( float ) { }

    // 设定以下的运算子为 virtual
    virtual void
    operator+=( const Point2d& rhs ) {
        _x += rhs.x();
        _y += rhs.y();
    }
    // ... more members

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

只有当我们企图以多型的方式（polymorphically）处理 2d 或 3d 坐标点，在设计之中导入一个 virtual 界面才显合理。也就是说，写下这样的码：

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

其中 p1 和 p2 可能是 2d 也可能是 3d 坐标点。这并不是先前任何设计所能支

援的。这样的弹性，当然正是对象导向程序设计的中心。支持这样的弹性，势必对我们的 Point2d class 带来空间和存取时间的额外负担：

- 导入一个和 Point2d 有关的 virtual table，用来存放它所宣告的每一个 virtual functions 的位址。这个 table 的元素个数一般而言是被宣告之 virtual functions 的个数，再加上一个或两个 slots（用以支援 runtime type identification）。
- 在每一个 class object 中导入一个 vptr，提供执行时期的联结，使一个 object 能够找到相应的 virtual table。
- 加强 constructor，使它能够为 vptr 设定初值，让它指向 class 所对应的 virtual table。这可能意味在 derived class 和每一个 base class 的 constructor 中，重新设定 vptr 的值。其情况视编译器的最佳化的积极性而定。第 5 章对此有比较详细的讨论。
- 加强 destructor，使它能够抹消「指向 class 之相关 virtual table」的 vptr。要知道，vptr 很可能已经在 derived class destructor 中被设定为 derived class 的 virtual table 地址。记住，destructor 的呼叫次序是反向的：从 derived class 到 base class。一个积极的最佳化编译器可以压抑大量的那些指定动作。

这些额外负担带来的冲击程度视「被处理的 Point2d objects 的个数和生命期」而定，也视「对这些 objects 做多型程序设计所得的利益」而定。如果一个应用程式知道它所能使用的 point objects 只限于二维坐标点或三维坐标点，这种设计所带来的额外负担可能变得令人无法接受[1]。

以下是新的 Point3d 宣告：

```
// 译注：以下的 Point3d 宣告请与 #101 页的宣告做比较
class Point3d : public Point2d {
public:
```

1 我不知道是否有哪个产品系统真正使用了一个多型的 Point 类别体系。

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

译注：上述新的（与 p.101 比较）Point2d 和 Point3d 宣告，最大一个好处是，你可以把 operator+= 运用在一个 Point3d 对象和一个 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 新值将是 (5.2, 5.4, 3.3);

虽然 class 的宣告语法没有改变，但每一件事情都不一样了：两个 z() member functions 以及 operator+=() 运算子都成了虚拟函式；每一个 Point3d class object 内含一个额外的 vptr member（继承自 Point2d）；多了一个 Point3d virtual table；此外每一个 virtual member function 的唤起也比以前复杂了（第 4 章对此有详细说明）。

目前在 C++ 编译器那个领域里有一个主要的讨论题目：把 vptr 放置在 class object 的哪里会最好？在 cfront 编译器中，它被放在 class object 的尾端，用以支持下面的继承类型，如图 **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;
```

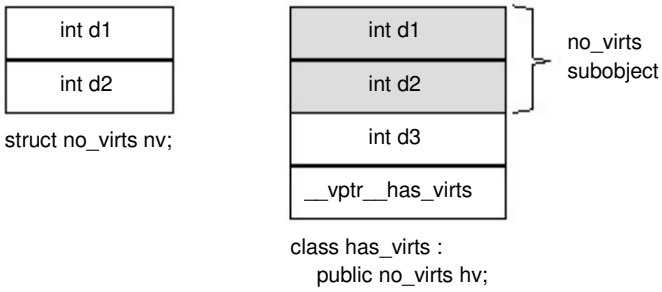


图 3.2a　　Vptr 被放在 class 的尾端

把 vptr 放在 class object 的尾端，可以保留 base class C struct 的对象布局，因而允许在 C 程序代码中也能使用。这种作法在 C++ 最初问世时，被许多人采用。

到了 C++ 2.0，开始支持虚拟继承以及抽象基础类别，并且由于对象导向典范（OO paradigm）的出头，某些编译器开始把 vptr 放到 class object 的起头处（例如 Martin O'Riordan，他领导 Microsoft 的第一个 C++ 编译器产品，就十分主张这种作法）。请看图 **3.2b** 的图解说明。

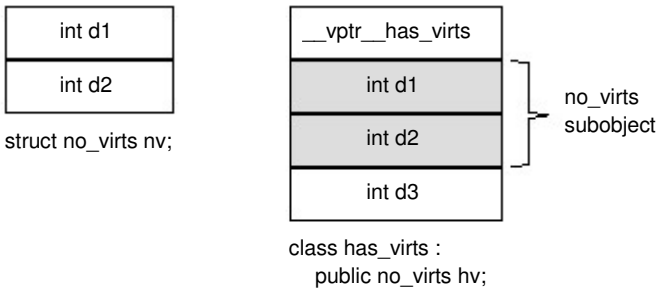


图 3.2b　Vptr 被放在 class 的前端

把 vptr 放在 class object 的前端，对于「在多重继承之下，透过指向 class members 的指标，唤起 virtual function」，会带来一些帮助（请参考 4.4 节）。否则，不仅「从 class object 起始点开始量起」 offset 必须在执行时期备妥，甚至与 class vptr 之间的 offset 也必须备妥。当然，vptr 放在前端，代价就是丧失了 C 语言兼容性。这种丧失有多少意义？有多少程序会从一个 C struct 衍生出一个具多型性质的 class 呢？目前我手上并没有什么统计数据可以告诉我这一点。

图 **3.3** 显示 Point2d 和 Point3d 加上了 virtual function 之后的继承布局。注意此图是把 vptr 放在 base class 的尾端。

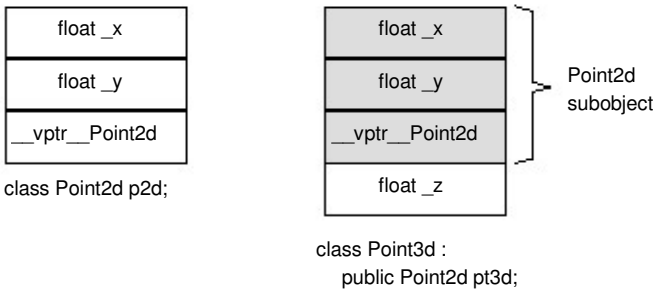


图 3.3　单一继承并含虚拟函式之情况下的数据布局

多重继承（Multiple Inheritance）

单一继承提供了一种「自然多型（natural polymorphism）」形式，是关于 classes 体系中的 base type 和 derived type 之间的转换 请看图 **3.1b** 图 **3.2a** 或图 **3.3**，你会看到 base class 和 derived class 的 objects 都是从相同的地址开始，其间差异只在于 derived object 比较大，用以多容纳它自己的 nonstatic data members。下面这样的指定动作：

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

把一个 derived class object 指定给 base class（不管继承深度有多深）的指标或 reference。这动作并不需要编译器去调停或修改地址。它很自然地可以发生，而且提供了最佳执行效率。

图 **3.2b** 把 vptr 放在 class object 的起始处 如果 base class 没有 virtual function 而 derived class 有（译注：正如图 **3-2b**），那么单一继承的自然多型（**natural** polymorphism） 就会被打破。这种情况下，把一个 derived object 转换为其 base 型态，就需要编译器的介入，用以调整地址（因 vptr 插入之故）。在既是多重继承又是虚拟继承的情况下，编译器的介入更有必要。

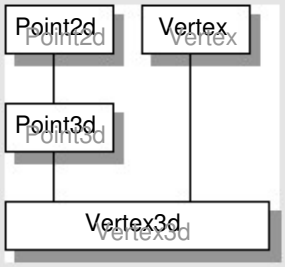
多重继承既不像单一继承，也不容易模塑出其模型。多重继承的复杂度在于 derived class 和其上一个 base class 乃至于上上一个 base class...之间的「非自然」关系。例如，考虑下面这个多重继承所获得的 class Vertex3d：

> 译注：原书的 p92~p94 有很多前后不一致的地方，以及很多「本身虽没有错误却可能误导读者思想」的叙述。程序代码和图片说明也不相符，简直一团乱！我已将之全部更正。如果您拿着原文书对照此中译本看，请不要乍见之下对我产生误会。

```
class Point2d {
public:
```

```
   // ... （译注：拥有 virtual 界面。所以 Point2d 对象之中 会有 vptr）
protected:
   float _x, _y;
};

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

class Vertex {
public:
   // ... （译注：拥有 virtual 界面。所以 Vertex 对象之中 会有 vptr）
protected:
   Vertex *next;
};

class Vertex3d : // 译注：原书误把 Vertex3d 写为 Vertex2d
   public Point3d, public Vertex { // 译注：原书误把 Point3d 写为 Point2d
public:
   // ...
protected:
   float mumble;
};
```

译注：至此，Point2d、Point3d、Vertex、Vertex3d 的继承关系如下：



多重继承的问题主要发生于 derived class objects 和其第二或后继的 base class objects 之间的转换；不论是直接转换如下：

```
extern void mumble( const Vertex& );
Vertex3d v;
...
// 将一个 Vertex3d 转换为一个 Vertex。这是「不自然的」。
mumble( v );
```

或是经由其所支持的 virtual function 机制做转换。因支援「virtual function 之唤起动作」而引发的问题将在 4.2 节讨论。

对一个多重衍生对象，将其地址指定给「最左端（也就是第一个）base class 的指标」，情况将和单一继承时相同，因为二者都指向相同的起始地址。需付出的成本只有地址的指定动作而已（图 **3.4** 显示出多重继承的布局）。至于第二个或后继的 base class 的地址指定动作，则需要将地址修改过：加上（或减去，如果 downcast 的话）介于中间的 base class subobject(s) 大小，例如：

```
Vertex3d v3d;
Vertex *pv;
Point2d *p2d;     // 译注：原书命名为 *pp，不符合命名原则。改为 *p2d 较佳。
Point3d *p3d;     // 译注：Point3d 的定义请看图 3.3 和 #108 页
```

那么下面这个指定动作：

```
pv = &v3d;
```

需要这样的内部转化：

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

而下面的指定动作：

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

都只需要简单地拷贝其地址就好。如果有两个指标如下：

```
Vertex3d *pv3d;    // 译注：原书命名为 *p3d，不符命名通则。改为 *pv3d 较佳。
Vertex   *pv;
```

那么下面的指定动作：

```
pv = pv3d;
```

不能够只是简单地被转换为：

```
// 虚拟 C++ 码
pv = (Vertex*)((char*)pv3d) + sizeof( Point3d );
```

因为如果 pv3d 为 0，pv 将获得 sizeof(Point3d) 的值。这是错误的！所以，对于指标，内部转换动作需要有一个条件测试：

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

至于 reference，则不需要针对可能的 0 值做防卫，因为 reference 不可能参考到「无物」（no object）。

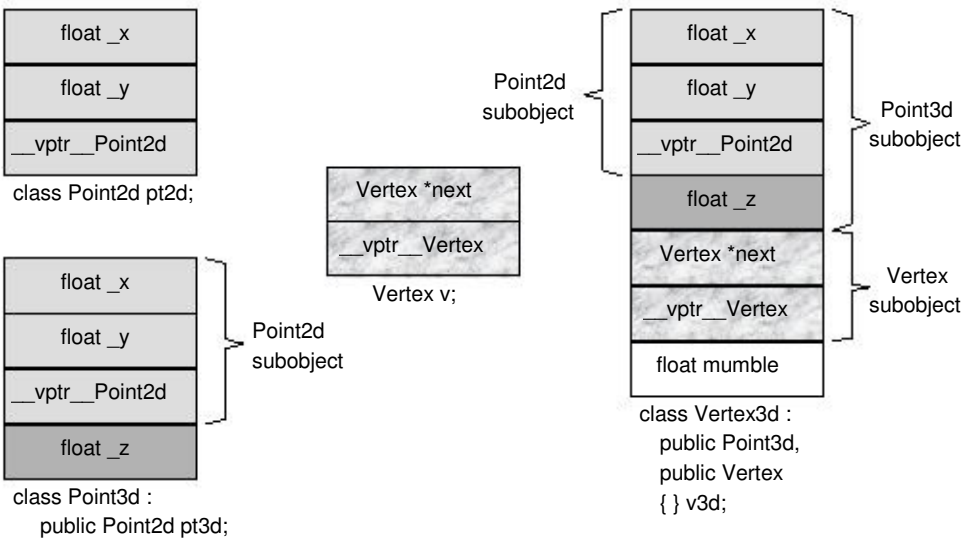


图 3.4 资料布局：多重继承（Multiple Inheritance）

译注：原书的图 **3.4** 只画出 Vertex2d，没有画出 Vertex3d。虽然其中的 Vertex2d 物件布局图「可能」是正确的（我们并没有在书中看到其宣告码），但我相信这其实是 Lippman 的笔误，因为它与书中的许多讨论没有关系。所以我把真正与书中讨论有关的 Vertex3d 的对象布局画于译本的图 **3.4**，如上。

C++ Standard 并未要求 Vertex3d 中的 base classes Point3d 和 Vertex 有特定的排列次序。原始的 cfront 编译器是把它们根据宣告次序来排列。因此 cfront 编译器制作出来的 Vertex3d 对象，将可被视为是一个 Point3d subobject（其中又有一个 Point2d subobject）加上一个 Vertex subobject，最后再加上 Vertex3d 自己的部份。目前各编译器仍然是以此方式完成多重 base classes 的布局（但如果加上虚拟继承，就不一样）。

某些编译器（例如 MetaWare）设计有一种最佳化技术，只要第二个（或后继）base class 宣告了一个 virtual function，而第一个 base class 没有，就把多个 base classes 的次序调换。这样可以在 derived class object 中少产生一个 vptr。这项最佳化技术并未得到全球各厂商的认可，因此并不普及。

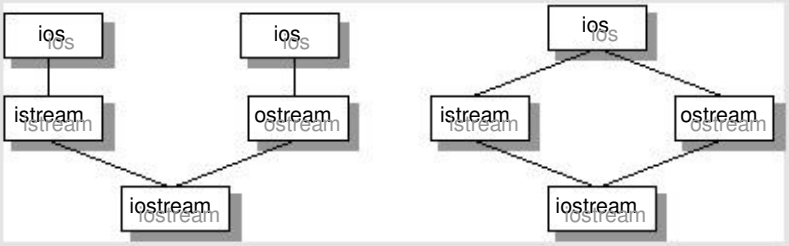
如果要存取第二个（或后继）base class 中的一个 data member，将会是怎样的情况？需要付出额外的成本吗？不，members 的位置在编译时就固定了，因此存取 members 只是一个简单的 offset 运算，就像单一继承一样简单 -- 不管是经由一个指标或是一个 reference 或是一个 object 来存取。

虚拟继承（Virtual Inheritance）

多重继承的一个语意上的副作用就是，它必须支持某种型式的「shared subobject 继承」。典型的一个例子是最早的 iostream library：

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

译注：下图可表现 iostream 的继承体系图。左为多重继承，右为虚拟多重继承。



不论是 istream 或 ostream 都内含一个 ios subobject。然而在 iostream 的对象布局中，我们只需要单一一份 ios subobject 就好。语言层面的解决办法是导入所谓的虚拟继承：

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

一如其语意所呈现的复杂度，要在编译器中支持虚拟继承，实在是困难度颇高。在上述 iostream 例子中，实作技术的挑战在于，要找到一个足够有效的方法，将 istream 和 ostream 各自维护的一个 ios subobject，折迭成为一个由 iostream 维护的单一 ios subobject，并且还可以保存 base class 和 derived class 的指标（以及 references）之间的多型指定动作（polymorphism assignments）。

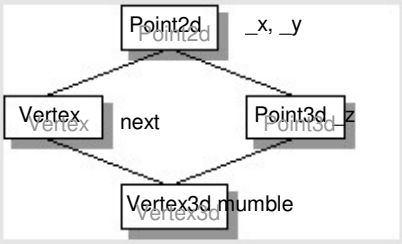
一般的实作法如下所述。Class 如果内含一个或多个 virtual base class subobjects，像 istream 那样，将被分割为两部份：一个不变区域和一个共享区域。不变区域

中的数据，不管后继如何衍化，总是拥有固定的 offset 从 object 的起头算起），所以这一部份数据可以被直接存取。至于共享区域，所表现的就是 virtual base class subobject。这一部份的数据，其位置会因为每次衍生动作而有变化，所以它们只可以被间接存取。各家编译器实作技术之间的差异就在于间接存取的方法不同。以下说明三种主流策略。下面是 Vertex3d 虚拟继承的阶层架构[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
   // 译注：原书上一行的两个 classes 次序相反。为与图 3.5ab 配合，故改之。
{
public:
   ...
protected:
   float mumble;
};
```

2 这个阶层架构是 [POKOR94] 所倡议的，那是一本很好的 3D Graphics 教科书，使用 C++ 语言。

译注：下图可表现 Point2d、Point3d、Vertex、Vertex3d 的继承体系：



一般的布局策略是先安排好 derived class 的不变部份，然后再建立其共享部份。

然而，这中间存在着一个问题：如何能够存取 class 的共享部份呢？cfront 编译器会在每一个 derived class object 中安插一些指标，每个指标指向一个 virtual base class。要存取继承得来的 virtual base class members，可以藉由相关指标间接完成。举个例，如果我们有以下的 Point3d 运算子：

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

在 cfront 策略之下，这个运算子会被内部转换为：

```
// 虚拟 C++ 码
__vbcPoint2d->_x += rhs.__vbcPoint2d->_x; // 译注：vbc 意为：
__vbcPoint2d->_y += rhs.__vbcPoint2d->_y; //          virtual base class
_z += rhs._z;
```

而一个 derived class 和一个 base class 的实体之间的转换，像这样：

```
Point2d *p2d = pv3d; // 译注：原书为 Vertex *pv = pv3d; 恐为笔误
```

在 cfront 实作模型之下，会变成：

```
// 虚拟 C++ 码
Point2d *p2d = pv3d ? pv3d->__vbcPoint2d : 0;
// 译注：原书为 Vertex *pv = pv3d ? pv3d->__vbcPoint2d : 0;
//       恐为笔误（感谢黄俊达先生与刘东岳先生来信指导）
```

这样的实作模型有两个主要的缺点：

1. 每一个对象必须针对其每一个 virtual base class 背负一个额外的指标。
   然而理想上我们却希望 class object 有固定的负担，不因为其 virtual base classes 的个数而有所变化。想想看这该如何解决？
2. 由于虚拟继承串链的加长，导至间接存取层次的增加。这意思是，如果我有三层虚拟衍化，我就需要三次间接存取（经由三个 virtual base class 指标）。然而理想上我们却希望有固定的存取时间，不因为虚拟衍化的深度而改变。

MetaWare 和其它编译器到今天仍然使用 cfront 的原始实作模型来解决第二个问题，它们经由拷贝动作取得所有的 nested virtual base class 指标，放到 derived class object 之中。这就解决了「固定存取时间」的问题，虽然付出了一些空间上的代价。MetaWare 提供一个编译时期的选项，允许程序员选择是否要产生双重指标。图 **3.5a** 说明这种「以指标指向 base class」的实作模型。

至于第一个问题，一般而言有两个解决方法。Microsoft 编译器引入所谓的 virtual base class table。每一个 class object 如果有一个或多个 virtual base classes，就会由编译器安插一个指标，指向 virtual base class table。至于真正的 virtual base class 指针，当然是被放在该表格中。虽然此法已行之有年，但我并不知道是否有其它任何编译器使用此法。说不定 Microsoft 对此法提出专利，以至别人不能使用它。

第二个解决方法，同时也是 Bjarne 比较喜欢的方法（至少当我还和他共事于 Foundation 专案时），是在 virtual function table 中放置 virtual base class 的 offset（而不是地址）。图 **3.5b** 显示这种 base class offset 实作模型。我在 Foundation

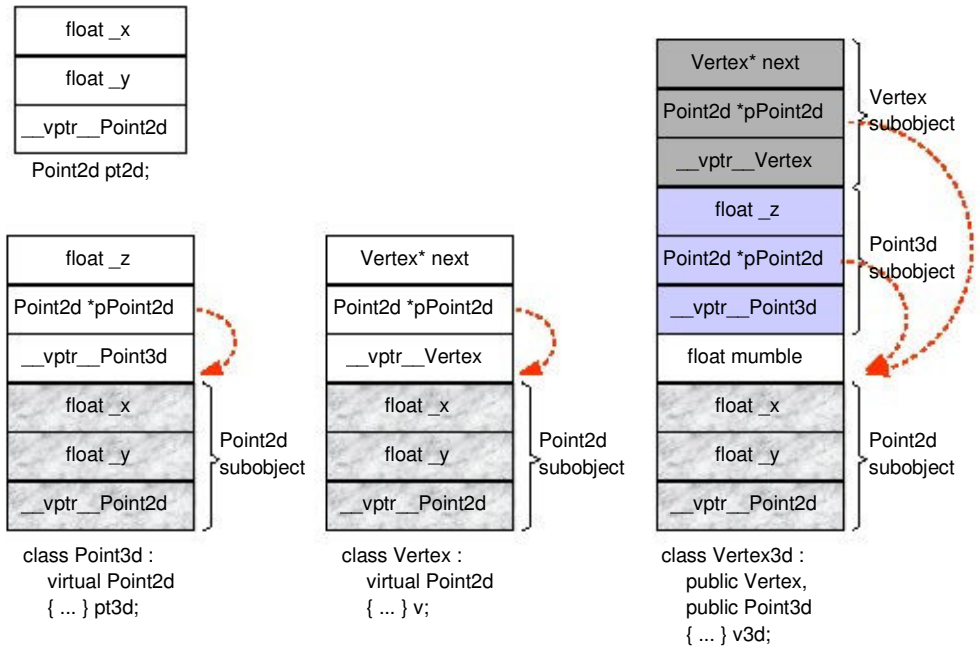


图 3.5a 虚拟继承，使用 Pointer Strategy 所产生的资料布局（译注：原书图 3.5a 把 Vertex 写为 Vertex2d，与书中程式码不符，所以我全部改为 Vertex）

专案中实作出此法，将 virtual base class offset 和 virtual function entries 混杂在一起。在新近的 Sun 编译器中，virtual function table 可经由正值或负值来索引。如果是正值，很显然就是索引到 virtual functions；如果是负值，则是索引到 virtual base class offsets。在这样的策略之下，Point3d 的 operator+= 运算子必须被转换为以下形式（为了可读性，我没有做型别转换，同时我也没有先执行对效率有帮助的地址预先计算动作）：

```
// 虚拟 C++ 码
(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;
```

虽然在此策略之下，对于继承而来的 members 做存取动作，成本会比较昂贵，不过此成本已经被分散至「对 member 的使用」上，属于区域性成本。Derived class 实体和 base class 实体之间的转换动作，例如：

```
Point2d *p2d = pv3d; // 译注：原书为 Vertex *pv = pv3d; 恐为笔误
```

在上述实作模型下将变成：

```
// 虚拟 C++ 码
Point2d *p2d = pv3d ? pv3d + pv3d->__vptr__Point3d[-1] : 0;
// 译注：上一行原书为：
//    Vertex *pv = pv3d ? pv3d + pv3d->__vptr__Point3d[-1] : 0;
//    恐为笔误（感谢黄俊达先生与刘东岳先生来信指导）
```

上述每一种方法都是一种实作模型，而不是一种标准。每一种模型都是用来解决「存取 shared subobject 内的数据（其位置会因每次衍生动作而有变化）」所引发的问题。由于对 virtual base class 的支持带来额外的负担以及高度的复杂性，每一种实作模型多少有点不同，而且我想还会随着时间而进化。

经由一个非多型的 class object 来存取一个继承而来的 virtual base class 的 member，像这样：

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

可以被最佳化为一个直接存取动作，就好像一个经由对象唤起的 virtual function 呼叫动作，可以在编译时期被决议（resolved）完成一样。在这次存取以及下一次存取之间，对象的型别不可以改变，所以「virtual base class subobjects 的位置会变化」的问题在此情况下就不再存在了。

一般而言，virtual base class 最有效的一种运用形式就是：一个抽象的 virtual base class，没有任何 data members。

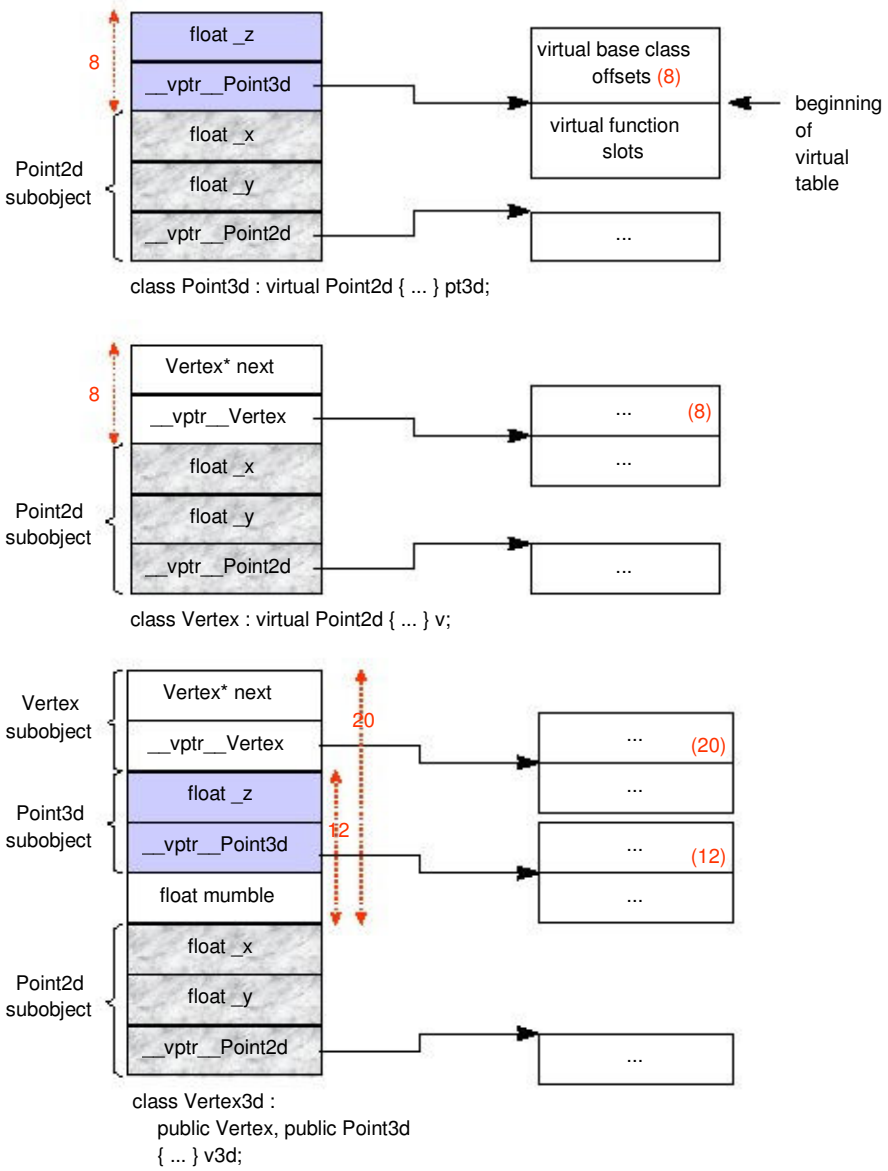


图 3.5b　虚拟继承，使用 Virtual Table Offset Strategy 所产生的资料布局（译注：原书图 3.5b 把 Vertex 写为 Vertex2d，与书中程式码不符，所以我全部改为 Vertex）

3.5 对象成员的效率（Object Member Efficiency）

下面数个测试，旨在量测聚合（aggregation）、封装（encapsulation）、以及继承（inheritance）所引发的额外负荷的程度。所有量测都是以个别区域变量的加法、减法、指派（assign）等动作的存取成本为依据：下面就是个别的区域变量：

```
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;
```

每个表达式需执行一千万次，如下所示　（当然啦，一旦坐标点的表现方式有变化，运算语法也就得随之变化）：

```
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;
}
```

我们首先针对三个 float 元素所组成的区域数组进行测试：

```
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 ];
}
```

第二个测试是把同质的数组元素转换为一个 C struct 数据抽象型别，其中的成员皆为 float，成员名称是 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;
}
```

更深一层的抽象化，是做出数据封装，并使用 inline 函式。坐标点现在以一个独立的 Point3d class 来表示。我尝试两种不同型式的存取函式，第一，我定义一个 inline 函式，传回一个 reference，允许它出现在 assignment 运算子的两端：

```
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;
};
```

那么真正对每一个坐标元素的存取动作应该是像这样：

```
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();
}
```

我所定义的第二种存取函式型式是，提供一对 get/set 函式：

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

于是对每一个坐标值的存取动作应该像这样：

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

表格 **3.1** 列出两种编译器对于上述各种测试的结果。只有当两个编译器的效率有明显差异时，我才会把两者分别列出。

表 格 3.1　不断加强抽象化程度之后，数据的存取效率

	最佳化	未最佳化
个别的区域变量	0.80	1.42
区域数组		
CC	0.80	2.55
NCC	0.80	1.42
struct 之中有 public 成员	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

这里所显示的重点在于，如果把最佳化开关打开，「封装」就不会带来执行时期的效率成本。使用 inline 存取函式亦然。

我很奇怪为什么在 CC 之下存取数组，几乎比 NCC 慢两倍，尤其是数组存取所牵扯的只是 C 数组，并没有用到任何复杂的 C++ 特性。一位程序代码产生（code generation）专家将这种反常现象解释为「一种奇行怪癖...与特定的编译器有关」。或许是真的，但它发生在我正用来开发软件的编译器身上耶！我决定挖掘其中秘密。叫我「爱挑毛病的乔治」吧，如果你喜欢的话！如果你对此一题目不感兴趣，请直接跳往下一个主题。

在下面的 assembly 语言输出片段中，l.s 表示载入（load）一个单精度浮点数，s.s. 表示储存（store）一个单精度浮点数，sub.s 表示将两个单精度浮点数相减。下面是两种编译器的 assembly 语言输出结果，它们都加载两个值，将某一个减去另一个，然后储存其结果。在效率较差的 CC 编译器中，每一个区域变量的位址都被计算并放进一个缓存器之中（addu 表示无正负号的加法）：

```
// 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 编译器的 assembly 输出中，加载（load）步骤直接计算地址：

```
// 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)
```

如果区域变量被存取多次，CC 策略或许比较有效率。然而对于单一存取动作，把变量地址放到一个缓存器中很明显地增加了表达式的成本。不论哪一种编译器，只要把最佳化开关打开，两段码都会变得相同，在其中，回路内的所有运算都会以缓存器内的数值来执行。

让我下一个结论 如果没有把最佳化开关打开，　　就很难猜测一个程序的效率表现，因为程序代码潜在性地受到专家所谓的「一种奇行怪癖...与特定编译器有关」的魔咒影响。在你开始「原始码层面的最佳化动作」以加速程序的运作之前，你应该先确实地量测效率，而不是靠着推论与常识判断。

在下一个测试中，我首先要介绍 Point 抽象化的一个三层单一继承表达法，然后再介绍 Point 抽象化的一个虚拟继承表达法。我要测试直接存取和 inline 存取（多重继承并不适用于此一模型，所以我决定放弃它）。三层单一继承表达法如下：

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

「单层虚拟继承」是从 Point1d 中 虚拟衍生出 Point2d；「双层虚拟继承」则又从 Point2d 中 虚拟衍生出 Point3d。表格 **3.2** 列出两种编译器的测试结果。同样地，只有当两种编译器的效率有明显不同时，我才会把两者分别列出。

表 格 3.2　在继承模型之下的数据存取

	最佳化	未最佳化
单一 继承		
直接存取	0.80	1.42
使用 inline 函式		
CC	0.80	2.55
NCC	0.80	3.10
虚拟继承（单层）		
直接存取	1.60	1.94
使用 inline 函式		
CC	1.60	2.75
NCC	1.60	3.30
虚拟继承（双层）		
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

单一 继承应该不会影响测试的效率，因为 members 被连续储存于 derived class object 中，并且其 offset 在编译时期就已知了。测试结果一 如预期，和表格 **3.1** 中 的抽象数据型别结果相同。这结果在多重继承的情况下 应该也是相同的，但我不能确定。

再一 次，值得注意的是，如果把最佳化关闭，以常识来判断，我们说效率应该相同（对于「直接存取」和「inline 存取」两种作法）。然而实际上 却是 inline 存取比较慢。我们再次得到教训：程序员如果关心其程序效率，应该实际量测，不要光凭推论或常识判断或假设。另一 个需要注意的是，最佳化动作并不一 定总是

能够有效运作，我不只一次以最佳化方式来编译一个已通过编译的正常程序，却以失败收场。

虚拟继承的效率令人失望！两种编译器都没能够辨识出对「继承而来的 data member pt1d::_x」的存取系透过一个非多型对象（因而不需要执行时期的间接存取）。两个编译器都会对 pt1d::_x（及双层虚拟继承中的 pt2d::_y）产生间接存取动作，虽然其在 Point3d 对象中的位置早在编译时期就固定了。「间接性」压抑了「把所有运算都移往缓存器执行」的最佳化能力。但是间接性并不会严重影响非最佳化程序的执行效率。

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

指向 data members 的指标，是一个有点神秘但颇有用处的语言特性，特别是如果你需要详细调查 class members 的底层布局的话。这样的调查可用以决定 vptr 是放在 class 的起始处或是尾端。另一个用途，展现于 3.2 节，可用来决定 class 中的 access sections 的次序。一如我曾说过，那是一个神秘但有时候有用的语言特性。

考虑下面的 Point3d 宣告。其中有一个 virtual function，一个 static data member，以及三个坐标值：

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

每一个 Point3d class object 含有三个坐标值，依序为 x, y, z，以及一个 vptr。至于 static data member origin 将被放在 class object 之外。唯一可能因编译器不同

而不同的是 vptr 的位置。C++ Standard 允许 vptr 被放在对象中的任何位置：在起始处，在尾端，或是在各个 members 之间。然而实际上，所有编译器不是把 vptr 放在对象的头，就是放在对象的尾。

那么，取某个坐标成员的地址，代表什么意思？例如，以下动作所得到的值代表什么：

```
& Point3d::z;     // 译注：原书的 & 3d_point::z; 应为笔误
```

上述动作将得到 z 坐标在 class object 中的偏移位置（offset）。最低限度其值将是 x 和 y 的大小总和，因为 C++ 语言要求同一个 access level 中的 members 的排列次序应该和其宣告次序相同。

然而 vptr 的位置就没有限制。不过容我再说一次，实际上 vptr 不是放在对象的头，就是放在对象的尾。在一部 32 位机器上，每一个 float 是 4 bytes，所以我们应该期望刚才获得的值要不是 8 就是 12 在 32 位机器上一个 vptr 是 4 bytes）。

然而，这样的期望却还少 1 bytes。对于 C 和 C++ 程序员而言，这多少算是个有点年代的错误了。

如果 vptr 放在对象的尾巴，三个坐标值在对象布局中的 offset 分别是 0, 4, 8。
如果 vptr 放在对象的起头，三个坐标值在对象布局中的 offset 分别是 4, 8, 12。
然而你若去取 data members 的地址，传回的值总是多 1，也就是 1, 5, 9 或 5, 9, 13 等等。你知道为什么 Bjarne 决定要这么做吗？

译注：如何取 & Point3d::z 的值并打印出来？以下是示范作法：

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

注意，不可以这么做：

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

否则会得到错误讯息：

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)

我使用的编译器是 Microsoft Visual C++ 5.0。为什么执行结果并不如书中所说增加 1 呢？原因可能是 Visual C++ 做了特殊处理，其道理与本章一开始对于 empty virtual base class 的讨论相近！

问题在于，如何区分一个「没有指向任何 data member」的指标，和一个指向「第一个 data member」的指标？考虑这样的例子：

```
float Point3d::*p1 = 0;
float Point3d::*p2 = &Point3d::x;
// 译注：Point3d::* 的意思是：「指向 Point3d data member」之指标型别。

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

为了区分 p1 和 p2，每一个真正的 member offset 值都被加上 1。因此，不论编译器或使用者都必须记住，在真正使用该值以指出一个 member 之前，请先减掉 1。

认识「指向 data members 的指标」之后，我们发现，要解释：

```
& Point3d::z;      // 译注：原书的 & 3d_point::z; 应为笔误
```

和

```
& origin.z
```

之间的差异，就非常明确了。鉴于「取一个 nonstatic data member 的地址，将会得到它在 class 中的 offset」，取一个「系结于真正 class object 身上的 data member」的地址，将会得到该 member 在内存中的真正地址。把

```
& origin.z
```

所得结果减（译注：原文为加，错误）z 的偏移值（相对于 origin 起始地址），并加 1（译注：原文为减，错误），就会得到 origin 起始地址。上一行的传回值型别应该是：

```
float*
```

而不是

```
float Point3d::*
```

由于上述动作所参考的是一个特定单一实体，所以取一个 static data member 的位址，意义也相同。

在多重继承之下，若要将第二个（或后继）base class 的指标，和一个「与 derived class object 系结」之 member 结合起来，那么将会因为「需要加入 offset 值」而变得相当复杂。例如，假设我们有：

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

void func1( int Derived::*dmp, Derived *pd )
{
   // 期望第一个参数得到的是个「指向 derived class 之 member」的指标。
   // 如果传进来的却是一个「指向 base class 之 member」的指标，会怎样？
   pd->*dmp;
}

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

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

当 bmp 被做为 func1() 的第一个参数，它的值就必须因介入的 Base1 class 大小而调整，否则 func1() 中 这样的动作：

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

将存取到 Base1::val1，而非程序员所以为的 Base2::val2。要解决这个问题，必须：

```
// 经由编译器内部转换
func1( bmp + sizeof( Base1 ), pd );
```

然而，一般而言，我们不能够保证 bmp 不是 0，因此必须特别护卫之：

```
// 内部转换
// 防范 bmp == 0
func1( bmp ? bmp + sizeof( Base1 ) : 0, pd );
```

译注：我实际写了一个小程序，打印上述各个 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)
```

经过 Visual C++ 5.0 编译后，执行结果竟然都是 0。(1)(2)(3)都是 0 是可以理解的（为什么不是 1？可能是因为 Visual C++ 有特殊处理；稍早 p.131 中 我的另一个译注曾有说明）。但为什么 (4) 也是 0，而不是 4？是否编译器已经内部处理过了呢？很可能（我只能如此猜测）。

如果我把 Derived 的宣告改为：

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

那么：

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

将得到 8，表示 vald 的前面的确有 val1 和 val2。

「指向 Members 的指标」的效率问题

下面的测试企图获得一些量测数据，让我们了解，在 3D 坐标点的各种 class 表现法之下，使用「指向 members 的指标」所带来的影响。一开始的两个案例并没有继承关系，第一个案例是要取得一个「已系结之 member」的地址：

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

然后施以指派（assignment）、加法、减法动作如下：

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

第二个案例则是针对三个 members，取得「指向 data member 之指针」的地址：

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

而指派（assignment）、加法和减法等动作，都是使用「指向 data member 之指标」语法，把数值系结到对象 pA 和 pB 中：

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

回忆 3.5 节中的直接存取动作，平均时间是 0.8 秒（当最佳化开启）或 1.42 秒（当最佳化关闭）。现在再执行这两个测试，结果列于表格 **3.3** 中。

表 格 3.3　存 取 Nonstatic Data Member

	最佳化	未最佳化
直接存取（请参考 3.5 节）	0.80	1.42
指标指向已系结之 Member	0.80	3.04
指标指向 Data Member		
CC	0.80	5.34
NCC	4.04	5.34

未最佳化的结果正如预期。也就是说，为每一个「member 存取动作」加上一层间接性（经由已系结之指标），会使执行时间多出一倍不止。以「指向 member 的指针」来存取数据，再一次几乎用掉了双倍时间。要把「指向 member 的指标」系结到 class object 身上，需要额外地把 offset 减 1。更重要的是，当然，最佳化可以使所有三种存取策略的效率变得一致，唯 NCC 编译器除外。你不妨注意一下，在这里，NCC 编译器所产生的码在最佳化情况下有着令人震惊的可怜效率，这反映出它所产生出来的 assembly 码有着可怜的最佳化动作，这和 C++ 原始码如何表现并无直接关系 -- 要知道，我曾检验过 CC 和 NCC 产生出来的未最佳化 assembly 码，两者完全一样！

下一组测试要看看「继承」对于「指向 data member 的指标」所带来的效率冲击。在第一个案例中，独立的 Point class 被重新设计为一个三层单一继承体系，每一个 class 有一个 member：

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

第二个案例仍然是三层单一继承体系，但导入一层虚拟继承：Point2d 虚拟衍生自 Point。结果，每次对于 Point::x 的存取，将是对一个 virtual base class data member 的存取。最后一个案例，实用性很低，几乎纯粹是好奇心的驱使：我加上第二层虚拟继承，使 Point3d 虚拟衍生自 Point2d。表格 **3.4** 显示测试结果。

注意：由于 NCC 最佳化的效率在各项测试中都是一致的，我已经把它从表格中剔除了。

表格 3.4 「指向 Data Member 的指标」存取方式

	最佳化	未最佳化
没有继承	0.80	5.34
单一继承（三层）	0.80	5.34
虚拟继承（单层）	1.60	5.44
虚拟继承（双层）	2.14	5.51

由于被继承的 data members 是直接存放在 class object 之中，所以继承的引入一点也不会影响这些码的效率。虚拟继承所带来的主要冲击是，它妨碍了最佳化的有效性。为什么？在两个编译器中，每一层虚拟继承都导入一个额外层次的间接性。在两个编译器中，每次存取 Point::x，像这样：

```
pB.*bx
```

会被转换为：

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

而不是转换最直接的：

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

额外的间接性会降低「把所有的处理都搬移到缓存器中执行」的最佳化能力。