1

|  |
| --- |
| 無限延伸你的視野  **物件導向的精髓**  多型與虛擬  **Polymorphism in C++**  侯俊傑 著  **松崗電腦圖書資料股份有限公司** |

多型與虛擬（*Polymorphism in C++*）

2

多型與虛擬（*Polymorphism in C++*）

讀者來函

1

读者来函

* BBFISH

《多型与虚拟》一书中的多型与物件导向概念，在我的物件导向学习路上点燃了一盏明灯。我本非科班出身，因兴趣使然，要投考资管所。在辛苦的 C 语言路上，前半段走的甚为艰辛，后来因为要准备考试，再度学起 C 语言。虽说考的只是观念罢了，却也让我对 C 语言燃起兴趣。后来，在学长的介绍下看起您的书评，得知很多好书。物件导向的观念，更有赖您的新书与《世纪末软件革命２》一书，让我得以很快进入状况。虽说您的新书很多地方我还看不懂，不过我却害怕缺货，马上就买了一本。

* ygwei

最近将您的那一本《多型与虚拟》看了一下，发现真的对我有很大的帮助。帮我厘清了很多 C++ 的问题。compiler 课程或系统程序课程，虽然有稍微提到有关方面的东西， 但都很少，而且也不是以我熟悉的 C++ 来解释，因此不是很懂。有很多东西是看了您的书之后才豁然开朗的。

* Kevin Lo

我是你的读者，住在台北。我是一个 C++ programming 新手，我刚读完你的《多型与虚拟》第一章，非常喜欢它。我要谢谢你，带领我经历 C++ 的奇历。顺带一提，我买了你截至目前所出版的所有书籍，有了它们，我有信心在短期的未来内，在 C++ programming 以及 Windows programming 中获得很大的进步。非常谢谢你，并祝你未来更成功。

多型与虚拟（*Polymorphism in C++*）

2

* David

讀者來函

我知道书本上的 1999 0928 是什么意义，就想在 0928 那天发一封 mail 给您祝福一下。在一片书海中总是会无意识地自动搜寻那熟悉的人头 mark，若有新发现总是欣喜地翻阅看看里面写了些什么。自从我略略有了幼幼班程度的 C++ 经验后，大略可以猜猜您在做的动作。我对您的书籍好有一比，一般医师只是拿起喷枪喷喷鼻子了事，我却意外发现一本精美说明书，从此走岔了路陶醉在您的喷枪彻底分析里头，那可真是比喷喷鼻子有趣多了 -- 只不过自我补强机械背景知识的过程中，让我这学医的人充满了郁郁寡欢的挫折感。看了你 post 的文章我觉得自己好像是披块白布在吸人家的鼻涕而不是着医师白袍在治疗病患。

对于耳鼻喉科刀法熟捻的教授们，我心中充满尊敬，在医学中心跟在他们身边亲炙大师的学术风范。日常应对则对一些前辈不自主地产生畏惧。但是很少有让我又敬又畏的。… 我回忆起在诊所见到你的名字出现在候诊单时的心头一震…他就是写出一篇篇文章的作者本人吗？他就是让你无数夜晚辗转难眠的那人吗？

有时对自己花太多时间在计算机上也会有些不安。我不是应该多看一些本行的专业书籍吗？或许我可以这么安慰自己，我是利用其他医生打高尔夫、唱卡拉 OK，喝酒的休闲时间来休闲我的计算机。这样一想好像不会那么罪恶感了。

COM 对我是个恶梦，它是 Chronic Otitis Medi（a 慢性中耳炎）。SOM：Serous Otitis Media

（浆液性中耳炎）是另一个梦魇。门诊里很难根治的两个疾病。Polymorphism？人与人的同源基因间都存在一些微常的差异， 遗传学上把这种差异叫做多形性

（Polymorphism）。

…我真希望哪一天我能够写信告诉您看了 *Polymorphism in C++* 时的喜悦，但我还不能够。虽然我们不常见面，当我写医普文章时，总觉得你就在身边。祝您佳作连连。

多型與虛擬（*Polymorphism in C++*）

讀者來函 3

* shincheng

前几天在书局看到侯老师所着的《多型与虚拟》一书，因为一直有个希望有一天我也能成为一个练书法的人，所以我就把书买回家，看了二三十页之后，觉得这本书没有白买， 学了不少以前不知道的观念。谢谢你，也希望你为国内信息界注入更多好书。如果有机会为我们这些新手写些入门书，相信会帮助更多的人。

* p60036

最近个人对 COM/DCOM 非常有兴趣，恰巧您最近出了一系列这方面的基础书籍，由于个人非常喜欢您的著作与译作，所以便全部买下。几天Ｋ下来，获益良多，以往在杂志以及其他书籍所获得的零碎知识总算可以获得归纳与了解。侯先生的著作与译作是国内作品中最棒的…。希望侯先生继续加油，让台湾的计算机出版界水平能够与国外一较长短，甚至更上层楼。

* cmeng

Dear Hou, I was so happy to know that you had written a new book titled *Polymorphism in C++*. I love this book very much... Anyway, I love reading your books.

* skchen

八月底于书局买了本您的著作《多型与虚拟》使我受益良多。希望您能再出些书来造福我们这些读者，因为我觉得学得他人的成功经验比自己摸索来得快又有效率。

* belldandy

我刚从传统程序导向的 programming 转入 OOP，《多型与虚拟》这本书让我获益良多， THANKS!

多型與虛擬（*Polymorphism in C++*）

4

* wking

讀者來函

《多型与虚拟》看得我心旷神怡，舒服得一塌糊涂，从来没有看什么书感觉这么好过（包括您的《深入浅出 MFC》；《多型与虚拟》看完，《深入浅出 MFC》的感觉估计也要出来了）。看您的书有种举重若轻，行云流水的感觉，美妙啊。完全是把艰深的学习变成了甜蜜的享受，把苦口的良药煮出了咖啡的浓香。幸甚至哉，得遇先生。

* Tsuhan Lee

我从前到现在买了您的一些 MFC 书籍，到现在的这本 *Polymorphism in C++*，觉得这本最让我收获良多，因为它真的让我的一些观念更清楚，在开发程序的时候，更知道自己在干什么。而且这种内容的书好像多半是原文的，此书真是国内中文书读者的一大福音。

* hjs

拜读您的大作 *Polymorphism in C++*，收益匪浅，减少了我在 C++ 领域苦修好几年。

* Patrick

I just read the book last Sunday. Thanks for writing such a wonderful book about the C++ polymorphism. I really learned much from it.

* teacher Lo

暑假中读到你的新书《多型与虚拟》及《深度探索 C++ 物件模型》，就一直想写信给你，感谢你的辛劳。《多型与虚拟》真是一本很好的书，读了之后既佩服又感动，已很久没有读到这样「延伸视野」的书了。

你一向建议读者要有命令列模式下的程序开发经验，我非常认同。感激你在书中详细解说如何编译程序、如何使用 debugger。阅读本书时我发现有些程序在命令列模式下编译有问题，例如第五章程式在 VC 5.0 中根本无法编译，但在 VC 4.0 则可顺利通过。反过来有些程序在 VC 5.0 可编译，在 4.0 却不行，相信是版本差异或安装时的设定所致。我认为如果能够在例题中尽量把编译器版本注明清楚，对读者的助益会更大。

多型与虚拟（*Polymorphism in C++*）

读者来函 5

* Patrick

读完《多型与虚拟》一书，有种充实与饱满的感觉，不仅唤起部份原已遗忘的记忆，同时也了解多型为何物，更体会出隐藏其后的运用机制，进而开始欣赏虚拟、多型与物件导向之美。每章以画龙点睛的方式传达意念，读来确实是种享受。这是一本雅俗共赏， 值得一读的好书。我想现在的我可以开始进入书法的境界了。

* ufjudy

我在光华商场和信息展分别购买了您所着的 *Polymorphism in C++* 和所译的 *Inside the C++ Object Model*，心中真是感触万千。依稀记得以前唸书时候的严谨与细致的心境， 而今见到阁下的著作，真是感动又有些许的感慨了!!

* ycchang

开始读《多型与虚拟》是在我得知大一下的课本是用侯老师您所著作的《多型与虚拟》一书于是我才赶紧买一本来研读。第一次看感觉就是不一样，有一定的深度，就好比外国作家写的书一样，很有说服力，很有自己的见解。虽然看到最后也让我有点想放弃的感觉，因为我感觉到 C++ 的背后又有那么多小细节，甚至没办法和多型与虚拟融为一体，常会问 "怎么才能说算学完C++，何时才能用C++ 写庞大的application 呢?"

* "chance"

10 天前买的书，尽快地将其看完，跳过 133~166 页。已经很少有书给我这种感觉，几乎可以说是欢喜赞叹！现在只想再挤出时间来看第二遍，同时犹豫着是不是要也买*Essential COM* 来看。sigh...

* wkc

Hi, dear Mr. Hou: I am an associate professor at TunhHai University. I am about to finish C++ course for our freshmen this semester. And I am searching a adequate book for next semester starting form February.

I was so excited as I found your book entitled *polymorphism in C++*. It is not so huge in

多型与虚拟（*Polymorphism in C++*）

6 读者来函

volume as other books, though most of OO books are not so deep enough, as you pointed. Besides, I agreed with you: the difficulty of learning C++ is not due to those learners, rather than the teachers.

Now I have decided to use your book as my textbook, although I don't like that you often

made use of the description style "I ......" I prefer to the stuff you covered and also the detail level you have made.

* "edward

我买您的第一本书是《多型与虚拟》（其实这不算第一本，我第一本是买 "侯捷" 的《无责任书评》，哈哈）。它所带给我的不只是技术上的，更是使我知道了不少好书，也让我知道是该多看点书了。接着下来就开始留意您的其他著作，呵，《大奥秘》也买了、

《深入浅出MFC》、《COM 本质论》、另外也在图书馆借了《深度探索 C++物件模型》、

《Inside VC++》、还有VxD 那一本。《多型与虚拟》算是我看得最透彻的一本（除了后面模拟 MFC 三大机制）。买了这么多书，却是一本也看不完，但又想想您说的，好书不要错过，就买了放架上了  。我还买了 *Programming Windows 5/e* 也还看不到十分之一，最近买了一本 MFC 入门书，想先会用，再把您的《深入浅出MFC》看透彻。真是每天看哪看，恨不得赶快把这么多书看完，精通十八般武艺。

* David Bai

侯先生你好：我一直视你的书如计算机书籍中的瑰宝。你的书（不论是译作或著作）养坏了我的胃口，使我在买书之前，必细心比较各书的内容及质感，让我觉得有了些许和读你书时相同的满足感后，才肯动荷包。也因为你，我才想报考元智大学信息研究所。百闻不如一见嘛！能上你的课想必是如沐春风。

多型与虚拟（*Polymorphism in C++*）

1

独沽一味

### （一版二刷感言）

松岗公司于《多型与虚拟》一书出版后短短 25 天，通知我即将进行二刷。此期间，我从授课学员的反应和读友的来函中，读到许多令人欣慰的讯息。

C++ 相关书籍，如天上繁星，如过江之鲫。广博如四库全书者有之（如 *The C++ Programming Language*、*C++ Primer*），深奥如重山复水者有之（如 *Inside The C++ Object Model*），独沽一味者有之（如 *C++ Programming Style*、*More Effective C++*）， 独树一帜者有之（如 *The Design and Evolution of C++*），另辟蹊径者亦有之（如 *STL tutorial Reference Guide*）。

我的理想是，以这本薄薄的小书，让即使「只对 C++ 语法有粗浅认识」的读者，仍然能够一步到位「以物件导向的精神来思考程序设计」。我的目标摆在polymorphism（多型）的运用，而终点前的基础作业，包括必要的 C++ 语法语意（syntax & semantics）、必要的 C++ 物件模型（object model）、必要的 C++ 型别转换（type casting），布置在前数章。

多型与虚拟（*Polymorphism in C++*）

2

在技术观念的演绎过程中，用词的精准非常重要。华邦林昆颖先生和徐逸辉先生，给我许多这方面的宝贵意见。此外，宏碁解翼彰先生与另一位工程师（很遗憾未能得知大名），以及网友陈崑泰先生和黄英典先生也予我指正，使我得以将第一刷中的误失更正过来。

拥有这样的读友，愿意花费额外的精力和时间，提供意见并与我讨论，使这本书得更精致，我深以为荣，深以为傲！

文字的修润与精准化，在为数颇多的页次上进行。其中许多页次虽只更动一两个字，却仍必须重新制版，耗费的额外成本甚多。这本书能够在第二刷就进行页数庞大的精致化动作，我要在此感谢松岗公司的全力配合。

侯俊杰 1998.08.19 于新竹

[jjhou@ccca.nctu.edu.tw](mailto:jjhou@ccca.nctu.edu.tw)

多型与虚拟（*Polymorphism in C++*）

3

虑而后能得

### （自序）

##### 故事

接触 C++ 大约是 1989 年的事。那时候的 PC 以现在的眼光看，除了「蛮荒」之外没有更合适的形容词了。横扫千军的 Windows 3.0 还没有诞生，如今以 C++ 编译器技术闻名于世的 Borland 公司那个时候也还没有踏进 C++ 领域， 遑论动作更慢的Microsoft。当时我用的编译器是 Zortech C++。

将近 10 年的历史，并不表示我在 C++ 技术上就怎么地超凡入圣了。事实上初期我很排斥这个语言。我没有用心去体会，也就看不到其中的美好。我用数年的时间把心力全花在 Windows 技术的钻研。那时候 C++ 风潮未开，我也不以为意。

决定好好研究 C++，不是为了要开发自己的 class library，而是为了要使用别人的 class library。92 年软件界开始出现一些所谓的 application framework，是一种用来帮助程序员发展 Windows 应用软件的大型 C++ class library（如今也有非 C++ 语言的类似产品）。为了运用这些有着巨大架构以及巨大能量的 C++ class library，我必须重拾 C++。然后，为了更能操控掌握那些复杂的架构，同时也因为对技术原理有一股强烈的好奇 心，我开始看 Microsoft Foundation Class（MFC）的原始码。

这使我有了一种异于常人的学习经验。我看的是营销全世界、无数程序员使用、市场占

多型与虚拟（*Polymorphism in C++*）

4

有率最高的一套 C++ class library 的原始码。从其中，我得到许多宝贵的知识与经验。然后我慢慢体会了物件导向（object-oriented）之美、多型（polymorphism）之美、虚拟

（virtual）之美。

学习的路线，不一定每个人都相同。但是黑暗中摸索的日子，实在不必人人走一遭。对于位居物件导向精髓地位的「多型与虚拟」观念有了深刻体会之后，我有一股强烈的欲望，要把心得写下来。

于是我写了这本书。

##### 层次

一般而言，C++ 是一个难学易用的语言。

C++ 的难学，初始在于其重重的布幕，布幕之中编译器对我们的程序码做了太多的手脚，使我们惯于循序思考的工程脑袋一无所措。及长又面临新的思维模式，使我们必须扭转惯常的思考习惯。

C++ 的易用则在于其巨大的弹性，能够以多型（polymorphism）、虚拟（virtual）、模板（template）等种种方式，让现有的码去处理未知的、未来的资料型态。

当然，易用必须先能用。用不好或不能用的话，「写 C++ 程序」最后就成了只是「使用 C++ 编译器」，这是大家常拿来彼此调侃的笑话。

在「难学」的背景下，「易用」是使我们依然前仆后继的动力。愈来愈多的大学信息科系把 C++ 开在大一课程，这虽然说明 C++ 是多么地重要，可也苦了信息新兵们。

其实「难学」的最大症结，在于很难得有一本书，能够一针见血地指出多型与虚拟的重要性；在我们粗具语法基础之后，直接把我们导引到最核心最重要的思想，并且在建立这个思想体系的过程中，提供足够的必要基础。

我希望这本书能够做到这一点。

多型与虚拟（*Polymorphism in C++*）

5

同样是使用 C++ 语言，有人进入了物件导向的殿堂，有人只学到 C++ 的语法；

有人练书法； 有人写毛笔字；

有人拿有毛的笔写字。

我们要向书法的境界挑战，不要只是拿有毛的笔写字。

##### 定位

讲述 C++ 语法（syntax）的书籍很多， 讲述 C++ 语意（semantics）的书籍很少，

讲述 C++ 物件导向（object oriented）精神的书籍更少， 深入 C++ 物件模型（object model）的书籍则几乎没有。

这本书第一章对 C++ classes 的语法和语意做了一个梗概整理。第二章谈 C++ 的物件模型，第三章谈型别转型（RTTI 的重要服务），第四章谈多型与虚拟的精神，第五章实作一个大型例子，示范如何在一个 class library 中完成 RTTI、dynamic creation、persistence 等基础建设。第六章为 Component Object Model（COM）奠基。

很容易看得出来，你不能够完全依赖这本书学习 C++ 语法（因为第一章不过才 96

页）。但是在学习了 C++ 语法之后，你可以依赖这本书学习物件导向的精髓。

我自己一直存在一个疑惑。别人常认为我写的是高阶技术书籍，我却认为我写的是高阶技术的入门书籍。也许我挖掘得很深很广，但，不就是要彻底了解重要的基础知识，才得登堂入室吗？

这本书，我仍旧挖得很深，尤其是第二章和第五章。但是我想粗具 C++ 语言基础的人应该都能循序接受。我渴望证明，C++ 的难学，问题不在学生，而在老师。

多型與虛擬（*Polymorphism in C++*）

6

当然，本书的读者，你，不能够只是一位程序设计领域的全然新手。

##### 厚薄

每次完成一本书，抱着厚厚的一叠完稿北上，父亲看到堆起来有 10 公分高的雪铜纸， 总是对我说：『这么厚的书，怎么看』？而我总要解释『这是雪铜纸，比较厚。而且单面印刷，厚度加倍。经过雷射打印机的高温，它又有点卷...』。

但是想想，1000 页左右的书籍，再怎么东折西扣也是厚得吓人。

渐渐地父亲不再问那个问题了，大概已经妥协于「计算机书都那么厚」的印象之中。而我更是早就习惯看大部头的书，写大部头的书。

大家都习惯了吧！这两年。

书籍的厚薄，是因于内容的选择，与读者群的界定。很难得有这么一个题材，我以 340

页的篇幅完成了它。

书籍很薄，又经过我精心的章节次序安排与起承转合，您就不要跳着看了吧。耐着性子， 一页一页，从头到尾把它好好咀嚼消化一遍。然后，如果你愿意，写信给我，我希望听到你的喜悦。

批评也是我所欢迎的。

侯俊杰 1998.07.02 于新竹

[jjhou@ccca.nctu.edu.tw](mailto:jjhou@ccca.nctu.edu.tw)

多型與虛擬（*Polymorphism in C++*）

目錄 1

目 录

虑而后能得（自序） / 003

目 录 / 009

导 读 / 019

第 1 章 Classes 的语法和语意 / 001

开宗明义：Classes 与Objects / 002

Classes 的宣告与实作（Classes Declaration and Implementation） / 005

符号命名习惯（Naming Convention） / 006

Function Overloading（函式多载化） / 007

Operator Overloading（运算子多载化） / 009 Insertion Operator for stdout / 010

Extraction Operator for stdin / 011

Insertion Operator for file / 011

Extraction Operator for file / 012

Name Mangling（符号的特殊处理） / 014

extern "C" / 015

三种封装层级：private、protected、public / 017

Struct 和 Class 的差异 / 019

Objects 的诞生（Objects Instantiation）与死亡 / 020 Objects 的生存范围与生命周期（Scope and Lifetime of Objects） / 021 Pointers 与 References / 023

Pointer / 023

Reference / 025

Constructors 和 Destructors（建构式和解构式） / 028 Default Constructors / 028

多型与虚拟（*Polymorphism in C++*）

2 目 录

Copy Constructors / 032

bitwise copy - 概括承受 / 034

memberwise copy - 选择性承受 / 034

Class Members / 036

Data Members / 036

Member Functions / 037

如何取用 Class Members / 038

特殊的 Static Members（静态成员） / 040

Static Data Members （静态资料成员） / 040

Static Member Functions（静态成员函式） / 042

Static Object Members （静态物件成员） / 044

Static Objects（静态物件） / 045

Inheritance（继承） / 045

单一继承（Single Inheritance） / 048

多重继承（Multiple Inheritance） / 048

虚拟继承（Virtual Inheritance） / 049

继承体系下的物件建构（由内而外，由上而下） / 050

继承体系下的物件解构（由外而内，由下而上） / 051

Initialization List / 052

Virtual Functions（虚拟函式） / 057

Virtual Destructor / 059

Composition（组合） / 061

Inline Function / 062

Inline Function 和Virtual Function 的关系 / 063 Template / 064

Template Functions / 065

Template Classes / 068

Standard Template Library（STL）or C++ Standard Library / 071 STL Containers / 072

STL Generic Algorithm（泛型算法） / 074

STL Iterators（迭代指标） / 076

一个完整实例 / 077

MFC Collection Classes / 081

认识MFC / 081

MFC Collection Classes 应用实例 / 082

多型与虚拟（*Polymorphism in C++*）

|  |  |  |
| --- | --- | --- |
| 目录 |  | 3 |
| RTTI（Runtime Type Identification） | / 086 |  |
| typeid operator | / 086 |  |
| 自己动手做RTTI | / 092 |  |
| Dynamic Creation（动态生成） | / 092 |  |
| 自己动手做 Dynamic Creation | / 093 |  |
| Name Space（命名空间） | / 094 |  |
| 第 2 章 C++ 物件模型（C++ Object Model） | / 097 |  |
| 薄幕重重：伟大的 C++ 编译器 | / 097 |  |
| C++ 物件模型 | / 098 |  |
| 暖身 | / 098 |  |
| this 指标 - data members 与 member functions 的桥梁 | / 100 |  |
| sizeof 的疑惑 | / 102 |  |
| 物件布局大局观：资料与函式 | / 103 |  |
| Data Members 的布局 | / 106 |  |
| vptr 座落何处 | / 109 |  |
| vtbl 的格式 | / 110 |  |
| Member Functions 的布局 | / 111 |  |
| Pointer to Data Members | / 116 |  |
| Pointer to Member Functions | / 117 |  |
| Bjarne Stroustrup 的解法： mptr 结构 | / 119 |  |
| Microsoft 的解法：vcall thunk | / 120 |  |
| 重回主题 | / 121 |  |
| 我的困扰 - 函式指标的转型 | / 123 |  |
| 经过继承的洗礼 | / 125 |  |
| Static Member 的继承 | / 125 |  |
| 单一继承（Single Inheritance）的物件模型 | / 127 |  |
| 多重继承（Multiple Inheritance）的物件模型 | / 128 |  |
| 自然多型（Natural Polymorphism） | / 129 |  |
| 非自然多型（Unnatural Polymorphism） | / 130 |  |
| 多重继承 | / 130 |  |
| 多重继承下 this 指标的调整 | / 135 |  |
| 再谈 sizeof 的迷思 | / 137 |  |
| 空空如也的class | / 138 |  |
| 虚拟继承对 object 大小的影响 | / 138 |  |

多型与虚拟（*Polymorphism in C++*）

4 目 录

base class subobject 在 derived class object 中的原样性 / 140

虚拟继承（Virtual Inheritance） / 142

Pointer Strategy / 142

Virtual Table Offset Strategy / 143

实例观察 / 145

利用除错器观察 object 布局和函式位址 / 150

产生一个project / 150

在 project 中加上 C++ 原始码 / 151

环境设定 / 152

建造程序 / 153

设立断点 / 153

除错 / 153

观察 object 布局 / 154

观察函式位址 / 155

观察位址空间 / 156

在 Visual C++ 中对于函式位址的观察 - 心得整理 / 157

第 3 章 型别转换（Type Casting） / 163

型别转换（Cast） / 163

Up Cast（向上转型） / 168

Down Cast（向下转型） / 169

dynamic\_cast 运算子 / 171

单一继承情况下 / 171

Up Cast / 171

Down Cast / 172

多重继承情况下 / 175

虚拟继承 / 176

横向转型 / 177

static\_cast 运算子 / 179

reinterpret\_cast 运算子 / 183

#（stringizing operator） / 184

##（merging operator or token-pasting operator） / 185

第 4 章 为什么需要多型（Why Polymorphism） / 187

Object Oriented Paradigm / 188

C++ 如何支援 Polymorphism / 189

多型与虚拟（*Polymorphism in C++*）

|  |  |  |
| --- | --- | --- |
| 目录 |  | 5 |
| 从继承开始谈起 | / 190 |  |
| 破题：无有多型，将奈何 | / 194 |  |
| 破题：无有虚拟，又如何 | / 196 |  |
| 指标型别的原罪 | / 199 |  |
| Virtual Functions（虚拟函式） | / 207 |  |
| Virtual Functions（虚拟函式）的额外负荷 | / 210 |  |
| Pure Virtual Function（纯虚拟函式） | / 212 |  |
| Polymorphism 与 Virtual Functions 归纳整理 | / 215 |  |
| Virtual Functions 的迷思与 Object Slicing | / 216 |  |
| 泛型（Generalization）的重要性：实例介绍 | / 219 |  |
| 第 5 章 Polymorphism 的应用 | / 227 |  |
| 目标：三大服务的公共基础建设 | / 229 |  |
| 模块切割 | / 231 |  |
| C++ 编译环境设定 | / 233 |  |
| Makefile 语意介绍 | / 233 |  |
| 本章所有的 classes | / 236 |  |
| 重新检讨 Shape Hierarchy | / 237 |  |
| 环环相扣的三层巨集 | / 239 |  |
| CRuntimeClass -- 构思 | / 240 |  |
| CRuntimeClass -- 实作 | / 243 |  |
| 巧妙运用巨集 | / 244 |  |
| 1. DYNAMIC（Run Time Type Identification）巨集 | / 245 |  |
| 巧妙运用 # 和 ## 运算子 | / 246 |  |
| 巧妙运用 constructor 完成 list 自动串接工作 | / 246 |  |
| 2. DYNCREATE（Dynamic Creation）巨集 | / 249 |  |
| 3. SERIAL（Persistence or Serialization）巨集 | / 251 |  |
| CObject | / 252 |  |
| RTTI 的运用之一 CObject::IsKindOf | / 253 |  |
| RTTI 的运用之二 | / 254 |  |
| CFile | / 255 |  |
| CArchive | / 256 |  |
| CArchive 与 CFile | / 256 |  |
| CArchive 与 Persistence | / 257 |  |
| 重回 CRuntimeClass 的讨论 | / 261 |  |

多型与虚拟（*Polymorphism in C++*）

6 目 录

CDWordArray / 263

CObList / 263

Serialize 函式 / 264

Shape Hierarchy / 264

测试 / 265

值得改善之处 / 268

结语 / 269

程序码列表 / 269

第 6 章 从 C++ Object Model 迈向 Component Object Model / 299

三套 Object Model / 300

C++ Object Model 所面临的问题 / 301

问题现场之一：新旧杂陈的 DLLs / 301

问题现场之二：不定时炸弹 / 302

问题现场之三 / 302

问题现场之四 / 302

所谓动态联结 / 303

程序实例之一 / 304

执行结果 / 307

C++ 缺乏二进制标准 / 308

DLL 和 Client 间的耦合（coupling）关系 / 311

抽取一个界面（interface） / 312

程序实例之二 / 315

执行结果 / 318

执行时期载入 / 319

Classes 的扩展性 / 321

扩展办法之一 / 322

扩展办法之二 / 322

物件生命的管理 / 327

程序总整理，IUnknown 现身 / 331

COM 只是一份规格 / 331

接下去的路 / 332

程序实例之三 / 334

附录A 参考书目（Bibliography） / 339

附录Ｂ C++ 的爱恋与沉迷 / 侯捷无责任书评 / 341

多型与虚拟（*Polymorphism in C++*）

导 读 1

导 读

本书在章节的启承转合上有密切的配合，所以并不建议你做跳跃式阅读。从第一页循序看到最后一页，是我以为比较好的方式。当然如果你对于某些主题十分熟稔，可以快速掠过进入下一主题。

##### 读者基础

你需要一些 C++ 语言基础，以及程序设计领域的某种程度的经验。这种经验很难加以量化，你可试着找出本书各主题中你最感兴趣或最有心得的一部份，试阅 3~5 页。如果读来可喜，有所领悟，就带回家去吧。

##### 章节划分

本书共分六章：

* 第 1 章 Classes 语法和语意

本章所有内容都围绕着 C++ classes 打转。本章对 C++ Classes 的重要性质做了一个梗概描述，算是搭起你（读者）和我（作者）之间的共识。这些 C++ 语法和语意大部份都是后续章节需要用到的。小部份虽然与后续章节无关，却是

* 第 2 章 C++ 物件模型（C++ Object Model）

C++ 物件模型的技术涵盖范围很广，本章只挑选和「多型与虚拟」有关的两大主题：(1)

多型與虛擬（*Polymorphism in C++*）

2 導 讀

物件实体的布局，(2) 虚拟机制的底层架构（vptr 和 vtbl）。这一章可以为你扎下非常好的底层基础，使你在面对一个虚拟函式时，清楚知道它如何被唤起、它的额外负担是什么、以及为什么能够靠它达到多型的终极目标。

* 第三章 型别转换（Type Casting）

一旦引入多型，物件指标就必须考虑 type-safe 的问题。本章介绍 C++ 的三个转型运算子，并把焦点放在和 RTTI 有关的 *dynamic\_cast* 运算子身上。本章也介绍了 C/C++ Standard 的 # 和 ## 运算子（或者你可以称它们为编译器前置处理器），帮助我们利用巨集让编译器为我们「写程序」。# 和 ## 这两个运算子在第五章的程序实例中扮演重要角色。

* 第四章 为什么需要多型（Why Polymorphism）

本章利用一个小例子，解说虚拟函式带来的影响，以及多型（polymorphism）的无远弗届，以及泛型（generalization）的重要性。本书前三章着重于本职学能的建立，这一章则开始进入逻辑层次的思考阶段。

* 第五章 Polymorphism 的应用

这一章以一个大型实例为主轴，一方面验证虚拟和多型的应用，一方面又以程序设计来弥补目前任何 C++ 编译器都未曾提供的两个物件导向重要特质：Dynamic Creation 和Persistence。本章程式码取材自 MFC source，再经我的删减修饰而成，适用于任何作业平台，非常具有参考价值，对于 C++ programming 技术可以带来开阔的眼界。

* 第六章 从 C++ Object Model 迈向 Component Object Model

C++ 有其可到之处，有其不可到之处。为了能够跨编译平台、跨语言平台、跨作业平台、跨网络传输，有必要把 C++ 语言层面的封装更进一步设计为二进制层面的封装。

Component Object Mode（l COM）就是这种思考下所制定的一个规格，而了解 C++ Object

Model 正是学习 Component Object Model 的最短路径。本章以三个实例逐次完成封装工作，使 C++ classes 成为所谓的 components，亦即所谓的「软件 IC」。

多型與虛擬（*Polymorphism in C++*）

導 讀 3

##### 使用工具

理论上，任何一个 C++ 编译器，都可以编译本书的任何一个程序。即使第五章那个脱胎自 Microsoft Foundation Classes（MFC）的大型实例，经过我的修改，一样可通过各家编译器。我的实验平台包括 Visual C++ 和 Borland C++Builder 两套 C++ 编译器。

前三章介绍各种语法或各种低阶实验性动作时，常会连带把测试动作的编译错误讯息列出（如果有的话）。我所列的是 Microsoft Visual C++ 的错误讯息和代码。

目前 PC 上任何一套 C++ 编译器都提供有整合开发环境（IDE）以及命令列（command line）两种模式。本书第 2 章 p.150 和第 5 章 p.233 分别示范在 Visual C++ 之下如何设定并使用这两种模式。

##### 程序码

第 5 章和第 6 章的范例程序，比较大也比较值得提供给读者。我把它放在松岗出版公司的网站上（[http://www.unalis.com.tw](http://www.unalis.com.tw/)），请自行下载。至于分散本书各处的小小测试程序，非常多，非常小，非常细琐，并不打算供应给读者。

##### 用词特色

C++ 以及物件导向领域中的一个特色就是，有着多如牛毛的术语，而且又臭又长。我决定尽量采用原文术语，目的有二：

1. 我清楚知道工程师们的习惯用语，我不想让他们看这本书时，时时得把中文名词倒译回去，或时时得查阅中英名词对照表。
2. 学子们对于原文术语的陌生度已经到了令我骇然的地步。我希望起一些引导作用。

当然，中英名词的取舍与选择，程度上的拿捏是很需要技巧的，而选择是很带个人色彩的。欢迎您给我意见，谢谢。

侯俊杰 1998.07.02 于新竹

[jjhou@ccca.nctu.edu.tw](mailto:jjhou@ccca.nctu.edu.tw)

多型與虛擬（*Polymorphism in C++*）

4 導 讀

多型與虛擬（*Polymorphism in C++*）

第１章 C++ Classes 的語法和語意

**第１章**

# C++ Classes 的

语法和语意

### （C++ Classes' Syntax and Semantics）

这本书谈的全是物件导向的 多型与虚拟 性质，这个主题可以说是 C++ 语言的精髓。这不是一本讲解 C++ 语法的书，不过，为了建立读者与我之间的共识， 同时也引导各位在C++ 高阶语法上有全面性的认识，我以小小的篇幅，将本书用到的C++ 高阶语法，做一个整理。这些语法几乎全部围绕在 class 身上打转。

如果你希望获得这些语法的更详细资料，我推荐两本 C++ 经典好书：

1. ***The C++ Programming Language***（Addison Wesley）：这是C++ 语言创始者

Bjarne Stroustrup 所着的「C++ 原典」。内容极为详尽，目前为第三版（1997）。

1. ***C++ Prime***（***r*** Addison Wesley）：这是全世界第一套 C++ 编译器的实作者 Stanly

B. Lippman 所着的经典书籍。内容极为详尽，而且编排比 Bjarne 的书更胜一筹、

更亲和易读（我个人看法）。目前也是第三版（1998）。

**1**

多型與虛擬（*Polymorphism in C++*）

开宗明义：Classes 与 Objects

本书讲的是 Object-Oriented（物件导向）的最重要精神：多型与虚拟，那么让我们先对 objects 做个了解。牛津计算机字典对于 objects 这个词条的解释是：

object ： 一个涵意松弛的术语， 用以描述软件系统中可资辨别的元件

（components），如今常被用来描述一个具有某种自我圆融（self-contained）意识并有着可识别界线的元件。在 object-oriented design 之中，object 是基本元素， 从中得以架构出整个系统模型（system model）。

在 object-oriented programming 领域中，object 有比较严谨的定义。所谓 object 是一个实体（instance），由资料结构以及用以处理该结构的程序（称为 methods） 所组成。这些 methods 藉着 object 所获得之 messages 的刺激而作用起来。object 的内部资料结构完全与其它 objects 隔离（ 这种性质称为「 封装」， encapsulation）。Objects 系根据一个 template（模板）产生出来。由某个 template 所生出的实体（instances）的集合称为 classes。有一个被称为 inheritance（继承） 的重要性质，允许新的 classes 架构在既有 classes 之上，继承既有 classes 的全部或部份特征， 并允许扩充新的特征。某些系统更实作出所谓的 multiple inheritance（多重继承），允许 class 继承一个以上的 parent classes。1

上述定义委实学术了一些！如果你有看没有懂，其实只要建立这样一个观念就 好：object 就是「东西」。东西必须占用一定的空间，在计算机之中也就是占用一定的存储器，所以我们又称它为一个实体（instance）。

上述定义有相当部份在描述「东西」的起源。它说「东西系根据一个模板产生出

1. 这段定义中的某些用词，与目前习用的名词稍有差异（例如 template）。某些名词非常抽象

（例如 message）。参考就好，切勿染著文字障。

**2**

第１章 C++ Classes 的語法和語意

来」，这个「模板」其实就是「东西的属性描述」，也就是所谓的 class（上述定义使用了 template 一词，易起混淆，因为 “template” 在C++ 语言中另有所指， 见本章末的 Template 一节）。东西的属性（class）由「资料」和「用以处理该资料的程序（在计算机世界里就是函式）」两者组成。

现在我们再看看牛津计算机字典对于 class 这个词条的解释：

class：一种由程序语言 SIMULA 所导入的设施（facility）。Class 提供了一种用以描述抽象资料型别（abstract data type）的型式，它也是 Smalltalk 及其他object-oriented languages 中所谓的 object 概念的基础。

请注意「抽象资料型别」几个字，这道出 classes 只是一种「属性的描述」，并非实体；根据 class 产生出来的 object，才是实体。「产生 object」的动作，我们称之为 instantiation。

再让我们看看牛津计算机字典对于 instantiation 的解释：

instantiation：根据一个 object class（或称为 generic unit 或 template）产生

出一个实体（instance）。

Object-oriented 的程序，是以objects 做为程序的机能主体。我们（程序员）可能花许多精力来描述 object 的属性，包括其资料和行为（处理资料的方法），并以classes 语法呈现出我们的设计。程序开始执行时，便根据设计好的 classes 一一产生出 objects。这些 objects 彼此合作，遂行程序机能。

从广义的角度去看，语言内建型别如 int, float, long 等等也都是一种 class，用以描述资料性质（但是没有相关的处理程序）。因此下面的 *i* 和 *f* 都是 objects：

int i; // i 是一个 int object. float f; // f 是一个 float object.

#### 3

多型與虛擬（*Polymorphism in C++*）

现在让我以一个实例说明 object 和 class 的关系。假设我的classes 设计如下：

struct Point

{

long x;

long y;

};

*CStroke* 描述「线条」应该有的特征（包括资料与行为）

class CStroke

{



public:

void DrawStroke( /\*...\*/ ) { }; // 画线条

void Serialize( /\*...\*/ ) { }; // 档案读写

// ... User Interface（UI） functions

protected:

vector<Point\*> m\_Array; // C++ standard library's array int m\_penColor; // 笔的颜色

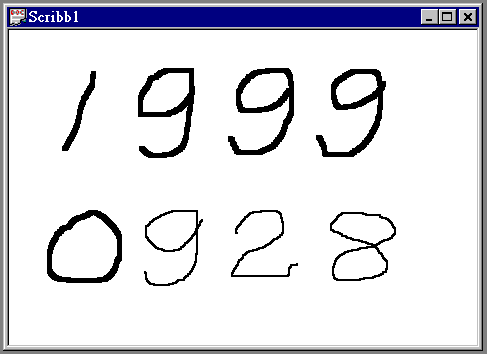
int m\_penWidth; // 笔的宽度

int m\_penStyle; // 笔的式样

};

于是每当我（程序设计者）允许使用者在荧幕上画一条线条（stroke），我便在程序中产生一个 *CStroke* object，其中有一个 array 准备用来储存线条上各点的坐标，以及三个 int 变量分别代表笔的颜色、宽度、式样。我必须设计一些 UI 函式，让使用者能够以鼠标代替画笔，直接在荧幕上输入构成线条的每一个点坐标。以下图为例，我（程序设计者）在程序中产生了 8 个 *CStroke* objects。

程序中拥有 8 个CStroke objects， 各有笔宽、颜色、式样、阵列。



#### 4

第１章 C++ Classes 的語法和語意

Classes 的宣告与实作（ Declaration & Implementation）

下面是 C++ class 的宣告形式（declaration），通常放在 .H 档中：

class declaration

#0001 class CPoint { class head

#0002 public:



#0003 CPoint( float x = 0.0 ) : \_x( x ) { } #0004

#0005 float x() { return \_x; }

#0006 void x( float xval ) { \_x = xval; } class body

#0007 // ...

#0008 protected:

#0009 float \_x; // data member #0010 };

称此为 class declaration 的代表人物与书籍是 Stanley Lippman 的 ***C++ Primer***。但是却也有人（例如 Bjarne Stroustrup 在其 ***The C++ Programming Language*** 中）把这样的东西称为 class definition。Bjarne 是 C++ 语言的创始者，Lippman 是全世界第一套 C++ 编译器（贝尔实验室之 cfront）的实作者。他们两人曾长期共事，都是 C++ 发展过程中的关键人物，怎么会在这么基本的用词上口径不一致呢？我很纳闷。

Class 内含两种东西：一是 data member（s 例如上面的 *\_x*），一是 member functions

（例如上述的 *CPoint()* 以及两个 *x()*）。

上例的三个 member functions 都直接在 class 中完成了实作内容，并因此自动成为一个 inline 函式（inline 函式将在稍后提及）。通常函式实作码极简易者，我们才会这么做。如果函式比较复杂，我们会在 class declaration 中完成 member functions 的宣告，并在 .CPP 档完成其实作部份（implementation），例如：

#### 5

多型與虛擬（*Polymorphism in C++*）

##### class declaration

class (member functions)

##### implementation

// in hello.h

#0001 class CPoint { #0002 public:



#0003 CPoint( float x = 0.0 ) : \_x( x ) { } #0004

#0005

#0006

float x();

// 只是 member function 的宣告

void x( float xval ); // 只是 member function 的宣告

#0007 // ...

#0008 protected:

#0009 float \_x;

#0010 };

#0003 float CPoint::x() { return \_x; }

// 實作內容

#0004 void CPoint::x( float xval ) { \_x = xval; } // 實作內容

// in hello.cpp

#0001 #include "hello.h"

#0002

称此为 function implementation 的代表人物与书籍是 Stanley Lippman 的 ***C++ Primer*** 。但是也有人（ 例如 Bjarne Stroustrup 在其 ***The C++ Programming Language*** 中）把上述这样的函式实作码称为 function definition。再一次，我很困惑，他们两人怎么会在这么基本的用词上口径不一致呢？

一个普遍被大家接受的说法是，会在存储器中配置空间者，谓之 definition。

符号命名习惯（ Naming Convention）

有一种常见的符号命名习惯是，在 class 名称前面加上 'C' 2，在 data members 名称前面加上 *m\_*。至于 member functions，并没有什么常见的命名习惯。

1. Microsoft MFC 的所有 classes 都以 'C' 做为名称前置词，Borland OWL 则以 'T' 做为所有classes 的名称前置词。MFC 和 OWL 是市场上两套有名的 C++ class libraries，由于它们具备某些高阶性质，所以也被称为 application framework。

#### 6

第１章 C++ Classes 的語法和語意

## Function Overloading（函式多载化）

所谓多载（overload），是指不同的函式实体共享同一个函式名称。例如稍早所提的 *CPoint* 之中，有两个 member functions 的名称同为 *x()*：

class CPoint { public:

...

float x();

void x( float xval );

};

其实作码如下：

float CPoint::x() { return \_x; }

void CPoint::x( float xval ) { \_x = xval; }

函式名称虽相同，但参数不同（个数不同，型别也不同），实作码也不相同。

C++ 之所以有 function overloading 的机制，是为了解决「只因资料型别不同，就必须设计算法完全相同但函式名称不同的多份函式码」的缺憾。举个例子，加法就是加法，如果为了 int、float、double、long... 各设计出对应的 *AddInt()*、*AddFloat()*、*AddDouble()*、*AddLong()*... 等加法函式，我们会觉得很琐屑，不漂亮。如果能够把它们全部都叫做 *Add()*，而我们不管对哪一种资料型别做加法运算， 都呼叫 *Add()* 就好，那么就逻辑层面来看就清爽多了。至于底层机制，则由编译器根据 *Add()* 所获得的参数型别，决定真正该唤起哪一个 *Add()* 函式实体。

于是，以加法为例，我们可以设计以下四个同名函式：

int Add(int a, int b); long Add(long a, long b);

double Add(double a, double b); float Add(float a, float b);

并这样使用之：

**7**

多型與虛擬（*Polymorphism in C++*）

int i = Add(3, 5); long l = Add(3L, 5L);

double d = Add(3.141592653, 5.141592653);

float f = Add(3.14, 5.14);

Function overloading 适用于 class member functions（如先前的 *CPoint::x()*），也适用于一般的 global functions（如上述的 *Add()*）。

Function overloading 无法适用于「函式名称相同，参数（包括个数与型别）也完全相同，只有回返值型别不同」的情况。这种情况将无法通过编译，会出现错误讯息如下（以 Visual C++ 为例）：

error C2556: 'Add' : overloaded functions only differ by return type

我想你已经看到了，所谓 function overloading，只不过是让诸多函式实体可以采用相同的函式名称，事实上你还是要撰写好几份不同（但逻辑十分类似）的函式实作码。为了能够「以相同的算法处理不同型别的资料」，比较更好的作法是， 把「参数的型别」亦视为一种参数。C++ template 可以办到这一点，稍后我再来谈这个非常棒的主题。

#### 8

第１章 C++ Classes 的語法和語意

## Operator Overloading（运算子多载化）

与 function overloading 异曲同工的是，C++ 提供所谓的 operator overloading（运算子多载化）。所谓 operators 是像 +（加）-（减）\*（乘）/（除）>>（位元右移）<<（位元左移）之类的符号，代表一种动作。

面对 operators，我们应该想象它是一种函式，只不过形式比较特殊罢了。一般函式的参数出现在括号 ( ) 之中，而 operator 的参数围绕着一个 operator 符号（之前或之后）。内建资料型别（如int、float、long 等）的operators 系由语言本身提供，并非借助于任何 libraries。

C++ 允许程序员为 class type’s objects 设计专属的 operators，使objects 的操作能够像内建型别一样地自然而直觉。让 operator 以多风貌的姿态呈现，但符号不变， 此即 operator overloading。

Operators overloading 的存在一直褒贬不一。赏识者认为它带给程序码更精简漂亮的形式，批评者则认为容易把程序员搞迷糊掉。但是，我想，谁都不否认下面这样的程序码实在是满「精简漂亮」的：

CString str1("Hello, I am J.J.Hou, "); CString str2("How are you?");

CString str3 = str1 + str2; // + 是 CString 的一个 overloaded operator

// 于是 str3 为 "Hello, I am J.J.Hou, How are you?"

int i = 5;

double pi = 3.141592653;

cout << "i=" << i << " and pi=" << pi << endl;

// 于是 console 荧幕上出现 "i=5 and pi=3.141592653"。

现在让我示范一个 overloaded operator 的作法。只要在先前的 *CPoint* 宣告之中加入以下的函式定义，即可完成 *CPoint* 的「加法运算子」：

**9**

多型與虛擬（*Polymorphism in C++*）

CPoint operator+(const CPoint& pt) const 3

{

return CPoint(\_x + pt.\_x);

}

以 *operator* 开头的函式，表示是个运算子。加法运算子需要左右两个操作数

（operands），左端操作数即成为本函式的 this 物件（关于 *this* 指标，第２章

p.100 有详细的说明），右端操作数则是本函式的参数 *pt*。上述函式码如果像下面这样表示，会比较清晰，但两者其实相同（因为 *this* 指标是个隐藏参数）：

CPoint operator+(const CPoint& pt) const

{

return CPoint(**this->**\_x + pt.\_x);

}

有了上述的 *CPoint::operator+*，现在你可以这么使用它：

CPoint pt1(7.2), pt2(5.3), pt3; pt3 = pt1 + pt2;

cout << "pt3.x = " << pt3.x() << endl; // 12.5

最著名而且使用率最高的 overloaded operators 大概非 << 和 >> 莫属。C++ iostream 把所有内建型别对标准输出入装置（stdout 和 stdin，也就是荧幕和键盘） 的读写动作都以这两个 operators 表示，带给我们极大的便利，可以取代形式麻烦的 *printf()* 和 *scanf()*。

#### Insertion Operator（ <<） for stdout

<< 本来是位元左移运算子，但是在C++ 标准的 *iostream* 中被改头换面，其左端操作数（operand）被指定为 cout（console output device），右端操作数是一个内建型别的 objects。我可以利用它很方便地对 cout 连续输出各种内建型别的资料

1. & 是所谓的 reference，一个像 pointer 之类的东西。稍后我会介绍。

#### 10

第１章 C++ Classes 的語法和語意

（也是一种 objects），不必再像 C 程序那样需要记忆不同型别的资料在 *printf()*

中的格式符号（如 %s、%d、%f...）：

#include <iostream.h>

...

int i = 5;

double pi = 3.141592653;

cout << "i=" << i << " and pi=" << f << endl;

#### Extraction Operator（ >>） for stdin

>> 本来是位元右移运算子，但是在C++ 标准的 *iostream* 中被改头换面，其左端操作数（operand）是 cin（console input device），右端操作数是一个内建型别的object。例如：

#include <iostream.h>

...

float f;

cout << "f = ? "; cin >> f;

程序使用者在键盘上的输入，会经由 cin 装置流往 *f* 变量，其作用相当于 C runtime library 中的 *scanf()*。

#### Insertion Operator（ <<） for file

就像 *iostream* 支援对 user's terminal 的输出入一样，C++ 另有一组 class types

支援对档案的输出入：

1. *ifstream* 衍生自 *istream*， 支援档案的输入。
2. *ofstream* 衍生自 *ostream*， 支援档案的输出。
3. *fstream* 衍生自 *iostream*， 支援档案的输出入。

如果需要这些 class type，必须含入对应的 header file。请注意，*fstream* 已经含入了 *iostream*，所以我们不需重复载入。

#### 11

多型與虛擬（*Polymorphism in C++*）

下面是将资料输入到档案的一个简单范例：

#0001 #include <fstream.h> // include iostream.h #0002

#0003 int main()

#0004 {

#0005 cout << "All your keyboard input in monitor " #0006 << "will into copy.txt.\n"

#0007 << "Terminate by ^Z\n\n"; #0008

#0009 ofstream outFile("copy.txt"); #0010 if (!outFile) {

#0011 cout << "Cannnot open copy.txt for output" << endl; #0012 return -1;

#0013 }

#0014

#0015 char c;

#0016 while (cin.get(c)) // get each char from keyboard #0017 outFile << c; // outFile.put(c) 亦可

#0018

#0019 return 0;

#0020 }

当程序执行起来，会先给一个提示，然后接收所有来自键盘（亦即显示于荧幕上） 的字符，一一放进 "copy.txt" 档案之中，直到接收 ^z 才停止。

#### Extraction Operator（ >>） for file

下面是将档案内容输出到荧幕上的一个简单范例：

#0001 #include <fstream.h> // include iostream.h #0002

#0003 int main()

#0004 {

#0005 cout << "All text in copy.txt "

#0006 << "will stream into monitor.\n\n"; #0007

#0008 ifstream inFile("copy.txt"); #0009 if (!inFile) {

#0010 cout << "Cannnot open copy.txt for input" << endl; #0011 return -1;

#0012 }

#### 12

第１章 C++ Classes 的語法和語意

#0013

#0014 char c;

#0015 while (inFile.get(c)) // get each char from file #0016 cout.put(c); // output to monitor

#0017

#0018 return 0;

#0019 }

当程序执行起来，会先给一个提示，然后从档案 "copy.txt" 中一一读出每一个字符，显示于荧幕上。上述 L15 如果改为：

while (inFile >> c) // get each char from file

虽然也能读出字符，但空格符会被忽略，并非我们所期望。

在 *iostream* 和 *fstream* 之中绝不只提供 operator<< 和 operator>> 而已，还有许多其他的 member functions，例如 *get()*、*put()*、*peek()*、*getline()*、*putback()*...。

virtual inheritance

istream

ostream

ofstream

fstream

ifstream

iostream

ios

fstream 的 operator<< 和operator>> 只接受语言内建资料型别。如果你这么做：

CPoint aPoint; // CPoint is your class type outFile << aPoint; // OO persistence!

// 第５章即致力要完成类似的行为

在 Visual C++ 编译器中会得到错误讯息如下：

error C2679: binary '<<' : no operator defined which takes a right-hand operand of type 'class Object' (or there is no acceptable conversion)

#### 13

多型與虛擬（*Polymorphism in C++*）

## Name Mangling（符号的特殊处理）

如果函式名称都相同（也就是被 overloaded），编译器在面对你的函式唤起动作时，究竟如何才能决定哪一份函式实体应该「起而行」呢？

事实上，编译器把所有同名的 overloaded functions 视为不同的函式，并且以特殊方式对它们的函式名称做了手脚。以四个 *Add()* 为例：

int Add(int a, int b); long Add(long a, long b);

double Add(double a, double b); float Add(float a, float b);

编译器真正为这四个函式所编码出来的函式名称是：

?Add@@YAHHH@Z 4

?Add@@YAJJJ@Z

?Add@@YANNN@Z

?Add@@YAMMM@Z

再以 *CPoint* 的两个 *x()* 为例：

class CPoint { public:

...

float x();

void x( float xval );

};

编译器真正为它们编出来的函式名称是（其中夹杂 class 名称、参数个数与型别）：

?x@CPoint@@QAEMXZ

?x@CPoint@@QAEXM@Z

1. 不同的编译器有不同的名称编码方式，本例使用 Visual C++。如果你想观察这些「真正的」函式名称，它们被记录在 map 档。要产生 map 档，以 Visual C++ 为例，你只需在 link.exe 后面加上 /map:filename 选项，即可获得 filename.map。

**14**

第１章 C++ Classes 的語法和語意

这样的「名称改变」行为称为 "name mangling"。遗憾的是，不同编译器有着不同的 name mangling 作法。这当然不好，表示我以 Microsoft C++ 编译器开发出来的程序没办法呼叫一个以 Borland C++ 编译器完成的函式。换句话说，函式名称并不存在着 binary standard，这就严重影响了 software components5 的发展。

Naming Mangling 不只施行于 overloaded functions 身上，C++ 程序中所有的

global 函式以及 classes 中的所有 members，都会被施以 naming mangling 手术。如果要抑制 naming mangling 的作用，就必须加上修饰词 *extern "C"*。

## extern "C"

对程序员而言，本来不必在意Naming mangling。但如果你的 C++ 程序想要汇出

（export）某些东西给外界使用，naming mangling 就会带来困扰。

能够在一个 C++ 程序中汇出（export）的东西，不外乎是 functions 和classes（data的汇出虽亦属可能，但是既然高唱「资料封装」，汇出 data 是极罕见并且应该避免的事）。Classes 无论是否汇出，无可避免都要被 name mangling。但是 functions 不一样，当我们企图汇出 functions，我们就不希望它们被编译器施以 naming mangling 手术，因为我们不希望这些 functions 的使用者被限制使用「与 functions 设计者相同的开发工具」。毕竟，到目前为止，各家 C++ 编译器厂商并没有协调出一个统一的 name mangling 作法。

因此，对于即将汇出（exported）的函式，为了抑制 naming mangling 作用，我们必须加上特殊的修饰词 *extern "C"*，例如：

extern "C" int Add(int a, int b);

1. Software components 的美丽远景是，希望开发出来的软件元件（软件 IC）能通行于不同的编译平台、甚至打破不同程序语言的藩篱。要达到这一点，就需要 binary standard。请参考***Essential COM***（Don Box/Addison Wesley/1998。中译本 COM 本质论，侯俊杰译，碁峰 1998）

**15**

多型與虛擬（*Polymorphism in C++*）

观察 map 档，就可以看见这个 *Add()* 函式的真实名称为 \_Add 6，而不是奇奇怪怪（经过mangling）的 ?Add@@YAHHH@Z。如果你这么写：

int Add(int a, int b); extern "C" long Add(long a, long b);

extern "C"

编译时会出错（以Visual C++ 为例）：

error C2733: second C linkage of overloaded function 'Add' not allowed

通常，准备汇出（exported）的 functions 会被设计在 DLLs（动态联结函式库） 之中，因此我们比较有机会在 DLLs 看到 *extern "C"* 的使用。举个例子，如果我以 C++ 撰写 Windows DLLs，并且决定写 *DllMain()*（这其实并非必要，视需求而定），由于这个 *DllMain()* 是 callback 函式（由系统回呼的函式），其名称与格式都已固定，为了阻止 C++ 编译器在我的 *DllMain()* 上施以 name mangling 手脚，我必须加上这么一行：

extern "C" int APIENTRY 7 DllMain(HINSTANCE hInst, DWORD dwReason, LPVOID lpReserved)

{

...

}

1. 在每一个函式名称之前加上 '\_'，是 C 编译器对于 C 函式名称的传统作法；C++ 也延续这个传统。曾经以 assembly 语言呼叫 C 函式的朋友，想必对此不陌生。
2. APIENTRY 是什么？在 \MSDEV\INCLUDE\WINDEF.H 中有这样的定义：

#define WINAPI stdcall （一种 calling convention，适用于win32） #define APIENTRY WINAPI

#### 16

第１章 C++ Classes 的語法和語意

三种封装层级：private、protected、public

再次以 *CPoint* 为例：

#0001 class CPoint { #0002 public:

#0003 CPoint( float x = 0.0 ) : \_x( x ) { } #0004

#0005 float x() { return \_x; }

#0006 void x( float xval ) { \_x = xval; } #0007 // ...

#0008 protected:

#0009 float \_x;

#0010 };

我们看到 class 宣告中有 public: 和 protected: 两个关键字。这一类关键字用来修饰 members（包括 data 和 functions）的处理层级（access level），也就是封装程度。这类关键字一共有三个：

* private: 表示以下 members 属于 class 私有，不允许在 class 范围以外被取用。
* protected: 表示以下 members 除了对自己这个 class 及其 derived class 开放外， 其余一概免谈。
* public: 表示以下 members 公开为外界使用，不限于 class 内部存取。

下面是封装层级的一个整理：

|  |  |  |  |
| --- | --- | --- | --- |
| 封装（保护）层级： | public | protected | private |
| in class (or friend)： | 可存取 | 可存取 | 可存取 |
| derived class： | 可存取 | 可存取 | 不可存取 |
| 程序其他地方： | 可存取 | 不可存取 | 不可存取 |

#### 17

多型與虛擬（*Polymorphism in C++*）

封装程度是可以继承的。我们可以在 classes 的继承8 关系上，指定一个封装关键字，做为「大体继承」，例如：

class CPoint2d : public CPoint { ... };

或是

class CPoint2d : protected CPoint { ... };

或是

class CPoint2d : private CPoint { ... };

至于「大体继承」和「细部继承」之间的交互作用，可以这张表做为说明：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| class 继承方式：  base class 封装层级 | | private | protected | public |  |
|  |  |  | | |  |
| public |  | private | protected | public |
| protected |  | private | protected | protected |
| private |  | N/A | N/A | N/A |
|  |  | | | |

图中粗框内即是在「base class 封装层级」与「derived class 继承方式」交互作用下，base class members 对于 derived class 的封装层级。

1. class 的继承，将在稍后讨论。

#### 18

第１章 C++ Classes 的語法和語意

## Struct 和 Class 的差异

C++ 的 struct 其实和 class 地位几乎完全相同，这恐怕和大家的传统印象大不相同。C++ struct 一样可以拥有 member functions，一样可以拥有 data members。只不过，class members 的预设封装层级是 private，而 struct members 的预设封装层级是 public。

预设归预设，如果你有特别指定，struct 和 class 两者可以彼此完全取代。举个例子，前面的 class *CPoint* 可以改写如下，意义完全相同：

#0001 struct CPoint { #0002 public:

#0003 CPoint( float x = 0.0 ) : \_x( x ) { } #0004

#0005 float x() { return \_x; }

#0006 void x( float xval ) { \_x = xval; } #0007 // ...

#0008 protected:

#0009 float \_x;

#0010 };

C++ struct 的一个最大应用，我想大概是 COM interfaces 了。COM（Component Object Model）要求 class 的界面和实作要彻底分开（这么一来界面部份才能有binary standard），而界面部份既然是要完全对外公开，根本不会（不该）有 private members ， 所以 COM interface 以 C++ struct 来完成将是非常合适。COM programming 中有一个 interface 关键字，事实上，它就是 struct。你可以在OBJBASE.H（COM 程序一定要含入的一个档案）中找到这一行：

// in MSDEV\INCLUDE\OBJBASE.H

#define interface struct

我将在第６章谈到 COM 的基本精神，并实作一个 COM 骨干程序。

**19**

多型與虛擬（*Polymorphism in C++*）

Objects 的诞生（Objects Instantiation）

根据前述对 class *CPoint* 的宣告，我们可以产生一个 object（物件），或说是产生一个 instance（实体）：

CPoint aPoint(7.2); // aPoint.\_x 初值为 7.2 aPoint.x(5.3); // aPoint.\_x 现值为 5.3

这样的 objects 可能放在函式的 stack 之中（如果上面的码位于函式范围内）， 也可能放在程序的 data segment 中（如果上面的码位于函式范围之外）。

我也可以这样产生一个 objects：

CPoint\* pPoint = **new** CPoint(3.6); // pPoint->\_x 初值为 3.6 pPoint->x(5.3); // pPoint->\_x 现值为 5.3 **delete** pPoint;

利用 *new* operator 所产生的objects，系放在程序的 heap 之内。

不管哪一种方式，我们把「根据某个 class，产生出一个 object」的动作称为instantiation（具现化、实体化）。Object 的诞生时刻和死亡时刻，各会自动引发class 中某个特殊的 member function，称为 constructor 和 destructor。稍后我再来细谈这两个特殊函式的写法。

* Constructor： object 诞生时会被自动唤起的 class member functions。此函式的命名必须与 class 相同，参数可自定，没有回返值。Class 可以拥有一个以上的 constructors， 其中无参数的那个特别称为 default constructor； 只有一个参数并且以该 class 为型别者， 特别称为 copy constructor。
* Destructor ： object 生 命 结 束 时 会 被 自 动 唤 起 的 class member function。一个 class 只能够有一个 destructor， 没有参数， 没有回返值， 其命名必须与 class 相同， 并以 ~ 为前置符号。

#### 20

第１章 C++ Classes 的語法和語意

Objects 的生命（Scope of Objects）

由于 objects 可能位于 stack 或 heap 或 data segment 之中，所以 objects 的生命周期就有了些许差异。

* 放在 stack 之中的称为 local objects，它的生命随着 objects 的产生而开始， 随着所在函式的结束而结束。
* 放在 data segment 之中的称为 global objects，它的生命随着程序的开始而开始（ 比程序进入点还早）， 随着程序的结束而结束。
* 放在 heap 之中的称为 heap objects， 它的生命随着 *new* operator

而展开， 随着 *delete* operator 而结束。

下面这个例子出现刚刚所提的三种不同生命周期的 objects。从程序的执行结果， 我们可以清楚看到三种 objects 的生命范围。其中用到的 constructors（建构式） 和 destructors（解构式）将在稍后详细介绍。

注意：#0027 的 static object 将于本章稍后提到，请暂时略过。程序中的标记请对照执行结果。

|  |  |  |
| --- | --- | --- |
| #0001 | #include <iostream.h> |  |
| #0002 | #include <string.h> |
| #0003 |  |
| #0004 | class CDemo |
| #0005 | { |
| #0006 | public: |
| #0007 | CDemo(const char\* str); | // constructor |
| #0008 | ~CDemo(); | // destructor |
| #0009 | private: |  |
| #0010 | char name[20]; |  |

#0011 };

#0012

#0013 CDemo::CDemo(const char\* str) // constructor #0014 {

#0015 strncpy(name, str, 20);

#0016 cout << "Constructor called for " << name << '\n';

#### 21

多型與虛擬（*Polymorphism in C++*）

#0017 }

#0018

#0019 CDemo::~CDemo() // destructor #0020 {

#0021 cout << "Destructor called for " << name << '\n'; #0022 }

#0023

#0024 void func()

#0025 {

#0026 CDemo **LocalObjectInFunc**("LocalObjectInFunc"); 

#0027 static CDemo **StaticObject**("StaticObject"); 

#0028 CDemo\* **pHeapObjectInFunc** = new CDemo("HeapObjectInFunc"); 

#0029

#0030 cout << "Inside func" << endl; 

#0031 

#0032 }

#0033

#0034 CDemo **GlobalObject**("GlobalObject"); 

#0035

#0036 void main()

#0037 {

#0038 CDemo **LocalObjectInMain**("LocalObjectInMain"); 

#0039 CDemo\* **pHeapObjectInMain** = new CDemo("HeapObjectInMain"); 

#0040

#0041 cout << "In main, before calling func\n"; 

#0042 func();

#0043 cout << "In main, after calling func\n"; 

#0044   

#0045 }

执行结果如下（注意，上例有 *new* 的动作，却忘了 *delete*，是个错误示范）：

|  |  |
| --- | --- |
| 1. Constructor called for GlobalObject |  |
| 2. Constructor called for LocalObjectInMain |  |
| 3. Constructor called for HeapObjectInMain |  |
| 4. In main, before calling func |  |
| 5. Constructor called for LocalObjectInFunc |  |
| 6. Constructor called for StaticObject |  |
| 7. Constructor called for HeapObjectInFunc |  |
| 8. Inside func |  |
| 9. Destructor called for LocalObjectInFunc |  |
| 10. In main, after calling func |  |
| 11. Destructor called for LocalObjectInMain |  |
| 12. Destructor called for StaticObject |  |
| 13. Destructor called for GlobalObject |  |

#### 22

第１章 C++ Classes 的語法和語意

## Pointers 与 References

Pointer 译为「指标」，相当贴切。Reference 译为「参考」，我总觉得不妥，字里行间很容易让人搞不清楚是动词还是名词、是一般所谓的「参考」还是一个 C++ 专有名词。所以，我将使用原文名词 reference。连带地，在这一节中我也使用原文名词 pointer。

#### Pointers

Pointer 是一个「指物器」，可以用来指向任何一个 objects，包括一般变量：

int i = 3; int\* pi = &i;

cout << pi << endl; // 0x0064FDF0

cout << \*pi << endl; // 3

此时 *pi* 本身内含 *i* 的位址（在 32 位元系统中位址为 32 位元）。欲取出 *pi* 所指的 object，可使用 \* 运算子（dereference operator）。

Pointer 也可以用来指向任何一个 class type objects。由于下面的 *pPoint* 指向一整个大结构（一个 object），如果要取用其中的 members （data members 或 member functions 皆可 -- 只要其封装层级是 public），必须使用 -> 运算子（arrow operator），例如：

class CPoint { public: float \_x, \_y, \_z; }; CPoint\* pPoint = new CPoint;

cout << pPoint << endl; // 0x00770560 pPoint->\_x = 9.28;

cout << pPoint->\_x << endl; // 9.28

Pointer 甚至可以指向一个不明物件（void）：

#### 23

多型与虚拟（*Polymorphism in C++*）

void\* pv;

pv = malloc(1024); // 配置 1024 bytes（来自 heap） cout << pv << endl; // 0x00760A14

如果要提取 *pv* 所指的物件，不可以，会出现编译错误：

cout << \*pv << endl; // error C2100: illegal indirection

也就是说，pointer 本身没有意义（好像尺度本身没有意义一样），它的意义来自于它的型别（好像尺度的意义来自于坐标单位一样）。因此，将 pointer 前进一个单位，究竟是前进多少个 bytes 呢？必须视 pointer 的型别而定。如果将前述的 int\* *pi* 和 CPoint\* *pPoint* 各加 1，得到：

cout << ++pi << endl; // 0x0064FDF4，比原先增加 4 cout << ++pPoint << endl; // 0x0077056C，比原先增加 12

这是因为在 32 位元系统中 int 为 4 bytes，而我们所设计的 *CPoint* 里面有三笔

float 资料，大小为 12 bytes9，所以各指标累加 1 时，分别前进 4 bytes 和 12 bytes。

如果把一个指向不明物件的指标（void\*）加 1，会得到什么结果？不会有结果， 只会有编译错误：

cout << ++pv << endl; // error C2036: 'void \*' : unknown size

当然，如果你做了强制转型（cast）动作，就可以解决「不明物件」的问题，因为你赋与了该指标一个明确的型别。例如：

double\* pd = (double\*)pv; // 強制轉型。double is 8 bytes. cout << pd << endl; // 0x00760A14

cout << ++pd << endl; // 0x00760A1C，比原先增加 8

1. member functions 并不包括在 object 大小之内。第２章会详细说明所谓的物件模型。

#### 24

第１章 C++ Classes 的語法和語意

下表是上述实验结果的一个整理：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pointer 型別 | pointer 值 | ++pointer | object 大小 | 提取（dereference） |
| int\* pi; | 0x0064FDF0 | 0x0064FDF4 | 4 bytes | 可 |
| CPoint\* pPoint; | 0x00770560 | 0x0077056C | 12 bytes | 可 |
| void\* pv; | 0x00760A14 | error | error: 'void \*' unknown size | error: illegal indirection |
| double\* pd = (double\*)pv; | 0x00760A14 | 0x00760A1C | 8 bytes | 可 |

当我们开始设计 classes 继承体系，有许多时候需要把一个 pointer 指向一个型别不符的 object（但彼此型别之间有继承的关系存在。这其实正是 polymorphism 的一个精髓，我将在第２、４章详细讨论之），这时候型别的转换就非常重要。强制转型（C 程序总是用这一招）太过粗暴，在不够安全的时候仍然硬ㄠ过去，有危险。C++ 需要其他更精致的转型工具（请看第３章）。

Pointer 不但可以指向 object，还可以指向 class 的 data members 或 member functions。它们的形式有点奇怪，结果也可能出人意表。我将在第２章 p.116 和

p.117 详细介绍这一部份。

#### References

与 pointer 常常相提并论并且常常被混淆不清的是所谓的 reference。Reference

（&）象是一个常数指标，可以被自动提取（dereference）。下面这个例子使 *r* 成为 *x* 的一个 reference：

int x;

int& r = x; // r is a reference of x

当一个 reference 产生，它必须被初始化为某个原已存在的 object，像上面那样。如果我写：

#### 25

多型與虛擬（*Polymorphism in C++*）

int& q = 12;

那么编译器会先配置一块 int 存储器空间，将内容设为 12，然后把 *q* 这个reference「系缚」到该空间上。重点是，任何 reference 都必须被「系缚」到某一个空间，成为一个...唔...「化身」。当你处理该 reference，你就是在处理那个被系缚的空间。如果：

int x = 0;

int& r = x; // r is a reference of x int\* p = &x; // p is a pointer to x

r++; // 請注意：sizeof(r) == sizeof(x)

那么 *r* 和 *x* 的现值都为 1，因为增加 *r* 的值就是增加 *x* 的值。

memory

注意：sizeof(r) == sizeof(x)



int\* p = &x;

int& r = x;

int x = 0;

###### r is a reference of x

p is a pointer, point to x

面对reference，最简单的想象就是，把它幻想为一个形式漂亮的 pointer。这个「形式漂亮的 pointer」好处是，我们不需要担心它是否被初始化（编译器会强迫做）， 也不需要担心如何提取（dereference）它（同样地，编译器会负责）。

Reference 虽然在本质上是个指标，在形式上却是个 object。也就是因为其形式漂亮而本质实用（用于 call by reference，或称 call by address），所以 reference 常被用于函式的自变量列（arguments list）和传回值（return value）。下面是个例子：

#### 26

第１章 C++ Classes 的語法和語意

#0001 int\* func1(int\* x) #0002 {

#0003 (\*x)++;

#0004 return x;

#0005 }

#0006

#0007 int& func2(int& x) #0008 {

#0009 x++;

#0010 return x;

#0011 }

#0012

#0013 main()

#0014 {

#0015 int a = 0;

#0016

#0017 // ugly but explicit，你可以清楚看到传给 func1() 的是个指标。

#0018 // 传回值的形式也很「难看」。

#0019 cout << \*func1(&a) << endl; // output: 1 #0020

#0021 // clean but hidden。传给 func2() 的其实是个指标（藉由 reference），

#0022 // 但你看不出来。传回的也是指标（藉由 reference），你也看不出来。#0023 cout << func2(a) << endl; // output: 2

#0024 }

Reference 对于 copy constructor 的设计十分重要。我将在稍后介绍 copy constructor。至于为什么 copy constructor 需要 reference，其底层机制请参考***Thinking in C++***（Bruce Eckel / Prentice Hall）一书第 9 章：Pointers & references。

#### 27

多型與虛擬（*Polymorphism in C++*）

Constructors & Destructors（建构式和解构式）

当 object 产生，有一个特殊的称为 constructor 的函式会自动执行。当 object 死亡，有一个特殊的称为 destructor 的函式会自动执行。Constructor 可以不只一个， 但 destructor 只能有一个。

Constructors （ 建构式） 就是与 class 同名的那些 member functions ， 例如*CPoint::CPoint()* 或 *CDemo::CDemo()*。Constructors 不能够指定回返值型别，也就是说它不必（但可以）return。你可以拥有一个以上的 constructors，各有不同个数以及不同型别的参数(s)。

Destructor（解构式）就是与 class 同名并且前面加上 '~' 符号的那个 member function，例如 *CPoint::~CPoint()* 或 *CDemo::~CDemo()*。Destructor 不能够指定回返值型别， 也就是说它不必（ 但可以） return。每一个 class 只能拥有一个destructor，并且不能有任何参数。

由于 global object 的诞生比程序进入点更早，所以 global object 的 constructor 执行时机更早于程序的进入点。这一点你一定已经在 p.21「Object 的生命」一节实例中注意到了。

Default Constructors

所谓 default constructor 就是没有指定任何参数的 constructor。如果我们的 class

*CA* 宣告如下：

class CA { public:

int getdata() { return \_data; };

CA() { \_data = 5; };

CA(int val) { \_data = val; };

// default constructor

// constructor

protected: int \_data;

};

**28**

第１章 C++ Classes 的语法和语意

*CA* 拥有两个 constructors，分别是 *CA(int)* 和 *CA()*。后者没有参数，正是所谓的

default constructor。当我产生一个 *CA* object 而没有指定任何参数：

CA aCA;

CA\* pCA = new CA;

编译器就为我呼叫 default constructor。

普遍存在于 C++ 程序员之间的一个误解是：如果我们没有为某个 class 设计constructor，编译器会自动为该 class 制造出一个 default constructor 来。这个说法也对也不对。这里我先提示一个结论（稍后有更多细节探讨）：

如果一个 class 没有任何 user-defined constructor，那么编译器可能会为此 classes 自动做出一个 default constructor。此一由编译器合成出来的 default constructor 只负责完成编译器所需的动作，无法照顾 user（programmer）的可能需求。

什么是「编译器所需」的动作？就是隐藏在 C++ 程序码语句下面，让 C++ 诸多特征得以实现的动作。包括以下三种情况：

CA object

CZ

object

1. class *CA* 内含 class *CZ* objects10。

当你产生一个 *CA* object，隐藏在底层的必要动作是：先将 *CZ* objects 初始化

（因为 *CA* object 之中有一个 *CZ* object）。

1. class *CA* 继承自 class *CZ*。

CA

object

CZ

subobject

当你产生一个 *CA* object，隐藏在底层的必要动作是，呼叫 *CZ* 的 default constructor（因为 *CA* object 之中有一个 *CZ* subobject）。

1. 这种情况称为 composition。此时的 *CZ* object 亦被称为 object members 或 embedded object。

#### 29

多型與虛擬（*Polymorphism in C++*）

vtbl

1. *CA* 是一个 polymorphic class，也就是说它拥有 virtual functions，或继承自拥有 virtual functions 的 class(es)（ 关于 virtual， 稍后有一节说明）。

当你产生一个 *CA* object，隐藏在底层的必要动作是：将虚拟机制所需的 vptr

|  |  |
| --- | --- |
|  | vptr |
|  |
| CA object | |

和 vtbl 初始化（第２章对于vptr 和vtbl 有详细的介绍）。

由于这三种情况皆有所谓「底层的必要动作」，所以编译器必须自动为 class *CA* 合成一个 default constructor（如果没有任何 user-defined constructor 的话），或暗中对已有的 user-defined constructors 动手脚（添加一些码）。合成出来的东西称为 implicit nontrivial default constructor11（这是 C++ Standard 的用语）。

如果没有上述需求，编译器就不会为 class 产生一个 default constructor。例如：

class CB { public:

int getdata() { return \_data; }; protected:

int \_data;

};

*CB* 没有符合虚拟机制（它没有 virtual functions，也不是继承自拥有 virtual functions 的 base classes），也没有内含 embedded objects，也没有继承自其他class，所以编译器不会为它合成一个 default constructor 出来。于是当我这么做：

CB aCB; // 应该呼叫 default constructor cout << "aCB.getdata()=" << aCB.getdata() << endl;

得到的结果是：

1. 很多人认为「无论如何，只要 class 之中没有 user defined constructor，编译器就会自动为它合成一个 default constructor。如果 class 不属于此处归纳的三种情况，合成出来的称为implicit trivial default constructor。」这个 trivial（什么事也没做）的函式是否真的存在？根据Lippman -- cfront（全世界第一套 C++ 编译器）实作者 -- 的说法，这个 trivial 函式其实并不存在。

#### 30

第１章 C++ Classes 的語法和語意

aCB.getdata()=4211382 // 莫名其妙的初值

这恐怕不是我所期望的。我期望 \_*data* 有个初值（最起码也该清为 0 吧），但这是程序员的需求，不是编译器的需求，所以我必须自求多福，明白地设计一个default constructor：

class CB { public:

int getdata() { return \_data; };

CB() { \_data = 5; }; // default constructor

protected: int \_data;

};

现在再执行以下动作，就符合我们对初值的期望了：

CB aCB; // 应该呼叫 default constructor cout << "aCB.getdata()=" << aCB.getdata() << endl;

// 输出结果：aCB.getdata()=5

注意，如果 class 已经有了任何 constructor 但非 default constructor，编译器绝不会为它合成一个 default constructor。如果 class *CZ* 正是如此的一个 class，那么当你想要产生一个 *CZ* object 并且没有指定参数时：

CZ \*aCZ; // error

编译器会发出抱怨的声音：

error C2512: 'CZ' : no appropriate default constructor available

注意：本小节对于 default constructor 的合成条件，只是一个概略性的介绍。***Inside the C++ Object Model*** （Lippman/Addison Wesley/1996 12）第２章对此有非常详细的描述与归纳。

1. 中译本为 深度探索 C++ 物件模型（侯俊杰译 / 碁峰 / 1998）

#### 31

多型与虚拟（*Polymorphism in C++*）

### Copy Constructors

所谓 copy constructor，是指那种「众多参数之中，有一个参数的型别是其 class type」的 constructor。例如：

#0001 class CA { #0002 public:

#0003 int getdata() { return \_data; };

#0004 CA() { \_data = 5; }; // default constructor #0005 CA(int val) { \_data = val; }; // constructor

#0006

#0007

#0008

#0009

CA(const CA& ca) {

// copy constructor

cout << "copy constructor" << endl;

\_data = 10;

};

#0010 protected:

#0011 int \_data;

#0012 };

以下两种情况，会唤起 copy constructor：

* 情况 1. 将一个 object 当做参数传给某一个函式。
* 情况 2. 将一个 object 当做函式回返值。

例如（沿用上面的 class *CA*）：

#0001 void foo2(CA aCA) // 情况 2（函式参数是个 object） #0002 {

#0003 cout << "in foo2(), aCA.getdata()=" << aCA.getdata() << endl; #0004 }

#0005

#0006 CA foo3()

#0007 {

#0008 CA aCA(3);

#0009 cout << "in foo3(), aCA.getdata()=" << aCA.getdata() << endl; #0010 return aCA; // 情况 3（函式传回值是个 object）

#0011 }

#0012

#0013 void main()

#0014 {

#0015 CA aCA1, aCA2(7);

#0016 cout << "aCA1.getdata()=" << aCA1.getdata() << endl; #0017 cout << "aCA2.getdata()=" << aCA2.getdata() << endl;

**32**

第１章 C++ Classes 的語法和語意

#0018

#0019 aCA2 = aCA1; // object assignment

#0020 cout << "aCA2.getdata()=" << aCA2.getdata() << endl; #0021

#0022 foo2(aCA1); // 情況 1（呼叫之前，aCA1.\_data 為 5）

#0023 aCA2 = foo3(); // 情況 2

#0024 cout << "aCA2.getdata()=" << aCA2.getdata() << endl; #0025 }

让我们看看执行结果：

aCA1.getdata()=5 aCA2.getdata()=7

aCA2.getdata()=5 // 经过 object assignment 之

后

copy constructor

in foo2(), aCA.getdata()=10 // 经过情况 1 之后

in foo3(), aCA.getdata()=3 copy constructor

aCA2.getdata()=10 // 经过情况 2 之后

这里有几点值得注意的地方：

* L19 将一个 object 指派（ assign）给另一个 object，这也是一种复制， 但它唤起的是所谓的 copy assignment operator。本例并没有特别设计copy assignment operator。
* 呼叫 *foo2()* 之前， \_*data* 为 5， 进入 *foo2()* 之后再印出， 已变成 10， 可见 copy constructor 的确在 *foo2()* 的参数复制时（ 情况 1） 发生。
* *foo3()* 内有一个 local object， 其 \_*data* 为 3， 把它当做回返值传出， 却变成了 10，可见 copy constructor 的确在 *foo3()* 的回返值复制时（ 情况 2） 发生。

当然，上述 class *CA* 的 copy constructor 设计内容是非常失败的，故弄玄虚，简直是拿石头砸自己的脚。为的只是要彰显 copy constructor 的发生。

许多人对于 copy constructor 有着和对 default constructor 一样的误解，以为如果没有 user-defined copy constructor，编译器便自动为我们做出一个。事实上，编译

#### 33

多型与虚拟（*Polymorphism in C++*）

器只有在必要的时候，才会这么做。

什么是必要的时候呢？就是当 class 不再展现 "bitwise copy" 意涵的时候。

#### bitwise copy - 概括承受

所谓 "bitwise copy"，是指「一个位元一个位元」地完整拷贝。一般的 objects 拷贝当然是如此，例如本节的 class *CA* 就展现了「bitwise copy 的意涵」。也就是说面对 class *CA*，其实你我不需要撰写 copy constructor，于是 p.32 的 *main()* 执行结果变成：

aCA1.getdata()=5 aCA2.getdata()=7 aCA2.getdata()=5

in foo2(), aCA.getdata()=5 in foo3(), aCA.getdata()=3 aCA2.getdata()=3

正確地滿足了需求。

但是另有一种拷贝情况，并不能够以「一个位元一个位元地完整拷贝」方式来进行，那种情况称为 "memberwise copy"。

#### memberwise copy - 选择性承受

当 classes 的设计融入了虚拟机制13，引进了一个额外的 data member：vptr（详见第２章 p.104），就不能够再以一个位元一个位元的方式对 object 做全额拷贝了，因为 vptr 需要有更复杂的考虑。这时候我们说这个 class「不再展现 bitwise copy 意涵」。

面對這樣的所謂 polymorphic classes，如果沒有 user-defined copy constructor，編

1. 如果 class 內含 virtual functions，或如果 class 繼承關係中出現虛擬繼承，那麼就有所謂的虛擬機制。

#### 34

第１章 C++ Classes 的語法和語意

譯器就會為它們合成出一個。但是，就如同先前討論的 default constructor 一樣， 這些「合成品」只滿足編譯器的需求；如果程式員另有需求，還是得自己寫出一個。

編譯器面對一個 user-defined copy constructor，會不會擴張它以加上編譯器自己的需求呢？讓我把本例的 copy constructor 改為：

CA(const CA& ca) { // copy constructor cout << "copy constructor" << endl;

// \_data = 10;

};

於是先前 p.32 的 *main()* 執行結果變成：

aCA1.getdata()=5 aCA2.getdata()=7

aCA2.getdata()=5 // 經過情況 1 之後

copy constructor

in foo2(), aCA.getdata()=4198864 // 經過情況 2 之後

in foo3(), aCA.getdata()=3 copy constructor

aCA2.getdata()=4222584 // 經過情況 3 之後

看来并没有「被扩张以加上编译器自己的需求（这里指的是 bitwise copy）」。可见我们自己撰写 copy constructor 时需非常注意其严谨度。

Copy constructor 的实作与 class reference（&）有密切关连，关系到函式的呼叫和传回、中断发生的需求及重进入（re-entrant）等细节，请参考 ***Thinking in C++***（Bruce Eckel / Prentice Hall）一书第 9 章：Pointers & references。

注意：本小节对于 copy constructor 的合成条件，只是一个概略性的介绍。***Inside The C++ Object Model***（Lippman/Addison Wesley/1996）一书第２章对此有非常详细的描述与归纳。

#### 35

多型與虛擬（*Polymorphism in C++*）

## Class Members

在 class 的宣告里头，真正有用的两样东西（其实也就只有这两样东西）是 data members 和 member functions：

* Data members 表示根据这个 class 所产生的 object（ 实体） 里头会有些什么东西，它事实上也是占据 object 存储器空间的唯一物质（ 除非引入虚拟机制）。通常， 为了资料的封装性， 我们把 data members 宣告为 private 或 protected。
* Member functions 是用来处理 data members 的函式。通常，为了界面的开放性， 我们把 member functions 设计为 public。

Data Members（资料成员）

Data members 的宣告和一般 non-class 的变量宣告形式一样。以 *CPoint* 为例：

#0001 class CPoint { #0002 public:

#0003 CPoint( float x = 0.0 ) : \_x( x ) { } #0004

#0005 float x() { return \_x; }

#0006 void x( float xval ) { \_x = xval; } #0007 // ...

#0008 protected:

#0009 float \_x; // data member

#0010 };

L9 的 *\_x* 就是一个 data members。

如果 data member 本身不是个一般变量，而是一个 class object，这种情况比较特殊，称为 composition（组合），而这种 objects 被称为 embedded objects 或object member。Composition 被用来描述 has a 的关系（例如汽车「有一个」引擎），inheritance（继承）用来描述 is a kind of 的关系（例如汽车「是一种」交通工具）。本章稍后我会分别描述composition 和 inheritance 两种情况。

**36**

第１章 C++ Classes 的語法和語意

### Member Functions（成员函式）

Member functions 用来处理 class data members。在资料封装的情况下，member functions 可以说就是一个 class（或说一个 object）的对外界面。这些 functions 并不占用 objects 的存储器空间，它被编译器处理过后，以完全等同于一般的global functions 的身份出现，独立于任何 object 之外。同一份函式码如何能够处理根据同一个 class 产生出来的不同 objects 的 data members 呢？关键在于有一个this 指标隐藏在member functions 的参数列之中，也隐藏在member functions 函式码对于 data members 的处理动作上，是编译器自动为我们加上去的。我将于第

２章 p.100 再谈这个主题。

class

instantiation

serve

member functions implementation code

object 1

object 5

object 4

object 3

object 2

data members

根据不同的 this 指标，指向不同的 objects 实体

**37**

多型与虚拟（*Polymorphism in C++*）

## 如何取用 Class Members

要取得 class members，如果是在 class scope 之内，直接使用其名称即可。但如果是在class scope 之外，你有三种方式可以办到（不管是对data 或对functions）：

1. 透过 dot operator（.）， 例如：

CPoint aPoint; // 产生一个 CPoint object，名为 aPoint aPoint.x(); // 唤起 CPoint::x()

aPoint.\_x = 7.0; // 使用 CPoint 的 data member

1. 透过 arrow operator（->）， 例如：

CPoint\* pPoint = new CPoint; // 产生一个 CPoint object，由 pPoint 指向它

pPoint->x(); // 唤起 CPoint::x()

pPoint->\_x = 7.0; // 使用 CPoint 的 data member

1. 透过 scope resolution operator（::）， 例如：

CPoint::foo(); // foo() must be a static member function CPoint::ratio = 0.2; // ratio must be a static data member

第三种作法并没有透过任何一个 object，显示这些 members 应该超越 objects 层次， 属于 classes 层次， 也就是所谓的 static members。如果不是这样， *CPoint::foo()* 函式呼叫动作会引发错误讯息如下：

error C2352: 'CPoint::foo' : illegal call of non-static member function

而 *CPoint::ratio* 资料存取动作会引发错误讯息如下（有点奇怪）：

error C2597: 'CPoint::ratio' : illegal reference to data member in a static member function

稍后我会介绍所谓的 static members。

**38**

第１章 C++ Classes 的語法和語意

如果函式呼叫时只指定 scope resolution operator（::），却没有指定 class 名称， 像这样：

::GetVersion();

表示这个函式不属于任何 classes 所有，是个 global function；通常也就是系统提供的 API 函式 – 如果你尽量不设计global 函式的话。虽然不使用 :: 仍然有可能达到呼叫 global 函式的效果，但我建议你明白地使用它，程序码会比较清晰。

#### 39

多型與虛擬（*Polymorphism in C++*）

## 特殊的 Static Members（静态成员）

Classes 中可以使用 static 关键字，使 members 成为静态。

静态的意思是，这份 members 属于 class 而非 objects 所有。听起来有点奇怪， 因为 class 只是属性，object 才是实体。很难一言以敝之，让我将 members 细分为 data 和 function 来讨论。

本书第５ 章范例程序中大量运用了 static members

当你看完以下的剖析，你会认为 static data members 和 global variables 很像，static member functions 和 global functions 很像。一点没错，从功能而言，global 可以取代 static。不过，以物件导向的观点来看，这些 members 如果在性质上与 class 息息相关，还是设计为 class 的 static members 为佳，因为在物件导向程序设计中，我们希望 global 的东西愈少愈好。

Static Data Members（静态资料成员）

##### SavingAccount object

m\_total

m\_addr

m\_name

SavingAccount ::m\_rate

##### （独立于 object 之外）

虽说 data 属于 objects 实体所有，但程序设计中有些资料的确希望由各 objects

共享，所以提升到 class 层级比较理想。例如，我们设计一个金融储户的 class：

class SavingAccount

{

private:

char m\_name[40]; // 储户姓名char m\_addr[60]; // 储户地址double m\_total; // 储簿金额double m\_rate; // 利率

...

};

这家行库采用浮动利率，每个账户的利息都是根据当天的挂牌利率来计算。这时候 *m\_rate* 就不适合成为每份 objects 的一笔资料了，否则每天一开市，光把所有账户内容叫出来，修改 *m\_rate* 的值，就花了不少时间。*m\_rate* 应该独立于各objects 之外，成为此一 class 独一无二的资料。怎么做？在 *m\_rate* 前面加上

**40**

第１章 C++ Classes 的語法和語意

static 修饰词即可：

class SavingAccount

{

private:

char m\_name[40]; // 储户姓名

char m\_addr[60]; // 储户地址

double m\_total; // 储簿金额

static double m\_rate; // 利率（注意，这只是宣告，不是定义，

... // 尚未配有存储器）

};

Static data members 不属于 object 的一部份，而是 class 的一部份，所以程序可以在还没有诞生任何 object 的时候就处理 static data members。但首先，你必须定义它（为它配置存储器）。

你应该在 .CPP 档中且为 class 以外的任何区域设定其初值；例如在 *main()* 之中，或 global functions 中，或任何函式之外：

double SavingAccount::m\_rate = 0.0075; // 定义并设初值

void main() // 不指定初值也可以

{

...

}

请注意用辞上的严谨度：上述「定义」动作常被误以为是「初始化」动作。如果只是单纯的初值设定动作，应该可以安排在 class constructor 中，但上一行绝对无法如此。此外，你也不应该把上述的定义安置于 .H 档中，因为这么一来它可能会被含入许多 .CPP 档，于是就重复定义了（会发生联结错误）。

上述动作有没有考虑 *m\_rate* 是笔 private 资料呢？没有关系，定义 static data members，不受任何封装层级的约束。请注意上述动作也指出 static data member 的型别，因为这是一个定义（definition）动作，不是一个数值指定（assignment） 动作。如果你没有做这个动作，*m\_rate* 就没有获得存储器，会发生联结错误：

error LNK2001: unresolved external symbol "private: static double SavingAccount::m\_rate"(?m\_rate@SavingAccount@@2HA)

#### 41

多型与虚拟（*Polymorphism in C++*）

下面是存取static data members 的一种方式。注意，此时还没有任何object 存在：

void main()

{

SavingAccount::m\_rate = 0.0072; // 此行欲成立，m\_rate 需改为 public

}

第二种方式是产生一个 object 后，透过 object 来处理 static data members：

void main()

{

SavingAccount myAccount;

myAccount.m\_rate = 0.0072; // 此行欲成立，m\_rate 需改为 public

}

请注意，static data members 并不是因为 *myAccount* object 的实现而才得以实现， 它本来就存在（你可以想象它是一个 global 变量）。因此，第一种处理方式在语法上比较不会给人错误的印象。

### Static Member Functions（静态成员函式）

既然 static data members 是超乎 object 之外存在，一般的 member functions 能否处理它？要知道，member function 得经由某个 object 才呼叫得到，该 object 位址则成为所谓的 *this* 指标（第２章介绍）！但由于 static data member 并不需要靠 *this* 指标的导引来决定其位址，所以其实任何函式（包括 global 函式），只要 access level 允许，都可以处理 static data member。

但「member function 得经由某个 object 才呼叫得到」这一事实毕竟还是带来某些影响！如果你在尚未产生任何 object 之前，就希望透过某个 member function 来更改前例的 static *m\_rate*，而它又是 private（以至于不能被 global 函式处理）， 那么你就一定得写一个 static member function 了：

#0001 class SavingAccount

#0002 {

#0003 private:

**42**

第１章 C++ Classes 的语法和语意

|  |  |  |
| --- | --- | --- |
| #0004 | char m\_name[40]; | // 储户姓名 |
| #0005 | char m\_addr[60]; | // 储户地址 |
| #0006 | double m\_total; | // 储簿金额 |
| #0007 | static double m\_rate; | // 利率 |
| #0008  #0009 | // ...  public: |  |

#0010 static void setRate(double newRate) { m\_rate = newRate; } #0011 // ...

#0012 };

#0013

#0014 double SavingAccount::m\_rate = 0.0075; // 设立初值

#0015

#0016 void main()

#0017 {

#0018 SavingAccount myAccount;

#0019

#0020 // 可以透过 object 呼叫 static member functions。

#0021 myAccount.setRate(0.0072); #0022 // 也可以直接呼叫 static member function 。

#0023 SavingAccount::setRate(0.0072);

#0024 }

由于 static member functions 不需要任何 object 的存在就可以被唤起，所以编译器不会为它暗加一个 *this* 指标。也因为如此，static member functions 无法处理任何 non-static data members，因为 member functions 之所以能够以单一一份函式实作码处理各个 objects 的资料而不紊乱，完全靠的是 *this* 指标14。

「Static member functions 没有 *this* 参数」这一性质，正是我们准备 system callback functions（被作业系统呼叫的函式）所必须（我曾经在 深入浅出 MFC 2/e 第６章示范过）。

1. 相关讨论，请参考本书第２章 p.100。

#### 43

多型与虚拟（*Polymorphism in C++*）

### Static Object Members（静态物件成员）

Object members 应该被视为和一般的 data members 一样地位，只不过它拥有的是class 型别，而一般data members 拥有的是内建型别。当它冠上 static，表示它所「寄生」之 class 一旦宣告完成（尚未产生任何 object），并且对 static members 做了初始化动作（其实就是实体配置），这个 static object member 就已经在存储器中有了实体。例如：

class Point3d { public:

// ...

public: // (protected: is better)

static Point3d origin; float x, y, z;

};

Point3d Point3d::origin; // 别忘了这个动作（其实就是实体配置）

void main()

{

奇怪的写法 

没有object，但 *origin*

已配有存储器 

printf("&Point3d::x = %p\n", &Point3d::x); // 00000004 printf("&Point3d::y = %p\n", &Point3d::y); // 00000008 printf("&Point3d::z = %p\n", &Point3d::z); // 0000000C

printf("&Point3d::origin = %p\n", &Point3d::origin); // 0040A450 printf("&Point3d::origin.x = %p\n", &Point3d::origin.x); // 0040A454 printf("&Point3d::origin.y = %p\n", &Point3d::origin.y); // 0040A458 printf("&Point3d::origin.z = %p\n", &Point3d::origin.z); // 0040A45C

|  |  |  |
| --- | --- | --- |
|  | Point3d pt3d; |  |
| *pt3d* 配有存储器  | cout << "&pt3d.x = " << &pt3d.x << endl;  cout << "&pt3d.y = " << &pt3d.y << endl; | // 0x0063FDEC  // 0x0063FDF0 |
|  | cout << "&pt3d.z = " << &pt3d.z << endl; | // 0x0063FDF4 |
| } |  |  |

你一定会对于上面第一组的写法和输出结果感到奇怪，它看起来象是 offset 之类的东西。那其实就是所谓的 pointer to data member。我将在第２章p.116 解释它。

**44**

第１章 C++ Classes 的語法和語意

### Static Objects（静态物件）

static 关键字也可以应用在 objects 身上。像这样：

static CPoint aPoint(3.6);

*aPoint* 于是就成为一个 static object，这个 object 的生命将持续到整个程序结束为止 – 不管这是个static global object15，或是个static local object。Static objects 的生命周期，可以从稍早 p.21「Objects 的生命」一节范例程序中验证。

## Inheritance（继承）

到目前为止，我所谈的都侷限在单一 classes 的情况。C++ 有一个很好的性质称为 inheritance（继承），就是宣告一个 class（称为 derived class），把另一个（或一些）class(es)（称为 base classes）的所有内容（包括 data members 和 member functions）统统继承下来（有无存取资格则是另一回事）。如果直接继承自一个base class，称为单一继承；如果直接继承自一个以上的classes，称为多重继承。语法如下（以下是三层单一继承）：

#0001 class CPoint { #0002 public:

#0003 CPoint( float x = 0.0 ) : \_x( x ) { } #0004

#0005 float x() { return \_x; }

#0006 void x( float xval ) { \_x = xval; } #0007 // ...

#0008 protected:

#0009 float \_x;

#0010 };

#0011 class CPoint2d : public CPoint { #0012 public:

1. 对一个 global object 加上 static 修饰词，主要意义在设定「可被看见的范围」只限于该档案本身。

**45**

多型與虛擬（*Polymorphism in C++*）

#0013 CPoint2d( float x = 0.0, float y = 0.0 ) #0014 : CPoint( x ), \_y( y ) { }

CPoint3d

CPoint2d

CPoint

#0015

#0016 float y() { return \_y; }

#0017 void y( float yval ) { \_y = yval; } #0018 // ...

#0019 protected:

#0020 float \_y;

#0021 };

#0022

#0023 class CPoint3d : public CPoint2d { #0024 public:

#0025 CPoint3d( float x = 0.0, float y = 0.0, float z = 0.0 ) #0026 : CPoint2d( x, y ), \_z( z ) { }

#0027

#0028 float z() { return \_z; }

#0029 void z( float zval ) { \_z = zval; } #0030 // ...

#0031 protected:

#0032 float \_z;

#0033 };

然后，我们就可以这样使用这些 classes：

CPoint3d aPoint3d(1.1, 2.2, 3.3);

cout << "x = " << aPoint3d.x() << endl; // 1.1 cout << "y = " << aPoint3d.y() << endl; // 2.2 cout << "z = " << aPoint3d.z() << endl; // 3.3

CPoint3d\* pPoint3d = new CPoint3d(4.4, 5.5, 6.6);

pPoint3d->x(4.1); // x() 是 CPoint's member function pPoint3d->y(5.2); // y() 是 CPoint2d's member function pPoint3d->z(6.3); // z() 是 CPoint3d's member function

cout << sizeof(CPoint) << endl; // 4 cout << sizeof(CPoint2d) << endl; // 8 cout << sizeof(CPoint3d) << endl; // 12

我们在 *CPoint3d* object 中使用继承而来的 *x()* 和 *y()* 函式，这两个函式将存取继承而来的 \_*x* 和 *\_y* 资料。

继承的最大用意，在于表现物件世界中 is a kind of（或说is a）的关系。以生物分类用的「界门纲目科属种」分类法为例，土拨鼠属于动物界、脊索动物门、

#### 46

第１章 C++ Classes 的語法和語意

哺乳纲、啮齿目、松鼠科，因此你可以说土拨鼠是一种松鼠，也是一种啮齿动物， 也是一种哺乳动物，也是一种脊索动物，并且（当然）是一种动物。这种 is a kind of 的关系，最适合以物件导向性质中的 inheritance 来表现。

务请把这个观念放在心上：derived class object 身上有着所有的 base classes subobjects。我们可以说一只土拨鼠身上有着动物、脊索动物、哺乳动物、啮齿动物、松鼠的性质（base class's subobject）。Subobject 当然不能够抽取出来成为一个独立的 objects，就好像土拨鼠身上不能抽出一只松鼠一样。但是，土拨鼠身上具有松鼠的特征，是非常合理的说法。

以本例 *CPoint3d* 而言，其 object 可以图示如下：

处理 CPoint subobject

8

bytes

4

bytes

12

bytes

CPoint subobject

CPoint member functions

处理 CPoint2d subobject

CPoint2d subobject

（涵盖 CPoint subobject）

CPoint3d object

CPoint2d member functions

处理 CPoint3d object

CPoint3d member functions

（涵盖 CPoint2d subobject

和 CPoint subobject）

Base class subobject 的讨论，涉及 C++ 物件模型以及 Polymorphism 的运用，我将在第２章和第４章更深入地讨论。

C++ 语言支援三种继承（当然，都是为了模拟真实世界）：

* 单一继承（ Single Inheritance）： direct base class 只有一个。
* 多重继承（ Multiple Inheritance）： direct base classes 不只一个。
* 虚拟继承（ Virtual Inheritance）： 多重继承之下， 让共同的「祖父级」base class object 能够被共享， 不至于浪费存储器空间。

#### 47

多型與虛擬（*Polymorphism in C++*）

单一继承（Single Inheritance）

所谓单一继承，就是每一个 class 的 directly base class 只能有一个。继承的层级数目并没有限制。上例的 *CPoint*、*CPoint2d*、*CPoint3d* 一脉相传就是一种单一继承：

CPoint3d

CPoint2d

CPoint

class CPoint { ... };

class CPoint2d : public CPoint { ... }; class CPoint3d : public CPoint2d { ... };

市面上有名的Microsoft MFC 就是一套单一继承的 C++ classes library（因它具备某些特殊性质，所以亦称为 application framework）。

### 多重继承（Multiple Inheritance）

所谓多重继承，是指一个 class 的 directly base classes 可以不只一个。多重继承的语法如下：

class CPoint { ... };

CVertex3d

CVertex

CPoint3d

CPoint2d

CPoint

class CPoint2d : public CPoint { ... }; class CPoint3d : public CPoint2d { ... }; class CVertex { ... };

class CVertex3d : public CPoint3d, public CVertex { ... };

上例的 *CVertex3d* 就是多重继承自 *CPoint3d* 和 *CVertex*。其中 *CPoint3d* 称为leftmost base class（最左端基础类别）。「最左端」带来什么影响？噢，多重继承的 base classes 排列次序会影响 object 的物件布局。我将在第２章p.128 再详细探讨这个问题。

市面上有名的 Borland OWL 就是一套多重继承的 C++ classes library（因它具备某些特殊性质，所以亦称为 application framework）。

**48**

第１章 C++ Classes 的語法和語意

### 虚拟继承（Virtual Inheritance）

所谓虚拟继承，就是在 class head 中，于 base class 的前方加上 virtual 关键字。虚拟继承的语法可以众所周知的 iostream library 为例，如下所示：

iostream

ostream

istream

ios

class ios { ... }; class istream : public

ios { ... };

virtual

class ostream : public virtual ios { ... };

class iostream : public istream, public ostream { ... };

我曾强调：derived class object 身上有着 base class subobjects。从语意来看，本例的 *iostream* object 中应该有一个 *istream* subobject，和一个 *ostream* subobject。两者之中又各自应该有一个 *ios* subobject。那么 *iostream* 的继承体系就应该变成这样了囉：

iostream

ostream

istream

ios

ios

如果上述程序码中没有 virtual 关键字，上图的确是正确的。但这么一来两个 *ios* subobject 带来困扰与浪费。如果加上 virtual 关键字，就只会有一个 *ios* subobject 产生。

虚拟继承的底层实作观念并不是那么直觉易懂，要在 C++ 编译器中支援虚拟继承，困难度颇高。我将在第２章 p.142 再详细探讨这个问题。

**49**

多型與虛擬（*Polymorphism in C++*）

继承体系下的物件建构（由内而外，由上而下）

假设我有 *CPoint*、*CPoint2d*、*CPoint3d* 的继承体系如下：

#0001 #include <iostream.h>

#0002

#0003 class CPoint

#0004 {

#0005 public:

#0006 CPoint() { cout << "CPoint's constructor." << endl; } #0007 ~CPoint() { cout << "CPoint's destructor." << endl; } #0008

CPoint3d

CPoint2d

CPoint

#0009 private:

#0010 float \_x;

#0011 };

#0012

#0013 class CPoint2d : public CPoint #0014 {

#0015 public:

#0016 CPoint2d() { cout << "CPoint2d's constructor." << endl; } #0017 ~CPoint2d() { cout << "CPoint2d's destructor." << endl; } #0018

#0019 private:

#0020 float \_y;

#0021 };

#0022

#0023 class CPoint3d : public CPoint2d #0024 {

#0025 public:

#0026 CPoint3d() { cout << "CPoint3d's constructor." << endl; } #0027 ~CPoint3d() { cout << "CPoint3d's destructor." << endl; } #0028

#0029 private:

#0030 float \_z;

#0031 };

这三个 classes 的 objects 内容如图所示：

#### 50

第１章 C++ Classes 的語法和語意

8

bytes

4

bytes

12

bytes

|  |
| --- |
| CPoint subobject |
| CPoint2d subobject |
| CPoint3d object |

由于「一个 derived class object 中有其 base class subobject 存在」，如果我产生一个 *CPoint3d*，必须有由内（base class）而外（derived class）的建构次序，才合理。因此，当我这么做：

CPoint3d\* p3d = new CPoint3d;

我将获得这样的结果：

CPoint's constructor. // 先建构 CPoint subobject CPoint2d's constructor. // 再建构 CPoint2d subobject CPoint3d's constructor. // 最后建构 CPoint3d object

继承体系下的物件解构（由外而内，由下而上）

继承上一节的讨论，如果我删除一个 *CPoint3d*，必须有由外（derived class）而内

（base class）的解构次序，才合理。因此，当我这么做：

delete p3d; // CPoint3d\* p3d = new CPoint3d;

我将获得这样的结果：

CPoint3d's destructor. // 先解构 CPoint3d 外围部份CPoint2d's destructor. // 再解构 CPoint2d 部份CPoint's destructor. // 最后解构 CPoint 部份

另有一种特殊的所谓 virtual destructor，稍后进行到 virtual function 时再讨论。

#### 51

多型与虚拟（*Polymorphism in C++*）

## Initialization List

请注意以下继承体系中各个 classes 的 constructors 写法：

#0001 class CPoint { #0002 public:

#0003 CPoint( float x = 0.0 )

#0004 : \_x( x ) { }

#0005 protected:

CPoint3d

CPoint2d

CPoint

#0006 float \_x;

#0007 };

#0008

#0009 class CPoint2d : public CPoint { #0010 public:

#0011 CPoint2d( float x = 0.0, float y = 0.0 )

#0012 : CPoint( x ), \_y( y ) { }

#0013 CPoint2d( ) { \_y = 0.0; } // default constructor #0014 protected:

#0015 float \_y;

#0016 };

#0017

#0018 class CPoint3d : public CPoint2d { #0019 public:

#0020 CPoint3d( float x = 0.0, float y = 0.0, float z = 0.0 )

#0021 : CPoint2d( x, y ), \_z( z ) { }

#0022 protected:

#0023 float \_z;

#0024 };

在 constructor 宣告之后有一个 : 符号，后面紧跟着一个（以上）的函式呼叫动作，这一行就是所谓的 initialization list。它的作用是在进入 constructor 主体动作之前，先唤起其中所列的函式。例如上面的：

* L4 ： 表示执行 *CPoint::CPoint(x)* 之前， 先执行 *\_x(x)*。16

1. 本章稍早我曾提过，以退化的角度看，语言内建型别如 int, float, long 等等也都是一种class。因为变量 *\_x* 的型别是 float，所以 \_x(x) 的意思是启动 "float class" 的 constructor，也就是把 *\_x* 的初值设为 x。

**52**

第１章 C++ Classes 的語法和語意

* L12： 表示执行 *CPoint2d::CPoint2d(x, y)* 之前， 先执行 *CPoint(x)* 和

*\_y(y)*。

* L21：表示执行 *CPoint3d::CPoint3d(x, y, z)* 之前，先执行 *CPoint2d(x, y)* 和 *\_z(z)*。

因此当我产生一个 *CPoint3d* object 如下：

CPoint3d aPoint3d(1.1, 2.2, 3.3);

会有以下六个动作依序被唤起：

\_x(1.1); // 相当于 \_x = 1.1;

CPoint::CPoint(1.1); // 本例没做什么事

\_y(2.2); // 相当于 \_y = 2.2;

CPoint2d::CPoint2d(1.1, 2.2); // 本例没做什么事

\_z(3.3); // 相当于 \_z = 3.3;

CPoint3d::CPoint3d(1.1, 2.2, 3.3); // 本例没做什么事

你可能会问，既然上一节提到继承体系中的建构方式是由内而外，由上而下，那么这里产生一个 *CPoint3d* object ， 必然会唤起 *CPoint2d* 和 *CPoint* 的constructors，而所有初始化动作都可以在其中完成，initialization list 的出现会不会因此显得多此一举？

做个实验就知道了。把上一段码的 *CPoint3d* constructor（L20）改为这样试试：

CPoint3d( float x = 0.0, float y = 0.0, float z = 0.0 ) { \_z = z; }

其中没有指定 initialization list。结果竟然无法通过编译：

error C2668: 'CPoint2d::CPoint2d' : ambiguous call to overloaded function

也就是说，当编译器根据继承体系往上一层呼叫 base class constructor 时，发现

*CPoint2d* 有两个constructors（L11 和 L13），而它不知道该呼叫哪一个。

#### 53

多型與虛擬（*Polymorphism in C++*）

这就是 initialization list 最明显的存在价值。如果本例的 *CPoint2d* 只有一个

constructor（不管是 L11 那一个或 L13 那一个），像这样：

class CPoint2d : public CPoint { public:

CPoint2d( ) { \_y = 0.0; } // default constructor protected:

float \_y;

};

或这样：

class CPoint2d : public CPoint { public:

CPoint2d( float x = 0.0, float y = 0.0 )

: CPoint( x ), \_y( y ) { }

protected: float \_y;

};

而 *CPoint3d* constructor 中没有列出 initialization list，像这样：

class CPoint3d : public CPoint2d { public:

CPoint3d( float x = 0.0, float y = 0.0, float z = 0.0 ) { \_z = z; } protected:

float \_z;

};

那么并不会出现前述的编译错误。

以上的讨论是针对 base class 的建构。同理对于 member class 也是一样。如果member class 有一个以上的 constructors，那么内含 embedded object 的那个 class 就必须在其 constructor 中指定 initialization list，否则一样会出现 "ambiguous call to overloaded function" 的编译错误。

Initialization list 到底会在编译器底层实作中发生什么影响呢？编译器会以「适当的次序」将 initialization list 中指定的 members 呼叫动作安插到 constructor 之内，并置于任何 user code 之前。下面这张图可以表现出编译器的插码结果：

#### 54

第１章 C++ Classes 的語法和語意

source snippet

CPoint3d( float x = 0.0, float y = 0.0, float z = 0.0 )

: CPoint2d( x, y ), \_z( z ) { /\* if any user code... \*/ }

被编译器扩充为

CPoint3d::CPoint3d( float x = 0.0, float y = 0.0, float z = 0.0 )

{

CPoint2d::CPoint2d( x, y );

source snippet

\_z( z );

/\* if any user code... \*/

呼叫

}

被编译器扩充为

d( float x = 0.0, float y = 0.0 )

nt( x ), \_y( y ) { /\* if any user code... \*/ }

CPoint2

: CPoi

|  |
| --- |
| CPoint::CPoint( float x = 0.0 )  {  \_x( x );  /\* if any user code... \*/  } |

有一些微妙的地方必须注意。编译器安插在 constructor 中的 members 宣告动作是以 members 在 class 中的宣告次序为根据，而不是以 initialization list 中的排序次序为根据。如果两者在外观上错乱，很容易引起程序设计时的一些困扰或疏失。例如：

source snippet

CPoint( float x = 0.0 )

: \_x( x ) { /\* if any user code... \*/ }

呼叫

被编译器扩充为

CPoint2d::CPoint2d( float x = 0.0, float y = 0.0 )

{

CPoint::CPoint( x );

\_y( y );

/\* if any user code... \*/

}

class X { public:

X(int val) : m\_data2(val), m\_data1(m\_data2) { } protected:

int m\_data1; int m\_data2;

};

我们很容易误以为在 *X* constructor 中是以 *val* 设定 *m\_data2*，再将 *m\_data2* 设

#### 55

多型與虛擬（*Polymorphism in C++*）

定给 *m\_data1*。但根据两个 data members 的宣告次序，实际发生的动作却是：

X::X(int val)

{

m\_data1(m\_data2); // 此时 m\_data2 尚未有初值，糟糕

m\_data2(val);

}

于是，当我们产生一个 *X* object：

X x(3);

其 data members 的内容可能成为这样：

x.m\_data1 = -2124198216 // 这不是我们希望的

x.m\_data2 = 3

一个比较好的做法是，把 class *X* 重新设计如下：

class X { public:

X(int val) : m\_data2(val) { m\_data1 = m\_data2; } protected:

int m\_data1; int m\_data2;

};

#### 56

第１章 C++ Classes 的語法和語意

## Virtual Functions（虚拟函式）

Virtual 这个关键字在多型性质中扮演一个绝对重要的角色。只要在 member functions 宣告之前方加上 virtual 关键字， 它就成为一个 virtual member function，简称为 virtual functions。

任何一个 class 如果拥有 virtual functions，就可获得 C++ 编译器的虚拟机制

（virtual mechanism）的服务。这个 class 的所有 derived classes，也都会继承这些 virtual functions。当 derived class 决定重新定义 virtual functions（而不再沿用base class 的定义），这动作称为 override（改写），这时候我们需要在 derived class 中重新宣告 virtual functions 并重新定义其实作码。重新宣告时不需特别再加上 virtual 字眼 -- 可说是「一代为 virtual，累世为 virtual」。

不过，我建议不厌其烦地在 derived classes 中为每一个你想要重新定义的 virtual functions 都加上 virtual 关键字，这对于将来检视程序码有帮助。

Virtual functions 的妙用，在Inheritance（继承）和Polymorphism（多型）情况下才会发挥得淋漓尽致，我们到第４章再来好好地研究这个主题。至于其实作方法

（编译器层面的技术），将在第２章介绍。以下我先以一个简单的例子，展示 virtual functions 带来的影响：

#0001 #include <iostream.h>

CRect

CShape

#0002

#0003 class CShape

#0004 {

#0005 public:

#0006 virtual void display() { cout << "CShape" << endl; }; #0007 protected:

#0008 int m\_color;

#0009 };

#0010

#0011 class CRect : public CShape #0012 {

#0013 public:

**57**

多型与虚拟（*Polymorphism in C++*）

#0014 void display() { cout << "CRect" << endl; }; #0015 };

#0016

#0017 void main()

#0018 {

#0019 CShape\* pShape1 = new CShape;

#0020 pShape1->display(); // output: CShape #0021 CShape\* pShape2 = new CRect; // polymorphic #0022 pShape2->display(); // output : CRect #0023 }

执行结果很妙。我们以相同型别的指标（*CShape*\*）呼叫相同的函式（至少表面上看来相同，因为名称都是 *display()*），结果却是：

CShape CRect

显然所唤起的 virtual function 因为指标真正指向的 object 不同而不同。

正是如此！ 以 base type pointer 指向 derived type object ， 就是所谓的polymorphism（多型），搭配 virtual functions，可以有非常大的威力（详见第３章）。但请注意，这种威力唯有透过指标的形式（像上面那样），才得呈现。

拥有 virtual function(s) 的 class，我们给予它一个特别的名称：polymorphic class。

#### 58

第１章 C++ Classes 的語法和語意

Virtual Destructor

我们已经知道了 polymorphism、virtual function、destructor 等观念，现在我可以来谈一谈所谓的 virtual destructor 了。

C++ Standard 明确规定，当一个 object 被摧毁（destroy）时，一定要由外而内地解构（这里所谓的内，是指 subobjects）。先前我已经在 p.51「继承体系下的物件解构」一节中做了展示。

现在我继续延用出现在p.50「继承体系下的物件建构」一节的 class 体系（*CPoint*、

CPoint3d

CPoint2d

CPoint

*CPoint2d*、*CPoint3d*），并在程序中这么做：

CPoint3d\* p3d = new CPoint3d;

delete p3d; // (1)

CPoint\* pt = new CPoint3d; // polymorphically delete pt; // (2)

第一组动作已在先前「继承体系下的物件建构、解构」中示范过，它的输出是：

CPoint's constructor. // 由内而外建构... CPoint2d's constructor.

CPoint3d's constructor.

CPoint3d's destructor. // 由外而内解构... CPoint2d's destructor.

CPoint's destructor.

很好。第二组动作的输出如下：

CPoint's constructor. // 由内而外建构... CPoint2d's constructor.

CPoint3d's constructor.

CPoint's destructor. // 不妙！

由于 *CPoint::~CPoint()* 不是 virtual function，导致编译器无法从所谓的虚拟机制

（ 第２ 章详论） 中找出外围的 destructors ， 至使 *CPoint2d::~CPoint2d()* 和

*CPoint3d::~CPoint3d()* 都没有被呼叫到，而它们原本（通常）用来让程序员完成

**59**

多型與虛擬（*Polymorphism in C++*）

各种善后清理（clean up）工作（虽然本例并未如此）。

改善方法是，把 destructor 宣告为 virtual，例如：

class CPoint

{

public:

virtual ~CPoint() { ... }

...

};

那么本例的 (2) *delete* 动作就如所期望了：

CPoint's constructor. // 由内而外建构... CPoint2d's constructor.

CPoint3d's constructor.

CPoint3d's destructor. // 由外而内解构... CPoint2d's destructor.

CPoint's destructor.

第６章谈到介绍 COM 的基础建设时，我还会再提到 virtual destructor。届时别忘了回来看看这一节。

#### 60

第１章 C++ Classes 的語法和語意

## Composition（组合）

有两个重要的模型，可以用来描述绝大部份的 objects 与 objects（或说 classes 与

classes）之间的关系。一个是 is a kind of，一个是 has a。前者以 inheritance

（继承）来描述极为合适，后者就需要以 composition（组合）来描述。

C++ 并没有特别为 composition 设计什么关键字或特殊语法。所谓composition ， 就是在一个 class 中拥有以另一个 class 所描述的 object members。我们可以说汽车「是一种」交通工具，我们也可以说汽车「有一个」引擎（虽然引擎构成了汽车的一部份，但我们不能够说汽车「是一种」引擎）：

class **CCar** : public **CVehicle** // 汽车是一种交通工具

{

protected:

**CEngine** m\_engine; // 汽车有一个引擎

...

};

当 composition 发生，上述的 *CCar* object 之中有一个完完整整、货真价实的embedded object（*CEngine* object），而不只是一个空有属性却抽离不出实体的所谓 subobject。

Composition 发生时，外围的 class 在设计 constructor 时需特别留神。请参考稍早 p.28 的「Constructor & Destructor」一节。

**61**

多型与虚拟（*Polymorphism in C++*）

## Inline Functions

定义一个小小的函式如下：

int max(int v1, int v2)

{

return (v1 < v2 ? v2 : v1)

}

把这么稀少的动作定义为一个函式，有数个优点：

1. 程序之中呼叫 *max()*， 要比直接做大小比较来得清爽干净。
2. 如果程序中处处需要做最大值比较， 直接把上述的比较动作放在每一个需要的地方， 对程序员而言将是一种折磨， 而且容易出错。万一函式中的动作稍微多一些， 那就不只是折磨， 简直是一场灾难了。
3. 函式有型别检验的能力， 比较容易检查出错误。

但是函式呼叫需要一些额外的成本：呼叫端需要一个 CALL 指令，之前还需要一小段码把 stack 备妥（内放参数以及函式回返位址）。函式回返前，亦需把 stack 恢复原状。这些在在都需要时间成本。

以 macro（巨集）来取代这种体积甚小的函式，是个不错的主意。虽然因此消除了函式呼叫的额外成本，但是 macro 也有缺点：

1. macro 只是一种「单纯的文字替代」，使用次数一多，程序的体积可能会暴涨。
2. macro 只是一种「单纯的文字替代」，所以没有什么型别检验。不小心的话甚至可能因为运算子的优先次序， 把传进去的参数弄混乱了（ 所以 macro 中对于参数的任何运算动作最好都加上括号）。

C++ 的 inline 可说是融合两者的发明。基本上你还是可以把 inline 函式视为「单

**62**

第１章 C++ Classes 的語法和語意

纯的替代」（所以你必须在每一个需要 inline 函式的程序档案中，备妥 inline 函式的定义），但C++ 提供给 inline 函式「参数检验」的好处。

Inline 函式只适合用在小型的、直率的、常被呼叫的程序单元。想要完成一个 inline

函式，有两个办法：

1. 直接把函式定义（ 实作码） 写到 class 宣告中， 那么它就自然地成为一个 inline member function。
2. 在函式的 implementation 前加上 inline 关键字。例如

**inline** void CShape::setColor(int color)

{

...

}

对编译器而言，inline 并非是一个强制条件。也就是说，你所使用的编译器不一定能实作出 inline 的要求，但它必须接受 inline 关键字（可能视而不见就是了）。为了避免 inline 函式在同一程序的不同档案中有不同的定义（这必然是因为程序员的疏忽），最好把 inline 函式的定义放在 .H 档中让 CPP 档含入，如此便万无一失了。

#### Inline function 和 virtual function

Virtual function 绝不可以是 inline，因为它的唤起系经由虚拟机制（也就是 vptr、vtbl 那套东西），不能够像 inline 的作法那样，将函式实作码以单纯的代换方式植入呼叫处（注：某些够聪明的编译器如 BCB 3，可将 virtual function 也做成inline，但仍保留一份正常的 virtual function body 以备需要）。

那么，先前我在p.57「Virtual Functions」那一节所展示的程序，直接把函式实作码写在 class 宣告之中，又会造成什么后果？别忘了，inline 是编译器的权利， 由它来判断可不可以让你 inlining。这种情况下，C++ 编译器不把它制作成 inline 函式就是了。

#### 63

多型与虚拟（*Polymorphism in C++*）

## Template

什么是 template？重要性如何？Kaare Christian 在 1994/01/25 的 ***PC Magazine***

上有一篇文章，把 template 说得生动又有趣，也点出了重点：

无性生殖并不只是存在于遗传工程，对程序员而言，它也是一个由来已久的动作。过去，我们只不过是以一个简单而基本的工具，也就是一个文字编辑器，剪剪贴贴地重制程序码。今天，C++ 提供给我们一个更好的繁殖方法：template。

复制一段既有的程序码，最平常的一个理由就是：为了改变资料型别。举个例子， 假设你写了一个绘图函式，使用型别为 long 的 x, y 坐标；突然之间你需要相同的算法，但坐标值改采 float。你当然可以使用一个文字编辑器把这段码拷贝一份，然后改变其中的资料型别。你甚至可以使用 C++ 的 overloaded functions，那么就可以继续使用相同的函式名称。Function Overloading 的确使我们有比较清爽的程序码，但你还是必须在程序的许多地方维护完全相同的算法。

C 语言对此问题的解决之道是：使用 macros。但 macros 有它自己的缺点，第一， 它只适用于简单的功能。第二个缺点比较严重：macros 不提供参数型别检验，这与 C++ 严格的型别检验格格不入。第三个缺点是：macro 并非 functions，程序中任何呼叫 macros 的地方都会被编译器前置处理器原原本本地插入 macros 所定义的那一整段码，而非只是一个 function call。因此你每使用一次 macros，你的执行档就会膨胀一些。

Templates 提供比较好的解决方案，它把「一般性的算法」和其「对资料型别的实际系结」区分开来。你可以先写出演算法的程序码，使用时才填入实际资料型别。C++ template 使「资料型别」也以参数的姿态出现。有了 template，我们可以拥有 macros「只写一次」的优点，以及 overloaded functions「型别检验」的优点。

**64**

第１章 C++ Classes 的語法和語意

Template 的好处其实不仅于此，你可以利用 template 设计出一般化（泛型）的算法，适用于「目前存在」以及「尚未被设计出来」的某种资料型别（某种 class type）。当然，如果要让新开发的 class types 能够适用于此泛型算法，那些classes 在设计时也必须配合某些事情。

稍后你就会从我所举的简单例子中看到这种伟大的弹性。

C++ template 有两种，一种针对 function，另一种针对 class。

Template Functions

假设我需要一个幂次方计算函式，名曰 *power()*。这个函式只接受正幂次方数，如果是负幂次方数，就让结果为 0。

对于 int 资料，*power()* 函式应该是这样：

#0001 int power(int base, int exponent) #0002 {

#0003 int result = base;

#0004 if (exponent == 0) return (int)1; #0005 if (exponent < 0) return (int)0; #0006 while (--exponent) result \*= base; #0007 return result;

#0008 }

对于 long 资料，*power()* 函式应该是这样：

#0001 long power(long base, int exponent) #0002 {

#0003 long result = base;

#0004 if (exponent == 0) return (long)1; #0005 if (exponent < 0) return (long)0; #0006 while (--exponent) result \*= base; #0007 return result;

#0008 }

对于 float 我们应该...，对于 complex 我们又应该...，噢，为什么不能够把资料

**65**

多型與虛擬（*Polymorphism in C++*）

型别也变成参数之一，在呼叫函式时加以指定呢？是的，这就是 template 的妙用，语法如下：

template <class T> T power(T base, int exponent);

写成两行或许比较清楚：

template <class **T**>

**T** power(**T** base, int exponent);

这样的函式宣告是以一个特殊的 template 字首开始，后面紧跟着一个参数列

（本例只有一个参数 T）。容易让人迷惑的是其中的 class 字眼，它并不一定表示 C++ 的 class，它可以只是一个普通的（内建的）资料型别（当然啦，以退化的角度看，内建型别也是一种class）。 <class T> 的意思是：T 是一种资料型别，此型别将在呼叫此函式时才由呼叫者指定。

下面就是 *power()* 的 template 版本。注意，传回值也必须吻合 template 函式的宣告：

#0001 template <class **T**>

#0002 **T** power(**T** base, int exponent) #0003 {

#0004 **T** result = base;

#0005 if (exponent == 0) return (**T**)1; #0006 if (exponent < 0) return (**T**)0; #0007 while (--exponent) result \*= base; #0008 return result;

#0009 }

下面是 template 函式的呼叫方法：

#0001 #include <iostream.h>

#0002 void main()

#0003 {

#0004 int i = power(5, 4);

#0005 long l = power(1000L, 3);

#0006 long double d = power((long double)1e5, 2); #0007

#0008 cout << "i= " << i << endl;

#### 66

第１章 C++ Classes 的語法和語意

|  |  |  |
| --- | --- | --- |
| #0009 |  | cout << "l= " << l << endl; |
| #0010 |  | cout << "d= " << d << endl; |
| #0011 | } |  |

执行结果如下：

i= 625

l= 1000000000 d= 1e+010

在第一次呼叫中我指定 T 为 int，在第二次呼叫中我指定 T 为 long。而在第三次呼叫中 T 又成了一个 long double。这些 template function 都可以应付。但如果函式呼叫时呼叫者自己混乱了资料型别，像这样：

int i = power(1000L, 4); // 基值是个 long，传回值却是个 int。错误！

编译时就会出错。

Template 函式的「型别参数」T 究竟可以适应多少种型别？我要说，任何内建

（build in）型别或使用者自定（user defined）型别（也就是 classes，包括 struct） 都可以。如果是个 class，此一 class 必须支援 *power()* 函式之中对于该型别（T） 的任何运算动作，否则编译器遇到那些运算时，不知道该怎么处理才好。

以 *power()* 为例，它对于型别为 T 的 *result* 变量和 *base* 变量有以下的运算：

|  |  |  |
| --- | --- | --- |
| (L4) | T result = base; | // assignment |
| (L5) | return (T)1; | // cast |
| (L6) | return (T)0; | // cast |
| (L7) | result \*= base; | // \*= |
| (L8) | return result; | // return (copy constructor, p.32) |

所有的 C++ 内建型别如 int 或 long 都支援上述运算动作。但如果你打算支援某个 C++ class，该 class 必须拥有可以支援上述运算动作的 member functions。

#### 67

多型與虛擬（*Polymorphism in C++*）

### Template Classes

除了 template functions，我们还可以建立 template classes，使它们能够神奇地操作任何型别的资料。下面这个例子，*CThree* 内含三个 data members，*Min()* 传回其中的最小值，*Max()* 传回其中的最大值：

#0001 template <class **T**> #0002 class CThree

#0003 {

#0004 public :

#0005 CThree(**T** t1, **T** t2, **T** t3); #0006 **T** Min();

#0007 **T** Max();

#0008 private:

#0009 **T** a, b, c; #0010 };

请你把 T 看成是熟悉的 int 或 float，语法就不至于太稀奇古怪了。以下是上述三个 member functions 的定义：

#0001 template <class T> #0002 T CThree<T>::Min()

#0003 {

#0004 T minab = a < b ? a : b; #0005 return minab < c ? minab : c; #0006 }

#0007

#0008 template <class T> #0009 T CThree<T>::Max()

#0010 {

#0011 T maxab = a < b ? b : a; #0012 return maxab < c ? c : maxab; #0013 }

#0014

#0015 template <class T>

#0016 CThree<T>::CThree(T t1, T t2, T t3) : #0017 a(t1), b(t2), c(t3)

#0018 {

#0019 return;

#0020 }

**68**

第１章 C++ Classes 的語法和語意

请注意，每一个 member function 都要在最前面加上 template <class T>，而且class 名称应该使用 *CThree<T>*，不是 *CThree*。

以下是 template class 的使用方式：

#0001 #include <iostream.h>

#0002 void main()

#0003 {

#0004 CThree<int> obj1(2, 5, 4); #0005 cout << obj1.Min() << endl; #0006 cout << obj1.Max() << endl; #0007

#0008 CThree<float> obj2(8.52, -6.75, 4.54); #0009 cout << obj2.Min() << endl;

#0010 cout << obj2.Max() << endl; #0011

#0012 CThree<long> obj3(646600L, 437847L, 364873L);

#0013 cout << obj3.Min() << endl; #0014 cout << obj3.Max() << endl; #0015 }

执行结果如下，一切正确：

2

5

-6.75

8.52

364873

646600

上一节我曾说过，当 class T 支援「template function 中对于 T 的所有运算动作」，T 才被视为对于该 template function 有效。此一限制在 template classes 亦属实。为了能够在使用 *CThree* 时指定以特殊的 T 型别，T 必须供应 copy constructor 以及 operator<，因为在 *Min()* 和 *Max()* 之中使用了这两个动作来操作 T class。

如果 template classes 是别人设计好的（例如一个 array），T 是我打算提供的（例如一个复数型别），我如何能够知道什么样的运算动作是 T 所必须供应的呢？ 唔，template classes 的文件中应该有所说明。如果没有，只有原始码才能揭露秘

#### 69

多型与虚拟（*Polymorphism in C++*）

密。C++ 内建型别如 int 和 float 等并不需要在意这份要求，因为所有的内建型别都支援所有的标准运算动作。

没有一位程序员在使用过 template 并享受过它的强悍威力后，不会热烈地爱上它。Template class 的弹性是如此地巨大，其底层实作机制相信会是一个吸引你的

兴趣的主题。***Inside the C++ Object Mode***（***l*** Stanly. Lippman / addison wesley / 1996）

一书第７章十分深入地谈到了 Template 的底层实作，以及一些细微的事项。现在让我们从使用层面先试着体会一下 C++ template classes 带来的利益，以下我将示范 C++ Standard Template Library（STL）和 MFC Collection classes 的用法。

以下试举一例，以 *CRect* 支援前述之 template class *CThree*：

class CRect

{

protected:

int x, y, width, height; public:

int area() { return (width \* height); }

CRect(int xval=0, int yval=0, int widthval=0, int heightval=0)

{ x=xval; y=yval; width=widthval; height=heightval; } CRect(const CRect& rect) // **copy constructor**

{ x=rect.x; y=rect.y; width=rect.width; height=rect.height; } bool **operator<**(CRect& rect)

{ return (area() < rect.area()); }

};

使用方式如下：

CRect rect1(0, 0, 10, 20); // area : 200

CRect rect2(0, 0, 15, 7); // area : 105

CRect rect3(0, 0, 11, 30); // area : 330

CThree<CRect> threerect(rect1, rect2, rect3);

cout << threerect.Min().area() << endl; // 105 cout << threerect.Max().area() << endl; // 330

#### 70

第１章 C++ Classes 的語法和語意

## Standard Template Library（ STL） or C++ Standard Library

C++ 提供一套标准的 library，称为 C++ standard library，完全以 template 完成， 所以又被称为 Standard Template Library（STL）。这套 library 专注于实现常用的资料结构（例如 array、list…）以及常用的算法（例如 push, pop, insert, delete, query, retrieval…）。

一般而言，STL 内含六个主要的零组件：

1. containers（ 泛型容器）
2. generic algorithms（ 泛型算法）
3. iterators（ 泛型指標）
4. function objects
5. adaptors
6. allocators

我将在这一节简单介绍前三个零组件，让大家领略一下 template 的伟大弹性。

如果你希望对 STL 有更多的了解，本书第一页推荐的两本书籍，***The C++ Programming Langauge*** 3/e 和 ***C++ Primer*** 3/e，都涵盖有这个主题。另有一本STL 专书也值得参考：***STL Tutorial and Reference Guide*** *- C++ Programming with the Standard Template Library* （David R. Musser & Atul Saini / Addison Wesley / 1996）

**71**

多型与虚拟（*Polymorphism in C++*）

STL Containers

STL 实现了一些常用的资料结构。这些结构用来收容许多资料，所以被称为

container（容器），主要分为两大类：

1. sequence container：内含有序而型别一致的元素。例如 vector（ 也就是 array） 和 list， 以及 deque。deque 的行为类似 vector， 但对于插入（ insertion） 和删除（ deletion） 第一个元素特别有效率。
2. associative container：此种资料结构对于查询（ query）或取出（ retrieval） 某一个元素的动作特别有效率。例如 map 和 set。所谓 map 是key/value 的成对组合： key 用来查询， value 内含真正的资料。电话簿和字典都适合以 map 来完成， 所以 map 又被称为 dictionary。

map 和 set 中的每一个 key 都只能出现一次。如果 key 必须出现一次以上，另有所谓的 multimap 和 multiset 适用。

以下我举一个 vector 实例，让大家见识一下 template classes 的弹性。

#0001 // build : cl /GX vector.cpp

#0002 #include <vector> // STL's vector #0003 #include <iostream>

#0004

#0005 // all the components of the C++ standard library are #0006 // declared and defined within a namespace called std. #0007 using namespace std; #0008

#0009 void main()

#0010 {

#0011 vector<int> ivec; #0012

#0013 cout << "ivec: size: " << ivec.size() << " " #0014 << "capacity: " << ivec.capacity() << endl; #0015

#0016 for (int ix = 0; ix < 24; ix++) #0017 {

#0018 ivec.push\_back( ix ); #0019 cout << "ivec: size: " << ivec.size() << " "

**72**

第１章 C++ Classes 的語法和語意

#0020 << "capacity: " << ivec.capacity() << endl; #0021 }

#0022 }

程序解说：

* L1. 欲使用 STL，必须将 exception handling 打开。/GX 是 Visual C++

的编译选项， 意思是： enable C++ exception handling。

* L2. 使用 STL vector，必须含入 <vector> 这个表头档。以 Visual C++ 为例，你可以在 \MSDEV\INCLUDE 中找到 <vector> 档案。请注意， 视你手上编译器的版本而定， 也可能是 vector.h（ 早期）。
* L7. STL 中所有的 classes、functions、templates、types 都定义于一个特殊的 namespace std 之中（ 关于 namespace 稍后另有一节说明）。L7 的用意是让我们的程序能够看到那个 std namespace。
* L11. 宣告一个 vector， 名为 *ivec*， 内放型别为 int 的元素。
* L13. 在尚未对 *ivec* 加入任何元素之前， 检查其大小与胃纳量。所谓大小是指 vector 内的元素个数，所谓胃纳量（ capacity）是指这个 vector目前可容多少元素。
* L16~L20. 加入 24 个元素到 *ivec* 中。每加入一个元素， 就检查 *ivec* 的 大 小 与 胃 纳 量 。 回 路 之 中我们用到了 vector 的三个 member functions： *push\_back()*、*size()*、*capacity()*。

執行結果如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ivec: | size: | 0 | capacity: | 0 |
| ivec: | size: | 1 | capacity: | 1 |
| ivec: | size: | 2 | capacity: | 2 |
| ivec: | size: | 3 | capacity: | 4 |
| ivec: | size: | 4 | capacity: | 4 |
| ivec: | size: | 5 | capacity: | 8 |
| ivec: | size: | 6 | capacity: | 8 |
| ivec: | size: | 7 | capacity: | 8 |
| ivec: | size: | 8 | capacity: | 8 |
| ivec: | size: | 9 | capacity: | 16 |

ivec: size: 10 capacity: 16

#### 73

多型與虛擬（*Polymorphism in C++*）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ivec: | size: | 11 | capacity: | 16 |
| ivec: | size: | 12 | capacity: | 16 |
| ivec: | size: | 13 | capacity: | 16 |
| ivec: | size: | 14 | capacity: | 16 |
| ivec: | size: | 15 | capacity: | 16 |
| ivec: | size: | 16 | capacity: | 16 |
| ivec: | size: | 17 | capacity: | 32 |
| ivec: | size: | 18 | capacity: | 32 |
| ivec: | size: | 19 | capacity: | 32 |
| ivec: | size: | 20 | capacity: | 32 |
| ivec: | size: | 21 | capacity: | 32 |
| ivec: | size: | 22 | capacity: | 32 |
| ivec: | size: | 23 | capacity: | 32 |
| ivec: | size: | 24 | capacity: | 32 |

我们可以从执行结果归纳出两个结论：

1. Visual C++ 内附之 STL（ HP 授权） 的 vector 胃纳量是可以变化的。
2. Visual C++ 内附之 STL（ HP 授权） 的 vector 胃纳量（ capacity） 变化， 是以 power of 2 累增（ 以本例之 vector 大小而言）。

### STL Generic Algorithm（泛型算法）

为了对付 STL 的各种 containers，以及语言本身的内建资料型别，STL 提供了一组所谓的 generic algorithm（泛型算法）。称之为 algorithm，是因为这些函式都用来执行最普遍的动作，诸如排序、搜寻、最大值、最小值等等。称之为 generic

（泛型）则是因为，它们适用于各种 container types，包括 STL 的vector, list, set, map，以及语言内建的 build-in array type。

STL 提供的 generic algorithm 非常多，我只举一个简单的「字符串逆转」实例。此例使用 *reverse()* 函式：

#0001 // build : cl reverse.cpp #0002 #include <iostream.h>

#0003 #include <algorithm> // STL #0004 #include <string.h>

#0005

#0006 using namespace std; // STL #0007

**74**

第１章 C++ Classes 的語法和語意

#0008 void main()

#0009 {

#0010 char\* string1 = "J.J.Hou was graduated from NCTU in 1983"; #0011 cout << string1 << endl;

#0012

#0013 int N1 = strlen(string1);

#0014 reverse(string1, string1 + N1);

#0015

#0016 cout << string1 << endl; #0017 }

執行結果：

J.J.Hou was graduated from NCTU in 1983 3891 ni UTCN morf detaudarg saw uoH.J.J

程序解说：

* L1. 虽然我在上一个例子中曾说过， 欲使用 STL， 必须将 exception handling 打开， 但因为本例只用到 generic algorithm， 没有用到 STL containers， 所以不需 /GX 编译选项（ for Visual C++）。
* L3. 使用 generic algorithm， 必须含入 <algorithm> 这个表头档。以Visual C++ 为例， 你可以在 \MSDEV\INCLUDE 中找到 <algorithm> 档案。请注意，视你手上编译器的版本而定，也可能是 algo.h（ 早期）。
* L6. 由于 STL 的所有符号命名都置于 std 这个 namespace 之中， 我们必须指定有 L6 才能够「看」到 std namespace。
* L10. C++ 字符串有两种，一种是 C-style 字符串，另一种是 string 型别（ 必须含入 <string>）。本例使用 C-style 字符串。
* L14. 使用 *reverse()*（ 一个 generic algorithm） 来逆转字符串。此函式的两个参数都是操作对象（ 某个 container） 的 iterators， 分别指向第一个元素和最后一个元素。所谓 iterator 是一种泛型指标， 稍后我会介绍。此例的操作对象是传统字符串， 其 iterator 就是一般的字符指标。

#### 75

多型與虛擬（*Polymorphism in C++*）

### STL Iterators（迭代指标）

上例的「字符串逆转」实例使用 C-style 字符串。所以使用一般的字符指标即可代表字符串（也被视为一个 container）的最前与最后两个元素。

但是 STL containers 的弹性很大，可以让你安放自定型别的资料。STL Container

之所以能够和STL generic algorithm 发生关系（系结），靠的是它的一对 iterators

（通常称为 *first* 和 *last*），使 generic algorithm 能够在 container 身上来回移动

（traverses）。

所谓 iterator，可以把它想象是一种泛型指标。它的作用是提供一般化的方法，使我们得以存取任何一种 container 中的任何一个元素。Iterator 其实也是一个template class，它最少必须提供四个 operators（以下以 *iter* 代表一个 iterator）：

1. increment operator： 用以移往下一個元素， 例如 ++*iter*。
2. dereference operator： 用以取出真正的元素個體， 例如 \**iter*。
3. equlity operator：用以判斷兩個 iterators 是否相等，例如 *iter1* == *iter2*。
4. inequlity operator：用以判斷兩個 iterators 是否不等，例如 *iter1* != *iter2*。

任何一个 STL container type 都必须提供两个 member functions：*begin()* 和*end()*，传回的 iterator 分别指向 container 的第一个和最后一个元素。下面是以先前的 vector<int> *ivec* 为例，示范 iterator 的宣告与运用：

vector<int> ivec;

...

vector<int>::iterator iter = ivec.begin(); vector<int>::iterator iter\_end = ivec.end(); for (; iter != iter\_end; ++iter)

cout << \*iter << endl;

最后的 for loop 可以把 *ivec* 的所有内容显示于荧幕上。

**76**

第１章 C++ Classes 的語法和語意

### 一个完整实例

以下我将示范 STL list 的用法。这里所举的例子可以和第４章「为什么需要多型」遥相呼应。

假设我想完成一个 list，每一个节点是一个指标，指向我的一个「形状物件」。形状物件的 class 体系图如下：

CTriangle

CSquare

CRect

CCircle

CEllipse

CShape

架构完成之后，我希望把整个 list 走一遍，将所有的形状显示出来。图 1.1 是范例程序。

图 1.1 Demo 程序， 示范 STL li st 的使用方法

#0001 // demo.h

#0002 class CShape

#0003 {

#0004 public:

#0005 virtual void display() { cout << "CShape" << endl; } #0006 };

#0007

#0008 class CRect : public CShape #0009 {

#0010 public:

#0011 void display() { cout << "CRect" << endl; }

**77**

多型与虚拟（*Polymorphism in C++*）

#0012 };

#0013

#0014 class CEllipse : public CShape #0015 {

#0016 public:

#0017 void display() { cout << "CEllipse" << endl; } #0018 };

#0019

#0020 class CTriangle : public CShape #0021 {

#0022 public:

#0023 void display() { cout << "CTriangle" << endl; } #0024 };

#0025

#0026 class CSquare : public CRect #0027 {

#0028 public:

#0029 void display() { cout << "CSquare" << endl; } #0030 };

#0031

#0032 class CCircle : public CEllipse #0033 {

#0034 public:

#0035 void display() { cout << "CCircle" << endl; } #0036 };

#0037

#0038 class CMyDoc 17

#0039 {

#0040 public:

#0041 list<CShape\*> m\_MyList; #0042 };

#0001 // demo.cpp (build : cl /GX demo.cpp) #0002 #include <iostream.h>

#0003 #include <list> // STL’s list #0004 using namespace std;

#0005 #include "demo.h"

#0006

#0007 void main()

#0008 {

1. 把程序的所有资料放在一个所谓的 "document class" 之中，是一种设计潮流，可以带来许多好处（但并不涵盖于本书范围）。

#### 78

第１章 C++ Classes 的語法和語意

#0009 CShape\* pShape[6];

|  |  |  |
| --- | --- | --- |
| #0010 | pShape[0] = new CShape; |  |
| #0011 | pShape[1] = new CEllipse; | // polymorphic |
| #0012 | pShape[2] = new CCircle; | // polymorphic |
| #0013 | pShape[3] = new CTriangle; | // polymorphic |
| #0014 | pShape[4] = new CRect; | // polymorphic |
| #0015 | pShape[5] = new CSquare; | // polymorphic |
| #0016 |  |  |
| #0017 | CMyDoc myDoc; |  |
| #0018 |  |  |
| #0019 | for (int i=0; i< 6; i++) |  |
| #0020 | myDoc.m\_MyList.push\_back | (pShape[i]); |

#0021

#0022

#0023

#0024

#0025

#0026

#0027

// retrieves each element of myDoc.m\_MyList list<CShape\*>::iterator iter = myDoc.m\_MyList.begin();

list<CShape\*>::iterator itere = myDoc.m\_MyList.end();

for (; iter != itere; ++iter)

(\*(\*iter)).display();

#0028 }

iterator



CMyDoc

list<CShape\*> m\_myList

list<CShape\*>::push\_back()

可以把一個 **CShape\*** 放入 **list** 尾端

圖 1.2 利用 C++ STL li st 完成一个串列， 内放多种「形状物件」

**79**

多型与虚拟（*Polymorphism in C++*）

Demo 程序执行结果如下：

CShape CEllipse CCircle CTriangle CRect CSquare

Demo 程序解说：

* demo.h L5： 由于这个例子应用了物件导向最重要的「多型与虚拟」特性（ 才能够在呼叫每一个元素的 *display()* 时唤起适当的 *display()* 动作）， 所以 *display()* 必须宣告为 virtual。我将在第４章详述「多型与虚拟」。
* demo.h L41： 表示我有一个 list， 其中每个元素都是一个指标， 指向

*CShape* object。这其实就是 polymorphism 的一种应用， 第４章详述。

* demo.cpp L9~L15：一共有 6 个 shapes，每一个都以 base type *CShape*\*

指向之。这也是所谓的 polymorphism， 第４章详述。

* demo.cpp L19~L20：把各种不同的「形状」放进 list 之中。*push\_back()* 可以办到这一点。
* demo.cpp L23~L27：利用 iterators 将 list 走访一遍，并呼叫每一个元素的 *display()* 函式。

#### 80

第１章 C++ Classes 的語法和語意

## MFC Collection Classes

Template classes 非常好用，真的非常非常好用，效率也不断在改善之中。但是 C++ standard library 绝对不是你的唯一选择。早在 STL 问世之前，就已经有一些商品化的 class library，有着类似的功能与威力。当然，它们也不断在演化进步之中。

这一节我将以 MFC（Microsoft Foundation Classes）为例，说明这类 class library 的使用法。

认识 MFC

MFC 是一套融合了视觉物件（我指的是视窗、绘图）以及资料结构（我指的是像先前所说的 container 那样的东西），希望让我们开发 Windows 应用程序时得以轻松自如的 C++ Class Library。在视觉物件部份，由于含入了 Windows 平台特有的讯息驱动机制，以及「将资料实体与展示界面分离」的Document / View 架构，所以规模复杂，许多特别的 classes 很难单独使用（即使勉强这么做，也是大才难为用），必须一股脑儿用一大伙。因此，这类 classes library 又被称为application framework（凝聚性很高的一组 classes）。

但在 MFC 之中，另有一大组被称为 collection classes 者，作用与 STL 相同， 都是为了实现一般化的资料结构与算法。这组 classes 的用法比较单纯，和视

觉物件一点关系也没有，适用于 GU（I Graphic User Interface）环境，也适用于 CUI

（Console User Interface）环境。图 1.3 就是 MFC collection classes 的架构体系图。

**81**

多型與虛擬（*Polymorphism in C++*）

|  |
| --- |
| CTypedPtrArray |
|  |
| CTypedPtrList |
|  |
| CTypedPtrMap |

圖 1. 3 MFC collection classes 的架构体系图（ 灰色为本例使用）

**Typed Template Collections**

Maps of user types

Arrays of user types

CMapStringToString

CWordArray

CMapStringToOb

CUIntArray

CMapStringToPtr

CStringArray

CMapWordToOb

Lists of user types

CPtrArray

CMapPtrToPtr

CStringList

CObArray

CMapPtrToWord

CObList

CDWordArray

CMapWordToPtr

CPtrList

CByteArray

CMap **(template)**

CList **(template)**

CArray **(template)**

CObject

### MFC Collection Classes 应用实例

我将以 MFC Collection Classe（s

和图 1.2 所显示的程序功能。

图 1.3）重新展现前一节图 1.1 的 class hierarchy

MFC Collection Classes 分为两大类：

* 简单型 - *CArray*、*CList*、*CMap*。它们都衍生自 *CObject*，所以它们都具备了一些很好的性质（ 档案读写 serialization、执行时期型别鉴识RTTI、动态生成 dynamic creation）。
* 指标型 - *CTypedPtrArray* 、*CTypedPtrList* 、*CTypedPtrMap*。这些类别要 求 我 们 在 参 数 中 指 定 base class ， 并 且 必 须 是 MFC 之 中 的non-template pointer collections，例如 *CObList* 或 *CPtrArray*。于是新的 class 将继承 base class 的所有性质。

我将尽量选用图 1.3 中的 MFC template classes 来完成图 1.4 的功能，因为这些

classes 是 "type-safe"。

**82**

第１章 C++ Classes 的語法和語意

我发现 *CTypedPtrList* 是个不错的 list 候选人，它的用法如下：

CTypedPtrList<BASE\_CLASS, TYPE>

其中 *TYPE* 用以指定我希望收集的 data type，可以是语言内建型别，也可以是自定型别。*BASE\_CLASS* 用来指定 base class（如前述）。

所以，我可以这么做：

class CMyDoc

{

public: // protected: is better

CTypedPtrList<CObList, CShape\*> m\_MyList;

};

这意思是，我有一个 list，拥有 *CObList* 的一切性质。MFC 手册告诉我们， *CObList* 是一个双向串列，每一个元素是一个指标，指向 *CObject* 的 derived class（这其实就是 polymorphism 的一种应用，第４章详述）。我已经在上面的程序码中指定：*CObList* 的每一个元素是一个 *CShape*\*。因此 *CShape* 必须衍生自 *CObject*。

接下来的工作是把各种不同的「形状」放进 *CMyDoc* object 之中。*AddTail()* 可以办到这一点。最后，我再使用 while loop 将 list 中的每一个元素走访一遍，并呼叫每一个元素的 *display()* 函式。

请注意，这个例子应用了物件导向最重要的「多型与虚拟」特性，所以才能够在呼叫每一个元素的 *display()* 时，唤起适当的 *display()* 动作。我将在第４章详细介绍「多型与虚拟」。

从这个实例看来，MFC 的 collection classes 其实与 STL 有许多相似之处：

1. *CTypedPtrList* 地位相当于 STL 的 *list* container。
2. *AddTail()* 地位相当于 STL *vector* 的 *push\_back()* member function。
3. *POSITION* 地位相当于 STL 的 iterator。

#### 83

多型與虛擬（*Polymorphism in C++*）

1. *GetHeadPosition()* 地位相当于 STL 的 container*::begin()*。
2. *GetNext()* 地位相当于 STL iterator 的 increment operator，而前者传回object pointer， 功能上又相当于 STL iterator 的 dereference operator。
3. 我们可以判断 *POSITION* 是否等于 NULL， 这个功能相当于 STL iterator 的 equlity operator 和 inequlity operator。

#### 程式修改

在 demo.h 方面，以下数行（灰色部份）需要修改（与前一节的 STL 程序比较）：

#0002 class CShape : public CObject

...

#0038 class CMyDoc18

#0039 {

#0040 public:

#0041 CTypedPtrList<CObList, CShape\*> m\_MyList; #0042 };

在 demo.cpp 方面，以下数行（灰色部份）需要修改（与前一节的 STL 程序比较）：

#0001 // build : cl /MT shape.cpp (use MFC, use multithreading) #0002 #include <iostream.h>

#0003 #include <afxtempl.h> // MFC template class

|  |  |  |
| --- | --- | --- |
| #0004 | #include "demo.h" |  |
| #0005 |  |
| #0006 | void main() |
| #0007 | { |
| #0008 | CShape\* pShape[6]; |
| #0009 | pShape[0] = new CShape; |
| #0010 | pShape[1] = new CEllipse; | // polymorphic |
| #0011 | pShape[2] = new CCircle; | // polymorphic |
| #0012 | pShape[3] = new CTriangle; | // polymorphic |
| #0013 | pShape[4] = new CRect; | // polymorphic |
| #0014 | pShape[5] = new CSquare; | // polymorphic |
| #0015 |  |  |
| #0016 | CMyDoc myDoc; |  |

1. 把程序的所有资料放在一个所谓的 "document class" 之中，是一种设计潮流，可以带来许多好处（但是不在本书范围）。

#### 84

第１章 C++ Classes 的語法和語意

#0017 for (int i=0; i< 6; i++)

#0018 myDoc.m\_MyList.AddTail(pShape[i]);

#0019

#0020 // retrieves each element of myDoc.m\_MyList

#0021 POSITION pos = myDoc.m\_MyList.GetHeadPosition(); #0022 while (pos != NULL)

#0023 {

#0024 CShape\* ps = myDoc.m\_MyList.GetNext(pos); #0025 ps->display();

#0026 }

#0027 }

POSITION pos



CMyDoc

CTypedPtrList<CObList CShape\*>

m\_myList

**AddTail() 可以**把一個

**CShape\*** 放入 **list** 尾端

圖 1.4 利用 MFC collection classes 完成一个串列， 内放多种「形状物件」

#### 85

多型与虚拟（*Polymorphism in C++*）

### RTTI（Runtime Type Identification）

所谓 RTTI，意思是，在执行时期判断某一个 object 或某一个 pointer 属于何种型别。它是 C++ 许多高级技术领域的基础。RTTI 的主要形式在 C++ 中是以*typeid* 和 *dynamic\_cast* operator 展现。

### typeid operator

typeid operator 的语法如下：

typeid(type-id)

例如：typeid(CPoint)

typeid(expression)

例如：typeid(pPoint) 或 typeid(\*pPoint)

*typeid()* 可以在程序执行期间决定一个 object 的型别（type）。它的传回值是一个 const *type\_info*&。C++ Standard 中对于 *type\_info* object 定义如下：

class type\_info { public:

virtual ~type\_info();

bool operator==( const type\_info& ) const; bool operator!=( const type\_info& ) const; bool before( const type\_info& ) const;

const char\* name() const; // 傳回 class 原始名稱

private:

// prevent memberwise init and copy type\_info( const type\_info& ); type\_info& operator=(const type\_info& );

// data members

};

有些文献曾提出以编码后的文字来代替实际的 class 名称，可能会获得比较好的执行效率。Visual C++ 就是这么做，在其 \msdev\include\typeinfo.h 中对 class *type*\_*info* 定义如下：

**86**

第１章 C++ Classes 的語法和語意

// declared in typeinfo.h class type\_info {

public:

\_CRTIMP virtual ~type\_info(); 19

\_CRTIMP int operator==(const type\_info& rhs) const;

\_CRTIMP int operator!=(const type\_info& rhs) const;

\_CRTIMP int before(const type\_info& rhs) const;

\_CRTIMP const char\* name() const;

\_CRTIMP const char\* raw\_name() const; private:

void \*\_m\_data; char \_m\_d\_name[1];

type\_info(const type\_info& rhs);

type\_info& operator=(const type\_info& rhs);

};

我们可以利用其中的 *name()* 和 *rawname()* 两个 member functions，取出我们所探询之 object 的型别名称。*name()* 传回的指标，指向一个人类可读的字符串， *rawname()* 传回的指标则指向一个编码后的字符串，非人类可读形式。此一编码后的字符串对于两个型别之间的比对较有效率。

如果你要使用 *typeid()*，程序中必须含入 typeinfo.h 档。

下面是个实际的例子：

#0001 // building : cl typeid.cpp #0002 #include <iostream.h>

#0003 #include <typeinfo.h> #0004

#0005 class B // base class, non-polymorphic #0006 {

#0007 public:

#0008 int \_data1;

#0009 };

#0010

#0011 class D : public B // derived class, non-polymorphic too. #0012 {

#0013 public:

1. Visual C++ 表头档定义： #define \_CRTIMP declspec(dllimport)

#### 87

多型與虛擬（*Polymorphism in C++*）

#0014 int \_data2;

#0015 };

#0016

#0017 void main()

#0018 {

#0019 B \*pb = new B; #0020 D \*pd = new D; #0021

#0022 cout << "B is Base type, D is Derived type" << endl; #0023

#0024 // check object pointer's type #0025 if (typeid(B) == typeid(\*pb)) #0026 cout << "pb is B type" << endl; #0027 else

#0028 cout << "pb is not B type" << endl; #0029

#0030 if (typeid(D) == typeid(\*pd)) #0031 cout << "pd is D type" << endl; #0032 else

#0033 cout << "pd is not D type" << endl; #0034

#0035 // check object pointer's type name

#0036 cout << "pb's type name = " << typeid(pb).name() << endl; #0037 cout << "pd's type name = " << typeid(pd).name() << endl; #0038

#0039 cout << "pb's type rawname = "

#0040 << typeid(pb).raw\_name() << endl; #0041

#0042 cout << "pd's type rawname = "

#0043 << typeid(pd).raw\_name() << endl; #0044

#0045 // check object's type name

#0046 cout << "\*pb's type name = " << typeid(\*pb).name() << endl; #0047 cout << "\*pd's type name = " << typeid(\*pd).name() << endl; #0048

#0049 cout << "\*pb's type rawname = "

#0050 << typeid(\*pb).raw\_name() << endl; #0051 cout << "\*pd's type rawname = "

#0052 << typeid(\*pd).raw\_name() << endl; #0053

#0054 // check object pointers' relationship #0055 if (typeid(pb).before(typeid(pd))) #0056 cout << "pb before pd" << endl; #0057 else

#0058 cout << "pb not before pd" << endl; #0059

#### 88

第１章 C++ Classes 的語法和語意

|  |  |  |
| --- | --- | --- |
| #0060 |  | // check objects' relationship |
| #0061 |  | if (typeid(\*pb).before(typeid(\*pd))) |
| #0062 |  | cout << "\*pb before \*pd" << endl; |
| #0063 |  | else |
| #0064 |  | cout << "\*pb not before \*pd" << endl; |
| #0065 |  |  |
| #0066 |  | B\* pb1 = new D; // base ptr point to derived object. |
| #0067 |  | cout << "pb1's type name = " << typeid(pb1).name() << endl; |
| #0068 |  | cout << "\*pb1's type name = " << typeid(\*pb1).name() << endl; |
| #0069 |  |  |
| #0070 |  | // check objects' relationship |
| #0071 |  | if (typeid(\*pb1).before(typeid(B))) |
| #0072 |  | cout << "\*pb1 before B" << endl; |
| #0073 |  | else |
| #0074 |  | cout << "\*pb1 not before B" << endl; |
| #0075 |  |  |
| #0076 |  | if (typeid(\*pb1).before(typeid(D))) |
| #0077 |  | cout << "\*pb1 before D" << endl; |
| #0078 |  | else |
| #0079 |  | cout << "\*pb1 not before D" << endl; |
| #0090 | } |  |

##### D\* pd

pd’s scope

##### D object

B object

m\_data2

m\_data1

##### B\* pb1 pb1’s scope

B\* pb pb’s scope

m\_data1

这个程序的输出结果如下：

B is Base type, D is Derived type pb is B type

pd is D type

pb's type name = class B \* // 原名称

pd's type name = class D \* // 原名称

pb's type rawname = .PAVB@@ // 编码后的名称

pd's type rawname = .PAVD@@ // 编码后的名称

#### 89

多型與虛擬（*Polymorphism in C++*）

\*pb's type name = class B // 原名称

\*pd's type name = class D // 原名称

\*pb's type rawname = .?AVB@@ // 编码后的名称

\*pd's type rawname = .?AVD@@ // 编码后的名称

pb before pd // before 的意思是在 classes hierarchy 的较顶层

\*pb before \*pd

okjo

pb1's type name = class B \*

\*pb1's type name = class B

\*pb1 not before B

\*pb1 before D

// 合理

// 想不到吧！

// 合理

// 想不到吧！

请注意，虽然 L66 的 *pb1* 指向一个真正的class *D* object，但利用 t*ypeid()* 所得的检查结果却显示，*pb1* 所指物件的 type name 是 "class B"，想不到吧！

如果我把 *typeid()* 应用于 polymorphic classes（简单地说，就是拥有 virtual functions 者）身上，执行结果会出现一点变化。现在我把前一个例子的 class *B* 和class *D* 做一些小修改如下（其余程序码完全不变）：

#0001 // building : cl -GR typeid.cpp #0002 #include <iostream.h>

#0003 #include <typeinfo.h>

#0004

#0005 class B // base class, polymorphic. #0006 {

#0007 public:

#0008 virtual void func() { }; #0009 int \_data1;

#0010 };

#0011

#0012 class D : public B // derived class, polymorphic too. #0013 {

#0014 public:

#0015 virtual void func() { }; #0016 int \_data2;

#0017 };

将 *typeid()* 应用于 polymorphic classes，在 Microsoft Visual C++ 环境下必须设定编译选项 /GR，否则会出现以下警告讯息：

warning C4541: 'typeid' used on polymorphic type 'class B' with /GR-; unpredictable behavior may result

#### 90

第１章 C++ Classes 的語法和語意

D\* pd

pd’s scope

D object

|  |
| --- |
| vptr |
| m\_data1 |
| m\_data2 |

B object

B\* pb1 pb1’s scope

B\* pb

pb’s scope

m\_data1

vptr

现在让我们看看执行结果。变化出现在 polymorphism 发生时，也就是当程序码以

base type pointer 指向 derived type object 时：

#0066 B\* pb1 = new D; // base ptr point to derived object.

此行之后的执行结果如下，请与先前的结果比较：

pb1's type name = class B \*

\*pb1's type name = class D

\*pb1 not before B

\*pb1 not before D

// 合理

// 好極了！

// 合理

// 合理

在 polymorphism 情况之下，虚拟机制确实记录了 *pb1* 所指物件的型别为 "class D" -- 虽然 *pb1* 本身的原始型别为 "class B \*"。这种能力，为我们带来许多好处。我将在第４章详细介绍多型与虚拟。

#### 91

多型與虛擬（*Polymorphism in C++*）

### 自己动手做 RTTI

在 polymorphism 的情况下，为了能够满足程序对于指标型别的检查（如前例的*pb1*），以便获得「指标真正所指物件之型别」，显然编译系统必须在某一个地方记录 class 的相关信息（至少也得包括 class name，以及各个 classes 之间的相对排序算法）。我们已经在上一节看到，这份信息显然是记录在 *type\_info* class 之中。但 *type\_info* 在哪里？和 objects 实体如何产生关联？

我将在第２章p.110 更进一步介绍 *type\_info* 在存储器中的位置。

早在 C++ 支援 RTTI 之前，一些商用的 C++ application framework 已经完成了这项功能。也就是说，它们巧妙地在语言之外，模拟实现了 *type\_info* 机制。其中的作法并不见得和 C++ 语言底层作法雷同（程序层面和语言层面毕竟是有大不同的），但是巧妙的手法却非常值得参考。我将在第５章实作出一套 RTTI 系统。

## Dynamic Creation（动态生成）

OO 术语之中有一个名词为 persistence，意思是永久保存。

Objects 放在 RAM 之中，生命受到电力的左右，不可能永久保存；唯一的办法是把它们写到档案去。

表面上看起来 persistence 与本节的主题 dynamic creation 没有什么关系。有的， 你把你的程序资料储存到档案，这些资料很可能（通常是）objects 的 data members。当程序再从档案中把它们读出来，势必要依据档案上的记载，重新 *new* 出那些个 objects 来，以便存放那些 data members。问题是，即使程序中已经宣告了那些 classes，你能够这么做吗：

**92**

第１章 C++ Classes 的语法和语意

char className[30] = getClassName(); // 从档案中获得一个类别名称

CMyObject\* obj = new classname; // error C2061: syntax error :

// identifier 'classname'

这样的需求即所谓的 dynamic creation：「在程序执行时期获得一个 class name， 然后产生出对应的 objects」。但是上述第二个动作过不了编译器的关口。事实上目前没有任何一个 C++ 编译器可以在语言层面实作出 dynamic creation。

为了实现 dynamic creation，有一个「笨」方法倒是可以用：

... // read classname to str if (str == "Class1")

new Class1;

else if (str == "Class2") new Class2;

else if (str == "Class3") new Class3;

else if (str == "Class4") new Class4;

...

这种穷举比对法，当然可以解决眼下的问题，但日后维护十分困难。每次新增加

classes，程序员必须记得在这一段码增加一小段比对动作。

自己动手做 Dynamic Creation

在 C++ 编译器没有支援 dynamic creation 的情况下，我们唯一可以使用的其实就是上述的穷举比对法，关键是如何把形式做得漂亮一点，容易维护一点。

我将在第５章实作出一套 dynamic creation 系统。此系统与上一节所说的 RTTI

自制系统必须搭配在一起。

**93**

多型與虛擬（*Polymorphism in C++*）

## Namespace（命名空间）

预设情况下，每一个 object、每一个 function、每一种 type，如果宣告于 global scope（全域范畴，或称为 global namespace scope）之中，就会拥有它自己的一个所谓的 global entity。在同一个程序中，不管是不是位于一份文字档（原始档）， objects 或 functions 或 types 的名称是不能够牴触的。这意味着你必须非常小心地不要和你所使用的 libraries 产生符号命名上的冲突。有些时候你想要小心谨慎亦不可得，因为你所使用的 libraries 可能并没有在其文件中详细说明它的各种命名。这种因命名而造成的冲突在我们大量使用各家厂商制造的各式各样的libraries 时尤其带来困扰。这种问题被称为 "global name space pollution"。

所谓 scope：在 C++ 语言中，scope（范筹）就是以左右大括号 { } 涵盖起来

的空间。objects 的生命、变量的生命，都以 scope 为依据。你可以在程序中使用

{ } 任意产生 scopes。

解决命名冲突的方法之一是，我们在自己程序的每一个命名符号之前，加上专属的前置词。于是名称变得又臭又长，在输入上和阅读上都形成一种负担。解决方式之二就是使用自定的 namespace。语法如下：

namespace polymorphism\_in\_cplusplus

{

class CPoint { /\* ... \*/ }; void printall(CPoint &);

}

出现在 namespace 之中的各种元素，包括 objects、template、functions、types 都是所谓的 namespace members。不同的 namespace 中的 members 可以有相同的命名。编译器如何对待它们呢？编译器会把上述的两个 namespace members 改为全名如下：

**94**

第１章 C++ Classes 的語法和語意

1. polymorphism\_in\_cplusplus::CPoint
2. polymorphism\_in\_cplusplus::printall()

持续使用又臭又长的符号命名规则：

namespace\_name::member\_name

是一件颇为累赘的事情。于是 C++ 又发展出一些所谓的 namespace aliases、using declarations、using directives 等等机制（请参考 C++ 书籍）。

如果我们的程序中想要「看见」某个 namespace，语法如下：

using namespace xxx; // xxx 是 namespace 的名称

举个例子，STL 中所有的 classes、functions、templates、types 都定义于一个名为 std 的 namespace 之中。任何一个 C++ 程序想要使用 STL，除了首先含入必要的 header file 之外，第一个程序动作就是：

using namespace std; // std is STL’s namespace

请参考p.78 程序码。

以下我再举一个小型实例，示范 namespace 的其他用法：

#0001 #include <iostream.h>

#0002

#0003 namespace ns1

#0004 {

#0005 class CPoint { /\* ... \*/ };

#0006 void print(CPoint&) { cout << "ns1::print()" << endl; }; #0007 void output() { cout << "ns1::output()" << endl; }; #0008 }

#0009

#0010 namespace ns2

#0011 {

#0012 class CPoint { /\* ... \*/ };

#0013 void print(CPoint &) { cout << "ns2::print()" << endl; }; #0014 void output() { cout << "ns2::output()" << endl; };

#### 95

多型與虛擬（*Polymorphism in C++*）

#0015 }

#0016

#0017 void main()

#0018 {

#0019 {

#0020 using namespace ns1; // note: using directive are scoped. #0021

#0022 CPoint aPoint; // if no "using namespace ns1;", #0023 // or "using namespace ns2;", then

#0024 // error C2065: 'CPoint' : undeclared identifier #0025 print(aPoint); // ns1::print()

#0026 }

#0027

#0028

#0029 {

#0030 using namespace ns2; // note: using directive are scoped. #0031

#0032 CPoint aPoint;

#0033 print(aPoint); // ns2::print() #0034 }

#0035

#0036 // try the scope operator for namespace #0037 ns1::CPoint pt1;

#0038 ns2::CPoint pt2;

#0039

#0040 ns1::output(); // ns1::output() #0041 ns2::output(); // ns2::output() #0042

#0043 ns1::print(pt1); // ns1::print() #0044 ns2::print(pt2); // ns2::print() #0045

#0046 /\*

#0047 ns1::print(pt2); // error C2664: 'print' : cannot convert param 1 #0048 // from 'class ns2::CPoint' to 'class ns1::CPoint &' #0049 ns2::print(pt1); // error C2664: 'print' : cannot convert param 1 #0050 // from 'class ns1::CPoint' to 'class ns2::CPoint &' #0051 \*/

#0052 }

#### 96

第２章 C++ 物件模型（C++ Object Model）

**第２章**

# C++ 物件模型

## （C++ Object Model）

#### 薄幕重重：「伟大」的 C++ 编译器

当我第一次接触 C++ 的时候，我非常不能够适应，常常出现「雄兔脚扑朔，雌兔眼迷离」的情况。或许因为我的血液中已经充满了 sequential language 的因子， 总奇怪为什么明明很多事情我没有做，却都会自动执行（好比说 constructor）？ 为什么有些动作虽然符合需要，却是那么地离奇（好比说 virtual functions）？

慢慢地，当我对 C++ 的语法（syntax）和语意（semantic）更了解，也学习了所谓的 C++ 物件模型之后，疑云就一层一层地褪去了。计算机的世界，说到底，没有什么是会自动发生的。我（程序员）没有指定的事情，之所以能够发生，是因 为 C++ 编译器幕后做的手脚、牵的线。于是，我开始喜欢形容 C++ 是一个布幕太多的语言：编译器为我们设下太多太多的布幕，使物件导向的舞台变幻万千， 但是布幕与布幕之间，象是米诺托斯（Minotauros，希腊神话中的牛头人身怪兽） 的迷宫，初学者极易迷惘（甚至丧生）于其中，或是在「为什么会这样、为什么

#### 97

多型與虛擬（*Polymorphism in C++*）

会那样」的情绪中郁郁寡欢。

我已经在第一章将 C++ class 的各种语法，都做了梗概整理。其中也揭露了一些C++ 编译器幕后的斧痕（例如 default constructor 和 copy constructor）。现在， 我要把另一些我认为很重要的 C++ 布幕揭开。

#### C++ 物件模型（ C++ Object Model）

C++ 编译器必须设计出一套架构，使 objects 的产生、摧毁、继承、虚拟、指标转型、RTTI、exception handling、template...有一个合理而可实现的运作模型，得以完成 C++ 的诸多物件导向性质。此即所谓的 C++ 物件模型。

物件模型的设计与实作，可能会随着不同的编译器而有所不同。C++ Standard 只规定 C++ 的语法（syntax）和语意（semantic），并没有规范什么样的语意必须用什么样的技术（物件模型）来达成。本章的各个实例探讨，除非有特别说明， 都是以 Microsoft C++ 编译器（Visual C++）为对象。

既然 C++ 物件模型并不存在所谓的标准（也就是 binary standard），我们探讨这个主题，意义何在？噢，了解物件模型底层机制（即使只不过是市面上某一种编译器的作法），有助于我们对 C++ 语意「知其然并知其所以然」。我们并不打算学习如何设计一个 C++ 编译器，我们的目的放在彻底了解物件导向的精髓「虚拟与多型」的运作方式，从而使我们在运用 C++ 的「虚拟与多型」性质时，更得心应手。也因此，在庞大的 C++ 物件模型中，我把焦点放在「物件布局」和

「虚拟机制」两大主题。

#### 暖身

首先让我再把 inheritance（继承）、virtual functions（虚拟函式）、polymorphic class

（多型类别）的观念做个整理。我以 class *B* 代表 base class，class *D* 代表 derived class，*vfunc()* 代表 virtual function：

#### 98

第２章 C++ 物件模型（C++ Object Model）

ClassB object

ClassD object

class B // base class

{

vptr

m\_dataB

public:

virtual void vfunc(); // virtual function private:

int m\_dataB;

};

class D : public B // derived class

vptr

m\_dataD

m\_dataB

{

public:

virtual void vfunc(); // override! private:

int m\_dataD;

};

B::vfunc()

B b;

b.vfunc(); // invoke B::vfunc() D d;

D::vfunc()

d.vfunc(); // invoke D::vfunc()

B\* pb = new B; // native

pb->vfunc(); // invoke B::vfunc() pb = new D; // polymorphic

pb->vfunc(); // invoke D::vfunc()

* inheritance ： 当 *D* 继承了 *B* ， *D* 所继承的是 *B* 全部的 data members 以及所有的 member functions。*D* object 的大小应该是 *B* object 大小（ 也就是 *B* subobject）加上 *D* data members 的大小。至于所谓「继承了 member functions」，并不是指在 object 体内复制一份函式实体，而是指 *D* object 拥有「唤起 *B* member functions 权力」的意思。
* virtual functions： 在 member functions 宣告之前加上 virtual 关键字，即成为 virtual functions。这种函式的特色是，*D* 继承之后， 如果重新定义（ override）， 存储器中就会出现 *B::vfunc()* 和 *D::vfunc()* 两个函式实体。当我们以 object 指标来呼叫 virtual functions，如果指标实际指向 *D* object， 唤起的就是 *D::vfunc()*； 如果指标实际指向 *B* object， 唤起的就是 *B::vfunc()*。
* polymorphic class： 具有 virtual funcion(s) 的 class（ 不论是继承而来或是本身新有）， 称为 polymorphic class。

#### 99

多型与虚拟（*Polymorphism in C++*）

this 指标 - Data members 与 member functions 的桥梁

让我们从隐藏的 *this* 指标开始这趟惊奇之旅。

当我产生一个 object，class member functions 并不占用object 存储器，而是独立于外。不管同一个 class 有多少 objects 被产生出来，这个 class 的 member functions 都只有一份实体。

同一份 member functions 函式实体，如何能够处理不同的 objects？唯一的可能是从参数中识别，但我们从 member functions 的参数中看不出任何端倪呀。

事情的真象是，每一个 member functions 都有一个隐藏参数，称为 *this* 的便是。说它是「隐藏」参数，因为这是编译器为我们加上去的，从 C++ 程序码中看不出来。假设我有一个 class 宣告为：

class CRect

{

protected:

int m\_color; public:

void setcolor(int color) { m\_color = color; }

};

当我这么做：

CRect aRect1, aRect2; aRect1.setcolor(2); aRect2.setcolor(3);

C++ 编译器为我做出来的码其实是：

CRect::setcolor(2, (CRect\*)&aRect1); CRect::setcolor(3, (CRect\*)&aRect2);

新增的第二参数，就是所谓的 *this* 指标。至于 *CRect* 中的 *setcolor()* 定义：

#### 100

第２章 C++ 物件模型（C++ Object Model）

class CRect

{

...

public:

void setcolor(int color) { m\_color = color; }

};

则被C++ 编译器改为：

class CRect

{

...

public:

void setcolor(int color, (CRect\*)this1) { this->m\_color = color; }

};

下图可以说明 *this* 指标的作用。



aRect1

aRect2

這個 **this** 參數是編譯器自行為我們加上的，

所以我說它是個 “隱藏指標”。

CRect::setcolor(int color,

**CRect\* this**)

{

**this->**m\_color = color;

}

this 指標 this 指標

or

aRect1.setcolor() 和 aRect2.setcolor() 呼叫的都是 CRect::setcolor()，

此函式之所以能分別處理相同 class 之不同 objects，完全是靠一個隱藏的 this 指標。

m\_color

m\_color

1. C++ Standard 并没有规定 *this* 指标应该被编译器加于第一个参数或是最后一个参数。

#### 101

多型与虚拟（*Polymorphism in C++*）

## sizeof 的疑惑

从这一节开始，我将对于 objects 在存储器中的布局（也就是 class data members、vptr、vtbl）以及 member functions 的位址，有许多探讨。这些位址及其内容都是亲自写码探索而得。然而现代化软件开发整合环境所附的除错器（debugger）其 实能够很方便地让我们随时随地观看 objects 的布局和函式位址。我将在本章最后一节介绍 Visual C++ 的除错环境。

假设我有一个 classes hierarchy 如图 2.1：

#0001 class ClassA

#0002 {

#0003 public:

#0004 int m\_data1;

#0005 int m\_data2;

#0006 void func1() { cout << "ClassA::func1" << endl; } #0007 void func2() { cout << "ClassA::func2" << endl; }

virtual

ClassC

ClassB

ClassA

#0008

#0009

#0010

#0011

#0012

#0013

#0014

#0015

void vfunc1() { cout << "ClassA::vfunc1" << endl; } virtual void vfunc2() { cout << "ClassA::vfunc2" << endl; }

};

class ClassB : public ClassA

{

public:

int m\_data3;

#0016 void func2() { cout << "ClassB::func2" << endl; } #0017 virtual void vfunc1() { cout << "ClassB::vfunc1" << endl; } #0018 };

#0019

#0020 class ClassC : public ClassB #0021 {

#0022 public:

#0023 int m\_data1;

#0024 int m\_data4;

#0025 void func2() { cout << "ClassC::func2" << endl; } #0026 virtual void vfunc1() { cout << "ClassC::vfunc1" << endl; } #0027 };

图 2.1 单一继承的 classes hierarchy

**102**

第２章 C++ 物件模型（C++ Object Model）

现在让我量测 *ClassA*、*ClassB*、*ClassC* 的大小：

cout << "sizeof(ClassA)=" << sizeof(ClassA) << endl; cout << "sizeof(ClassB)=" << sizeof(ClassB) << endl; cout << "sizeof(ClassC)=" << sizeof(ClassC) << endl;

得到的结果是：

sizeof(ClassA)=12 sizeof(ClassB)=16 sizeof(ClassC)=24

但如果我把程序码 L8, L9, L17, L26 的 virtual 关键字拿掉，得到的结果是：

sizeof(ClassA)=8 sizeof(ClassB)=12 sizeof(ClassC)=20

为什么virtual 修饰词的有无，会造成 object 大小的改变？member functions 实作码不是独立于 object 之外的吗？

## 物件布局大局观：资料与函式

先前我一再强调过一个观念：一个 object 实体之内只内含 class data members； class member functions 独立于 object 实体之外，是独一无二的一份函式实体。因此，在没有 virtual functions 的情况下，如果我产生三个 objects：

ClassA a; ClassB b; ClassC c;

这三个 objects 的物件布局如图 2.2 右。C++ 中凡是处于同一个 access section 的资料，必定以其宣告次序出现在存储器布局当中。但如果是被放置在多个 access sections 中的各笔资料，排列次序就不一定。

**103**

多型與虛擬（*Polymorphism in C++*）

加上 virtual functions 之后（成为所谓的polymorphic class），三个 objects 的大小都多了 4 bytes（如图 2.2 左）。而且只要 base class 有一个 virtual function， 就会造成这种结果 -- 不论 derived classes 有没有改写（override）继承得来的virtual functions 或创造它自己新的 virtual functions。多出来的 4 bytes，是个指标

（或称为 virtual pointer，vptr），指向一个由函式指标所形成的阵列（表格，或称为 virtual table，vtbl）。表格中的每一个函式指标指向一个 virtual function 函式实体，如图 2.3。

vptr 和 vtbl 正是形成曼妙的虚拟机制的关键。

12

bytes

a (ClassA object)

m\_data2

m\_data1

vptr

8

bytes

a (ClassA object)

m\_data2

m\_data1

16

bytes

24

bytes

b (ClassB object)

|  |
| --- |
| vptr |
| m\_data1 |
| m\_data2 |
| m\_data3 |

c (ClassC object)



|  |
| --- |
| vptr |
| ClassA::m\_data1 |
| m\_data2 |
| m\_data3 |
| m\_data1 |
| m\_data4 |

ClassA subobject

ClassB subobject



12

bytes

20

bytes

b (ClassB object)

c (ClassC object)



|  |
| --- |
| m\_data1 |
| m\_data2 |
| m\_data3 |

ClassA subobject

ClassB subobject



|  |
| --- |
| ClassA::m\_data1 |
| m\_data2 |
| m\_data3 |
| m\_data1 |
| m\_data4 |

当 ClassA 宣告有 virtual function(s) 时， 三个 objects 的大小及存储器布局

当 ClassA 没有任何 virtual function 时， 三个 objects 的大小及存储器布局

图 2.2 object 的大小忠实反应出 class data members 的大小（ 本图以图 2.1 的 class 体系为例）。但如果是 polymorphic class，每个 object 又多出 4 个 bytes， 即所谓的 vptr 指标。

#### 104

第２章 C++ 物件模型（C++ Object Model）

0x0063FDEC

0x409004

vptr ClassA’s vtbl

0x00409004

0x00409008

0x0040900C

0x00409010

b (ClassB object)

0x409014

vptr

0x00409014

0x00409018

0x0040901C

0x00409020

ClassB’s vtbl

**？**

**？**

c (ClassC object)

ClassC’s vtbl

0x00409024

0x00409028

0x0040902C

0x00409030

**？**

0

0

**0x401F10**

0x401FF0

0x4096A0

0

**0x401F10**

0x401F80

m\_data3

m\_data2

m\_data1

0x409650

0

**0x401F10**

0x401ED0

m\_data2

m\_data1

ClassC::vfunc1()

ClassB::vfunc1()

**ClassA::vfunc2()**

ClassA::vfunc1()

ClassC::func2()

ClassB::func2()

ClassA::func2()

ClassA::func1()

0x0063FDF0

0x0063FDF4

a (ClassA object)

functions implementation

0x0063FDDC

0x0063FDE0

0x0063FDE4

0x0063FDE8

0x0063FDC4

0x0063FDC8

0x0063FDCC

0x0063FDD0

0x0063FDD4

0x0063FDD8

请注意：图中各位址的排列颇为有趣，是连续的。

|  |  |  |
| --- | --- | --- |
| 0x409024 |  | vptr |
|  |  |
| ClassA::m\_data1 | |
| m\_data2 | |
| m\_data3 | |
| m\_data1 | |
| m\_data4 | |

图 2.3 根据图 2.1 classes 所实现的三个 objects 的物件布局（ in VC++）

Polymorphic class object 中多出来的 4 bytes，是一个所谓的 vptr 指标，指向一个所谓的 vtbl（virtual table）表格。vtbl 表格内的每一笔项目都是一个函式指标2， 分别指向该 class 的 virtual functions 函式实体。

图 2.3 中的存储器位址与内容的获得方法，皆在稍后披露。

1. 有些编译器会把支援 RTTI 所需的 *type\_info* class object 位址放置于 vtbl 的第一笔项目中。p.110 另有说明。

#### 105

多型与虚拟（*Polymorphism in C++*）

## Data Members 的布局

图 2.3 的 objects *a*, *b*, *c* 中的 data members 位址，可以这样获得（每一行程序码后面的注解，即为输出结果）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| cout | << | &a << endl; |  | // | 0x0063FDEC |
| cout | << | &(a.m\_data1) << endl; |  | // | 0x0063FDF0 |
| cout | << | &(a.m\_data2) << endl; |  | // | 0x0063FDF4 |
| cout | << | &b << endl; |  | // | 0x0063FDDC |
| cout | << | &(b.m\_data1) << endl; |  | // | 0x0063FDE0 |
| cout | << | &(b.m\_data2) << endl; |  | // | 0x0063FDE4 |
| cout | << | &(b.m\_data3) << endl; |  | // | 0x0063FDE8 |
| cout | << | &c << endl; |  | // | 0x0063FDC4 |
| cout | << | &(c.ClassA::m\_data1) | << endl; | // | 0x0063FDC8 |
| cout | << | &(c.m\_data2) << endl; |  | // | 0x0063FDCC |
| cout | << | &(c.m\_data3) << endl; |  | // | 0x0063FDD0 |
| cout | << | &(c.m\_data1) << endl; |  | // | 0x0063FDD4 |
| cout | << | &(c.m\_data4) << endl; |  | // | 0x0063FDD8 |

请注意，*ClassC* 有一个 *m\_data1*，和 *ClassA* 的 *m\_data1* 同名，在这种情况下， 我必须在程序中完整写出 data members 的全名（如上面的灰色部份），才能够区分之。下面是全部使用完整名称的程序写法及其结果：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| cout  cout | <<  << | &a << endl;  &(a.ClassA::m\_data1) | << | endl; | //  // | 0x0063FDEC  0x0063FDF0 |
| cout  cout cout | <<  <<  << | &(a.ClassA::m\_data2)  &b << endl; &(b.ClassA::m\_data1) | <<  << | endl;  endl; | //  //  // | 0x0063FDF4  0x0063FDDC  0x0063FDE0 |
| cout | << | &(b.ClassA::m\_data2) | << | endl; | // | 0x0063FDE4 |
| cout  cout cout | <<  <<  << | &(b.ClassB::m\_data3)  &c << endl; &(c.ClassA::m\_data1) | <<  << | endl;  endl; | //  //  // | 0x0063FDE8  0x0063FDC4  0x0063FDC8 |
| cout | << | &(c.ClassB::m\_data2) | << | endl; | // | 0x0063FDCC |
| cout | << | &(c.ClassA::m\_data2) | << | endl; | // | 0x0063FDCC |
| cout | << | &(c.ClassB::m\_data3) | << | endl; | // | 0x0063FDD0 |
| cout | << | &(c.ClassC::m\_data1) | << | endl; | // | 0x0063FDD4 |
| cout | << | &(c.ClassC::m\_data4) | << | endl; | // | 0x0063FDD8 |

注意：灰色部份的两行表示法，代表的是同一个 data member。这表示 *m\_data2* 既属于 *ClassB* subobject，也属于 *ClassA* subobject。

**106**

第２章 C++ 物件模型（C++ Object Model）

上一页的前后两种写法（习惯写法和完整写法），结果当然完全相同。对 data members 书写全名，你必须具备一个观念：objects 实体中拥有其 class 及其 base class(es) 中的所有 data members。也因此，我们可以把一个 derived class object 视为其各个 base class subobjects 的组合。

Vtbl 应该是「同一个 class 所衍生的每一个objects」的共同参考部份（稍后我会详细介绍 vtbl 的用途），编译器有必要为它做出两份来吗？显然没有必要，就好像程序码也是独一无二的一份就好，道理相同。以下验证这项事实，图 2.4 表现出这样的观念。

ClassB b1, b2, b3;

cout << "b1 address=" << &b1 << endl; // 0x0063FD70 cout << "b2 address=" << &b2 << endl; // 0x0063FDA0 cout << "b3 address=" << &b3 << endl; // 0x0063FD8C

long \*v1, \*v2, \*v3; // 代表 b1, b2, b3 的 vptr

v1 = (long \*)&b1; v2 = (long \*)&b2; v3 = (long \*)&b3;

printf("%x \n",\*v1);

// 0x409014

printf("%x \n",\*v2); // 0x409014 printf("%x \n",\*v3); // 0x409014

以上使用 *printf* 输出而未使用cout 输出，是因为 \**v1*, \**v2*, \**v3* 都是 long，需另以 iostream's **parameterized manipulators** 才能输出 16 进位格式。如欲使用iostream's parameterized manipulators，可这么做：

// build by VC: cl /GX xxx.cpp #include <iostream>

#include <iomanip> using namespace std;

…

int i = 0x1234;

cout << setw(8) << setfill('0') << hex << i << endl; // 00001234

#### 107

多型与虚拟（*Polymorphism in C++*）

0x0063FD70

0x0063FDA0

b1 (ClassB object)

|  |
| --- |
| 0x409014 |
| m\_data1 |
| m\_data2 |
| m\_data3 |

vptr

b2 (ClassB object)

0x409014

vptr

ClassB’s vtbl

0x00409014

b3 (ClassB object)

m\_data3

m\_data2

m\_data1

0x0063FD8C

vptr

|  |
| --- |
| 0x409014 |
| m\_data1 |
| m\_data2 |
| m\_data3 |

圖 2.4 同一个 class 所衍生的每一个 objects， 共同参考同一份 vtbl

#### 108

第２章 C++ 物件模型（C++ Object Model）

vptr 座落何处

我在图 2.2 ~ 图 2.4 中，都把 vptr 画在 object 的最起始处。C++ Standard 可曾这样规定？没有，C++ Standard 只要求编译器能够做出各种必要的语意特征，并未规定实作细节。因此，vptr 放在 object 顶端或尾端，或甚至各个 data members 之间，都是可以的。不过，放在 data members 之间会使问题过于复杂，通常编译器不会如此做。图 2.3 显示 Microsoft C++ 编译器把 vptr 放在 object 顶端。

在贝尔实验室的 CFront 编译器中，设计者 Lippman 把 vptr 放在 object 的尾端。这么做的好处是可以支援对原本的 C struct 的继承。例如：

struct baseStruct { int d1, d2;

};

class derivedClass : public baseStruct { public:

virtual void vfunc();

// ... protected:

int d3;

};

如果 vptr 放在 object 尾端， *derivedClass* object 之中就能够保有 *baseStruct*

subobject 的完整型态。

*derivedClass*

object

*baseStruct*

subobject

|  |
| --- |
| data1 |
| data2 |
| data3 |
| vptr |

**109**

多型与虚拟（*Polymorphism in C++*）

但是自从 C++ 2.0 开始支援虚拟继承以及抽象基础类别（abstract base classes）之后，由于物件导向的程序观念大为抬头，愈来愈多的 C++ 编译器不再亟思与 C 兼容（真是成了龙形再现爪呀3），因此喜欢把 vptr 改放在object 的顶端。

### vtbl 的格式

Vtbl 内部存放的都是函式指标，如图 2.3 所示。但是自从 C++ 支援 RTTI 之后，情况有些改变。

想一想，能够在执行时期做型别鉴定的工作，必然是因为每一个 classes 都保持有一份 *type\_info* 信息（请参考第１章 p.86）。有些编译器把这份信息放在 vtbl 的第一笔项目（对表格而言，编号为 0）之中。这种情况下 vtbl 如下图所示：

VTBL

*type\_info*

class object

virtual function’s implementation

virtual function’s implementation

我们可以从图 2.3 看出，Visual C++ 显然并不是这一类编译器。至于Visual C++

之中的 object 如何与其 class *typeinfo* 搭上关系，我手上并没有这方面的资料。

1. 中國雲南省諺語

**110**

第２章 C++ 物件模型（C++ Object Model）

## Member Functions 的布局

有了 vptr，接下来我想验证 vtbl 的内容，并确定它是否真的指向各个 virtual functions。假设目前的我并不知道 vtbl 中有几笔资料，所以我总是从 vtbl 起头处开始，列出 6 笔来瞧瞧。vtbl 的内容取得，稍微需要一些技巧（主要在于指标型别的转换）：

ClassA a; ClassB b; ClassC c;

long\* v; // v is the address of vptr

long\* vtbl0; // vtbl0 is the address of vtbl[0]

v = (long \*)&a; vtbl0 = (long\*)(\*v);

cout << vtbl0 << endl; for (int i=0; i< 6; i++)

printf("ClassA vtbl[%d]=%#x \n", i, \*(vtbl0+i));

v = (long \*)&b; vtbl0 = (long\*)(\*v);

cout << vtbl0 << endl; for (i=0; i< 6; i++)

printf("ClassB vtbl[%d]=%#x \n", i, \*(vtbl0+i));

v = (long \*)&c; vtbl0 = (long\*)(\*v);

cout << vtbl0 << endl; for (i=0; i< 6; i++)

printf("ClassC vtbl[%d]=%#x \n", i, \*(vtbl0+i));

获得结果如下（已显示于图 2.3 中）。每一行程序输出后面的注解表示该笔内容所占用的存储器位址（注意各组输出之中有部份重叠）：

0x00409004

ClassA vtbl[0]=0x401ed0 // 0x00409008

ClassA vtbl[1]=0x401f10 // 0x0040900C

ClassA vtbl[2]=0 // 0x00409010

ClassA vtbl[3]=0x409650 // 0x00409014

**111**

多型與虛擬（*Polymorphism in C++*）

|  |  |  |
| --- | --- | --- |
| ClassA vtbl[4]=0x401f80 | // | 0x00409018 |
| ClassA vtbl[5]=0x401f10 | // | 0x0040901C |
| 0x00409014 |  |  |
| ClassB vtbl[0]=0x401f80 | // | 0x00409018 |
| ClassB vtbl[1]=0x401f10 | // | 0x0040901C |
| ClassB vtbl[2]=0 | // | 0x00409020 |
| ClassB vtbl[3]=0x4096a0 | // | 0x00409024 |
| ClassB vtbl[4]=0x401ff0 | // | 0x00409028 |
| ClassB vtbl[5]=0x401f10 | // | 0x0040902C |
| 0x00409024 |  |  |
| ClassC vtbl[0]=0x401ff0 | // | 0x00409028 |
| ClassC vtbl[1]=0x401f10 | // | 0x0040902C |
| ClassC vtbl[2]=0 | // | 0x00409030 |
| ClassC vtbl[3]=0 | // | 0x00409034 |
| ClassC vtbl[4]=0x8 | // | 0x00409038 |
| ClassC vtbl[5]=0x4096f0 | // | 0x0040903C |

看来 vtbl 似乎是以 0 为结束符号4。其中每一笔资料（一个函式指标），根据***Inside The C++ Object Model*** 5 一书的说法，都应该指向一个 virtual functions 实体。是否真是如此？我得验证一下。

如果我能取得所有 classes member functions 的起始位址，而其中的 virtual functions 位址出现在 vtbl 的适当位置中，那就可以验明正身。取出函式位址， 需要一些不寻常的语法技巧，以下连续三个动作，请注意其语法：

1. void (ClassA::\***pmfa**)(); 意思是宣告 *pmfa* 为一个 *ClassA* member function pointer。
2. **pmfa** = ClassA::func1; 意思是令 *pmfa* 等于 *ClassA*::*func1()* 函式位址。请注意， 在 C/C++ 语言中， 函式名称即函式位址。
3. 不过，在我所做的更大型 classes hierarchy 测试中，我发现有时候（通常是最末端 class 的） vtbl 尾端并非以 0 结束。这一部份真象如何，必须依赖编译器公开其实作方法，我们才能知道。
4. ***Inside The C++ Object Mode***（***l*** Lippman/Addison Wesley/1996），中译本为 深度探索 C++ 物件模型（侯俊杰译/碁峰/1998）

#### 112

第２章 C++ 物件模型（C++ Object Model）

1. (a.\***pmfa**)(); 意思是透过 object *a*，呼叫 *pmfa*（ *ClassA* 的 member function pointer） 所指向的函式实体。

根据上面三个语法技巧，我利用以下程序码取得所有 classes member functions 的起始位址（每一行程序码后面的注解代表输出的结果）：

void (ClassA::\***pmfa**)(); // pmfa is a ptr to ClassA member function ClassA a;

pmfa = ClassA::func1;

printf("ClassA::func1=%#x \n", pmfa); // ClassA::func1=0x401da0 (a.\*pmfa)(); // ClassA::func1

pmfa = ClassA::func2;

printf("ClassA::func2=%#x \n", pmfa); // ClassA::func2=0x401de0 (a.\*pmfa)(); // ClassA::func2

pmfa = ClassA::vfunc1;

printf("ClassA::vfunc1=%#x \n", pmfa); // ClassA::vfunc1=0x402030 (a.\*pmfa)(); // ClassA::vfunc1

pmfa = ClassA::vfunc2;

printf("ClassA::vfunc2=%#x \n", pmfa); // ClassA::vfunc2=0x402040 (a.\*pmfa)(); // ClassA::vfunc2

void (ClassB::\***pmfb**)(); // pmfb is a ptr to ClassB member function ClassB b;

pmfb = ClassB::func2;

printf("ClassB::func2=%#x \n", pmfb); // ClassB::func2=0x401e20 (b.\*pmfb)(); // ClassB::func2

pmfb = ClassB::vfunc1;

printf("ClassB::vfunc1=%#x \n", pmfb); // ClassB::vfunc1=0x402030 (b.\*pmfb)(); // ClassB::vfunc1

pmfb = ClassB::vfunc2;

printf("ClassB::vfunc2=%#x \n", pmfb); // ClassB::vfunc2=0x402040 (b.\*pmfb)(); // ClassA::vfunc2

void (ClassC::\***pmfc**)(); // pmfc is a ptr to ClassC member function ClassC c;

#### 113

多型與虛擬（*Polymorphism in C++*）

pmfc = ClassC::func2;

printf("ClassC::func2=%#x \n", pmfc); // ClassC::func2=0x401e60 (c.\*pmfc)(); // ClassC::func2

pmfc = ClassC::vfunc1;

printf("ClassC::vfunc1=%#x \n", pmfc); // ClassC::vfunc1=0x402030 (c.\*pmfc)(); // ClassC::vfunc1

pmfc = ClassC::vfunc2;

printf("ClassC::vfunc2=%#x \n", pmfc); // ClassC::vfunc2=0x402040 (c.\*pmfc)(); // ClassA::vfunc2

整理上述的程序码与执行结果，得图 2.5。

图 2.5 Class member functions 的函式实体位址， 及透过函式指标所唤起的结果。

|  |  |  |  |
| --- | --- | --- | --- |
| Member Functions | 程序中取得的  函式位址 | 透过函式指标所呼  叫的结果（文字输出） | 注 |
| ClassA::func1 | 0x401da0 | ClassA::func1 | non-virtual |
| ClassA::func2 | 0x401de0 | ClassA::func2 | non-virtual |
| ClassA::vfunc1 | 0x402030 \* | **ClassA**::vfunc1 \* | virtual |
| ClassA::vfunc2 | 0x402040 \* | ClassA::vfunc2 | virtual |
| ClassB::func2 | 0x401e20 | ClassB::func2 | non-virtual |
| ClassB::vfunc1 | 0x402030 \* | **ClassB**::vfunc1 \* | virtual |
| ClassB::vfunc2 | 0x402040 \* | ClassA::vfunc2 | virtual |
| ClassC::func2 | 0x401e60 | ClassC::func2 | non-virtual |
| ClassC::vfunc1 | 0x402030 \* | **ClassC**::vfunc1 \* | virtual |
| ClassC::vfunc2 | 0x402040 \* | ClassA::vfunc2 | virtual |

* + 注意：灰色部份的位址，竟然两两相同，但呼叫结果却又不完全相同（稍后另有讨论）

当然你也可以直接利用函式名称取得函式位址，因为 C/C++ 中的函式名称即为函式位址。因此也可以这么写：

#### 114

第２章 C++ 物件模型（C++ Object Model）

printf("ClassA::func1=%#x \n", ClassA::func1); // 0x401da0 printf("ClassA::func2=%#x \n", ClassA::func2); // 0x401de0 printf("ClassA::vfunc1=%#x \n", ClassA::vfunc1); // printf("ClassA::vfunc2=%#x \n", ClassA::vfunc2); // 0x402040 printf("ClassB::func2=%#x \n", ClassB::func2); // 0x401e20 printf("ClassB::vfunc1=%#x \n", ClassB::vfunc1); // 0x402030 printf("ClassB::vfunc2=%#x \n", ClassB::vfunc2); // 0x402040 printf("ClassC::func2=%#x \n", ClassC::func2); // 0x401e60 printf("ClassC::vfunc1=%#x \n", ClassC::vfunc1); // 0x402030 printf("ClassC::vfunc2=%#x \n", ClassC::vfunc2); // 0x402040

0x402030

结果与图 2.5 完全一样。

图 2.5 有两点令人大惑不解：

1. 不同 classes 的 *vfunc1()* 竟然有着相同的位址（ *vfunc2()* 也是如此）。而呼叫之后，却又有着不同的结局（ 如图 2.5 的 \* 部份）。虽然，以我们对 virtual functions 的认知而言（ 第１章 p.57 曾提过）， 这个程序对于 virtual function 的唤起结果是合理的， 但为什么相同的位址， 可以唤起不同的函式呢？
2. 没有一个 virtual functions 的位址出现于图 2.3 的 vtbl 之中。这与我们的推测不符合。

事实的真象究竟如何呢？这一节在取得函式位址的过程中，我使用了 "pointer to member function" 这个不寻常的语法。让我先对 pointer to members 多做一些解释，再回到主题。

Pointer to members 分为指向 data members 和指向 member functions 两种。

#### 115

多型與虛擬（*Polymorphism in C++*）

#### Pointer to Data Members

众所周知， data members 是因为 object 的实现才存在（ 除非是 static data members，它属于 class 范筹而非 object 范筹），所以如果我这么做：

ClassA a;

cout << &(a.m\_data1) << endl; // 0x0063FDF0 cout << &(a.m\_data2) << endl; // 0x0063FDF4

很明显是取得 object *a* 的两个 data members 在位址空间中的位址。但如果在尚未产生任何 object 实体之前，下面这样做会得到什么结果呢：

printf("&(ClassA::m\_data1)=%p \n", &(ClassA::m\_data1)); printf("&(ClassA::m\_data2)=%p \n", &(ClassA::m\_data2)); printf("&(ClassB::m\_data3)=%p \n", &(ClassB::m\_data3)); printf("&(ClassC::m\_data1)=%p \n", &(ClassC::m\_data1)); printf("&(ClassC::m\_data4)=%p \n", &(ClassC::m\_data4));

注意，上例输出时不能使用 cout operator<<，否则会出现类似这样的编译错误（以Visual C++ 为例）：

error C2679: binary '<<' : no operator defined which takes a right-hand operand of type

'int ClassC::\*' (or there is no acceptable conversion) (new behavior; please see help)

答案竟然是 object 实体布局中的 offset ：

&(ClassA::m\_data1)=00000004 &(ClassA::m\_data2)=00000008 &(ClassB::m\_data3)=0000000C &(ClassC::m\_data1)=00000010 &(ClassC::m\_data4)=00000014

（注意：此为 Visual C++ 的结果）

这些 offset 值必须依附（系结）在一个 object 实体位址上，才有真正的意义。也就是说，当我产生一个

offset 00

04

|  |
| --- |
| vptr |
| ClassA::m\_data1 |
| ClassA:: m\_data2 |
| ClassB:: m\_data3 |
| ClassC:: m\_data1 |
| ClassC:: m\_data4 |

08

0C

10

14

ClassC object layout

object，各 data members 的位址即是 object 位址加上对应的 offset 值。

#### 116

第２章 C++ 物件模型（C++ Object Model）

#### Pointer to Member Functions

既然 pointer to data members 的实值意义只是个 offset，我们很自然推想 pointer to member functions 的实值意义会不会也只是个 offset 呢？Lippman 在其 ***Inside The C++ Object Model*** 第四章提到：

对一个 virtual member function 取其函式位址，获得的将是一个 vtbl 索引值。

换句话说，假设下面这段程序码：

class CPoint

{

public:

virtual ~CPoint(); float x();

float y();

virtual float z();

...

};

class CPoint3d : public CPoint

{

public:

virtual float z();

...

};

CPoint’s vtbl

#0



#1

#2

type\_info

~Point() z()

那么当我们取 *CPoint::~CPoint()* 的函式位址，应该得到 16；取 *CPoint::z()* 的函式位址，应该得到 2。当我们以 pointer to member functions 来唤起 virtual function，像这样：

typedef float (CPoint::\*POINTERTOMEMFUNC)(); POINTERTOMEMFUNC pmf;

pmf = &CPoint::z; // 现在 pfm 应为 2，因为 Point::z() 是 virtual。

CPoint \*ptr = new CPoint3d; // 多型：base ptr 指向 derived object.

1. Lippman 说 vtbl 的 #0 元素应该放置 RTTI 所需的 *type\_info* class object。不过并不是每个编译器都如此。从图 2.3 可知，显然 Microsoft 编译器就不是如此。

**117**

多型与虚拟（*Polymorphism in C++*）

ptr->z(); // (1) 根据多型的游戏规则，这应该唤起 CPoint3d::z() (ptr->\*pmf)(); // (2) 根据多型的游戏规则，这应该唤起 CPoint3d::z()

第一个式子毫无问题会唤起 *CPoint3d::z()*（注意 *z()* 是 virtual function）。第二个式子在编译时期会被转化为如下形式：

(\* ptr->vptr[(int)pmf])(ptr); // pmf == 2

于是也唤起 *CPoint3d::z()*。但是你知道，我也可以拿 *pmf* 来放置 nonvirtual functions *x()* 的位址，因为它和 *z()* 的函式型态是一样的。所以我可以这样做：

pmf = &CPoint::x; // 此时 pmf 不为 vtbl 索引值，而是一个真正的函式位址

(ptr->\*pmf)(); // (3) 这和多型无关，应该唤起 Point::x()

为了满足这两种情况，很显然 *pmf* 必须有两个「分身」（一个代表存储器位址， 一个代表 vtbl 索引），并且可以被编译器区别意义，才能分别在 (2) 和 (3) 式子中扮演正确的角色。

Lippman 的 cfront 2.0 pre-release 使用一个技巧如下，来解决上述的问题。一旦面对这样的唤起动作：

(ptr->\*pmf)(); // 可能是上述的 (2)，也可能是上述的 (3)

编译器会把它转化为：

((( int ) pmf ) & ~127 ) // pmf 不大于 127

? // non-virtual invocation

(\*pmf)(ptr) // pmf 被视为函式指标

: // virtual invocation

(\* ptr->vptr[(int)pmf](ptr)); // pmf 被视为 vtbl 索引

这种简单的解决方法，必须假设在继承体系中最多只有 128 个 virtual functions， 不甚理想。再者，由于多重继承的引入，我们需要更弹性的实作技巧。

#### 118

第２章 C++ 物件模型（C++ Object Model）

**Bjarne Stroustrup 的解法： *\_\_mptr* 结构**

为解决这个问题，Stroustrup 设计出这样一个结构体：

// 用以支援多重继承情况下的 pointer to member functions struct mptr {

int delta;

};

int index; union {

ptrtofunc faddr; int v\_offset;

};

struct *mptr*

or

|  |
| --- |
| delta |
| index |
| faddr |

struct *mptr*

|  |
| --- |
| delta |
| index |
| v\_offset |

并且令 pointer to member function 其实不为真正的函式位址，而是上述结构体。当前述的 *pmf* 指向 virtual member function 时，结构体内的 *index* 内含 vtbl 索引；当 *pmf* 指向non-virtual member function 时，结构体内的 *faddr* 内含其真实位址而 *index* 被设为 -1。此外，*delta* 栏位表示 *this* 指标的 offset 值（稍后我会谈到这个主题），*v\_offset* 栏位放的是 virtual base class 的 vptr 位置（如果 vptr被编译器放在 class object 的第一个栏位，*v\_offset* 就没有作用了）。*delta* 和*v\_offset* 两个栏位只有在多重继承或虚拟继承的情况下才有必要性（稍后介绍）。

在此模型之下，这样的唤起动作：

(ptr->\*pmf)();

会变成：

( pmf.index < 0 )

? // non-virtual invocation

(\*pmf.faddr)(ptr+pmf.delta) // ptr 调整后成为 this 指标

: // virtual invocation

(\* ptr->vptr[pmf.index](ptr+pmf.delta)); // ptr 调整后成为 this 指标

图 2.6 表现出这种行为。

#### 119

多型與虛擬（*Polymorphism in C++*）

##### *pmf* is a pointer to member function



vptr

vtbl

if *pmf* present a pointer to a virtual function

struct *mptr*

)

*f* index

faddr

(2)

if *pmf* present a pointer to a non-virtual function

(1

delta

virtual member function’s

implementation

non-virtual member function’s implementation

*pm*

目前暂不考虑虚拟继承， 所以 *mptr* 的第三个栏位解释为 faddr 而非 v\_offset

圖 2.6 Bjarne Stroustoup 设计的结构， 用来解决 pointer to member function 的问题。

#### Microsoft 的解法： vcall thunk

问题是，Bjarne Stroustrup 的解法带来一些额外负担，因为每一个 member functions 的唤起动作都得做上述检查（检查是否为 virtual），影响程序执行效率。Microsoft 不做如此设计，而是导入一个所谓的 vcall thunk，使 *pmf* 代表一个 non-virtual function 时，*faddr* 内含真正的函式位址，而当 *pmf* 代表一个 virtual function 时， *faddr* 内含 vcall thunk 位址。Vcall thunk 会负责选出相关的 vtbl 中的适当 slot 并呼叫之。于是，virtual 或 non-virtual 的 member functions 唤起动作就可以一致。图 2.7 表现出这种行为。

事实上有许多编译器都针对不同的 classes 特征（多重继承啦、虚拟继承啦）提供不同结构的 pointer to member functions。Microsoft 就提供了三种7：

1. 这个说法来自 ***Inside The C++ Object Model*** 第四章。但是资料有限，我没有办法以我手上的 Visual C++ 编译器实证出来。

#### 120

第２章 C++ 物件模型（C++ Object Model）

1. 针对单一继承。指标实体内含实际的函式位址（ 或 vcall thunk 位址）
2. 针对多重继承。指标实体内含 *faddr* 和 *delta* 两个栏位。
3. 针对虚拟继承。指标实体内含四个栏位。

*pmf*

*pmf* is a pointer to member function

if *pmf* point to a non-virtual function

if *pmf* point to a virtual function

member function’s implementation

**vc thu**

**all nk**

圖 2.7 Microsoft 對於 pointer to member function 的解法

#### 重回主題

现在让我们再看一次图 2.5 的内容（注意，灰色部份的位址两两相同）：

|  |  |  |  |
| --- | --- | --- | --- |
| Member Functions | 函式位址（？） | 函式指标呼叫结果 | 注 |
| ClassA::func1 | 0x401da0 | ClassA::func1 | non-virtual |
| ClassA::func2 | 0x401de0 | ClassA::func2 | non-virtual |
| ClassA::vfunc1 | 0x402030 | ClassA::vfunc1 | virtual |
| ClassA::vfunc2 | 0x402040 | ClassA::vfunc2 | virtual |
| ClassB::func2 | 0x401e20 | ClassB::func2 | non-virtual |
| ClassB::vfunc1 | 0x402030 | ClassB::vfunc1 | virtual |
| ClassB::vfunc2 | 0x402040 | ClassA::vfunc2 | virtual |
| ClassC::func2 | 0x401e60 | ClassC::func2 | non-virtual |
| ClassC::vfunc1 | 0x402030 | ClassC::vfunc1 | virtual |
| ClassC::vfunc2 | 0x402040 | ClassA::vfunc2 | virtual |

#### 121

多型與虛擬（*Polymorphism in C++*）

ClassC::vfunc1() implementation

其中所有的 virtual function 位址（灰色部份），根据上一节的说法，我猜想应该都是 vcall thunk 的位址吧（我用的是Microsoft 编译器）。这样才能够合理解释为什么「同一个位址竟然能够唤起不同的函式实体」。这些 vcall thunk 的位址， 都没有出现在各个 class 的 vtbl 之中，似乎也还合理，因为 vtbl 中的元素所代表的，恐怕应该是 virtual functions 的真正函式位址（真是如此吗？请看本章稍后利用除错器的观察心得）。

注意：以上纯为个人推论，因为我手上没有 Microsoft Visual C++ 的实作技术资

料。这样的推论，根据各项知识基础的交叉辅证，尚称合理（I hope so ）。

圖 2.8 是 vcall thunk 的示意圖。

pmfa = &(ClassA::vfunc1);

0x402030

pmfb = &(ClassB::vfunc1);

ClassA, ClassB, ClassC 之 vtbl 内容请参考图2.3

Vcall thunk 负责选出相关的 vtbl 中的适当 slot 并呼叫之。

**vcall thunk**

pmfc = &(ClassC::vfunc1);

ClassA::vfunc1() implementation

ClassB::vfunc1() implementation

pmfa = &(ClassA::vfunc2);

0x402040

pmfb = &(ClassB::vfunc2);

ClassA, ClassB, ClassC 之 vtbl 内容请参考图2.3

Vcall thunk 负责选出相关的 vtbl 中的适当 slot 并呼叫之。

**vcall thunk**

pmfc = &(ClassC::vfunc2);

ClassA::vfunc2() implementation

圖 2.8 Microsoft 的 vcall thunk 扮演的角色

#### 122

第２章 C++ 物件模型（C++ Object Model）

#### 我的困扰 - 函式指标的转型

欲做最后验证，可将图 2.3 的 vtbl 项目内容当做是函式指标，试试看能否唤起图中所指之函式（可从函式输出字符串判断之）。但，我找不到合适的语法来写出这样的需求。我曾尝试如此（*ClassA*, *ClassB*, *ClassC* 请见p.102）：

#0001 typedef void (ClassA::\*ClassAMemFuncPtr)(void); #0002

#0003 void main()

#0004 {

#0005 ClassA a;

#0006 long\* v; // *v* will be used to point to vptr

#0007 long\* vtbl0; // *vtbl0* will be used to point to vtbl[0] #0008

#0009 v = (long\*)&a;

#0010 printf("vptr = %#x \n", \*v); // 0x00409004 #0011 vtbl0 = (long\*)(\*v);

#0012 printf("vtbl[0] = %#x \n", \*vtbl0); // 0x00401ED0 #0013

#0014 // try to invoke vtabl's entry (ptr to v-mem-func) directly #0015 ClassAMemFuncPtr pmfa = (ClassAMemFuncPtr)(\*vtbl0); // error #0016 (a.\*pmfa)();

#0017 }

但是 L15 过不了编译器这一关，会出现错误讯息如下：

error C2440: 'type cast' : cannot convert from 'long' to 'void (ClassA::\*)(void)' There are no conversions from integral values to pointer-to-member values

改成这样也不行（注意 L6 和 L7 的资料型别）：

#0001 typedef void (ClassA::\*ClassAMemFuncPtr)(void); #0002

#0003 void main()

#0004 {

#0005 ClassA a;

#0006 void\* v; // *v* will be used to point to vptr #0007 void\* vtbl; // *vtbl* will be used to point to vtbl #0008

#0009 v = (void\*)&a;

#0010 printf("vptr = %#x \n", \*(long\*)v);

#### 123

多型與虛擬（*Polymorphism in C++*）

#0011 vtbl = (void\*)(\*(long\*)v);

#0012 printf("vtbl[0] = %#x \n", \*(long\*)vtbl); #0013

#0014 // try to invoke vtbl's entry (ptr to v-mem-func) directly #0015 ClassAMemFuncPtr pmfa = (ClassAMemFuncPtr)(\*vtbl); // error #0016 (a.\*pmfa)();

#0017 }

L15 会出现编译错误如下：

error C2100: illegal indirection

C++ 编译器对型别的检验非常严格，似乎没有任何办法可以把一个 long（虽然我们明明知道它是一个指标值）强制转型为函式指标（不论是 member function 或是一般的 global function），除非那个 long 数值原先就是从一个函式指标转型过来的：

void func(void)

{ cout << "hello" << endl; }

typedef void (\*FuncPtr)(void); void main()

{

long l = (long)func; // 函式名称就是函式指标

FuncPtr pfn = (FuncPtr)l;

(\*pfn)(); // 执行结果：输出 "hello"

}

如果你有什么好构想可以解决这个问题，请告诉我（[jjhou@ccca.nctu.edu.tw](mailto:jjhou@ccca.nctu.edu.tw)），谢谢。

注：本题经多位读者来函指点，已顺利解决。以下采用陈星光先生的意见，将上

页第一种作法修改如下，即可顺利达到目的：

#0015 ClassAMemFuncPtr pfna = (ClassAMemFuncPtr)(\*(ClassAMemFuncPtr\*)vtbl0);

此行意义：vtbl0 是个「函式指标的指标」。

#### 124

第２章 C++ 物件模型（C++ Object Model）

## 经过继承的洗礼

容我先把继承的观念再复习一次：

所谓继承，不仅继承了 base class 的各项 data members，也继承了 base class 的各个 member functions：

1. 对于 data members 的继承， 是指这些 members 在 derived object 和

base object 中各有一份实体（ static members 除外）；

1. 对于 member functions 的继承，则与「多长出一份实体」的观念无关， 而是指继承了对 base class's member functions（ 包括 non-virtual、virtual、static） 的呼叫权利。
2. Static data members 也可以继承，但其观念类似于对 member functions 的继承： 继承得到的是存取权利， 而非在 derived object 中多一份实体空间。

**顺带一提， Static Members 的继承**

下面这个简单的例子可以说明 static 的一些继承疑点：

#0001 class A

#0002 {

#0003 public: // use public is convenient for access #0004 double m\_total;

#0005 static double m\_rate; #0006 public:

#0007 static void setRate(double newRate) { m\_rate = newRate; } #0008 };

#0009

#0010 class B : public A #0011 {

#0012 };

#0013

#0014 double A::m\_rate = 0.0075; // 设立 static data member 的初值

#0015

#0016 void main()

#### 125

多型與虛擬（*Polymorphism in C++*）

#0017 {

#0018 cout << sizeof(A) << endl; // 8 #0019 cout << sizeof(B) << endl; // 8 #0020

#0021 cout << A::m\_rate << endl; // 0.0075 #0022

#0023 A::setRate(0.0072); // (1) call static member function directly #0024 cout << A::m\_rate << endl; // 0.0072

#0025

#0026 A a;

#0027 a.setRate(0.0070); // (2) call static member function via object #0028 cout << a.m\_rate << endl; // 0.007

#0029

#0030 B b;

#0031 b.setRate(0.0068);

#0032 cout << A::m\_rate << endl; // 0.0068 #0033 cout << b.m\_rate << endl; // 0.0068 #0034

位址都相同

#0035 cout << &(A::m\_rate) << endl; // 0x0040B040 #0036 cout << &(B::m\_rate) << endl; // 0x0040B040 #0037 cout << &(a.m\_rate) << endl; // 0x0040B040 #0038 cout << &(b.m\_rate) << endl; // 0x0040B040

#0039

#0040 cout << &(a.m\_total) << endl; // 0x0064FDF0 #0041 cout << &(b.m\_total) << endl; // 0x0064FDE8

位址各不同

#0042 }

0x0064FDF0



8

class A object

0x0040B040

double m\_total

A::m\_rate

0x0064FDE8

###### 8

class B object

double m\_total

###### accessable

A::setrate()

member functions 以及static data member 都不會佔據 object 實體空間

对于 base class 的 member functions 和 static data members， derived class 继承的是其存取权力，而不是复制一份存储器空间。

#### 126

第２章 C++ 物件模型（C++ Object Model）

下面是程序的重点说明：

* L14： 对于 static data members 一定要有明显的设立初值动作。
* L18 ： 显然 class *A* object 中没有 *m\_rate* 的存在， 只有 *m\_total*（ 8 bytes）。
* L19： 显然 class *B* object 中也没有 *m\_rate* 的存在， 只有继承得来的

*m\_total*（ 8 bytes）。

* L21：在尚未产生任何 class *A* object 的时候，就已经可以使用 class *A*

的 static data member。

* L23：在尚未产生任何 class *A* object 的时候，就已经可以呼叫 class *A*

的 static member functions。

* L33：透过 class *B* object 可以取用 class *A* 的 static member functions。
* L35~L38： 从 class *A* 或 class *B* 的角度， 或是从 object *a* 或 object *b* 的角度，都可以取用 class *A* 的 static data members，它们其实是同一份实体（ 位址都相同）。
* L40~L41：一般的 non-static data members，会在 base object 和 derived object 中各有一份实体， 所以位址不相同。

单一继承（Single Inheritance）的物件模型

所谓单一继承，是指每一个 class 的direct base class 只能有一个。继承的深度没有限制。

单一继承后的物件模型，我们可以从先前对图 2.1 ~ 图 2.3 的种种讨论获得认识。

**127**

多型與虛擬（*Polymorphism in C++*）

### 多重继承（Multiple Inheritance）的物件模型

所谓多重继承，是指每一个 class 可以拥有一个以上的direct base classes。继承的深度亦没有限制。

多重继承所制造的最大迷思（亦即物件模型的设计困难点）就是，如何在一个derived object 中拥有多个 direct base classes 的 subobjects，其中的难处在于对vptr 的处理。毕竟，C++ 语言保证，出现在 derived class object 中的 base class subobject 有其完整原样性。

此外，当多型（polymorphism）情况发生，例如以一个 base type pointer 指向一个derived type object 时，该如何移动 *this* 指标使能够透明化（对程序员而言）地指向正确的 subobject，亦是一个关键点。这可区分为自然多型（natural polymorphism） 和非自然多型（unnatural polymorphism）两种情况来讨论。

**128**

第２章 C++ 物件模型（C++ Object Model）

#### 自然多型（ natural polymorphism）

如果没有 virtual functions，class object 中就不会产生 vptr。那么不管任何继承情况，到目前为止我们所讨论的 C++ 物件模型都能够维持 base class subobject 的完整原样性。

现在我以一个 base type pointer 指向一个 derived type object：

class CPoint3d : public CPoint2d { ... }; CPoint2d\* p2d = new CPoint3d;

// 多型：以 base type pointer 指向 derived type object

如果 base class 本身有 virtual function(s)，object 布局如下图，依然保有 subobject 的完整原样性8，base type pointer（*this* 指标）直接落在 base class subobject 身上， 不需要编译器多费力气移动它，这便是所谓的自然多型（natural polymorphism）：

base class object derived class object

this

vptr

|  |
| --- |
| vptr |
|  |
|  |

指向 derived class object 起头，也指向 base class subobject 起头。

#### 非自然多型（ unnatural polymorphism）

##### base class subobject

1. 所谓保有 subobject 的完整原样性，是指架构（格式）相同，不是指内容必须完全一样。举个例子，base class object 的 vptr 和 base class subobject 的 vptr 就绝对不可能一致，一致的话就糟了。这也就是为什么不能对于 polymorphic class 采用 bitwise copy 的缘故。我曾在第１章

p.34 谈过 bitwise copy。

#### 129

多型與虛擬（*Polymorphism in C++*）

假设 base class 本身没有 virtual function(s) 而 derived class 有，当 vptr 位在object 顶端（目前许多编译器的确是如此设计），natural polymorphism 的局面就会被打破，编译器必须介入，在必须存取 subobject members 时，移动 *this* 指标， 才能保证 polymorphism 的成功，如下图所示：

base class object derived class object

this

|  |
| --- |
| vptr |
|  |
|  |

this

base class subobject

#### 多重繼承

多重继承的复杂度就像上述的 unnatural polymorphism 一样。常常需要编译器的介入。假设 classes 的继承状况如下：

class CPoint3d : public CPoint2d, public CVertex { ... };

CPoint3d

CVertex

CPoint2d

CPoint2d\* p2d = new CPoint3d; // (1) CVertex\* pv = new CPoint3d; // (2)

如果 *CPoint2d* 没有 virtual function，那么 (1) 和 (2) 两个式子都是 unnatural polymorphism。如果 *CPoint2d* 拥有 virtual function(s)，那么 (1) 式是 natural polymorphism 而 (2) 式是 unnatural polymorphism。

让我举一个实例，在程序中模拟 C++ 编译器对于多重继承情况下所发生的

unnatural polymorphism 的处理方式。这个例子的 classes 体系如图 2.9 所描述。

#### 130

第２章 C++ 物件模型（C++ Object Model）

圖 2.9 多重繼承的 classes 體系

#0001 class CPoint2d

#0002 {

#0003 public:

#0004 CPoint2d( float x = 0.0, float y = 1.0 ) #0005 : \_x( x ), \_y( y ) { };

#0006

#0007 float x() { return \_x; } #0008 float y() { return \_y; } #0009

CVertex3d

CPoint3d

CVertex

CPoint2d

#0010 void x( float newX ) { \_x = newX; } #0011 void y( float newY ) { \_y = newY; } #0012

#0013 // 預留 z 的彈性空間（目前什麼也沒做）

#0014 virtual float z() { return 0.0; } #0015 virtual void z( float ) { }

#0016

#0017 public: // protected: is better #0018 float \_x, \_y;

#0019 };

#0020

#0021 class CPoint3d : public CPoint2d #0022 {

#0023 public:

#0024 CPoint3d( float x = 0.0, float y = 1.0, float z = 2.0 ) #0025 : CPoint2d( x, y ), \_z( z ) { };

#0026

#0027 virtual float z() { return \_z; }

#0028 virtual void z( float newZ ) { \_z = newZ; } #0029

#0030 public: // protected: is better #0031 float \_z;

#0032 };

#0033

#0034 class CVertex

#0035 {

#0036 public:

#0037 virtual void foo() { } ; #0038

#0039 public: // protected: is better #0040 CVertex \*next;

#0041 };

#0042

#0043 class CVertex3d : public CPoint3d, public CVertex #0044 {

#### 131

多型與虛擬（*Polymorphism in C++*）

#0045 public:

#0046 CVertex3d(float m = 5.0) { mumble = m; } #0047

#0048 public: // protected: is better #0049 float mumble;

#0050 };

首先询问 class 的大小：

cout << sizeof(CPoint2d) << endl; // 12 cout << sizeof(CPoint3d) << endl; // 16 cout << sizeof(CVertex) << endl; // 8 cout << sizeof(CVertex3d) << endl; // 28

由于每一个 classes 都是 polymorphic classes（因为都有 virtual functions），所以每一个 class 的大小都是 data members（包括继承得来的）加上一个 vptr 的总和。

但 *CVertex3d* 是多重继承，比较特殊。如果它的 direct base classes 中有 n 个polymorphic classes，那么 *CVertex3d* object 内就有 n 个 vptrs。本例 n 为 2，所以 *CVertex3d* 有两个 vptrs，对应至两个vtbls（请参考图 2.10）。

注意，如果本例的 *CVertex::foo()* 不为 virtual，于是 *CVertex* 不为 polymorphic class，那么上面所说的 n 为 1，而各个 classes 大小如下：

|  |  |  |
| --- | --- | --- |
| cout << sizeof(CPoint2d) | << endl; | // 12 |
| cout << sizeof(CPoint3d) | << endl; | // 16 |
| cout << sizeof(CVertex) | << endl; | // 4 |
| cout << sizeof(CVertex3d) | << endl; | // 24 |

現在讓我們看看 *CVertex3d* object 的佈局：

CVertex3d av3d;

// check the address of data members

cout << &av3d << endl; // 0x0063FDC4

cout << &(av3d.\_x) << endl; // 0x0063FDC8

cout << &(av3d.\_y) << endl; // 0x0063FDCC

cout << &(av3d.\_z) << endl; // 0x0063FDD0

cout << &(av3d.next) << endl; // 0x0063FDD8

#### 132

第２章 C++ 物件模型（C++ Object Model）

cout << &(av3d.mumble) << endl; // 0x0063FDDC

|  |  |
| --- | --- |
| // check the value of data members |  |
| printf("%f \n", \*(float\*)((char\*)&av3d + 4)); | // 0.000000 |
| printf("%f \n", \*(float\*)((char\*)&av3d + 8)); | // 1.000000 |
| printf("%f \n", \*(float\*)((char\*)&av3d + 12)); | // 2.000000 |
| printf("%#x\n", \*(long\* )((char\*)&av3d + 16)); | // 0x40a018 |
| printf("%#x\n", \*(long\* )((char\*)&av3d + 20)); | // 0x40b004 |
| printf("%f \n", \*(float\*)((char\*)&av3d + 24)); | // 5.000000 |

从上面的结果我已经可以断定 vptrs 的位址，现在准备开始把对应的 vtbls 内容挖出来看。这个技巧我们应该已经很熟悉了，前面已经操练过：

CVertex3d av3d;

void\* v; // v is the address of vptr void\* vtbl; // vtbl is the address of vtbl

v = (void\*)&av3d;

printf("1st vptr = %#x \n", \*(long\*)v); vtbl = (void\*)(\*(long\*)v);

for (int i=0; i< 6; i++)

printf("1st vtbl[%d]=%#x \n", i, \*((long\*)vtbl+i));

获得结果如下（第一个 vtbl）：

1st vptr = 0x40a020 1st vtbl[0]=0x401640 1st vtbl[1]=0x401620 1st vtbl[2]=0x4016b0 1st vtbl[3]=0x7830 1st vtbl[4]=0x7025 1st vtbl[5]=0

现在我把指标移动到第二个 vtbl，再次挖宝：

v = (void\*)((char\*)&av3d + sizeof(CPoint3d)); printf("2nd vptr = %#x \n", \*(long\*)v);

vtbl = (void\*)(\*(long\*)v); for (int i=0; i< 6; i++)

printf("2nd vtbl[%d]=%#x \n", i, \*((long\*)vtbl+i));

#### 133

多型與虛擬（*Polymorphism in C++*）

获得结果如下（第二个 vtbl）：

2nd vptr = 0x40a018 2nd vtbl[0]=0x4016b0 2nd vtbl[1]=0

2nd vtbl[2]=0x401640 2nd vtbl[3]=0x401620 2nd vtbl[4]=0x4016b0 2nd vtbl[5]=0x7830

运用同样的技巧，我再产生一个 *CPoint3d* object 和一个 *CPoint2d* object，并把其object 布局以及 vtbl 内容显示出来。所有结果整理于图 2.10。

0x0064FD74



a2d (*CPoint2d* object)

0x40A000

vptr

virtual functions implementation

0x0040A000 0x4015D0

0x0040A004

0x0040A008

vptr 0x0040A00C a3d (*CPoint3d* object) 0x0040A010 0x40A010 0x0040A014

0x0040A018

0x0040A01C

0x0040A020

0x0040A024

0x0040A028

0x401640

0x401620

0x4016B0

0x401640

0x4016B0

av3d (*CVertex3d* object) 0x0040A02C

0x40A020

1st vptr

this 指標需要調整

this 指標不需調整

*CVertex3d* 的

primary virtual table

0x40A018

*CVertex3d* 的

secondary virtual table

2nd vptr

多重繼承下的物件模型（以 Visual C++ 為例）

mumble (5.0)

next (0x40B004)

\_z (2.0)

\_y (1.0)

\_x (0.0)

\_z (2.0)

\_y (1.0)

\_x (0.0)

0x7830

0x401620

0

0

0

0x4015B0

\_y (1.0)

\_x (0.0)

void CVertex::foo(void)

void CPoint3d::z(float)

float CPoint3d::z(void)

void CPoint2d::z(float)

float CPoint2d::z(void)

0x0064FD78

0x0064FD7C

0x0064FD88

0x0064FD8C

0x0064FD90

0x0064FD94

0x0063FDC4

0x0063FDC8

0x0063FDCC

0x0063FDD0

0x0063FDD4

0x0063FDD8

圖 2.10 CVertex 3d classes 体系（ 图 2.9） 的 vptr 和 vtbl 布局

#### 134

第２章 C++ 物件模型（C++ Object Model）

从图 2.10 中可以看出，*CVertex3d* 有两个 vtbls。当我们以下列动作：

CVertex3d\* pv3d = new CVertex3d; // (1) CPoint3d\* p3d = new CVertex3d; // (2) CPoint2d\* p2d = new CVertex3d; // (3)

产生三个 objects，经由 *pv3d* 或 *p3d* 或 *p2d* 唤起 *CVertex3d's* virtual functions

的过程中，编译器都应该使用 primary vtbl 做为导引。而当我们以：

CVertex\* pv = new CVertex3d; // (4)

产生一个 objects，经由 *pv* 唤起 *CVertex3d*'s virtual functions 的过程中，编译器应该使用 secondary vtbl 做为导引。

你一定会奇怪，为什么 primary vtbl 中除了列有最左端 base class（本例为*CPoint3d*）的 virtual functions，还列有后继所有 direct base classes（本例为*CVertex*）的 virtual functions。这是因为当我们这么做：

CVertex3d\* pv3d = new CVertex3d; // (1)

編譯器用的是 primary vtbl；而我們可以在程式中透過 *pv3d* 這麼做：

pv3d->foo(); // 喚起 CVertex::foo()

所以 primary vtbl 中必須有些項目關係到 *CVertex::foo()*。

### 多重继承下 this 指标的调整

上面所讨论的 (1),(2),(3),(4) 不就是以 base type pointer 指向 derived type object 所发生的种种状况吗！由于 *this* 指标是沟通 member functions 和 data members 的桥梁，因此在某些情况下，编译器就必须调整 *this* 指标。从下图我们可以清楚看出 *this* 指标应该怎么调整。C++ 编译器只要把这样的调整策略实作出来即可。

**135**

多型與虛擬（*Polymorphism in C++*）

###### this



CPoint3d subobject

如果將 CVertex3d\* 轉型為 CVertex\*， this 指標就應該移到這裡。

CVertex3d object

CPoint2d subobject

|  |
| --- |
| vptr |
| \_x |
| \_y |
| \_z |
| vptr |
| next |
| mumble |

CVertex subobject

下面这段码就针对非自然多型（unnatural polymorphism）的 *this* 指标调整做了一个程序层面的模拟。我继续延用图 2.9 的 classes 体系：

#0001 CVertex3d av3d;

#0002 CPoint2d \*p2d = &av3d; // natural polymorphism #0003 CPoint3d \*p3d = &av3d; // natural polymorphism #0004 CVertex \*pv = &av3d; // unnatural polymorphism #0005 CVertex3d \*pv3d = NULL;

#0006

#0007 cout << &av3d << endl; // 0x0064FDD8 #0008 cout << pv << endl; // 0x0064FDE8 #0009

#0010 // 模擬 C++ 編譯器對於 unnatural polymorphism 的 this 指標調整動作

#0011 pv = (CVertex\*)(((char\*)&av3d) + sizeof( CPoint3d )); #0012 cout << pv << endl; // 0x0064FDE8

#0013

#0014 cout << p2d << endl; // 0x0064FDD8 #0015 cout << p3d << endl; // 0x0064FDD8 #0016

#0017 pv = pv3d; // pointer assignment #0018 cout << pv << endl; // 0x00000000

#0019

#0020 // 模擬 C++ 編譯器面對 null 指標時的 this 指標調整動作

#0021 pv = pv3d ? (CVertex\*)((char\*)pv3d) + sizeof( CPoint3d ) : 0;

#0022 cout << pv << endl; // 0x00000000

L8 和 L12 的結果相同，正是我們所期望看到的。

#### 136

第２章 C++ 物件模型（C++ Object Model）

## 再谈 sizeof 的迷思

既然 object 的大小只与 data members 以及 vptr 有关，我们再来看另一个例子：

class X { };

A

Z

Y

X

X

class Y : public X { }; class Z : public X { };

class A : public Y, public Z { }; // 多重继承

上述 *X*, *Y*, *Z*, *A* 等 classes 之中没有任何一个classes 内含明显的 data members， 也都没有 virtual functions，其间只有继承关系。请注意 *A* 多重继承了 *Y* 和 *Z*。那么是不是每一个 class object 的大小都应该是 0 呢？不是！

cout << "sizeof(X)=" << sizeof(X) << endl; // VC:1, BCB:4 cout << "sizeof(Y)=" << sizeof(Y) << endl; // VC:1, BCB:4 cout << "sizeof(Z)=" << sizeof(Z) << endl; // VC:1, BCB:4 cout << "sizeof(A)=" << sizeof(A) << endl; // VC:1, BCB:8

如果把 classes 继承关系改为虚拟继承如下：

class X { };

A

Z

Y

X

class Y : public virtual X { }; // 虚拟继承class Z : public virtual X { }; // 虚拟继承class A : public Y, public Z { }; // 虚拟多重继承

则 class object 的大小变成了：

cout << "sizeof(X)=" << sizeof(X) << endl; // VC:1, BCB:4 cout << "sizeof(Y)=" << sizeof(Y) << endl; // VC:4, BCB:8 cout << "sizeof(Z)=" << sizeof(Z) << endl; // VC:4, BCB:8 cout << "sizeof(A)=" << sizeof(A) << endl; // VC:8, BCB:12

（注意：这一部份随各家编译器而异。不同的编译器可能有不同结果）

看来，第一种情况中，应该空空如也的 objects，大小却为 1。而第二种情况中， 虚拟继承似乎带来类似 virtual functions 的存储器空间额外成本。

**137**

多型與虛擬（*Polymorphism in C++*）

#### 空空如也的 class

一个空空如也的 class：

class X { };

其 object 大小不为 0，而为 1，是编译器安插进去的一个 char，为的是让某两个objects 得以比较其间不同的存储器座落位置：

X x1, x2;

int i = (&x1 != &x2) ? 1 : 0;

cout << i << endl; // output : 1

当 *Y*, *Z*, *A* 的继承都不牵扯到 virtual：

class X { };

class Y : public X { }; class Z : public X { };

class A : public Y, public Z { };

它们的 objects 大小也都是 1（以Visual C++ 而言），道理相同。

#### 虚拟继承对 object 大小的影响

然而当继承体系牵扯到 virtual：

class X { }; class Y : public

X { };

virtual

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

objects 的大小就受到了下列因素的影响：

1. 语言的额外负担：当 virtual 继承发生，就像 virtual 函式带来的额外负担一样，编译器会在 object 体内增加一个指标（ 你可以想象是指向virtual base class's subobject， 后述）。
2. 编译器的特殊处理：empty virtual base class *X* 的 object 大小是 1，应该被继承到 derived class 身上。但有些编译器会对这种情况做最佳化

#### 138

第２章 C++ 物件模型（C++ Object Model）

处理，原因是既然 derived class's object 已有一个指标，不至于发生空物件所造成的存储器位置比对上的困难， 那么编译器实在是不需要额外加一个 char。

1. alignment 的影响： 在大部份机器上， 一个 object 的大小， 会被限制与 data bus 的宽度吻合，以便做最有效的存取与传输。在 32 位元机器上， alignment 的边界是 4 bytes。

以本例而言，*Y* 和 *Z* 的大小可以計算如下：

* 使用具备最佳化能力之编译器（ 例如 Visual C++）：

4 + 0 + 0 = 4 bytes

* 使用没有具备最佳化能力之编译器：

4 + 1 + 3 = 8 bytes

class *A* 的 object 大小计算又更复杂些。虚拟继承的一个重点是，virtual base class subobject 只存在一份，被大家共享。以下是 class *A* object 大小的几项考虑因素：

1. shared virtual base class 的大小。本例为 sizeof(X)。由于 X 是一个空的 class， 所以又引发最佳化与否（ 如稍早所讨论者） 两种情况。
2. base class *Y* 的大小扣除 shared virtual base class *X* 的大小。如果使用最佳化编译器，那么此值在本例中为 (0+4)-0=4。如果使用的不是最佳化编译器， 那么此值为 (1+4)-1=4。
3. base class *Z* 的大小扣除 shared virtual base class *X* 的大小。如果使用最佳化编译器，那么此值在本例中为 (0+4)-0=4。如果使用的不是最佳化编译器， 那么此值为 (1+4)-1=4。
4. class *A* 本身大小。由于是空的 class， 所以也分为最佳化与否（ 如稍早所讨论者） 两种情况。
5. alignment 的影响。

#### 139

多型與虛擬（*Polymorphism in C++*）

其中容易被误解的便是第 2 项和第 3 项。在未采取最佳化行动的编译器中，可能你会以为应该如此计算：(1+4+3)-1=7，其中 3 是因为 alignment 而附加。然而alignment 的影响不应该在这时候就计入，否则就会出现太多非必要的影响。

以本例而言，class *A* object 的大小可以计算如下：

* 具备最佳化能力之编译器（ 例如 Visual C++）：

0 + 4 + 4 + 0 + 0 = 8 bytes

* 没有具备最佳化能力之编译器：

1 + 4 + 4 + 1 + 2 = 12 bytes

#### base class subobject 在 derived class object 中的原样性

C++ 语言保证，出现在 derived class object 中的 base class subobject 有其完整的原样性。让我们看个实际例子：

|  |
| --- |
| int val |
| char c1 |
| char c2 |
| char c3 |
| padding |

4

1

###### 1

1

###### 1

class A object

class A { int val;

char c1, c2, c3;

};

class *A* object 的大小应为 (4+1+1+1)+1=8。最后 1 bytes 为 alignment 的结果。

如果我们以这样的继承体系来表示上述的 class *A*：

class A { int val; char c1;

};

class B : public A { char c2;

};

class C : public B { char c3;

};

#### 140

第２章 C++ 物件模型（C++ Object Model）

逻辑上，它们与先前单一 class 没有什么差别，甚至可能因为抽象分割而更理想。它们的 object 大小是多少？如果未经深思熟虑，可能你会回答都是 8，也就是让derived class *B* 和 *C* 新增的 1 byte 占用原先因为 alignment 而附加的空间。但事实上是：

cout << sizeof(A) << endl; // (4+1)+3=8 cout << sizeof(B) << endl; // (8+1)+3=12 cout << sizeof(C) << endl; // (12+1)+3=16

下图可以表现其布局方式。这样的布局安排，为的就是要保持「base class subobject

在 derived class object 中的原样性」。

4

|  |
| --- |
| int val |
| char c1 |
| padding 3 bytes |

1

###### 3

class A object

###### 8

1

###### 3

|  |
| --- |
| class A subobject |
| char c2 |
| padding 3 bytes |

class B object

###### 12

1

###### 3

padding 3 bytes

char c3

class B subobject

class A subobject

class C object

以下我要继续探讨虚拟继承，看看编译器面对 polymorphic classes（有 virtual functions 者），在虚拟继承方面需要对 vptr 和 vtbl 动些什么特殊手脚。

#### 141

多型與虛擬（*Polymorphism in C++*）

##### 虚拟继承（ Virtual Inheritance）

所有的规则，只要遇上 virtual，统统自动缴械！

是的，一有了 virtual functions，编译器便自动为 object 加上一个 vptr，又在存储器的另一个地方制作出一份 vtbl。而一旦继承关系是以 virtual 来维系，再加上polymorphic classes （有 virtual functions 者），情况更是复杂得无法三言两语说得清楚。

虚拟继承的一个重点是，virtual base class subobject 只需在任何层次的 derived class object 中存在一份。为了完成这个目标，目前开发出两种物件模型：

#### Pointer Strategy

编译器在 derived object 之中，针对 virtual 继承，多安插一个指标，指向virtual base class。于是在更深层的 derived class object 中，就可以只存在一个 shared virtual base class。以下面的classes 体系为例：

\_z xt

\_x, \_y

ne

mumble

CVertex3d

CVertex

CPoint3d

CPoint2d

图 2.11 可以表现所谓的 Pointer Strategy 模型。

#### 142

第２章 C++ 物件模型（C++ Object Model）

|  |
| --- |
| CPoint2d’s vptr |
| float \_x |
| float \_y |

CPoint2d ap2d;



CVertex subobject

CPoint3d subobject

CPoint2d subobject

|  |
| --- |
| CVertex’s vptr |
| CVertex\* next |
| CPoint2d \*pP2d |
| CPoint3d’s vptr |
| float \_z |
| CPoint2d \*pP2d |
| float mumble |
| CPoint2d’s vptr |
| float \_x |
| float \_y |

Point2d subobject



|  |  |
| --- | --- |
| CPoint3d’s vptr |  |
| float \_z |
| CPoint2d \*pP2d |
|  |
| CPoint2d’s vptr |
| float \_x |
| float \_y |

|  |  |
| --- | --- |
| CVertex’s vptr |  |
| CVertex\* next |
| CPoint2d \*pP2d |
|  |
| CPoint2d’s vptr |
| float \_x |
| float \_y |

Point2d subobject

class CPoint3d : virtual CPoint2d

{ ... } ap3d;

class CVertex : virtual CPoint2d



{ ... } av;

class CVertex3d : public CVertex, public CPoint3d

{ ... } av3d;

图 2.11 虚拟继承中的 Pointer Strategy 物件模型。本图参考 *Inside The C++ Object Model*（ 侯俊杰译： 「深度探索 C++ 物件模型」， 碁峰/1998） 图 3.5a。

#### Virtual Table Offset Strategy

编译器利用原就存在的 vtbl 来放置 virtual base class subobject 在 object 中的offset 值。通常的技巧是把这些 offset 放在 vtbl 的负索引处，以便与 vtbl 原本的内容有明显的区隔。再以上一页的同一个 classes 体系为例，图 2.12 可以表现出这种 Virtual Table Offset Strategy 模型。

#### 143

多型與虛擬（*Polymorphism in C++*）

|  |
| --- |
| CPoint2d’s vptr |
| float \_x |
| float \_y |

CPoint2d ap2d;



8



virtual function slots

virtual base class

offsets (8)

float \_y

float \_x

CPoint2d’s vptr

float \_z

CPoint3d’s vptr

...

CPoint2d subobject

beginning of

virtual table

class CPoint3d : virtual CPoint2d { ... } ap3d;

8



...

... (8)

float \_y

float \_x

CPoint2d’s vptr

CVertex\* next

CVertex’s vptr

...

CPoint2d subobject

CVertex subobject

CPoint3d subobject

CPoint2d subobject

class CVertex : virtual CPoint2d { ... } av;

##### 

|  |  |  |
| --- | --- | --- |
| CVertex’s vptr |  | |
| 20 |  |
| Vertex\* next |
| CPoint3d’s vptr |  | |
| 12 |  |
| float \_z |
| float mumble |  | |
| CPoint2d’s vptr |
|  |  |
| float \_x |
| float \_y |  | |

...

class CVertex3d : public CVertex, public CPoint3d



...

(12)

...

...

(20)

...

{ ... } av3d;

圖 2.12 虚拟继承中的 Virtual Table Offset Strategy 物件模型。本图参考 *Inside The C++ Object Model*（ 侯俊杰译：「深度探索 C++ 物件模型」， 碁峰/1998） 图 3.5b。

#### 144

第２章 C++ 物件模型（C++ Object Model）

这两种作法的优劣点，以及各家编译器实际设计时的改良作法，请参考 ***Inside The C++ Object Model*** 第３章（Stanley Lippman / Addison Wesley / 1996；侯俊杰译 / 碁峰 / 1998）。

现在让我再次利用先前讨论多重继承时的 classes 体系（图 2.9），改装为虚拟继承，并使用 Visual C++ 编译器，看看实际情况如何。我对图 2.9 的改装动作有二：

1. 改用虚拟多重继承。
2. 本案情况复杂， 我必须使用除错器观察（ 用法请参考本章最后一节

p.150） 。为了能够在除错器的「Memory 视窗」中轻易观看数值， 我把原先的 float members 全改为 long。

图 2.13 虚拟继承的 classes 体系

#0001 class CPoint2d

#0002 {

CVertex3d

CVertex

CPoint3d

CPoint2d

#0003 public:

#0004 CPoint2d( long x = 0, long y = 1 ) #0005 : \_x( x ), \_y( y ) { };

#0006

#0007 long x() { return \_x; } #0008 long y() { return \_y; } #0009

#0010 void x( long newX ) { \_x = newX; } #0011 void y( long newY ) { \_y = newY; } #0012

#0013 // 加上 z 的保留空間（目前什麼也沒做）

#0014 virtual long z() { return 0; } #0015 virtual void z( long ) { } #0016

#0017 public: // protected: is better #0018 long \_x, \_y;

#0019 };

#0020

#0021 class CPoint3d : public virtual CPoint2d #0022 {

#0023 public:

#0024 CPoint3d( long x = 0, long y = 1, long z = 2 ) #0025 : CPoint2d( x, y ), \_z( z ) { };

#### 145

多型與虛擬（*Polymorphism in C++*）

#0026

#0027 virtual long z() { return \_z; }

#0028 virtual void z( long newZ ) { \_z = newZ; } #0029

#0030 public: // protected: is better #0031 long \_z;

#0032 };

#0033

#0034 class CVertex : public virtual CPoint2d #0035 {

#0036 public:

#0037 CVertex() { next = NULL; } #0038 virtual void foo() { } ; #0039

#0040 public: // protected: is better #0041 CVertex \*next;

#0042 };

#0043

#0044 class CVertex3d : public CPoint3d, public CVertex #0045 {

#0046 public:

#0047 CVertex3d(long m = 5) { mumble = m; } #0048

#0049 public: // protected: is better #0050 long mumble;

#0051 };

首先检查各个 class object 的大小（以下为Visual C++ 执行结果）：

|  |  |  |
| --- | --- | --- |
| cout << sizeof(CPoint2d) << endl; | // | 12 |
| cout << sizeof(CPoint3d) << endl; | // | 24 |
| cout << sizeof(CVertex) << endl; | // | 24 |
| cout << sizeof(CVertex3d) << endl; | // | 40 |

以下可以取出所有 data members 的位址：

CVertex3d av3d;

cout << &av3d << endl; // 0x65FDCC

cout << &(av3d.\_x) << endl; // 0x65FDEC

cout << &(av3d.\_y) << endl; // 0x65FDF0

cout << &(av3d.\_z) << endl; // 0x65FDDC cout << &(av3d.next) << endl; // 0x65FDD4 cout << &(av3d.mumble) << endl; // 0x65FDE0

#### 146

第２章 C++ 物件模型（C++ Object Model）

以下可以把 *av3d* object 的所有內容列印出來：

// check the value of data members

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| printf("%#x\n", | \*(long\* | )((char\*)&av3d | + | 0 )); | // | 0x419064 |
| printf("%#x\n", | \*(long\* | )((char\*)&av3d | + | 4 )); | // | 0x419068 |
| printf("%#x\n", | \*(long\* | )((char\*)&av3d | + | 8 )); | // | 0 |
| printf("%#x\n", | \*(long\* | )((char\*)&av3d | + | 12)); | // | 0x419078 |
| printf("%ld\n", | \*(long\* | )((char\*)&av3d | + | 16)); | // | 2 |
| printf("%ld\n", | \*(long\* | )((char\*)&av3d | + | 20)); | // | 5 |
| printf("%#x\n", | \*(long\* | )((char\*)&av3d | + | 24)); | // | 0 |
| printf("%#x\n", | \*(long\* | )((char\*)&av3d | + | 28)); | // | 0x419058 |
| printf("%ld\n", | \*(long\* | )((char\*)&av3d | + | 32)); | // | 0 |
| printf("%ld\n", | \*(long\* | )((char\*)&av3d | + | 36)); | // | 1 |

根据我的判断，这些栏位所代表的意义如下（有些栏位的意义并不清楚）：

av (CVertex object)

CVertex’s vptr a pointer



|  |  |
| --- | --- |
|  | |
|  |  |
|  |
| next | |
|  | |
| \_x | |
| \_y | |

next CPoint2d’s vptr

###### \_x

\_y

a3d (CPoint3d object)

###### CPoint3d’s vptr

|  |
| --- |
|  |
| \_z |
| ??? (0) |
|  |
| \_x |
| \_y |

\_z

###### ???

CPoint2d’s vptr

###### \_x

\_y

av3d (CVertex3d object)

###### CVertex’s vptr a pointer



CVertex3d\*

|  |  |  |
| --- | --- | --- |
| CVertex\* 0x65FDCC | 0x419064 | |
| 0x65FDD0  CPoint3d\* 0x65FDD4 | 0x419068 |  |
|  |
| next (null) | |
| 0x65FDD8  0x65FDDC  0x65FDE0  CPoint2d\* 0x65FDE4 | 0x419078 | |
| \_z (2) | |
| mumble (5) | |
| ??? (0) | |
| 0x65FDE8  0x65FDEC  0x65FDF0 | 0x419058 | |
| \_x (0) | |
| \_y (1) | |

next CPoint3d’s vptr

###### \_z mumble

???

###### CPoint2d’s vptr

\_x

###### \_y

CVertex3d 和 CVertex 之间是属于 natural polymorphism

（this 指标不必移动）

CVertex3d 和 CPoint2d 以及 CPoint3d 之间是属于

unnatural polymorphism（this 指标必须移动）

#### 147

多型與虛擬（*Polymorphism in C++*）

以下动作可以检验 polymorphism 发生时，*this* 指标的位移：

CPoint2d \*p2d = &av3d; // unnatural polymorphism CPoint3d \*p3d = &av3d; // unnatural polymorphism CVertex \*pv = &av3d; // natural polymorphism

cout << &av3d << endl; // 0x0065FDCC

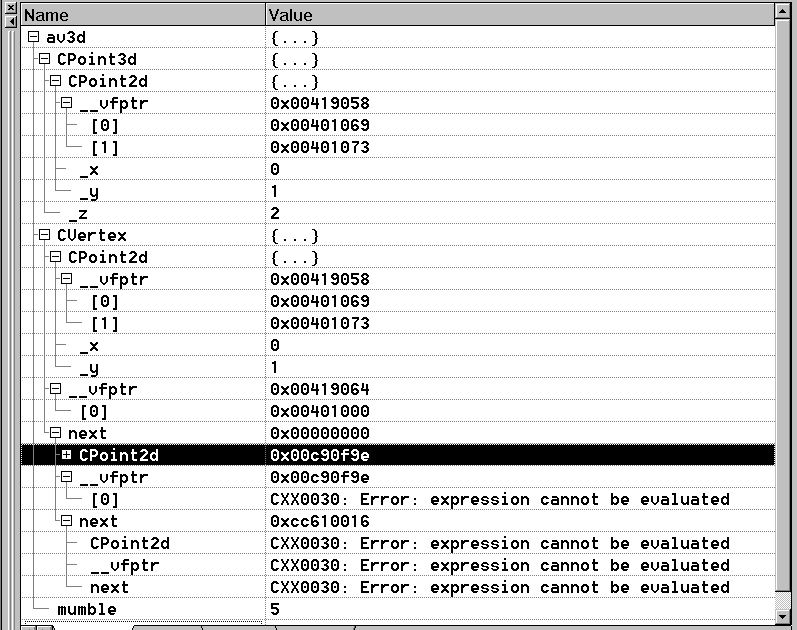
cout << pv << endl; // 0x0065FDCC

cout << p2d << endl; // 0x0065FDE8

cout << p3d << endl; // 0x0065FDD8

利用除错器（请参考 p.150「利用除错器观察 object 布局和函式位址」一节）， 我可以把 *CVertex3d* object 的布局表现如图 2.14。这个模型和图 2.11 及图

2.12 的理论架构并不吻合。我只能提醒你，C++ Standard 并没有硬性规定物件

模型的标准，各家编译器厂商可以自由发挥，只要能够维系住物件导向以及多型的语意，就可以。下面是以 Visual C++ 除错器观察的 object 布局结果：

#### 148

第２章 C++ 物件模型（C++ Object Model）

av3d (CVertex3d object)

vptr 0x419064

vptr 0x419068

0x401069

0x401073

0x419078

vptr 0x419058

0x419058

0x41905C

0x419060

0x419064

0x419068

0x41906C

0x419070

0x419074

0x419078

0x41907C

0x419080

0x419084

0x419088

0x41908C

0x419090

0x419094

0x419098

0x10

this 指標調整：

65FDD4+18=65FDEC

this 指標調整：

65FDDC+10=65FDEC

0x401078

0x401064

0

0

0

0

0

0

0

0x18

0xFFFFFFFC (-4)

0x401000

0

\_y (1)

\_x (0)

??? (0)

mumble (5)

\_z (2)

next (null)

0x65FDD4

0x65FDDC

0x65FDEC

virtual functions implementation

long Point3d::z(void)

void Vertex::foo(void)

void Point3d::z(long)

long Point2d::z(void)

void Point2d::z(long)

圖 2.14 虚拟继承下的 object 布局（ 使用 Visual C++编译器）。除非拿到 Visual C++ 的实作技术资料， 否则很难推测出每一个栏位的意义与运用方式。

#### 149

多型與虛擬（*Polymorphism in C++*）

## 利用除错器观察 object 布局和函式位址

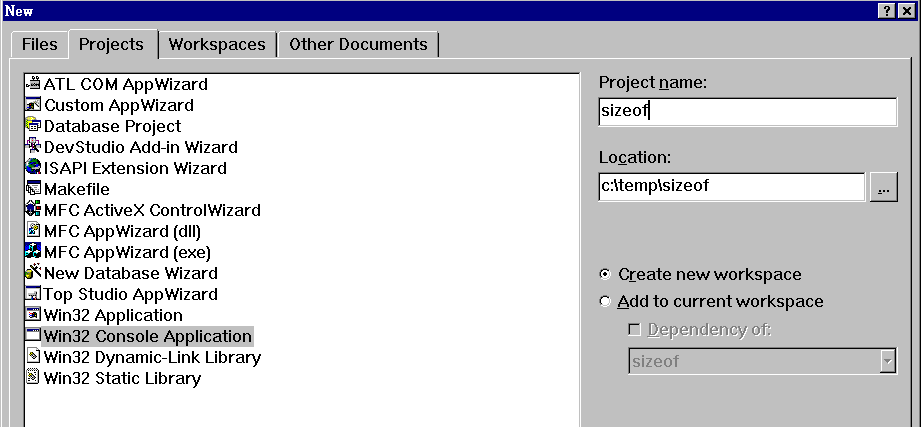
我相信，本章先前以 DIY（Do It Yourself）的方式探索 object 布局和函式位址， 可以让大家学习到对于指标的各种操控。虽然麻烦，却也不是没有收获。但如果要探索更复杂更多的资料，一一写码就真的很麻烦了。

这些工作，除错器都可以代劳。以下我以 Microsoft Visual C++ 为例，示范除错器的功能与使用。即使你手上是另一套编译器，我想触类旁通，并不困难。

我将示范如何在 Visual C++ 整合环境中写一个 console 程序，并使用除错器观看任何你想看的东西，包括定址空间中的任何一个位址及其内容。我将以本章一开始「sizeof 的疑惑」一节所出现的程序做为示范，project 名称为 sizeof，其中只有一个 sizeof.cpp 档案（classes 的宣告与实作都在其中）。

**产生一个 project：**

启动 Visual C++。选按【File/New】，选择【Project】附页下的【Win32 Console Application】，并在右侧输入你的 project 档案目录名称，然后按下【OK】。

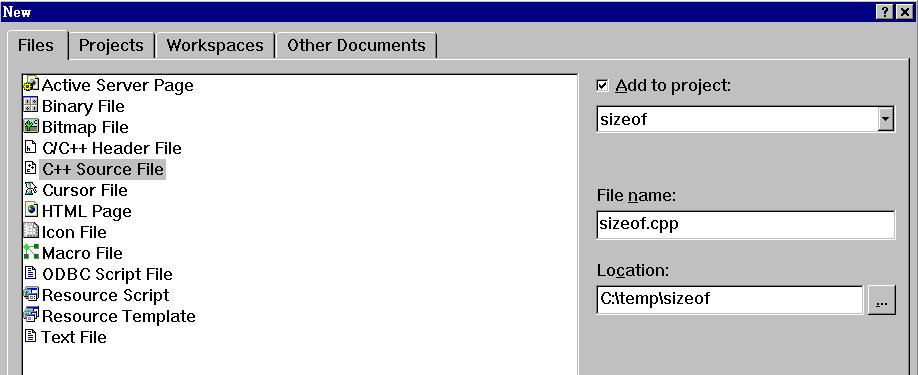


#### 150

第２章 C++ 物件模型（C++ Object Model）

#### 在 project 中加上 C++ 原始码

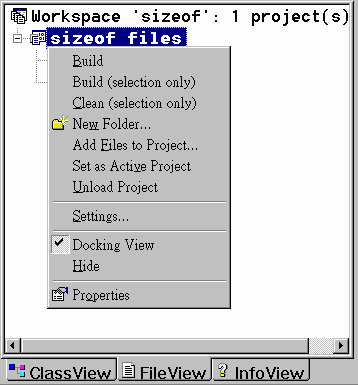
如果你要在整合环境中写码，请选按【File/New】并在【Files】附页中选择【C/C++ Header File】或【C/C++ Source File】，并在右侧填入文件名称：



然后就可以在随后出现的编辑视窗中开始写码。

如果你已经在别的文字编辑器上写好程序，那么可以在 Visual C++ 管理视窗（通常在荧幕左侧）的【File View】附页所出现的【sizeof files】资料夹上按右键，获

得 popup menu 如左。选择其中的



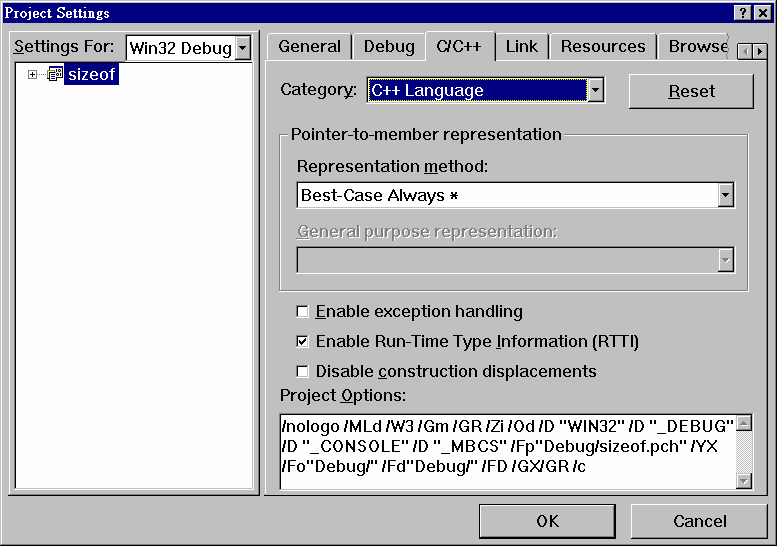
【Add Files to project...】，然后在出现的档案对话盒中选择一个已经写好的程序（.CPP）。这个档案即被加入你的 project 之中。

#### 151

多型與虛擬（*Polymorphism in C++*）

#### 环境设定

选按主选单中的【Project/Setting...】。为了稍后能够使用除错器，请在对话盒左 上方选择【Win32 Debug Mode】。对话盒右侧有许多附页，提供各种环境设定。与 C/C++ programming 有关的，大约都集中在【C/C++】附页中：

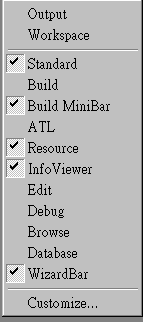


请拉下【Category:】，选择【C++ Language】（原本是【General】）。本书第一章曾说过，若使用 C++ STL，必须把 exception handling 打开，这时候就需要选按【Enable exception handling】，而下方的【Project Options:】第一行会多出 /GX 项目。本书第一章亦曾说过使用 *typeid* operator 于 polymorphic classes 时，必须把 RTTI 打开，这时候你必须选按【Enable Run-Time Type Information (RTTI)】， 于是下方的编译选项第一行会多出 /GR 项目。

#### 152

第２章 C++ 物件模型（C++ Object Model）

#### 建造（ build） 程式

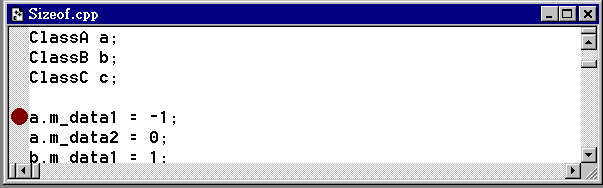


选择【Build/Build sizeof.exe】。编译联结的结果会出现在荧幕最下方（通常）的

【Output】视窗中。这个视窗如果没有出现，可在大视窗任何空白地点按右键， 并在冒起式选单中选择【Output】以显示之（如左图）。前述的管理视窗如果没有出现，也可选按左图的【Workspace】显示之。

#### 設立中斷點（ break point）

除错之前，可以下列方式设立断点（break point）：将鼠标移到文字编辑视窗中的目标行，按下 F9（或工具列上的「手形钮」）此时目标行的最前面会出现一个



大红点，表示断点已设立妥当。

除错过程中，程序遇到断点即暂停执行，你可以利用这个时候观察各个变量内容、存储器内容、暂存器内容 ...。

#### 除错

选按【Start to Debug...】即可开始除错。此时选单中的【Project】变成【Debug】， 必须再选按【Stop Debugging】才能够结束除错模式。下面是除错工具列：

各种除错手法如下：

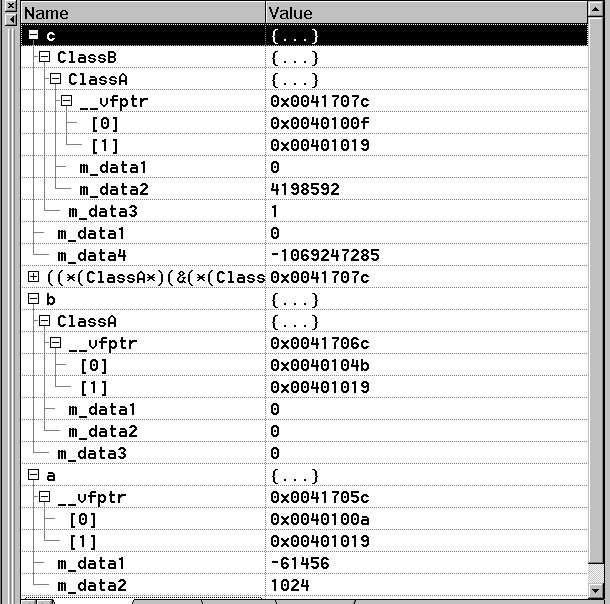
#### 153

多型與虛擬（*Polymorphism in C++*）

* Go F5 - 前进（ 执行） 至下一个断点
* Step Into F11 - 遇到函式呼叫时， 进入函式体内除错
* Step Over F10 - 遇到函式呼叫时， 不进入函式体内除错
* Step Out Shift+F11 - 在函式内一直执行前进，直到跳出函式。
* Run to Cursor Ctrl+F10 - 执行到游标所在那一行

#### 观察 object 布局

以 sizeof.cpp 为例，我在产生出一个 *ClassC* object 之后设立断点，于是就可以观察 object 的布局（如下）。这个视窗可以在选按主选单中的【View/Debug Windows/Watch】后获得。要把一个 object 加到 Watch 视窗中，很简单，请直接在编辑视窗上将该 object 名称以鼠标圈选出（highlight），然后按右键，并于

冒 起 式 选 单 中 选 择

【QuickWatch 】， 等出现

【QuickWatch】视窗后再选按其右上角的【Add Watch】钮，即可常驻于【Watch】视窗中随时观察之。

根据本例的 *ClassA*, *ClassB*, *ClassC* 的物件布局（ 如左），很容易就可以整理出本章先前的图 2.3 画面。

#### 154

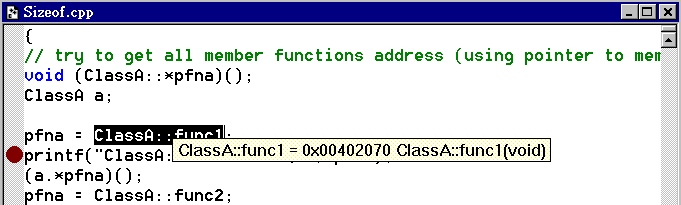
第２章 C++ 物件模型（C++ Object Model）

#### 观察函式位址

sizeof.cpp 之中有这样的动作：

void (ClassA::\*pmfa)();

pmfa = ClassA::func1; // (1)

把断点设在 (1) 之下，Go 之后遇断点即暂停执行，此时可将鼠标移到文字视窗中的 *ClassA::func1* 之上，出现函式位址（因为函式名称即函式位址）：

同理可对其他 member functions（包括 virtual member functions）依法炮制。

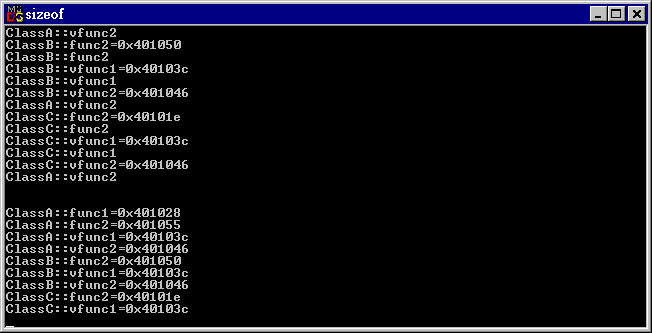
奇怪的是，除错器告诉我们的函式位址（如上图），和我们自己以程序方法取出的函式位址不相同。如果拿 sizeof.exe 在 MS-DOS 视窗中的执行结果，和除错器中观察所得两相比较，不一样是应该的（可能的），因为谁知道 sizeof.exe 一旦进入除错器中，和单纯的 console mode sizeof.exe 有何差别?! 程序区段或资料区段有所膨胀是极有可能的事。但我是拿 sizeof.exe 除错时同时输出于 DOS 视窗的结果（如下）与除错器观察所得两相比较，却不相同，这是我所无法理解的。或许和 Microsoft 的 vcall thunk 有关，谁知道！

此外，除错器中观察到的 virtual member functions 位址，也没有出现在各个objects 的 vtbl 之中。这一点还可以解释，大约是因为 vcall thunk 之故。我好像把所有无法解释的观察结果都推给 vcall thunk 了！实在是因为我们无法获知Visual C++ 编译器中 vcall thunk 的真实结构，所以很难实证之。我把函式位址的观察放在稍后p.157 的一个小节之中，做成一份心得整理。

#### 155

多型與虛擬（*Polymorphism in C++*）

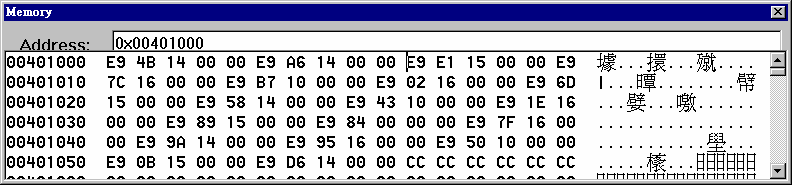
这是 sizeof.exe 在除错过程中的输出画面：



#### 观察位址空间

选按【View / Debug Windows / Memory】，即可获得一个 Memory 视窗。你可以在视窗上方的【Address:】处键入 4GB 位址空间中的任何位址，Memory 视窗便

会将该区域的内容显示给你。



#### 156

第２章 C++ 物件模型（C++ Object Model）

在 Visual C++ 除错器中对函式位址的观察 - 心得整理

我们探讨 Object Model，是为了对 object 布局以及 vptr、vtbl 有一个了解，俾能对于 polymorphism 的背后机制「知其所以然」。以实用的角度而言，能够实证出图 2.2 和图 2.3，已经足够。

由于 Microsoft 的 vcall thunk 机制的搅和，我在 Visual C++ 中获得的和在除错器中观察所得的 member function 位址有些奇怪，和理论架构之间并没有一一映合。虽然继续深入探索这一部份，其实用价值并不高，而且最终我也没有获得什么肯定的结论，不过我还是利用小小的篇幅，把观察心得整理出来，或许有兴趣的读者可以继续下去，并进而解释一些目前我不了解的真象。

我必须说，这么低阶的动作实在不是我的专长。请把以下的整理视为我个人的一

些经历，抱以适度的怀疑。如果您能指出其中错误或愿意加上补充并 email 给我

（[jjhou@ccca.nctu.edu.tw](mailto:jjhou@ccca.nctu.edu.tw)），我会很感谢您并在新刷中修正之。

对于 member functions 位址，目前有三个方式可以观察，但互不一致：

1. 第一组位址：从 vtbl 观察到的 virtual member functions「位址」。此组资料可以程序方法取出，或从除错器观察（ 如图 2.3 或 p.154 图）。
2. 第二组位址： 在除错器中直接把游标移到 member functions 名称之上， 观察所得（ 如 p.155 图）。
3. 第三组位址：在程序中取 member functions 的位址（ 如 p.113~p.115）。

在某一次实验过程中，三组数据如下（classes 体系请见图 2.1）：

注意：以下 memory dump 中屡屡出现形式为 "E9 xx xx xx xx" 的内容。E9 是个machine code，意思是：JMP label，也就是 near jump 的意思，在Win32 环境下， 后接一个 DWORD。

**157**

多型與虛擬（*Polymorphism in C++*）

函式名稱 (1) vtbl 內容 (2)除錯器觀察得 (3)程式中取得

ClassA::func1() x 0x402070 0x401028

ClassA::func2() x 0x402530 0x401055

**ClassA::vfunc1()** 0x40100F 0x4025F0

0x40103C

0x401046

**ClassA::vfunc2()** 0x401019 0x402620

ClassB::func2() x 0x402560 0x401050

**ClassB::vfunc1()** 0x40104B 0x4020A0

0x40103C

0x401046

**ClassB::vfunc2()** 0x401019 0x402620

ClassC::func2() x 0x402590 0x40101E

**ClassC::vfunc1()**

**ClassC::vfunc2()**

0x40100A

0x401019

0x402690

0x402620

0x40103C

0x401046

为了了解这些位址究竟是代表什么，我必须到除错器的【Memory 视窗】中去看看它们所指的内容：

* 第一组位址（ vtbl 内容。可在程序中根据 vptr 取得）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函式 | vtbl entry | 存储器内容 | JMP 至 | 「JMP 目标地」的存储器内容 |
| ClassA::func1() | x |  |  |  |
| ClassA::func2() | x |  | （註） |  |
| ClassA::vfunc1() | 0x40100F | **E9** 7C 16 00 00 | 0x402690 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 48 A3 41 |
| ClassA::vfunc2() | 0x401019 | **E9** 02 16 00 00 | 0x402620 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 20 A3 41 |
| ClassB::func2() | x |  |  |  |
| ClassB::vfunc1() | 0x40104B | **E9** 50 10 00 00 | 0x4020A0 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 34 A3 41 |
| ClassB::vfunc2() | 0x401019 | **E9** 02 16 00 00 | 0x402620 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 20 A3 41 |
| ClassC::func2() | x |  |  |  |
| ClassC::vfunc1() | 0x40100A | **E9** E1 15 00 00 | 0x4025F0 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 0C A3 41 |
| ClassC::vfunc2() | 0x401019 | **E9** 02 16 00 00 | 0x402620 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 20 A3 41 |

\*註：040100F + 5(instruction length) + 167C(operand) = 402690

#### 158

第２章 C++ 物件模型（C++ Object Model）

* 第二组位址（ 从除错器中直接观察所得， 如 p.155 图）：

|  |  |  |
| --- | --- | --- |
| 函式 | 位址 | 该处内容 |
| ClassA::func1() | 0x402070 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 CC A2 41 |
| ClassA::func2() | 0x402530 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 DC A2 41 |
| ClassA::vfunc1() | 0x4025F0 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 0C A3 41 |
| ClassA::vfunc2() | 0x402620 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 20 A3 41 |
| ClassB::func2() | 0x402560 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 EC A2 41 |
| ClassB::vfunc1() | 0x4020A0 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 34 A3 41 |
| ClassB::vfunc2() | 0x402620 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 20 A3 41 |
| ClassC::func2() | 0x402590 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 FC A2 41 |
| ClassC::vfunc1() | 0x402690 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 48 A3 41 |
| ClassC::vfunc2() | 0x402620 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 20 A3 41 |

* 第三組位址（ 利用程式方法取得的函式位址）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函式 | 位址 | 内容 | JMP 至 | 「JMP 目标地」的存储器内容 |
| ClassA::func1() | 0x401028 | **E9** 43 10 00 00 | 0x402070 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 CC A2 41 |
| ClassA::func2() | 0x401055 | **E9** D6 14 00 00 | 0x402530 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 DC A2 41 |
| ClassA::vfunc1() | 0x40103C | **E9** 7F 16 00 00 | 0x4026C0 | 8B 01 FF 20 CC CC CC CC CC CC CC CC CC CC |
| ClassA::vfunc2() | 0x401046 | **E9** 95 16 00 00 | 0x4026E0 | 8B 01 FF 60 04 CC CC CC CC CC CC CC CC CC |
| ClassB::func2() | 0x401050 | **E9** 0B 15 00 00 | 0x402560 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 EC A2 41 |
| ClassB::vfunc1() | 0x40103C | **E9** 7F 16 00 00 | 0x4026C0 | 8B 01 FF 20 CC CC CC CC CC CC CC CC CC CC |
| ClassB::vfunc2() | 0x401046 | **E9** 95 16 00 00 | 0x4026E0 | 8B 01 FF 60 04 CC CC CC CC CC CC CC CC CC |
| ClassC::func2() | 0x40101E | **E9** 6D 15 00 00 | 0x402590 | 55 8B EC 51 89 4D FC 68 23 10 40 00 68 FC A2 41 |
| ClassC::vfunc1() | 0x40103C | **E9** 7F 16 00 00 | 0x4026C0 | 8B 01 FF 20 CC CC CC CC CC CC CC CC CC CC |
| ClassC::vfunc2() | 0x401046 | **E9** 95 16 00 00 | 0x4026E0 | 8B 01 FF 60 04 CC CC CC CC CC CC CC CC CC |

#### 159

多型與虛擬（*Polymorphism in C++*）

为了查证函式的起头是什么格式，我在另一个简单程序中设计了一个 global function 如下：

void func1() { cout << "func1" << endl; }

...

cout << &func1 << endl; // 取出 func1() 的位址

程序的输出是 0x401037，该处内容是 E9 54 00 00 00，也就表示要 JMP 到绝对位址 0x401037+0x0054+5=0x401090 处，而除错器所显示的 *func1()* 函式位址也是0x401090：

0x401090（根据程序方法所取得的函式位址）

55 8B EC 68 1E 10 40 00 68 50 6D 41 00 B9 30 A1

0x401090（除错器所显示的函式位址）

这个位址的确是函式的真正起始处，因为一开始的两个 machine code：

55 ： push ebp 8BEC ： mov ebp,esp

正是 C/C++ 语言的函式 prologue（前置码），用来布置标准的 stack frame9。

1. 函式呼叫端（caller）会有一些由编译器加入的程序码，将参数压入 stack 之中，然后再由CALL 指令将函式回返位址（return address）压入 stack，并将控制权（IP，instruction pointer） 指向函式起头处。函式一开始的这个由编译器自动加入的prologue（前置码），即是将 ebp 准备妥当，以便稍后能够在 statck 中保留 local 变量的空间以及抓取函式参数。

#### 160

第２章 C++ 物件模型（C++ Object Model）

前述第二、三组数据中的 non-virtual member functions 都符合这种模式：

第三组位址（以程序方法取得）的起始处

函式名稱ClassA::func1() ClassA::func2()

55 8B EC 51 89 4D FC 68 23 10 40 00 68 CC A2 41

55 8B EC 51 89 4D FC 68 23 10 40 00 68 DC A2 41

55 8B EC 51 89 4D FC 68 23 10 40 00 68 EC A2 41

55 8B EC 51 89 4D FC 68 23 10 40 00 68 FC A2 41

ClassB::func2()

ClassC::func2()

第二组位址（除错器观察所得）的起始处

但是三组数据中的 virtual member functions 则有不同的模式，这是我所无法理解的（或许你有什么心得？）：

* 第一组位址（根据 vtbl 内容所导出的函式位址）：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 函式 | JMP 至 | 內容 |  | | | |
| ClassA::vfunc1() | 0x402690 | 55 8B EC 51 | 89 | 4D | FC | 68 23 10 40 00 68 48 A3 41 |
| ClassA::vfunc2() | 0x402620 \* | 55 8B EC 51 | 89 | 4D | FC | 68 23 10 40 00 68 20 A3 41 |
| ClassB::vfunc1() | 0x4020A0 | 55 8B EC 51 | 89 | 4D | FC | 68 23 10 40 00 68 34 A3 41 |
| ClassB::vfunc2() | 0x402620 \* | 55 8B EC 51 | 89 | 4D | FC | 68 23 10 40 00 68 20 A3 41 |
| ClassC::vfunc1() | 0x4025F0 | 55 8B EC 51 | 89 | 4D | FC | 68 23 10 40 00 68 0C A3 41 |
| ClassC::vfunc2() | 0x402620 \* | 55 8B EC 51 | 89 | 4D | FC | 68 23 10 40 00 68 20 A3 41 |

* 第二组位址（利用除错器直接观察到的函式位址）：
* 第三组位址（以程序方法取得的函式位址）： 函式 JMP 至 内容

ClassA::vfunc1() 0x4026C0 \* ClassA::vfunc2() 0x4026E0 \* ClassB::vfunc1() 0x4026C0 \* ClassB::vfunc2() 0x4026E0 \* ClassC::vfunc1() 0x4026C0 \* ClassC::vfunc2() 0x4026E0 \*

疑点一：为什么没有完全相同？

（两组的 #1 行和 #5 行刚好相反）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 函式 | 函式位址 | 內容 |  | | | | | |
| ClassA::vfunc1() | 0x4025F0 | 55 8B | EC | 51 | 89 | 4D | FC | 68 23 10 40 00 68 0C A3 41 |
| ClassA::vfunc2() | 0x402620 \* | 55 8B | EC | 51 | 89 | 4D | FC | 68 23 10 40 00 68 20 A3 41 |
| ClassB::vfunc1() | 0x4020A0 | 55 8B | EC | 51 | 89 | 4D | FC | 68 23 10 40 00 68 34 A3 41 |
| ClassB::vfunc2() | 0x402620 \* | 55 8B | EC | 51 | 89 | 4D | FC | 68 23 10 40 00 68 20 A3 41 |
| ClassC::vfunc1() | 0x402690 | 55 8B | EC | 51 | 89 | 4D | FC | 68 23 10 40 00 68 48 A3 41 |
| ClassC::vfunc2() | 0x402620 \* | 55 8B | EC | 51 | 89 | 4D | FC | 68 23 10 40 00 68 20 A3 41 |

疑点二：这是什么？vcall thunk 吗？

CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC

8B 01 FF 20 CC

8B 01 FF 60 04

8B 01 FF 20 CC

8B 01 FF 60 04

8B 01 FF 20 CC

8B 01 FF 60 04

#### 161

多型與虛擬（*Polymorphism in C++*）

#### 162

第３章 型別轉換（Type Casting）

**第３章**

# 型别转换

## （Type Casting）

到目前为止，我所举出的最接近多型（polymorphism）应用的例子，是第１章末 利用 STL 以及利用 MFC Collection Class 所做的「形状串列」。我以 base type pointers（*CShape*\*）指向各种 derived type object（矩形、正方形、三角形、圆形、椭圆形），加入串列之中，然后再将串列走访一遍，呼叫每一个「形状物件」的*display()*，这是一个从 *CShape* 开始即出现的 virtual function。于是各个 *display()* 函式依序被唤起。

这个例子过于简单，没有出现型别转换（尤其是指标的型别转换）动作。

我在第２章探讨 object 布局时曾经一再强调 C++ Standard 所强调的一个观念： 一个 derived type object 中存在有其 base type subobjects。我也曾经以土拨鼠为例，说明土拨鼠身上拥有松鼠、啮齿动物、哺乳动物、脊索动物、以及（当然） 动物的特征。当我以一个「动物指标」指向一只「土拨鼠实体」，是被允许的， 而且也是符合自然的。如果我再把这个指标转型为「啮齿动物」指标，这时候指

**163**

多型與虛擬（*Polymorphism in C++*）

标所指的就是土拨鼠中的「啮齿动物 subobject」。subobject 虽然不能够抽出成为一个独立实体，但它在 object 的运作上是有意义的。

在多重继承情况下，由于 subobjects 在 object 中的排列次序，多型因而分为自然多型（natural polymorphism）和非自然多型（unnatural polymorphism）两种。它们关系到 *this* 指标的移动。不过请记住，多型的自然或非自然，由编译器去操心就可以了，对我们并没有影响。

在多型应用上，指标的型别转换是必要的功课。我们先在这一章完成基本功，然后再在后两章实例探讨中检视其运用。

## 型别转换（cast）

只要是面对指标，就常常要面对型别转换的问题。

从过去的 16 位元时代到今天的 32 位元时代，指标本身一直占用 4 bytes。对于指标的各种操作（例如将它前进一个单元）的结果，视指标型别（type）而定。例如：

char str[] = "Hello"; char \*pc = (char\*)&str;

printf("%c \n", \*pc); // 'H'

printf("%c \n", \*(pc+1)); // 'e' printf("%X \n", pc); // 63FDE4

printf("%X \n", pc+1); // 63FDE5

*pc* 指向 char，那么 (*pc*+1) 将前进 1 byte（因为一个 char 是 1 byte）。如果指标指向 long：

long larray[3] = { 11, 22, 33 };

long \*pl = (long\*)&larray; // 等同于 (long\*)&larray[0]

printf("%ld \n", \*pl); // 11 printf("%ld \n", \*(pl+1)); // 22

**164**

第３章 型別轉換（Type Casting）

printf("%X \n", pl); // 63FDEC

printf("%X \n", pl+1); // 63FDF0

那么 (*pl*+1) 将前进 4 byte（因为一个 long 是 4 byte）。注意，前述的两个转型动作：

char str[] = "Hello"; char \*pc = (char\*)&str;

long larray[3] = { 11, 22, 33 }; long \*pl = (long\*)&larray;

如果没有明白地指定型别，而改为这个样子：

char str[] = "Hello";

char \*pc = &str; // error

long larray[3] = { 11, 22, 33 }; long \*pl = &larray; // error

将无法通过型别检验极为严格的 C++ 编译器，编译时会出现错误讯息如下（以

Visual C++ 为例）：

error C2440: 'initializing' : cannot convert from 'char (\*)[6]' to 'char \*' (new behavior; please see help)

error C2440: 'initializing' : cannot convert from 'long (\*)[3]' to 'long \*' (new behavior; please see help)

C++ 对于型别检验极为严格，并支援四种转型运算子（cast operator），分别是：

1. *reinterpret\_cast*
2. *static*\_*cast*
3. *dynamic*\_*cast*
4. *const\_cast*（ 本书并不讨论此运算子）

我将先在这里预演一遍，看看结果，精确意义则留待稍后各节再深入介绍。

#### 165

多型与虚拟（*Polymorphism in C++*）

前面的两个转型动作如果改采 *reinterpret\_cast* 如下：

char str[] = "Hello";

char \*pc = reinterpret\_cast<char\*>(&str);

long larray[3] = { 11, 22, 33 };

long \*pl = reinterpret\_cast<long\*>(&larray);

可以顺利通过编译。可见 *reinterpret*\_*cast* 似乎与 C 语言时代的强制转型有异曲同工之妙。是否真是如此，待会儿就知道。如果改为 *static*\_*cast*：

char str[] = "Hello";

char \*pc = static\_cast<char\*>(&str);

long larray[3] = { 11, 22, 33 };

long \*pl = static\_cast<long\*>(&larray);

却又不行，会出现编译错误讯息如下：

error C2040: 'static\_cast' : 'char \*' differs in levels of indirection from 'char (\*)[6]'

error C2040: 'static\_cast' : 'long \*' differs in levels of indirection from 'long (\*)[3]'

改为 *dynamic*\_*cast* 也不行：

char str[] = "Hello";

char \*pc = dynamic\_cast<char\*>(&str);

long larray[3] = { 11, 22, 33 };

long \*pl = dynamic\_cast<long\*>(&larray);

会出现编译错误讯息如下：

error C2680: 'char \*' : invalid target type for dynamic\_cast error C2680: 'long \*' : invalid target type for dynamic\_cast

稍后我会对于这三个型别转换运算子做更详细的解释。

#### 166

第３章 型别转换（Type Casting）

C++ 的型别转换比起 C 复杂得多，因为 C 的指标通常用来指向资料（在 C 语言中使用函式指标，算是高阶技法；把函式指标拿来转型，更是少之又少），而 C++的指标常常用来指向 object。透过指向 object 的指标，可以唤起 object 的member functions，也可以（可能）取得 object 的 data members。但是我们知道 C++ classes 有继承的能力，而 base type pointer 可能指向 derived object（在多型应用下尤其常见）。因此，如果型别转换没有搞清楚，没有 "type-safe" 的话，程序逻辑很容易出错。

面对 C++ classes 的继承体系，我们操作指标时（例如透过指标唤起 member functions），操作结果主要是以指标的型别为依据（除非是指向 virtual member function）。然而指标真正指向哪一个 object，object 的型别是否与指标型别相符， 却只有在执行时期才能确定。是的，我们无法从指标型别去推论或判断它真正所指的 object（尤其在多型发生的时候）。这正是关键所在。

谈到指标在 classes 继承体系中游走（转型），我们必须先了解两种游走方向的意义。这两种游走方向是「向上游走（upcast）」和「向下游走（downcast）」。

#### 167

多型与虚拟（*Polymorphism in C++*）

## Up Cast（向上转型）

所谓 up cast，意思是把 derived type 指标，转型为 base type 指标。由于我们通常把型别继承体系图（type hierarchy）画成 base type 在上而 derived type 在下， 所以这样的转型动作称为 up cast（向上转型）。例如：

class B { };

class C : public B { }; class D : public C { };

void main()

{

D\* pd = new D; // (1) pd point to a complete D object. C\* pc = pd; // (2) ok, up casting

B\* pb = pd; // (3) ok, up casting

}

Up cast 是一种安全的行为，也是一种不必明言的（implicitly）型别转换。上述程序码 (1) (2) (3) 的意义是：

1. *pd* 指向一个完整的 *D* object；
2. *pc* 指向该 *D* object 中的 *C* subobject（ 合法， 没问题）；
3. *pb* 指向该 *D* object 中的 *B* subobject（ 合法， 没问题）。

*pc* 和 *pb* 的活动范围（权力范围）只分别在 *C* subobject 和 *B* subobject 之中：

D object

C\* pc

D\* pd

class D

class C

class B

B

subobject

C

subobject

pd’s scope

pb’s scope

B\* pb

pc’s scope

class hierarchy

**168**

第３章 型別轉換（Type Casting）

但是不要忘了，*pc* 和 *pb* 虽然受限于其形式（也就是受到其 type 的束缚），毕竟它们真正所指的还是个完整的 *D* object（这开始有点多型的味道了）。如果我再把它们还原为 *D* type 呢？像这样：

D\* pd2 = pb; // error, down casting

会出现编译错误讯息如下：

error C2440: 'initializing' : cannot convert from 'class B \*' to 'class D \*' (new behavior; please see help)

这样的编译错误，对执行时期实际情况而言，是不合理的，因为 *pb* 的型别虽为*B*\*，毕竟真正指向的是一个 *D* object。但这个编译错误，对程序开发阶段而言却又是一种良好的保护，因为从语法来分析，编译器真的不知道在执行时期 *pb* 会指向 *D* object，编译器只认为 *pb* 会指向 *B* object。

有没有什么办法，既可以有编译时期的良好保护，又兼顾执行时期的多型可能性呢？我将在 *dynamic*\_*cast* 那一节（p.171）回答这个问题。

## Down Cast（向下转型）

所谓 down cast（向下转型），意思是把 base type 指标转型为 derived type 指标。由于我们通常把型别继承体系图（type hierarchy）画成 base type 在上，derived type在下，所以这样的转型动作称为 down cast（向下转型）。例如：

class B { };

class C : public B { }; class D : public C { };

void main()

{

B\* pb = new B; // (1) pb point to a complete B object. C\* pc = pb; // (2) **error**, down casting

D\* pd = pb; // (3) **error**, down casting

}

**169**

多型與虛擬（*Polymorphism in C++*）

###### D object

C\* pc

###### D\* pd B\* pb

class D

class C

class B

B

subobject

pb’s

###### pd’s scope C

scope

###### pc’s scope

subobject

###### class hierarchy

上面程序码 (1) (2) (3) 的意义是：

1. *pb* 指向一个完整的 *B* object；
2. *pc* 指向该 *B* object
3. *pd* 指向该 *B* object

但是 (2) 和 (3) 会出现编译错误讯息如下：

error C2440: 'initializing' : cannot convert from 'class B \*' to 'class C \*' (new behavior; please see help)

error C2440: 'initializing' : cannot convert from 'class B \*' to 'class D \*' (new behavior; please see help)

这是一种很好的保护。从形式（型别）的角度来看，*pc* 的活动范围（权力范围） 有 *C* type 那么大（*C* 是 *B* 的超集），*pd* 的活动范围（权力范围）有 *D* type 那么大（*D* 也是 *B* 的超集），如果透过 *pc* 或 *pd* 去取 *C* type 或 *D* type 之中超越 *B* type 的东西（不管是 data members 或 member functions），就形式而言是应该被允许的，而就实质而言却又会造成程序崩溃。所以编译器才要把你拦下来， 不让通关。

接下来让我们看看C++ 对于型别转换提供了哪些协助。

#### 170

第３章 型別轉換（Type Casting）

## dynamic\_cast 运算子

C++ 語法：

dynamic\_cast<type-id>(expression)

这个运算子可以把 expression 转换为一个 type-id 物件型别，其中 type-id 必须是程序中已定义之 class 的指标或 reference，或者是一个 *void*\*。如果 type-id 是指标，expression 也必须是指标；如果 type-id 是 reference，expression 必须是一个L-Value。

### 单一继承（Single Inhertance）情况下

如果 type-id 是个指标，指向 expression 的「可存取，且直接（或间接）」的 base class，那么 *dynamic*\_*cast* 运算子将传回一个指标，指向 type-id subobject。例如：

class B { ... };

class D

class C

class B

class C : public B { ... };

class D : public C { ... };

void f(D\* pd)

{

C\* pc = dynamic\_cast<C\*>(pd);

// 没问题，因为 C 是 D 的一个 direct base class。

// 现在 pc 指向 pd 所指之 D object 中的 C subobject。

B\* pb = dynamic\_cast<B\*>(pd);

// 没问题，因为 B 是 D 的一个 indirect base class。

// 现在 pb 指向 pd 所指之 D object 中的 B subobject。

...

}

#### Up Cast

上面这两个转换都是一种 up-cast，因为指标往类别继承体系的上层转换。up-cast

是一种隐喻的，不必言明的（implicit）转换，一定可以成功。所以上述两个转换

#### 171

多型與虛擬（*Polymorphism in C++*）

其实不动用 *dynamic*\_*cast* 运算子也没有关系。

如果 type-id 是 *void*\*，那么传回的指标就指向 expression 真正所指的完整物件。至于 expression 真正指向什么，执行时期才能决定。例如：

class A { ... };

class B { ... }; // 注意 A 和 B 之间没有继承关系。

class A

void f()

{

class B

A\* pa = new A; B\* pb = new B;

void\* pv = dynamic\_cast<void\*>(pa); // 現在 pv 指向 A object

pv = dynamic\_cast<void\*>(pb); // 現在 pv 指向 B object

...

}

但请注意，以此例的 *A* 和 *B* 而言（它们之间没有继承关系），如果你这么做：

void f()

{

A\* pa = new A; assert(pa);

B\* pb = dynamic\_cast<B\*>(pa); // 轉型失敗。

assert(pb); // pb 將為 null。

}

执行时会出现：

Assertion failed: pb, file xxx.cpp, line xxx abnormal program termination

也就是说，*dynamic\_cast* 检查出 *A* 和 *B* 并没有继承关系，所以把 *pa* 的转型结果（也就是 *pb*）设为 null。

#### Down Cast

如果 expression 是 type-id 的一个 base class，这就等于我们要把一个指标往类别继承体系的下层转换，也就是 down-cast。这不一定能够转型成功，它会引发一个

#### 172

第３章 型別轉換（Type Casting）

执行时期的检查（RTTI），看看是否 expression 确实指向一个 type-id 完整物件。如果是，就传回指标，指向一个 type-id 完整物件；如果不是，就传回 null。例如：

class B { ... };

class D : public B { ... };

class D

class B

void f()

{

B\* pb1 = new D; // base type 指標指向 derived object B\* pb2 = new B;

D\* pd1 = dynamic\_cast<D\*>(pb1);

// 没问题，因为 pb1 本就指向完整的 D object。

D\* pd2 = dynamic\_cast<D\*>(pb2);

// 转换失败，因为 pb2 原本指向 B object 而非 D object。

// 现在，pd2 将等于 null。

...

}

下面是一個完整的程式實例：

#0001 // build : cl /GR dyncast.cpp #0002 #include <iostream.h>

#0003 #include <assert.h>

#0004

class D

class C

class B

#0005 class B { public: virtual void func() { }; }; // polymorphic #0006 class C : public B { };

#0007 class D : public C { }; #0008

#0009 void main()

#0010 {

#0011 D\* pd = new D; #0012

#0013 C\* pc = dynamic\_cast<C\*>(pd); // upcast

#0014 assert(pc); // it will be ok #0015

#0016 B\* pb = dynamic\_cast<B\*>(pd); // upcast

#0017 assert(pb); // it will be ok #0018

#0019 B\* pb1 = new D; // unclear but ok. #0020 B\* pb2 = new B;

#0021 assert(pb1); // it will be ok #0022 assert(pb2); // it will be ok #0023

#### 173

多型與虛擬（*Polymorphism in C++*）

#0024 D\* pd1 = dynamic\_cast<D\*>(pb1); // downcast but ok #0025 D\* pd2 = dynamic\_cast<D\*>(pb2); // downcast，error

#0026 assert(pd1); // it will be ok

#0027 assert(pd2); // it will be fail (pd2 will be null)

#0028 }

此程式之輸出結果如下：

Assertion failed: pd2, file dyncast.cpp, line 27 abnormal program termination

是的，我们可以在 *dynamic\_cast* 之后，检查转型结果是否为 null，以判断转型是否成功。

请注意了，上例的 *B* 有一个 virtual function，为的是使 *B* 以下的整个继承体系成为 polymorphic type。原因是，如果发生 down-cast，那么 *dynamic*\_*cast* 只有在面对 polymorphic type 时才有正常的演出，而我们都知道，在 C++ 语言中要制造出一个 polymorphic type 的唯一方法就是在 class 中设计 virtual function(s)。

当然，我想你必然好奇地想知道 *dynamic*\_*cast* 施行于 non-polymorphic type 的情况。如果上一个程序的 class *B* 没有任何 virtual function，那么施行 *dynamic*\_*cast* 的那两行码（L24, L25）会出现编译错误：

line(24) error C2683: dynamic\_cast : 'B' is not a polymorphic type line(25) error C2683: dynamic\_cast : 'B' is not a polymorphic type

但 L13 和 L16 可以安全通过编译，因为它们是 up-casting。

另请注意，你必须在 Visual C++ 编译器上设定 /GR 选项（意思是要拥有 RTTI

能力），否则 *dynamic*\_*cast* 施行于 down-cast 情况时，会出现警告讯息如下：

line(24) warning C4541: 'dynamic\_cast' used on polymorphic type 'class B' with /GR-; unpredictable behavior may result

line(25) warning C4541: 'dynamic\_cast' used on polymorphic type 'class B' with /GR-; unpredictable behavior may result

此时 *dynamic\_cast* 不保证能够发挥它的正常功能。不理会上述警告而强渡关山的

#### 174

第３章 型別轉換（Type Casting）

结果是：

Assertion failed: pd1, file dyncast.cpp, line 26 abnormal program termination

这不是正确的执行结果，L26 的 *pd1* 本来应该是可以转型成功的！

### 多重继承（Multiple Inhertance）情况下

在多重继承的情况下：

class A { public: virtual void func() { }; }; // polymorphic type class B : public A { };

class D

class C

class B

class A

class A

class C : public A { };

class D : public B, public C { };

*D* 物件指标可以被安全地转型为 *B* 或 *C* 型别。然而如果 *D* 指标被转型为 *A* 型别，它应该指向哪一个 *A* 呢？这是一种暧昧不明（ambiguous）的情况：

void f() // 注意，别忘了在 VC++ 中使用 /GR 编译选项

{

D\* pd = new D;

A\* pa = dynamic\_cast<A\*>(pd); // 错误，暧昧不明，

// pa 不知指向哪一个 A object。

}

编译错误出现在 *dynamic*\_*cast* 那一行：

warning C4540: dynamic\_cast used to convert to inaccessible or ambiguous base; run-time test will fail ('class D \*' to 'class A \*')

这种情况可以藉着两个明确的型别转换来解决：

void f()

{

D\* pd = new D;

B\* pb = dynamic\_cast<B\*>(pd); // 先转型为 B。

A\* pa = dynamic\_cast<A\*>(pb); // 再转型为 A，指向图左的 A object

}

上述的最后两个动作显然是 up-cast，所以，写成这样也可以：

**175**

多型與虛擬（*Polymorphism in C++*）

void f()

{

D\* pd = new D;

B\* pb = pd; // upcast. 先转型为 B。

A\* pa = pb; // upcast. 再转型为 A（指向图左的 A object）。

}

### 虚拟继承（Virtual Inhertance）

如果使用虚拟继承，情况会更加暧昧不明。以下一个例子而言：

class A { public: virtual void func() { }; }; // polymorphic type class B : public virtual A { };

class E

class D

class C

class B

class B

class A

class C : public B { }; class D : public B { };

class E : public C, public D { };

*A* 是一个 virtual base class。如果我们令指标 *A*\* *pa* 指向一个完整的 *E* object（以形式而言 *pa* 指向 *E* object 中的 *A* subobject）。现在把该指标以 *dynamic*\_*cast* 转型为 *B*\*，会出现暧昧不明的情况：

void f()

{

A\* pa = new E;

B\* pb = dynamic\_cast<B\*>(pa); // 转型失败。fail due to ambiguity.

assert(pb);

}

程序执行时会失败于 assert(*pb*) 那一行：

Assertion failed: pb, file xxx.cpp, line xxx abnormal program termination

解决之道是，先把该 *pa* 转型指向完整的 *E* object（虽然是 down-cast，但符合成功条件），再依次做 up-cast 动作，转型为我们要的 *B* object：

void f()

{

**176**

第３章 型別轉換（Type Casting）

A\* pa = new E;

E\* pe = dynamic\_cast<E\*>(pa); // downcast to complete E object first. assert(pe);

C\* pc = dynamic\_cast<C\*>(pe); // upcast to the C subobject. assert(pc);

B\* pb = dynamic\_cast<B\*>(pc); // upcast to the B subobject. assert(pb);

}

这一次就不会再出现 Assertion failed 的情况了。

### 横向转型（cross cast）

假设一个继承体系如下：

class A { public: virtual void funcA() { }; }; // polymorphic type class B : public A { };

class C : public A { };

class D { public: virtual void funcD() { }; }; // polymorphic type class E : public B, public C, public D { };

class E

class D

class C

class B

class A

class A

如果我令指标 *D*\* *pd* 指向一个完整的 *E* objec（t 以形式而言 *pd* 指向 *E* object 中

的 *D* subobject）。现在我把该指标以 *dynamic*\_*cast* 转型为最左边的 *A*\*，会出现暧昧不明的情况：

void f()

{

D\* pd = new E;

A\* pa = dynamic\_cast<A\*>(pd); // 转型失败。fail due to ambiguity.

assert(pa);

**177**

多型與虛擬（*Polymorphism in C++*）

}

程序执行时会失败于 assert(*pa*) 那一行：

Assertion failed: pa, file xxx.cpp, line xxx abnormal program termination

解决之道是，先把 *pd* 转型指向完整的 *E* object（虽然是 down-cast，但符合成功条件），再依次做 up-cast 动作，转型为我们要的 *B* object：

void main()

{

D\* pd = new E;

E\* pe = dynamic\_cast<E\*>(pd); // downcast to the complete E object first. assert(pe);

B\* pb = pe; // upcast assert(pb);

A\* pa = pb; // upcast assert(pa);

}

事实上，从本例的 *D*\* 转型为 *B*\*，是一种横向转型（cross cast）。*dynamic*\_*cast*

可用于横向转型，上述程序码可简化为：

void main()

{

D\* pd = new E;

B\* pb = dynamic\_cast<B\*>(pd); // cross cast assert(pb);

A\* pa = pb; // upcast assert(pa);

}

#### 178

第３章 型別轉換（Type Casting）

## static\_cast 运算子

C++ 語法：

static\_cast<type-id>(expression)

此运算子可以将一个 base class 指标转指向 derived object。这种 down-cast 动作并不总是安全。例如：

class B { ... };

class D

class B

class D : public B { ... };

void f(B\* pb, D\* pd)

{

D\* pd2 = static\_cast<D\*>(pb); // down-cast，不安全，

// 因為 pb 可能只是指向 B object。B\* pb2 = static\_cast<B\*>(pd); // 安全。

}

此运算子和 *dynamic\_cast* 不同的地方在于，以上例而言，它对于 *pb* 不会有执行时期检验动作，我们还是获得了一个转型后的 non-null *pd2*。当 *pb* 指向的不是一个 *D* object，而是一个 *B* object，程序中使用转型后的结果 *pd2*，就可能导至惨重的灾情。

写一个实例瞧瞧：

#0001 // build : cl stacast.cpp #0002 #include <iostream.h>

class D

class B

#0003 #include <assert.h>

#0004

#0005 class B { #0006 public:

#0007 B::B() { m\_B = 3; }

#0008 void funcB() { cout << "B::funcB() : m\_B=" << m\_B << endl; }; #0009 private:

#0010 int m\_B;

#0011 };

#0012

#0013 class D : public B {

**179**

多型與虛擬（*Polymorphism in C++*）

#0014 public:

#0015 D::D() { m\_D = 5; }

#0016 void funcD() { cout << "D::funcD() : m\_D=" << m\_D << endl; }; #0017 private:

#0018 int m\_D;

#0019 };

#0020

#0021 void f(B\* pb, D\* pd) #0022 {

#0023 D\* pd2 = static\_cast<D\*>(pb); // downcast，unsafe

#0024 assert(pd2);

#0025 B\* pb2 = static\_cast<B\*>(pd); // upcast，safe。#0026 assert(pb2);

#0027

#0028 pb->funcB(); // 3

#0029 pd->funcB(); // 3

#0030 pd->funcD(); // 5

#0031

#0032 pb2->funcB(); // 3

#0033 pd2->funcB(); // 3

#0034 pd2->funcD(); // 0 <-- error

#0035 }

#0036

#0037 void main()

#0038 {

#0039 D\* pd = new D; #0040 B\* pb = new B; #0041 f(pb, pd);

#0042 }

我把 *B::m\_B* 設初值為 3，*D::m\_D* 設初值為 5。在 *f()* 函式中：

#0023 D\* pd2 = static\_cast<D\*>(pb); // pb 被轉換為 D\* 並指定給 pd2，

...

#0034 pd2->funcD(); // 利用 pd2 呼叫 D::funcD()

这些码都可以通过编译，也可以执行，但执行结果却有误，L34 根本就抓到了不该抓的东西。我可以下面这张图来说明这个结果：

#### 180

第３章 型別轉換（Type Casting）

D\* pd

pd’s scope

D object

B\* pb2 D\* pd2 pb2’s

scope

m\_B (3)

pd2’s scope

m\_D (5)

B object

m\_D (?)

m\_B (3)

B\* pb

pb’s scope

當然，如果程式是這樣：

#0037 void main()

#0038 {

#0039 D\* pd = new D;

#0040 B\* pb = new D; // 注意，pb 指向一個完整的 D object #0041 f(pb, pd);

#0042 }

执行结果就是正确的：

|  |  |  |
| --- | --- | --- |
| #0032 | pb2->funcB(); | // 3 |
| #0033 | pd2->funcB(); | // 3 |
| #0034 | pd2->funcD(); | // 5 <-- OK |

所以， 为了确保拥有正确的转型， 不至发生误谬的执行结果，我们应该以*dynamic*\_*cast* 取代 *static*\_*cast*，那么一旦有不正确的转型动作，就会得到 null 指标，而我们可以在程序中判断并决定接下来的动作：

#0021 void f(B\* pb, D\* pd) #0022 {

#0023 D\* pd2 = dynamic\_cast<D\*>(pb); // downcast #0024 if (pd2) {

#0025 ...

这么说来，*dynamic*\_*cast* 比起 *static*\_*cast* 完美，为什么还需要 *static*\_*cast* 呢？ 原因是成本问题。*static*\_*cast* 只依赖出现在 cast 运算子上的信息（也就是操作数的型别）来做决定，*dynamic*\_*cast* 却还需要取 object 的 *type*\_*info*（也是一个 class object）来比对，效率上当然差些。

#### 181

多型与虚拟（*Polymorphism in C++*）

*static*\_*cast* 能否应用在继承体系以外的型别转换呢？答案是可以，但相同地，转型后的结果是否安全，程序员自己心里得有个谱！没有任何人可以帮你。延续上例，如果另加上一个 class *D1*：

class D1 { }; // 與 class B 和 class D 沒有任何關連

class D

class B

那么这样的动作：

D1\* pd1 = new D1;

D\* pd = static\_cast<D\*>(pd1); // fail due to bad\_cast B\* pb = static\_cast<B\*>(pd1); // fail due to bad\_cast pd->funcD();

class D1

pb->funcB();

虽然可以执行，结果却不可预期。

同样的道理，*static*\_*cast* 可应用于 class types 以外的 data types，如语言内建的int、float、char 等等。你可以这么做：

char ch; int i = 65;

float f = 2.5; double dbl;

ch = static\_cast<char>(i); // int to char

dbl = static\_cast<double>(f); // float to double

i = static\_cast<unsigned char>(ch); // ch to unsigned char

把 int 转型为 char，可能使得 char 没有足够的位元来放置完整的 int 内容。但这是你自己必须负起的责任（既然你坚持要这么使用 *static*\_*cast*）。

另外，请注意，任何 expression 都可以被 *static*\_*cast* 运算子转换为 void 型别， 而且可以选择性地加上 const、volatile 或 unaligned 等修饰属性。

#### 182

第３章 型別轉換（Type Casting）

## reinterpret\_cast 運算子

C++ 語法：

reinterpret\_cast<type-id>(expression)

此运算子允许任何指标转换为任何其他型别。所以如果程序员自己不注意，使用这个运算子很容易出错（因为编译器不会为你做检查）。

这个运算子最为接近旧时代的 C 转型动作。

## const\_cast 运算子

C++ 語法：

const\_cast<type-id>(expression)

此运算子用来转换expression 的「const 属性」，唯不在本书讨论之列。

除了介绍上述的三个型别转换运算子，接下来我还要介绍两个很特殊的运算子， 分别是 # 和 ##。这两个运算子在本书第５章的 Polymorphism 实例中，有吃重的演出。你也可以在 Bjarne Stroustrup 所着的 ***The C++ Programming Language 2/e*** p609~p610 看到这两个运算子的介绍。奇怪的是，我却无法在同一本书的更新版（***The C++ Programming Language 3/e***）索引中找到 # 运算子，我只找到了 ## 运算子。

#### 183

多型與虛擬（*Polymorphism in C++*）

##### #（ stringizing operator）

# 是所谓的 "stringizing"（字符串化）运算子，可以把巨集的参数转换为字符串常数， 等于是让编译器为我们产生原始文字码。它只适用于带有参数的巨集。

由于将参数转换为字符串，所以字符串中的空格符或任何需要 escape sequence 的符号（例如双引号 " 或倒斜线 \），就需要特别处理。巨集中的第一个 token 之前以及最后一个 token 之后的空格符，都会被忽略不处理。例如：

#define stringer( ) printf( #x "\n" )

x

巨集参数 *x* 之前和之后各有一个空格符，都不会被处理。如果两个 token 之间有许多个空格符，会被视为单独一个空格符，例如我们这样呼叫 *stringer* 巨集：

stringer( );

token1 token2

在 *token1* 和 *token2* 之间有 5 个空格符，但巨集扩展后得到的是：

printf( "\n" );

"token1 token2"

// token1 和 token2 之間只留有 1 個空白字元

如果我们这样呼叫巨集：

stringer( In quotes in the printf function call\n );

*stringer* 巨集将会扩展为：

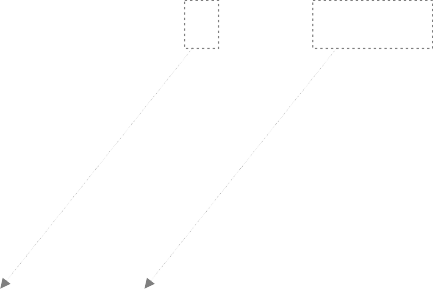
printf( "In quotes in the printf function call\n" "\n" );

至于那些需要 escape sequence 的符号（例如双引号 " 或倒斜线 \）及其注意事项，请参考编译器说明文件。

#### 184

第３章 型別轉換（Type Casting）

##### ##（ merging operator or token-pasting operator）

## 是所谓的 "merging"（合并）运算子，或称为 "token-pasting" 运算子。它会把两个 token 合并为一个 token，然后再交给编译器去解析（parsing）。等于是让编译器为我们产生原始文字码。举个实例：

#define paster(n) printf("token" #n "=%d", token##n ) int token9 = 9;

如果我们这样呼叫 *paster* 巨集：

paster(9);

巨集展开后导至：

printf("token" "=%d", );

"9"

token9

也就是：

printf("token9=%d", token9); // output: token9=9

# 和 ## 运算子在 MFC source 中有颇为吃重的运用，它们被安放在一组极重要的巨集之内：

DECLARE\_DYNAMIC IMPLEMENT\_DYNAMIC

这组巨集与执行时期型别鉴定（RTTI）有重要关联。而这一组巨集又是以下两组巨集的基础：

DECLARE\_DYNCREATE IMPLEMENT\_DYNCREATE

DECLARE\_SERIAL IMPLEMENT\_SERIAL

前一组巨集与动态生成（dynamic creation）有关，后一组巨集与物件永久保存

#### 185

多型與虛擬（*Polymorphism in C++*）

（persistence）有关。本书第５章将以 MFC 的技术为蓝本，实作一套具有 RTTI 和 Dynamic Creation 机能的 class hierarchy 出来，做为多型应用的一个大型实例。

#### 186

第５章 Polymorphism（多型）的應用

**第５章**

# Polymorphism

的应用

纳须弥于芥子

C++ 的运用有两种层次，一种是自行开发 classes，一种是选用功能齐全威力强大的市售 class libraries。从轮子开始做起的人不多，也不够聪明。能够站在别人的肩膀上，运用别人的心血结晶，并且有能力衍生（derived）适用于自己项目目的之 classes，才是运用C++ 物件导向性质的最佳路线。

有什么方法可以既学习到前人种树的功力，又享受到后人乘凉的福利？最好的训练莫过于观摩名家作品了。一来学习大家风范，二来用心观摩之后必然对于该作品了如指掌，使用起来也就反掌折枝，了无罣碍。

虽然市售的 class libraries 总是附有原始码（这是这个领域一向的习惯），但是冒冒然一头埋进庞大的架构，在重峦叠嶂里头迷路是想当然耳的事情。行家眼中的壮丽山川在你看来却成了穷山恶水，那可不妙。面面俱到（因而复杂无比）的设计，固然有利于商品化，对学习与教学的效果却一定会大打折扣。因此，如果要

**227**

多型與虛擬（*Polymorphism in C++*）

以这些极其庞大极其复杂的市售 class libraries 为技术学习对象，一定要简化、简化、再简化。

这一章，我挑选在 C++ class libraries 市场上占有率最高的 MFC（Microsoft Foundation Classes），在特定范围内将它极简化之后，引介各位体会 polymorphism

（多型）的实作技术与运用之美。我会在简化地点加上说明，提醒你值得改善的方向。为了让它成为一个纯粹的 C++ 程序，不涉及任何特定作业平台，我把 MFC 原始码中凡是用到 Win32 API 的地方，完全改用 ANCI C runtime library，以完成诸如字符串比对、档案读写、存储器配置等等底层动作。

以下的所有 classes declaration、implementation、macro，都是从 MFC 中剪裁并稍加修改而来。为了解说语气的顺畅，我必须使用第一人称来说明这些设计理念， 但文中的「我」，一旦涉及程序实作码，绝大部份其实代表的是 Microsoft's MFC team。

我曾经在 深入浅出 MFC 第一版和第二版（侯俊杰着/松岗）第３章对于类似主题有所介绍，可能已有不少读者看过该书。本章和「深」书第３章有些什么异同呢？技术上两者都涵盖 RTTI 和 Dynamic Creation，但在本章之中我对于设计理念着墨较多，对于物件导向的观念、多型的观念、C++ 高阶技巧、为什么这样设计、为什么那样设计... 有较多的解释。技术层面上最大的差异在于，「深」书第

３章还涵盖了对 MFC message mapping 和 command routing 的模拟，然而讯息是Windows 平台的特性，当然我不会在本章讨论之。此外，「深」书第３章还涵盖了隐藏于 MFC 应用程序表面之后的四大类别（ *CWinApp* 、*CFrameWnd* 、*CDocument*、*CView*）的最初实现（instantiation）过程，同样地，我也不会在一本专讲 C++ 物件导向观念的书中谈它。至于本章在技术上特别突出的是，模拟出persistence 性质。这项性质关系到 archive、file、list、array 的实作，涵盖面很广，

我相信对你的物件导向技术必定有非常大的助益。

#### 228

第５章 Polymorphism（多型）的應用

目标 - 三大服务的公共基础建设

我的目标是希望呈现一个 class library 基础建设，这个基础建设提供三大服务：

1. Run Time Type Identification（ RTTI） - 可以在执行时期供应

objects type 的查询服务。

1. Dynamic Creation - 可以在执行时期产生 class libraries（ 及你据所衍生） 之任何一个 class 的 objects。
2. Persistence - 可以将程序之中由许多 objects 组成的资料结构永久保存起来， 也就是储存到档案里头。当然， 更重要的是必须能够读回来并原貌重组资料结构。读回来的过程， 必须一一产生原来的 class objects， 也就是必须用到项目 **2** 的 dynamic creation 服务以及项目 **1** 的 RTTI 服务。

图 5.1 展示这三大性质的图解说明。

这三项都是物件导向领域里头的高难度动作。项目 1 目前已由 C++ 语言本身支援，项目 2 和项目 3 则尚未有语言层面的解决方案。我们可以从本章实例中看到名家（我是指 Microsoft）如何漂亮地解决这些问题。我所谓漂亮，意思是尽量贴近（模拟）语言层次，只以一个 statement（本例中其实是个 macro）就完成上述功能的铺设。从历史上看，C++ 语言原本并没有支援 RTTI，后来才以 *type\_info* 和 *typeid()* 支援之（见第１章），那么，历史未尝不可能重演，谁知道将来 C++ 会不会加上类似 *declare\_dyncreate* 、 *implement\_dyncreate* 运算子或是诸如*declare\_persistence*、*implement\_persistence* 等运算子来完成项目 2 和 3 的功能

 ?! 编译器把本章的这些技术（或类似技术）含入，实非难事。

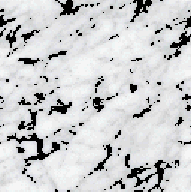
**229**

多型与虚拟（*Polymorphism in C++*）

Document（资料结构设计）

...

RTTI &



View（资料的显示）



Persistence

RTTI &

Dynamic Creation

###### Disk File

000000: 06 00 08 00 43 45 6C 6C 69 70 73 65 00 00 40 40 CEllipse..@@

000010: 00 00 40 40 00 00 E0 40 00 00 A8 41 07 00 43 43 ..@@...@...A..CC

000020: 69 72 63 6C 65 00 00 A0 40 00 00 A0 40 00 00 E0 ircle...@...@...

000030: 40 09 00 43 54 72 69 61 6E 67 6C 65 00 00 00 00 @..CTriangle....

000040: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00 .......?........

000050: 00 00 80 3F 05 00 43 52 65 63 74 33 33 B3 40 9A ...?..CRect33.@.

000060: 99 D9 40 00 00 40 40 00 00 10 41 07 00 43 53 71 ..@..@@...A..CSq

000070: 75 61 72 65 CD CC 4C 40 9A 99 89 40 00 00 C0 40 uare..L@...@ @

000080: 07 00 43 53 74 72 6F 6B 65 07 00 01 00 00 00 02 ..CStroke.......

000090: 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00 06 ................

0000A0: 00 00 00 07 00 00 00 00 00 00 00 00 00 00 00 00 ................

图 5.1 物件导向三大特性

上述三大服务的基础建设完成之后，为了示范其威力，我必须实作一个 class library 验证之。对于 RTTI 的验证很简单，我只需以 C++ 语言之 *typeid()* 运算子为蓝本。Dynamic Creation 和 Persistence 两项性质的验证就稍麻烦，因为我必须真的制作一个资料结构（最好能够稍稍复杂些，例如拥有 list、array 等等）， 把它们统统写到档案去，再把它们读出来。

#### 230

第５章 Polymorphism（多型）的應用

#### 模块切割

我将完成一个能够支援上述三大服务的 class library。由于实作码脱胎自 MFC， 所以我把这个轻型的 class library 称为MFCLITE（MFC-Lite）。

我还将完成一个 "Shape" hierarchy，示范如何继承 MFCLITE 的所有美好性质。

最后我将完成一个 Test 程序，示范如何根据 MFCLITE 和 Shape hierarchy，设计出一个可容纳各种几何形状的工程图资料结构，并以此资料结构示范上述三大服务的运用。

本章所有的程式模組安排如下：

* + MFCLITE.H： 内含 MFCLITE 的所有 classes 宣告及其 inline 函式， 以及各种 macros 定义。任何欲使用 MFCLITE 的程序都必须含入此档。
  + MFCLITE.CPP： 内含 MFCLITE 的所有 classes 实作码。此档案需含入 MFCLITE.H。如果将这个模块制作成 DLL， 最能够模拟真实应用情况。但欲在 DLL 之中开放（ 汇出， export） 各个 classes， 视各作业平台之不同， 可能我们需要为每一个 classes 多做一些动作1。为了尽量与平台无关，我决定放弃制作为 DLL，编译后仅以 .OBJ 档出现。
  + SHAPE.H：内含 Shape class hierarchy 的宣告，示范如何使用 MFCLITE

（ 所以必须含入 MFCLITE.H）。此应为 MFCLITE user 的项目部份。

* + SHAPE.CPP ： 内含 Shape class hierarchy 的实作内容。此档需含入SHAPE.H。此应为 MFCLITE user 的项目部份， 编译后以 .OBJ 档出现。

1. 例如在Windows 系统之中，为了开放 DLL classes 供外界使用，必须在 class head 的 "class"

关键字之后以及 class 名称之前，加上 \_declspec(dllexport)，如下所示：

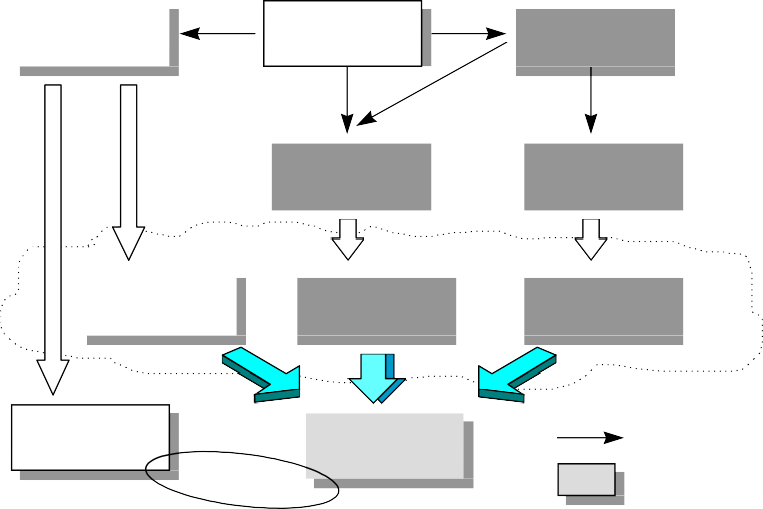
class \_declspec(dllexport) CObject { ... }

#### 231

多型与虚拟（*Polymorphism in C++*）

 TEST.CPP：测试用的主程序。此应为 MFCLITE user 的项目部份，需含 入 MFCLITE.H 和 SHAPE.H ， 并 与 上 述 的 MFCLITE.OBJ 和SHAPE.OBJ 联结。

理想的情況



static link

MFCLITE.DLL

be included to

project files

MFCLITE.H

TEST.EXE

SHAPE.OBJ

TEST.OBJ

MFCLITE.LIB

SHAPE.CPP

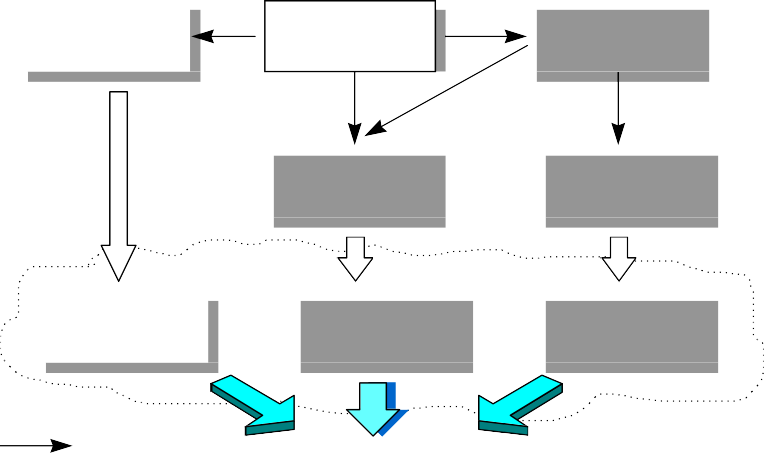
TEST.CPP

SHAPE.H

MFCLITE.CPP

dynamic link

权宜之计



static link

be included to

project files

MFCLITE.H

TEST.EXE

SHAPE.OBJ

TEST.OBJ

MFCLITE.OBJ

SHAPE.CPP

TEST.CPP

SHAPE.H

MFCLITE.CPP

#### 232

第５章 Polymorphism（多型）的应用

#### C++ 编译器环境设定

程序设计开始之际，你当然必须决定使用哪一套 C++ 编译器。目前 PC 上绝大部份的 C++ 编译器都提供两种开发环境：

1. 命令列模式： 可直接在系统提示号如 MS-DOS（ 或 Windows 中的DOS 视窗） 的 ">" 或 UNIX 的 "%" 之下， 下达编译指令和联结指令。
2. 整合环境： 在图形界面（ 如 Windows 或 OS/2 Presentation Manager） 中使用各种视窗工具。

强烈建议各位要有命令列模式下的开发经验，对于编译、联结过程才会比较熟悉。任何 C++ 编译器的命令列模式都要求你设定三个环境变量：

1. PATH - 指向编译器（ compiler）、联结器（ linker）、MAKE 工具的所在目录。
2. LIB - 指出 C/C++ 的 libraries（.LIB） 所在目录。
3. INCLUDE - 指出 C/C++ 的 include files（.H） 所在目录。

我选用 Microsoft Visual C++ 做为本章的开发工具。以下是其环境设定，可写成一个批次档（.BAT）备用：

@echo off

rem \DevStudio\VC is VC 5 default directory. change it in your case. set PATH=E:\DevStudio\VC\BIN;E:\DevStudio\SharedIDE\BIN

set INCLUDE=E:\DevStudio\VC\INCLUDE set LIB=E:\DevStudio\VC\LIB

#### Makefile 语意简介

面对先前的模块切割，我当然可以设计一个批次档（.BAT）如下，包办所有编译联结动作，其中 cl(.exe) 和link(.exe) 是 Visual C++ 的编译器名称和联结器名称：

#### 233

多型與虛擬（*Polymorphism in C++*）

cl -c mfclite.cpp cl -c shape.cpp cl -c test.cpp

link test.obj mfclite.obj shape.obj

但是每次稍微修改任何一个档案，不论是 .H 或 .CPP，便全部重新编译联结，有一点伤及无辜的感觉。有一种工具，几乎任何 C++ 编译器都会内含，称为make.exe，可以读取所谓的makefile，让你在makefile 中指定各档案的依存关系。

下面这个 makefile 可以建造出本章所有的 OBJ 档和 EXE 档：

圖 5.2 test.mak

#0001 # filename : test.mak

#0002 # makefile for test.exe (plain C++ application)

#0003 # build : setting proper environment for VC++ any version, #0004 # then C:\> nmake test.mak <Enter>

#0005

#0006 all:test.exe

#0007

#0008 test.exe : test.obj mfclite.obj shape.obj

#0009 link /out:test.exe test.obj mfclite.obj shape.obj #0010

#0011 test.obj : test.cpp mfclite.h #0012 cl -c /GR test.cpp

#0013

#0014 mfclite.obj : mfclite.cpp mfclite.h #0015 cl -c mfclite.cpp

#0016

#0017 shape.obj : shape.cpp shape.h mfclite.h #0018 cl -c shape.cpp

test.mak 说明如下：

 L1~L4 # 符号表示以下为文字说明（ 类似 C++ 的 // 符号）

 L6 all: 表示这个 makefile 最终目的是要做出 test.exe 档

 L8 makefile 语法是以 : 符号为中心。MAKE 工具会比较 : 符号前后档案的制造先后次序。如果发现右边的任何一个档案比左边的档案

#### 234

第５章 Polymorphism（多型）的应用

新， 就执行下一行动作（ 只要是可在命令列中执行的动作， 都可以指定）。

 L9 任何一个 obj 档更新， 便执行联结动作。

 L11, L14, L17 任何原始档更新， 便执行对应的编译动作。

虽说尽量不要涉及任何作业平台和编译平台，但真正面临实作关头，总是得选择一个编译器，这就可能影响我们所下的编译选项（compile option）。上列 L12 的

/GR 就是因为我在程序中使用 *dynamic\_cast* 运算子（请参考第３章）而下的

Visual C++ 编译选项。

下面就是我在 Windows 95 的 DOS 视窗中执行 test.mak 的画面。环境变量必须先设定好。其中nmake(.exe) 是Visual C++ 的 MAKE 工具名称。

H:\G001\PROG\BIG.05>nmake test.mak

Microsoft (R) Program Maintenance Utility Version 1.62.7022 Copyright (C) Microsoft Corp 1988-1997. All rights reserved.

cl -c /GR test.cpp

Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 11.00.7022 for 80x86 Copyright (C) Microsoft Corp 1984-1997. All rights reserved.

test.cpp

cl -c mfclite.cpp

Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 11.00.7022 for 80x86 Copyright (C) Microsoft Corp 1984-1997. All rights reserved.

mfclite.cpp

cl -c shape.cpp

Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 11.00.7022 for 80x86 Copyright (C) Microsoft Corp 1984-1997. All rights reserved.

shape.cpp

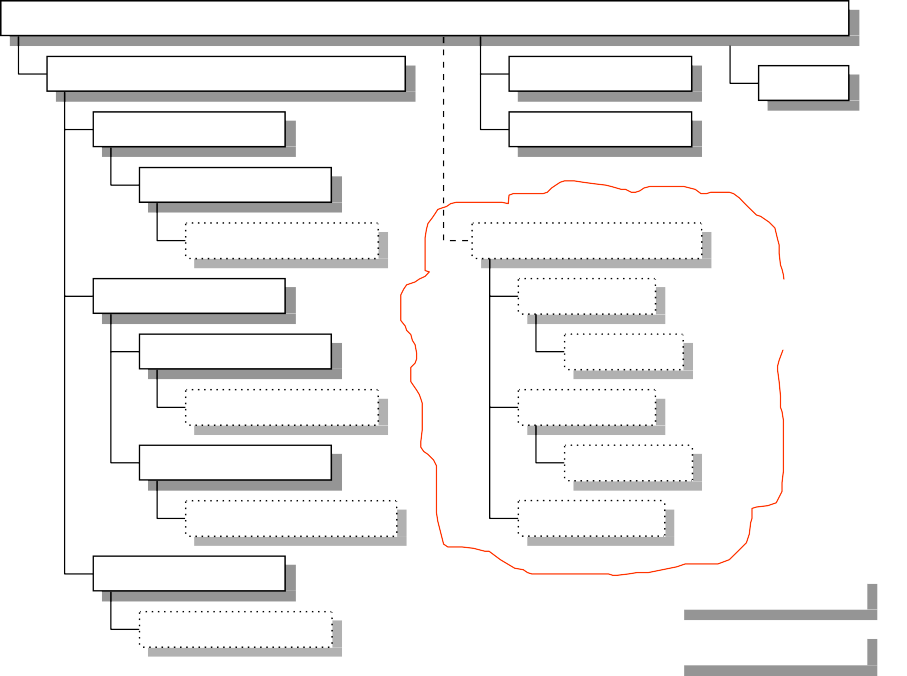
link /out:test.exe test.obj mfclite.obj shape.obj Microsoft (R) 32-Bit Incremental Linker Version 5.00.7022 Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

#### 235

多型與虛擬（*Polymorphism in C++*）

#### 本章所有的 classes

图 5.3 便是本章所完成的 classes 体系。MFCLITE classes 之中真正有大动作的是 *CRuntimeClass*、*CObject*、*CFile*、*CArchive*、*CDWordArray*、*CObList*，其他 classes 在真正的 MFC programming 中很重要，但在此处只是为了用以模拟 MFC 的部份继承体系，所以我把它们都设计为 empty classes。



CMyWinApp

CShape

CSquare

shape hierarchy (for my project)

CRect

CMyView

CCircle

CEllipse

CMyFrameWnd

CTriangle

CDocument

CFrameWnd

CView

CWnd

CWinApp

CDWordArray

CWinThread

CObList

CCmdTarget

CObject

(**MFC-Lite**)

CFile

CMyDoc

CRuntimeClass

CArchive

圖 5.3 本章所有的 classes

#### 236

第５章 Polymorphism（多型）的應用

#### 重新检讨 Shape Hierarchy

图 5.3 中有一大块 classes 是我所谓的 Shape hierarchy，以 *CShape* 为首。这组

classes 体系与先前在第４章出现过的 Shape 不太一样：



CTriangle

CCircle

CEllipse

CSquare

CRect

CShape

CTriangle

CEllipse

CCircle

CRect

CSquare

CShape

图 5.4 Shape hierarchy。图左为本章的设计， 图右为第４ 章的设计。

如何抽取特性、如何分类、如何设计 classes，本是属于 OOA/OOD（物件导向分析/物件导向设计）的范畴。不过这里既然有这么个 shape 例子，我也不妨谈一些基本概念。

由于对 object 布局的认识（第２章），我们知道，一个 derived class object 继承了 base class object 的所有 data members。那么正方形是一种矩形？抑或矩形是一种正方形？圆形是一种椭圆形？抑或椭圆形是一种圆形？

 以数学的角度看， 由于正方形是矩形的一种退化， 圆形是椭圆形的一种退化，所以我们说正方形「是一种」矩形，圆形「是一种」椭圆形。这是为什么我在第４章很直观地把 shape hierarchy 设计成图 5.4 右侧的原因。

 但是以 object 布局来看， derived class 应该有比 base class 更多（ 或至少相等） 的资料。 我们可以依赖一个坐标点 (x, y) 和一个边高

（ height）就描述出正方形，但我们需要一个坐标点 (x, y) 和一个边高

#### 237

多型與虛擬（*Polymorphism in C++*）

（ height）以及一个边宽（ width）才能够描述矩形。从资料继承的角度看， 我们可以说「矩形是一种正方形」！

(x, y) (x, y)

height height

rectangle

square

##### width

* + 同样道理，我们可以依赖一个中心点 (x, y) 和一个半径（ r）就描述出圆形，但我们需要一个中心点 (x, y) 和一个短轴半径（ r）以及一个长轴半径（ r2）才能够描述椭圆形。从资料继承的角度看，我们可以说「椭圆形是一种圆形」！

circle ellipse



r

(x, y)

r

(x, y)

r2

换句话说，物件导向的角度与数学的角度可能不一样，和其他惯常的思考也可能不一样。根据物件布局的思考，我设计出新的 Shape hierarchy 如图 5.4 左。

我甚至看过有人把三角形设计为「是一种」四边形，因为他的四边形的 data members 是两个矢量，各代表四边形的两个边。而三角形也可以使用两个矢量来描述：

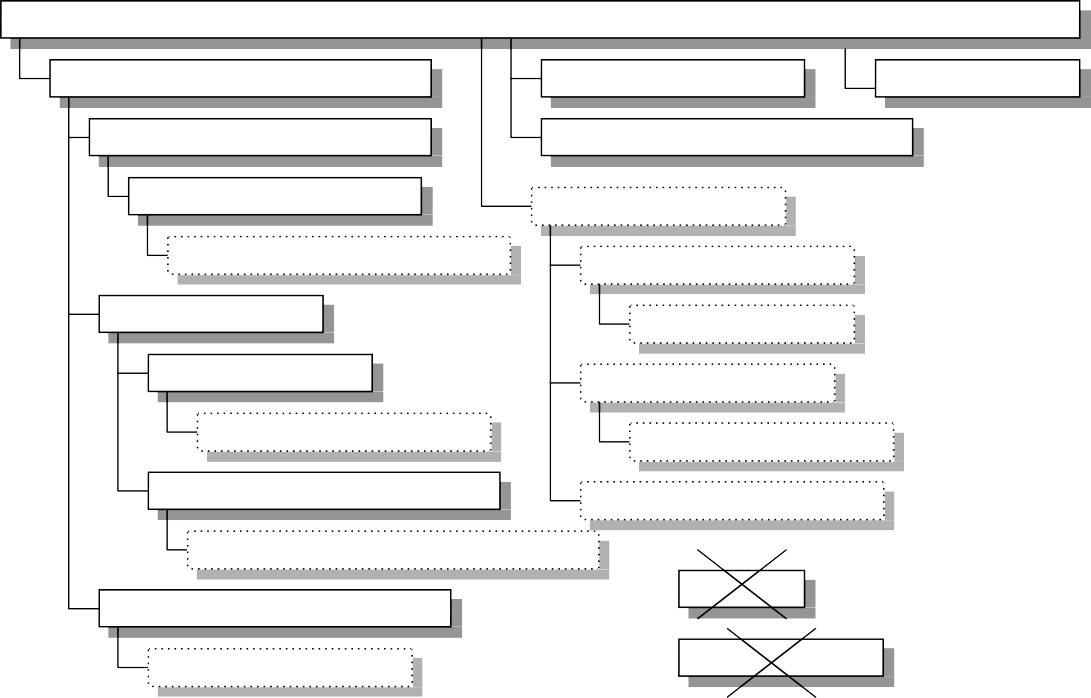
**238**

第５章 Polymorphism（多型）的應用

## 环环相扣的三层巨集

MFCLITE 的三个物件导向重要性质，最根源技术是必须实现 RTTI，否则一切免谈。要做出 RTTI，就必须在存储器中维护一个个资料结构，用以表现被注册过的 class 的各种内部信息。C++ Standard 不也说每一个注册过的 class 应该有一个对应的 *type\_info* 结构体吗 ?!

由于这个结构体只与 class 有关，与「class 有多少个 objects」无关，所以应该把它宣告为 class 的 static data member（ 请参考第１章 p.40 「Static Data Member」及第２章 p.125 「static members 的继承」）。我希望 MFCLITE classes 及其 derived classes 在宣告完毕之后，存储器中能够呈现出图 5.5 的架构。



CWinApp::classCWinApp

CShape::classCShape

CMyWinApp::classCMyWinApp

CSquare::classCSquare CRect::classCRect

CCircle::classCCircle

CMyView::classCMyView

CEllipse::classCEllipse

CTriangle::classCTriangle

CMyFrameWnd::classCMyFrameWnd

CArchive

CMyDoc::classCMyDoc

CRuntimeClass

CDocument:::classCDocument

CFrameWnd::classCFrameWnd

CView::classCView

CWnd::classCWnd

CDWordArray::classCDWordArray

CWinThread::classCWinThread

CObList::classCObList

CCmdTarget::classCmdTarget

CObject::classCObject

CFile::classCFile

圖 5.5 CFoo::classCFoo 必须是 CFoo 的一个 static member。 我们要想办法组织这样一个 tree， 以支援「三大服务」。

**239**

多型与虚拟（*Polymorphism in C++*）

请注意，图 5.5 中各结构体的命名与其「主人」（class）的名称有一致的关系。在 class 名称前面再加上 "class"，即成为结构体的名称。这种一致性带来一个莫大好处，我们可以利用特殊的运算子（编译器前置处理器）为我们产生程序码。稍后详述。

至此，我希望每一个想要获得三大服务的 classes 都有以下形式，其中的结构体沽且名为 *CRuntimeClass*：

class CFoo : public CObject

{

static CRuntimeClass classCFoo; // 注意命名方式

...

};

CRuntimeClass - 构思

图 5.5 中的结构体应该设计成什么样子？为了支援 RTTI，结构体中内含 class 名称是绝对必要的。class 大小（*sizeof* 的结果）最好也放进去，以备不时之需。另外，为了支援 Dynamic Creation，应该有一个栏位放置函式指标；如果class 名为*CFoo*，此栏位所指之函式应该有这样的内容：

{ return new CFoo; }

如此一来，程序在执行时期动态获得一个 class 名称之后，即可想办法取得图 5.5 中对应的结构体，并呼叫其中的「函式指标栏位」所指的函式，*new* 出一个 object 来。这样就解决了我在第１章 p.92 「Dynamic Creation（动态生成）」一节所提出的技术困扰。

此外，为了支援 persistence，应该有一个栏位放置 class 版本号码（所谓的 schema number）。这可以帮助程序员发现「旧版程序读新版资料档」或「新版程序读旧版资料档」的不当情况。不过在本章的简化原则下，我并没有利用它。

**240**

第５章 Polymorphism（多型）的應用

图 5.5 的各个结构体组织成一个 tree，然而要在程序中实作出一个 tree，困难度比较高，list（串列）是大家比较熟悉的资料结构。另一个原因是，某些 classes 可能不属于 tree（例如图 5.3 的 *CArchive*），但还是要把它纳入 RTTI 体系之中呀，所以使用 list 最方便2。

欲享有 RTTI 能力的 classes，都应该在图 5.5 中有一个对应的结构体。由于一个 class library（本例为 MFCLITE）再加上项目所衍生的 classes（本例为 Shape hierarchy）不可能太多（1000 个已经很不得了了），即使循序搜寻，速度亦十分可以接受，因此采用单向 list 已是绰绰有余，不需要双向 list。但是一维的 list 不能表现二维的tree，除非另有一个栏位记录着base（parent node）是谁。

现在，整块结构体已经呼之欲出了：

**CRuntimeClass** object

|  |  |
| --- | --- |
| m\_lpszClassName |  |
| m\_nObjectSize |
| m\_wSchema |
| m\_pfnCreateObject |
|  |
| m\_pBaseClass |
|  |
| m\_pNextClass |
|  |

**CRuntimeClass::pFirstClass**

（static member，only one instance）

我把它命名为 *CRuntimeClass*（图 5.3 最右下）。图 5.5 的 tree 于是可以图 5.6

的 list 来表现。

1. 但由于稍后我们以巨集的方式来自动产生程序码，对于这种不衍生自 *CObject* 的 classes， 处理上比较麻烦。所以虽然理论可行，为了简化，我（以及 MFC）规定，欲获得三大特性， classes 必须衍生自 *CObject*。

#### 241

多型與虛擬（*Polymorphism in C++*）

**CObject::classCObject**

“CWinThread”

NULL

NULL

**CFrameWnd::classCFrameWnd**

“CFrameWnd”

**CWnd::classCWnd**

“CWnd”

**CWinApp::classCWinApp**

m\_pfnCreateObject

m\_pBaseClass m\_pNextClass

**CDocument::classCDocument**

“CDocument”

**CView::classCView**

“CView”

**CObList::classCObList**

“CObList”

m\_pfnCreateObject m\_pBaseClass

m\_pNextClass

m\_pfnCreateObject m\_pBaseClass

m\_pNextClass

m\_pfnCreateObject

m\_pBaseClass m\_pNextClass

m\_pNextClass

m\_pBaseClass

m\_pfnCreateObject

m\_pNextClass

m\_pBaseClass

m\_pfnCreateObject

**CCmdTarget::classCCmdTarget**

**CWinThread::classCWinThread**

##### 

|  |  |  |
| --- | --- | --- |
|  | “CObject” |  |
|  |
|  |
| m\_pfnCreateObject |
| m\_pBaseClass |  |
|  | m\_pNextClass |

|  |  |  |
| --- | --- | --- |
|  |  | “CCmdTarget” |
|  |
|  |
| m\_pfnCreateObject |
| m\_pBaseClass |
|  | m\_pNextClass |

|  |  |  |
| --- | --- | --- |
|  | |  |
| “CWinApp” |  |
|  |
|  |
| m\_pfnCreateObject |
| m\_pBaseClass |
| m\_pNextClass |  | |

**CRuntimeClass:: pFirstClass**

圖 5.6 以 li st 來表現圖 5.5 的 tree。每一個 li st node 是一個

CRuntimeClass object。

#### 242

第５章 Polymorphism（多型）的應用

### CRuntimeClass - 實作

維護一個最簡單形式的 list（串列），只需要一個頭部指標。由於這樣一個頭部指標在整個 list 中只需唯一一個，當然應該把它設計為 *CRuntimeClass* 的一個static member。設計為 global 變數雖然也可以，但既然它與 *CRuntimeClass* 關係密切，從物件導向的封裝角度去考慮，設計為 *CRuntimeClass* 的 data member 還是比較理想。

下面就是 *CRuntimeClass* 的實作碼：

|  |
| --- |
| struct **CRuntimeClass**  {  char\* m\_lpszClassName; int m\_nObjectSize;  UINT m\_wSchema; // schema number of the loaded class CObject\* (\* m\_pfnCreateObject)();  CRuntimeClass\* m\_pBaseClass;  // CRuntimeClass objects linked together in simple list CRuntimeClass\* m\_pNextClass; // next node  static CRuntimeClass\* pFirstClass; // start node |
| // for Dynamic Creation CObject\* CreateObject();  // for Persistence  void Store(CArchive& ar) const;  static CRuntimeClass\* Load(CArchive& ar, UINT\* pwSchemaNum); |
| }; |

最尾端有三個 member functions 與稍後另兩個主題有關，目前暫時不討論它們。

Static member *pFirstClass* 必須有初始化動作（這一點常被忽略）：

CRuntimeClass\* CRuntimeClass::pFirstClass = NULL;

**243**

多型與虛擬（*Polymorphism in C++*）

### 巧妙運用巨集

請注意各個 class 的 *CRuntimeClass* member 的命名方式，與其「主人」的命名方式有一致的關連。把 class 名稱前面再加上 "class"，即成為 *CRuntimeClass* member 的名稱。這種一致性帶來一個莫大好處：我們可以利用巨集，使圖 5.6 的架設動作幾乎貼近於 C++ 語言層面：只要一行 statement 就全部做掉。這其中就密切關係到第３章所提的 ## 運算子和 # 運算子。

我希望達到的最高境界是像下面這樣（然後這些 classes 便擁有三大特性）：

// in declaration file (.H) class CCmdTarget : public CObject

{

DECLARE\_DYNAMIC(CCmdTarget) // 視為 C++ 運算子，Hmmm, good.

...

};

class CWnd : public CCmdTarget

{

DECLARE\_DYNCREATE(CWnd) // 視為 C++ 運算子，Hmmm, good.

...

};

class CDWordArray : public CObject

{

DECLARE\_SERIAL(CDWordArray) // 視為 C++ 運算子，Hmmm, good.

...

};

// in global scope of implementation file (.CPP) IMPLEMENT\_DYNAMIC(CCmdTarget, CObject)

IMPLEMENT\_DYNCREATE(CWnd, CCmdTarget)

IMPLEMENT\_SERIAL(CDWordArray, CObject, 0)

為此，先看看第一組巨集（*DYNAMIC*）該如何設計。

**244**

第５章 Polymorphism（多型）的應用

### 1. DYNAMIC（Run Time Type Identification）巨集

#define **DECLARE\_DYNAMIC**(class\_name) \ public: \

static CRuntimeClass class##class\_name; \ virtual CRuntimeClass\* GetRuntimeClass() const;

#define **IMPLEMENT\_DYNAMIC**(class\_name, base\_class\_name) \

\_IMPLEMENT\_RUNTIMECLASS(class\_name, base\_class\_name, 0xFFFF, NULL)

不 直 接 設 計 出 *IMPLEMENT\_DYNAMIC()* 巨 集 ， 卻 另 外 又 架 構 出 一 層

*\_IMPLEMENT\_RUNTIMECLASS()* 巨集如下，純粹是為了模組規劃考量（因為稍後第二組巨集 *DYNCREATE* 也要用到它）：

#define **\_IMPLEMENT\_RUNTIMECLASS**(class\_name, base\_class\_name, wSchema, pfnNew) \ static char \_lpsz##class\_name[] = #class\_name; \

CRuntimeClass class\_name::class##class\_name = { \

\_lpsz##class\_name, sizeof(class\_name), wSchema, pfnNew, \ RUNTIME\_CLASS(base\_class\_name), NULL }; \

static AFX\_CLASSINIT \_init\_##class\_name(&class\_name::class##class\_name); \ CRuntimeClass\* class\_name::GetRuntimeClass() const \

{ return &class\_name::class##class\_name; } \

看起來很複雜，其實沒什麼，純粹的文字代換而已。以 *CFoo* 為例：

class CFoo : public CObject

{

DECLARE\_DYNAMIC(CFoo)

...

};

IMPLEMENT\_DYNAMIC(CFoo, CObject)

經過巨集的擴展之後，變成這樣的碼：

**245**

多型與虛擬（*Polymorphism in C++*）

class CFoo : public CObject

{

public:

static CRuntimeClass classCFoo;

virtual CRuntimeClass\* GetRuntimeClass() const;

...

};

static char \_lpszCFoo[] = "CFoo"; CRuntimeClass CFoo::classCFoo = {

\_lpszCFoo, sizeof(CFoo), 0xFFFF, 0, (&CObject::classCObject), 0 };

static AFX\_CLASSINIT \_init\_CFoo(&CFoo::classCFoo); CRuntimeClass\* CFoo::GetRuntimeClass() const

{ return &CFoo::classCFoo; }

*DYNAMIC* 巨集

要點說明：

#### 巧妙運用 # 和 ## 運算子

1. 利用 # 運算子（ 請參考第３章 p.184）， 為我們產生程式碼， 合成出class 名稱字串。
2. 利用 ## 運算子（ 請參考第３ 章 p.185） ， 為我們產生程式碼， 合成出 *CRuntimeClass* member 的名稱字串。
3. 為 了 設 定 classes 的 繼 承 關 係 ， 我 必 須 取 得 base class 的*CRuntimeClass* member 位址，所以另外設計一個 *RUNTIME*\_*CLASS* 巨集。此巨集極其簡單：

#define RUNTIME\_CLASS(class\_name) \ (&class\_name::class##class\_name)

#### 巧妙運用 constructor 完成 list 自動串接工作

1. 「自動串接 list」是個大難題。前面都只是 *CRuntimeClass* 的欄位設定，list 的前後串接卻非「程式動作」不行。而你要知道，圖 5.6 的整個 list 必須在程式進入點（ *main()*，或 Windows 程式中的 *WinMain()*） 之前完成。哪一段程式碼可以比 *main()*更早執行？ 只有 global object 或static object 的 constructor 了（ 請參考第１章 p.21「object 的生命」）。

#### 246

第５章 Polymorphism（多型）的應用

所以， 我設計一個 struct3 如下， 其中沒有任何 data members， 只有一個 constructor 函式：

struct AFX\_CLASSINIT

{ AFX\_CLASSINIT(CRuntimeClass\* pNewClass); };

這樣的設計只是為了在產生這種 struct object 時喚起其 constructor：

AFX\_CLASSINIT::AFX\_CLASSINIT(CRuntimeClass\* pNewClass)

{

pNewClass->m\_pNextClass = CRuntimeClass::pFirstClass; CRuntimeClass::pFirstClass = pNewClass;

}

很明顯這是做 list 的串接工作，如下頁圖示。

請注意， 由於上述的 constructor 函式中使用了 *CRuntimeClass* 的 static member：*pFirstClass*，所以它也必須被宣告為 static 函式4。

1. 為 class 自動合成一個 member function： *GetRuntimeClass()*， 使外界能夠很方便地取得這個 *CRuntimeClass* member， 遂行三大服務。

這一組巨集，可成功地套用在任何一個衍生自 *CObject* 的 classes 身上。單單一行 statement，卻做了那麼多工作，確實是非常高明的設計。

1. 在 C++ 之中，struct 和 class 是幾乎一樣的，只有 default access level 不同。請參考第１章

p.19「Struct 和 Class 的差異」。

1. 請參考第１章 p.40「特殊的 Static Members」。

#### 247

多型與虛擬（*Polymorphism in C++*）

###### 串接前：

CRuntimeClass::pFirstClass

（static 變數）

CObject::classCObject

|  |  |  |
| --- | --- | --- |
| “CObject” |  | |
| sizeof(CObject) |
| 0xFFFF |
| m\_pfnCreateObject |
|  |  |
| m\_pBaseClass |
|  |
| m\_pNextClass |
|  | |

NULL

###### 串接後：

CRuntimeClass::pFirstClass

（static 變數）

CFoo::classCFoo

|  |  |
| --- | --- |
| “CFoo” |  |
| sizeof(CFoo) |
| 0xFFFF |
| m\_pfnCreateObject |
|  |
| m\_pBaseClass |
|  |
| m\_pNextClass |

**CObject::classCObject**

|  |  |  |
| --- | --- | --- |
| “CObject” |  | |
| sizeof(CObject) |
| 0xFFFF |
| m\_pfnCreateObject |
|  |  |
| m\_pBaseClass |
|  |
| m\_pNextClass |
|  | |

NULL

#### 248

第５章 Polymorphism（多型）的應用

### DYNCREATE（Dynamic Creation）巨集

欲完成 Dynamic Creation，必須先擁有 RTTI 的能力。因此，第二組巨集必須涵蓋第一組巨集，設計如下（灰色表示新增或修改部份）：

|  |
| --- |
| #define **DECLARE\_DYNCREATE**(class\_name) \ DECLARE\_DYNAMIC(class\_name) \ |
| static CObject\* CreateObject(); |
| #define **IMPLEMENT\_DYNCREATE**(class\_name, base\_class\_name) \ |
| CObject\* class\_name::CreateObject() \  { return new class\_name; } \ |
| \_IMPLEMENT\_RUNTIMECLASS(class\_name, base\_class\_name, 0xFFFF, \ class\_name::CreateObject) |

要點說明：

* 1. *DYNCREATE* 巨集完全涵蓋 *DYNAMIC* 巨集， 並增加一些額外動作。
  2. 額外動作之一：新增一個 static *CObject*\* *CreateObject()* 函式。此函式只是簡單地 *new* 一個 class object。請注意， 這種藉由編譯器之助， 將巨集擴展開來，為我們添加程式碼的作法，成功地解決掉第１ 章 p.92

「Dynamic Creation（ 動態生成）」所提出的問題。

為什麼要把 *CreateObject()* 設計為 static 函式？因為很有可能在尚未產生任何 *CXxx* object 之前，就需要呼叫 *CXxx*::*CreateObject()* 函式（這個函式的目的不就是要動態生成一個 *CXxx* object 嗎 ?!）。能夠不透過 object 實體而呼叫（所謂 statically invocation）的函式，顯然必須是個 static 函式（請參考第１章 p.42「Static Member Functions」一節）。

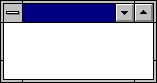
* 1. 額外動作之二： 將 *CRuntimeClass* member 的第四個欄位（ 函式指標） 設定為剛剛所製造出來的 *CreateObject()* 函式位址。

於是，任何 class 只要使用了 *DYNCREATE* 巨集，一旦程式執行時期獲得一個

**249**

多型與虛擬（*Polymorphism in C++*）

class 名稱，便可以走訪圖 5.6 的整個 list，尋找「class 名稱（第一個欄位）相符」的 *CRuntimeClass* 結構體，找到後呼叫其 *m\_pfnCreateObject* 欄位所指之函式，便可以 *new* 得一個相關的 object 出來：

1



**pFirstClass**



執行時期動態獲得一個 class name

**...**

CFoo::classCFoo

NULL

2

走訪整個 *CRuntimeClass* list 之後， 搜尋到一個 class name 吻合的節點

呼叫 (\**m\_pfnCreateObject*)()



**3**

|  |  |
| --- | --- |
| “CFoo” |  |
| sizeof(CFoo) |
| 0xFFFF |
| m\_pfnCreateObject |  |
| m\_pBaseClass |
|  |
| m\_pNextClass |

即可產生一個對應的 object

static CObject\* CFoo::CreateObject()

{ return new CFoo; }

注意：傳回的是



**4**

generic type：*CObject*\*

當然，這「走訪圖 5.6 整個 list」的動作也必須實作出來，稍後我再來談它。

這裡我們看到了多型的一個大大用途。*CreateObject()* 傳回的是一個 *CObject*\*， 而不是任何特定的 object 指標。以 base type pointer 指向 derived type object 當然是被允許的事情。就巨集的設計技術而言，我們當然也可以強迫讓擴展後所得的 *CFoo*::*CreateObject()* 傳回 *CFoo*\*，但這沒有必要，並且會妨礙程式員將此object 指標做更廣泛的用途。

#### 250

第５章 Polymorphism（多型）的應用

##### SERIAL（ Persistence or Serialization） 巨集

要獲得 persistence 能力，必須先具備 dynamic creation 的能力。因此，第三組巨集必須涵蓋第二組巨集，設計如下（灰色表示新增部份）：

|  |
| --- |
| #define DECLARE\_SERIAL(class\_name) \ DECLARE\_DYNCREATE(class\_name) \ |
| friend CArchive& operator>>(CArchive& ar, class\_name\* &pOb); |
| #define IMPLEMENT\_SERIAL(class\_name, base\_class\_name, wSchema) \ CObject\* class\_name::CreateObject() \  { return new class\_name; } \  \_IMPLEMENT\_RUNTIMECLASS(class\_name, base\_class\_name, wSchema, \ class\_name::CreateObject) \ |
| CArchive& operator>>(CArchive& ar, class\_name\* &pOb) \  { pOb = (class\_name\*)ar.ReadObject(RUNTIME\_CLASS(class\_name)); return ar; } |

要點說明：

* 1. *SERIAL* 巨集完全涵蓋 *DYNCREATE* 巨集， 並增加一個額外動作。
  2. 額 外 動 作 就 是 ， 新 增 一 個 operator>> 函 式 （ 也 就 是 extraction operator）。此運算子的動作對象（ 運算元， operand） 是：
     + 左運算元： 一個 *CArchive* object。資料準備從這裡讀出。
     + 右運算元： 一個 object 指標。資料準備放進這裡。

這個 operator>> 函式內部將呼叫 *CArchive::ReadObject()*，邏輯頗為複雜。稍後進行到 *CArchive* 時，我再對這個 operator>> 多做解釋。

資料的永久保存（persistence）可不能夠光讀不寫，也不能夠光寫不讀。相對於operator>>，當然也有 operator<<（insertion operator），我將在稍後進行到 *CArchive*時，一併對這個 operator<< 再做解釋。

現在，任何 class 只要使用了 *SERIAL* 巨集，一旦程式執行時期想要永久保存所

#### 251

多型與虛擬（*Polymorphism in C++*）

有的 objects（persistence），便可以使用 << 和 >> 兩個極便利的運算子對某個

*CArchive* object（暫時請想像為一個磁碟檔案）做讀寫動作了。

### CObject

上述三組巨集，形式上非常漂亮，好像語言層面的一個 statement。但請注意，由於 *CObject* 是 root class，不再有 base class，所以沒辦法使用 *DYNAMIC* 巨集來設定上述的 *CRuntimeClass* 內容5，遑論後面兩個更高階的巨集。因此前述的每一個技術點，都必須針對 *CObject* 另做處理：

class CObject

{

public:

static CRuntimeClass classCObject;

virtual CRuntimeClass\* GetRuntimeClass() const;

...

};

// static members initialization static char szCObject[] = "CObject";

struct CRuntimeClass CObject::classCObject =

{ szCObject, sizeof(CObject), 0xFFFF, NULL, **NULL**, NULL }; static AFX\_CLASSINIT \_init\_CObject(&CObject::classCObject);

CRuntimeClass\* CObject::GetRuntimeClass() const

{

return &CObject::classCObject;

}

1. 註：如果你為 *CObject* 使用 *DYNAMIC* 巨集如下：

IMPLEMENT\_DYNAMIC(CObject, NULL)

那麼巨集展開後的這一行：RUNTIME\_CLASS(NULL) 會出現編譯錯誤，因為編譯器沒有辦法處理 &0::class0 這樣的句子。

#### 252

第５章 Polymorphism（多型）的應用

### RTTI 的運用之一 CObject::IsKindOf()

雖然圖 5.6 的基礎建設可以支援 RTTI，但我們總是得寫出一個「走訪函式」， 做為外界使用的介面。每個 classes 都需要提供這個函式，而且沒有必要改寫它， 因此提昇到 *CObject* 成為一個 non-virtual member function 最是理想：

BOOL CObject::IsKindOf(const CRuntimeClass\* pClass) const

{

CRuntimeClass\* pClassThis = GetRuntimeClass(); while (pClassThis != NULL)

{

if (pClassThis == pClass) return TRUE;

pClassThis = pClassThis->m\_pBaseClass; // not m\_pNextClass

}

return FALSE; // walked to the top, no match

}

用法如下（其中的 *RUNTIME*\_*CLASS* 是個巨集，先前已介紹過）：

CFoo aFoo;

if (aFoo.IsKindOf(RUNTIME\_CLASS(CObject)))

// aFoo 是 CObject 的一個 derived type object

...

else

// aFoo 不是 CObject 的 derived type object

...

這就是RTTI 的一種運用。完整範例程式請看本章最後的程式列表。

**253**

多型與虛擬（*Polymorphism in C++*）

### RTTI 的運用之二

如果在程式進行之中突然想針對某個 object 指標查詢其 class 名稱，可以這麼做：

CRuntimeClass\* pClassRef = objptr->GetRuntimeClass(); cout << pClassRef->m\_lpszClassName;

這也是RTTI 的一種運用。完整範例程式請看本章最後的程式列表。

接下來，我要把討論焦點放到檔案的處理身上，然後才能夠繼續討論 dynamic creation 和 persistence 的對外介面（基礎建設已經完備）。*CRuntimeClass* 至今還有三個 member functions 沒有探討，它們都與 dynamic creation 和 persistence 有關：

1. *CRuntimeClass::CreateObject()*
2. *CRuntimeClass::Load()*
3. *CRuntimeClass::Store()*

稍後我再來討論它們。

**254**

第５章 Polymorphism（多型）的應用

### CFile

*CFile* 只是把檔案動作包裝起來，基本上並沒有什麼值得大書特書之處。它提供以下兩個 data members：

HFILE m\_hFile; // 記錄檔案的 handle6 char m\_strFileName[40]; // 記錄檔案名稱

並提供以下函式（實作方法請參考本章最後的程式列表）：

1. CFile(char\* lpszFileName, UINT nOpenFlags);

這是 constructor，呼叫 *CFile::Open()* 準備打開檔案。

1. virtual ~CFile();

這是 destructor，呼叫 *CFile::Close()* 準備關閉檔案。

1. virtual BOOL Open(char\* lpszFileName, UINT nOpenFlags);

根據檔名及開檔模式（可指定為 *modeRead* 或 *modeWrite*），呼叫 *::fopen()* 打開檔案。

1. virtual void Close();

呼叫 *::fclose()* 關閉檔案。

1. virtual UINT Read(void\* lpBuf, UINT nCount);

呼叫 *::fread()* 讀檔。

1. virtual void Write(const void\* lpBuf, UINT nCount);

呼叫 *::fwrite()* 寫檔。

1. 由於我使用 ANSI 標準的 file stream 來存取檔案，所以我把 HFILE 定義為 FILE\*。MFC 使用 Win32 APIs 函式（如 *::CreateFile*、::*CloseHandle*、::*ReadFile*、::*WriteFile*）來存取檔案， 所以它把 HFILE 定義為 HANDLE。

**255**

多型與虛擬（*Polymorphism in C++*）

#### CArchive

*CArchive* 原本只是扮演 buffering file 的角色7，用以改善檔案讀寫效率。但後來在 object persistence 方面的地位反而日趨重要。圖 5.7 可以說明 Persistence 和*CArchive* 以及 *CFile* 的關係。

::fread() m\_hFile

modeRead

m\_pFile->Read()

(buffering)

operator>>

Data Structure

CArchive object

CFile object

Disk

::fwrite() m\_hFile

modeWrite

m\_pFile->Write()

(buffering)

operator<<

Data Structure

CArchive object

CFile object

Disk

圖 5.7 Persistence 和 CArchive 以及 CFile 的關係

#### CArchive 與 CFile

在與 *CFile* object 有關的業務之中，*CArchive* 提供了兩筆 data members：

BOOL m\_nMode; // open mode，可以是 store 或 load CFile\* m\_pFile; // 相關連的 CFile\* member

以及下列數個 member functions：

CArchive(CFile\* pFile, UINT nMode);

~CArchive();

BOOL IsLoading() const; // 檢查 open mode 是否為 load BOOL IsStoring() const; // 檢查 open mode 是否為 store

1. Buffering 不是我要示範的技術。為了簡化，我把這項功能拿掉。

#### 256

第５章 Polymorphism（多型）的應用

UINT Read(void\* lpBuf, UINT nMax); // 呼叫 CFile::Read() void Write(const void\* lpBuf, UINT nMax); // 呼叫 CFile::Write() void Close();

此外， 還有一大堆針對不同型別（ 都是內建型別）而設計的 operator>> 和

operator<< 函式，例如：

CArchive& operator<<(int i); CArchive& operator<<(short w); CArchive& operator>>(int& i); CArchive& operator>>(short& w);

這些 operator 函式的實作都非常簡單，以 inline 完成極為理想。當然，inline 函式應該放在 .H 檔中（請參考第１章 p.62「Inline Functions」）。

#### CArchive 與 Persistence

為了提供一個高階介面，能夠在面對 class type objects 時，也像面對 C++ 內建型別那樣地以 operator<< 和 operator>> 做檔案讀寫動作，顯然我們必須對這兩個運算子施以 operator overloading，將流程導引到我們的控制之中：

inline CArchive& operator<<(CArchive& ar, const CObject\* pOb)

{ ar.WriteObject(pOb); return ar; }

inline CArchive& operator>>(CArchive& ar, CObject\*& pOb)

{ pOb = ar.ReadObject(NULL); return ar; }

inline CArchive& operator>>(CArchive& ar, const CObject\*& pOb)

{ pOb = ar.ReadObject(NULL); return ar; }

此外 *SERIAL* 巨集也為我們合成出一個類似的 operator>>（以 *CFoo* 為例）：

CArchive& operator>>(CArchive& ar, CFoo\* &pOb) \

{ pOb = (CFoo\*)ar.ReadObject(RUNTIME\_CLASS(CFoo)); return ar; }

現在我們面臨重大挑戰了：究竟把一個 object 儲存到檔案時，應該儲存什麼樣的資訊，才能夠再把它原貌讀出來？下面是幾點考量：

#### 257

多型與虛擬（*Polymorphism in C++*）

1. 必 須 儲 存 object 的 class 名稱， 以便讀出時可以根據此名稱做

dynamic creation。

1. 既然 class type object 不像內建型別那樣只是個純量， 而是一種container 8， 寫檔時勢必將 container 的元素個數存進去， 讀檔時才知道該讀出多少筆元素。
2. 如果 class 名稱無法表現出每筆元素的大小， 那麼還應該寫入元素大小， 以便讀檔時知道每一筆元素該讀多少個 bytes。

為此，我再為 *CArchive* 添加四個非常重要的 member functions（實作方法請參考本章最後的程式列表）：

1. void WriteObject(const CObject\* pOb);

此函式首先呼叫 *WriteClass()* 將 class 相關資訊寫檔，然後要求 object 自己負責把 raw data 寫入檔案。也就是說，每一個支援 persistence 性質之 class 應該有一個函式，專門等待被此處呼叫，負責將它自己的 raw data 寫入檔案。這個函式被命名為 *Serialize*()，稍後再討論。

1. void WriteClass(const CRuntimeClass\* pClassRef);

這個函式將 class 相關資訊（顯然必須靠 RTTI 性質的協助）寫到檔案之中。它會呼叫 *CRuntimeClass::Store()* 幫忙（稍後介紹）。

1. CObject\* ReadObject(const CRuntimeClass\* pClass);

這個函式首先呼叫 *ReadClass()* 將 class 相關資訊從檔案中讀出，並傳回*CRuntimeClass*\* 指標，然後以 dynamic creation 方式產生出對應的 object，然後再要求該 object 自己負責把 raw data 讀出檔案。也就是說，每一個支援persistence 性質之 class 應該有一個函式專門等待被此處呼叫，負責將它自己

1. 用來收集資料的結構，我們習慣稱之為 container，如 array 或 list 或 map。

#### 258

第５章 Polymorphism（多型）的應用

的 raw data 讀出檔案。這個函式也被命名為 *Serialize()*，稍後再討論。

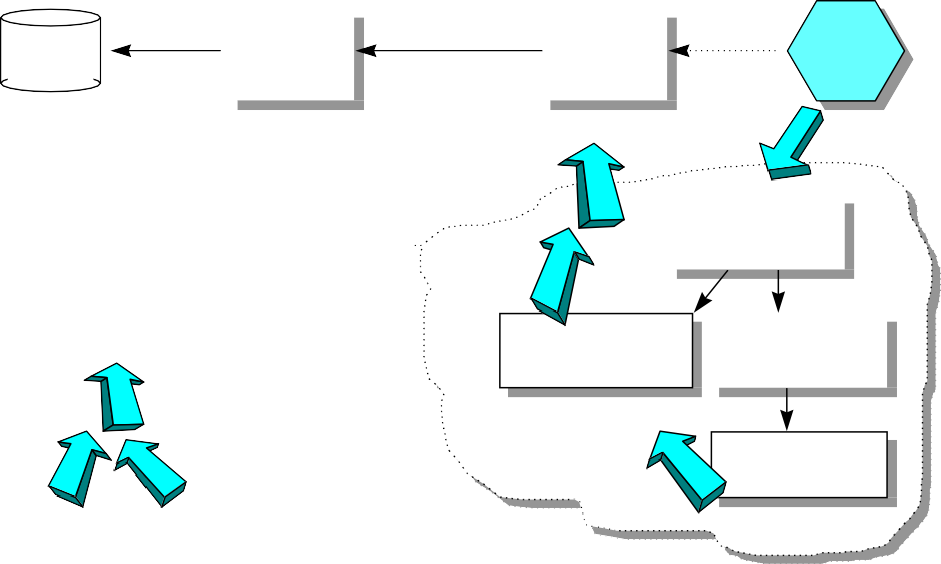
1. CRuntimeClass\* ReadClass(const CRuntimeClass\* pClassRefRequested = NULL);

這個函式從檔案中讀出 class 相關資訊。它會呼叫 *CRuntimeClass::Load()* 幫忙（稍後介紹），並取得圖 5.6 *CRuntimeClass* list 中的對應節點。有了這個節點（一個 *CRuntimeClass* object），要完成 dynamic creation 就不是問題了。

圖 5.8 可以說明整個流程。詳細的程式碼請參考本章末的程式列表。

#### 259

多型與虛擬（*Polymorphism in C++*）



modeWrite

Disk

::fwrite()

m\_hFile

m\_pFile->Write()

(buffering)

Data Structure

operator<<

(3)

Serializable\_class:: Serialize()

(1)

(2)

CRuntimeClass:: Store()

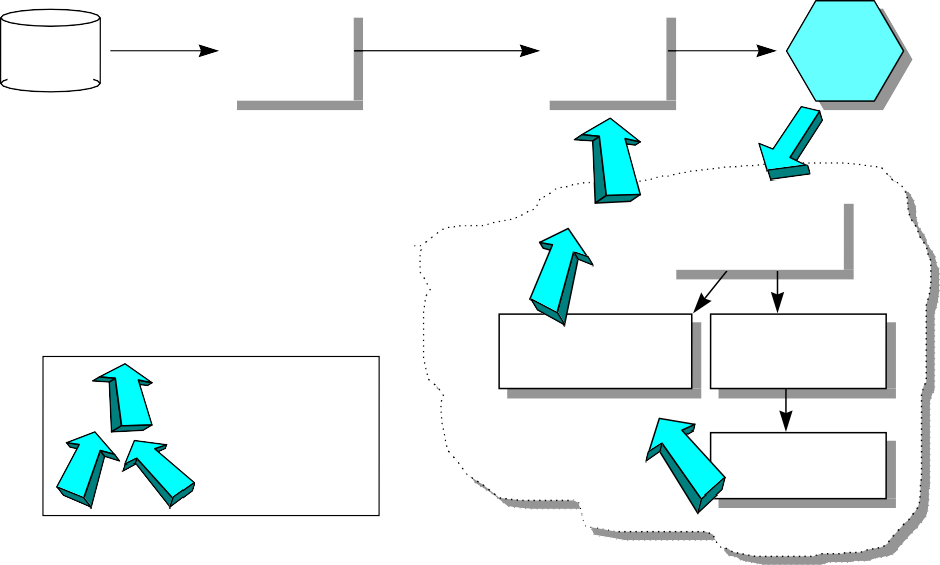
CArchive object

：向 CArchive 發出寫檔要求

CArchive:: WriteClass()

CArchive:: WriteObject()

CFile object



modeRead

Disk ::fread() m\_hFile

m\_pFile->Read()

(buffering)

Data Structure

operator>>

(3)

Serializable\_class:: Serialize()

(1)

CArchive:: ReadClass()

：向 CArchive 發出讀檔要求

(2)

CRuntimeClass:: Load()

CArchive object

CFile object

CArchive:: ReadObject()

圖 5.8 CArchive 和 CRuntimeClass 以及 Serializable Classes 共同合作， 完成 persistence 服務。

#### 260

第５章 Polymorphism（多型）的應用

#### 重回 CRuntimeClass 的討論

先前介紹 *CRuntimeClass* 時我遺留了三個缺口，現在讓我把它補全。

|  |
| --- |
| void CRuntimeClass::Store(CArchive& ar) const |
| // stores a runtime class description  {  WORD nLen = (WORD)::strlen(m\_lpszClassName); ar << nLen;  ar.Write(m\_lpszClassName, nLen\*sizeof(char));  } |

*Store()* 把 *CRuntimeClass* object 所記錄的檔案名稱及其長度寫入一個 archive 之中（當然也就是寫入一個檔案中）。「寫入」遠比「讀出」輕鬆，因為「寫入」不需引發dynamic creation 而「讀出」需要。

|  |
| --- |
| CRuntimeClass\* CRuntimeClass::Load(CArchive& ar,  UINT\* pwSchemaNum) |
| // loads a runtime class description  {  WORD nLen;  char szClassName[64]; CRuntimeClass\* pClass;  ar >> nLen;  ar.Read(szClassName, nLen\*sizeof(char)); szClassName[nLen] = '\0';  for (pClass = pFirstClass; pClass != NULL;  pClass = pClass->m\_pNextClass)  {  if (::strcmp(szClassName, pClass->m\_lpszClassName) == 0) return pClass;  }  TRACE1("Error: Class not found: %s \n", szClassName); return NULL; // not found  } |

#### 261

多型與虛擬（*Polymorphism in C++*）

*Load()* 從一個 archive（代表一個檔案）中讀出 class 名稱長度，再依據這個長度讀出適當的字元，做為 class 名稱。然後以此名稱為依據，將圖 5.6 的整個*CRuntimeClass* list 走訪一遍；如果 class 名稱吻合，即將該節點的位址傳回。

*CRuntimeClass::Load()* 的呼叫者，也就是 *CArchive::ReadObject()* 函式，正是dynamic creation 的發生地點。當它收到 *Load()* 傳回「型別為 *CRuntimeClass*\*」的指標，並確定不是 NULL 之後9，便呼叫 *CRuntimeClass::CreateObject()*：

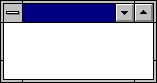
|  |
| --- |
| CObject\* CRuntimeClass::CreateObject() |
| { |
| if (m\_pfnCreateObject == NULL) |
| { |
| TRACE1("Error: class not DECLARE\_DYNCREATE or \n" |
| "DECLARE\_SERIAL : %s\n", m\_lpszClassName); |
| return NULL; |
| } |
| CObject\* pObject = NULL; |
| pObject = (\*m\_pfnCreateObject)(); |
| return pObject; |
| } |

下面這張圖可以說明 *CRuntimeClass::Load()* 和 *CRuntimeClass::CreateObject()*

的合作過程：

1. 有可能是 NULL！也許你的程式從檔案中讀到一個 class 名稱，而程式中的該 class 宣告卻未使用 *DECLARE\_SERIAL* 巨集。不過如果該檔案是由 MFCLITE 程式寫入，就應該不會發生這種事情，畢竟，在MFCLITE 程式中，能夠寫入檔案的 object，一定都在其class 宣告時使用了 *DECLARE\_SERIAL* 巨集。

#### 262

第５章 Polymorphism（多型）的應用



CFoo::classCFoo



**1**

執行時期動態獲得一個 class name

NULL

**pFirstClass**

**2**

走訪整個 *CRuntimeClass* list 之後， 搜尋到一個與 class name 吻合的節點

**...**

|  |  |
| --- | --- |
| “CFoo” |  |
| sizeof(CFoo) |
| 0xFFFF |
| m\_pfnCreateObject |  |
| m\_pBaseClass |
| m\_pNextClass |

呼叫 CRuntimeClass::CreateObject() CRuntimeClass::CreateObject()

呼叫 (\**m\_pfnCreateObject*)()

static CObject\* CFoo::CreateObject()

{ return new CFoo; }



**3**



**4**

於是產生一個對應的 object



**5**

注意：傳回的是 generic type：*CObject*\*

### CDWordArray

為了實證整個基礎建設的正確性，我完成一個 *CDWordArray*（當然，仍然是從MFC source 改裝而來）。最外緣的介面是 *Add()* 和 *operator*[ ]，用以加入和取出元素內容。每一個元素是一個 DWORD，也就是 unsigned long。

### CObList

為了實證整個基礎建設的正確性，我完成一個 *CObList*（當然，仍然是從 MFC source 改裝而來）。最外緣的介面是 *AddTail()*、*GetHeadPosition()* 和 *GetNext()*， 用以加入和取出元素內容。每一個元素是一個型別為 *CObject*\* 的指標（大小固定為 4 bytes），這又是多型的一個大大用途，我已經在第４章末 p.219「泛型的重要性」一節以一個類似的程式為例，討論過了。

#### 263

多型與虛擬（*Polymorphism in C++*）

### Serialize 函式

MFC 把檔案讀寫動作（也就是物件導向的 persistence）命名為 *Serialize()*，我也隨俗使用。

我想你已經在「*CArchive* 與 Persistence」一節的討論中看到了：

1. 從檔案中讀到任何一個 class 名稱，取得對應的 *CRuntimeClass* 節點之後， 即可 *new* 出一個 object 來。
2. *CArchive* 並不知道該如何讀取接踵而來的 object’s raw data 。 唯有container 自己才知道。因此， 每一個 container classes 如果要享有persistence（ serialization） 的能力， 都必須準備一個 *Serialize()* 函式， 負責讀取自己的 raw data。此函式將被 *CArchive::ReadObject()* 呼叫。
3. 這樣的函式， 同名而且應該被每一個 container classes 重新定義（ 改寫，override），因此以 virtual function 來完成最是理想。它應該被提昇到最高層的 *CObject* 之中。
4. 被 步 驟 1. *new* 出 來 的 object ， 其 指 標 型 別 是 *CObject*\* -- 整個MFCLITE 中最根源的型別， 因此步驟 2. 所呼叫的 *Serialize()* 函式也就獲得了多形的好處： 就像第４章那個簡單的 shape 程式一樣， 一旦derived class 改寫了它， 就呼叫之； 如果 derived class 沒有改寫， 就呼叫 derived class 所繼承者。當然， 做為一個希望有檔案讀寫能力的class （ 不只在 MFCLITE 中， 也包括專案中任何一個有此欲望的classes ， 如本例的 Shape hierarchy 的各個 classes） ， 都應該改寫*Serialize()*。

### Shape Hierarchy

所有的 shape classes，都為了表現某種形狀。為獲得上述三大服務，我把它們統統繼承自 *CObject*，並使用 *SERIAL* 巨集，並改寫 *Serialize()* 函式。同時我也設計了一個 *display()* 函式，並且根據第４章的分析，把它提昇為 *CShape* 的 virtual function，再讓各個 shape classes 去改寫。

#### 264

第５章 Polymorphism（多型）的應用

## 測試

我在 TEST.CPP 中首先產生 *pShape*[8]，放置不同形狀的物件指標，其中的*CStroke* object 內含一個 *CDWordArray*，為了能夠對它安放初值，我必須轉型：

CShape\* pShape[8];

// prepare raw data

pShape[0] = new CEllipse(3.0, 3.0, 7.0, 21.0);

pShape[1] = new CCircle(5.0, 5.0, 7.0);

pShape[2] = new CTriangle(0.0, 0.0, 1.0, 0.0, 0.0, 1.0);

pShape[3] = new CRect(5.6, 6.8, 3.0, 9.0);

pShape[4] = new CSquare(3.2, 4.3, 6.0);

pShape[5] = new CStroke;

CStroke\* pStroke = dynamic\_cast<CStroke\*>(pShape[5]); assert(pStroke);

pStroke->aDArray.Add(1);

...

pShape[6] = new CTriangle(0.0, 0.0, 3.5, 0.0, 0.0, 5.3);

pShape[7] = new CRect(9.9, 9.9, 4.8, 9.8);

然後我選擇 *CObList* 做為我的資料收集器（container），並將 *pShape*[ ] 陣列元素一一放進 list 之中成為節點。*CObList* 要求每一個節點都必須是 *CObject* 的derived class object pointer，而 *pShape*[ ] 的每一個元素的確符合此一條件：

CObList myList; // my container

for (int i=0; i<8; i++) // prepare container's data myList.AddTail(pShape[i]);

然後我將所有的資料列印出來驗證：

// retrieval all data

POSITION pos = myList.GetHeadPosition(); while (pos != NULL) {

CShape\* pS = (CShape\*)myList.GetNext(pos); pS->display(); // show it.

}

得到的結果是：

**265**

多型與虛擬（*Polymorphism in C++*）

|  |
| --- |
| CEllipse: x=3, y=3, r=7, r2=21 CCircle: x=5, y=5 ,r=7  CTriangle: x1=0, y1=0 x2=1, y2=0 x3=0, y3=1 CRect: x=5.6, y=6.8, height=3, width=9 CSquare: x=3.2, y=4.3, height=6  CStroke: 1 2 3 4 5 6 7  CTriangle: x1=0, y1=0 x2=3.5, y2=0 x3=0, y3=5.3 CRect: x=9.9, y=9.9, height=4.8, width=9.8 |

接下來我將它寫入檔案：

CFile\* pFile = new CFile("output.dat", CFile::modeWrite); CArchive ar(pFile, CArchive::store);

myList.Serialize(ar); // persistence, write to file ar.Close();

pFile->Close();

然後再讀出到另一個 container（也是 *CObList*）之中：

CFile\* pFile = new CFile("output.dat", CFile::modeRead); CArchive ar(pFile, CArchive::load);

CObList myNewList; // another container myNewList.Serialize(ar); // persistence, read from file

ar.Close(); pFile->Close();

再將所有資料列印出來驗證：

// retrieval all data

POSITION pos = myNewList.GetHeadPosition(); while (pos != NULL) {

CShape\* pS = (CShape\*)myNewList.GetNext(pos); pS->display();

}

#### 266

第５章 Polymorphism（多型）的應用

得到的結果是：

|  |
| --- |
| CEllipse: x=3, y=3, r=7, r2=21 CCircle: x=5, y=5 ,r=7  CTriangle: x1=0, y1=0 x2=1, y2=0 x3=0, y3=1 CRect: x=5.6, y=6.8, height=3, width=9 CSquare: x=3.2, y=4.3, height=6  CStroke: 1 2 3 4 5 6 7  CTriangle: x1=0, y1=0 x2=3.5, y2=0 x3=0, y3=5.3 CRect: x=9.9, y=9.9, height=4.8, width=9.8 |

與寫入檔案前完全一致，證明 persistence 的一切運作都正確無誤！

另外，我以下列動作驗證 RTTI 和 *IsKindOf()*，一切都合乎理想。程式碼後的註解即為輸出結果：

// try IsKindOf() and RTTI (class name query) CDocument\* pDoc = new CDocument;

CView\* pView = new CView; CRect\* pRect = new CRect;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| cout | << | pDoc->IsKindOf(RUNTIME\_CLASS(CDocument)) << endl; | // | 1 |
| cout | << | pDoc->IsKindOf(RUNTIME\_CLASS(CObject)) << endl; | // | 1 |
| cout | << | pDoc->IsKindOf(RUNTIME\_CLASS(CCmdTarget)) << endl; | // | 1 |
| cout | << | pDoc->IsKindOf(RUNTIME\_CLASS(CView)) << endl; | // | 0 |
| cout | << | pView->IsKindOf(RUNTIME\_CLASS(CObject)) << endl; | // | 1 |
| cout | << | pView->IsKindOf(RUNTIME\_CLASS(CShape)) << endl; | // | 0 |
| cout | << | pRect->IsKindOf(RUNTIME\_CLASS(CRect)) << endl; | // | 1 |
| cout | << | pRect->IsKindOf(RUNTIME\_CLASS(CSquare)) << endl; | // | 1 |
| cout | << | pRect->IsKindOf(RUNTIME\_CLASS(CShape)) << endl; | // | 1 |
| cout | << | pRect->IsKindOf(RUNTIME\_CLASS(CObject)) << endl; | // | 1 |
| cout | << | pRect->IsKindOf(RUNTIME\_CLASS(CView)) << endl; | // | 0 |
| cout | << | pDoc->GetRuntimeClass()->m\_lpszClassName << endl; | // | CDocument |
| cout | << | pView->GetRuntimeClass()->m\_lpszClassName << endl; | // | CView |
| cout | << | pRect->GetRuntimeClass()->m\_lpszClassName << endl; | // | CRect |

#### 267

多型與虛擬（*Polymorphism in C++*）

## 值得改善處

我希望對 *Serialize* 後的檔案格式多有一些瞭解。使用任何一個可以觀看二進位傾印碼的工具（我用的是 TDUMP.EXE）來觀察上述的 output.dat：

注意，Intel CPU 的資料排列採 little-endian 形式，必須兩兩顛倒看

一開始的 0008 表示整個list 有 8 個元素

H:\g001\prog\big.05>tdump output.dat

|  |
| --- |
| Turbo Dump Version 3.1 Copyright (c) 1988, 1992 Borland International Display of File OUTPUT.DAT  000000: 08 00 08 00 43 45 6C 6C 69 70 73 65 00 00 40 40 CEllipse..@@  000010: 00 00 40 40 00 00 E0 40 00 00 A8 41 07 00 43 43 ..@@...@...A..CC  000020: 69 72 63 6C 65 00 00 A0 40 00 00 A0 40 00 00 E0 ircle...@...@...  000030: 40 09 00 43 54 72 69 61 6E 67 6C 65 00 00 00 00 @..CTriangle....  000040: 00 00 00 00 00 00 80 3F 00 00 00 00 00 00 00 00 .......?........  000050: 00 00 80 3F 05 00 43 52 65 63 74 33 33 B3 40 9A ...?..CRect33.@.  000060: 99 D9 40 00 00 40 40 00 00 10 41 07 00 43 53 71 ..@..@@...A..CSq  000070: 75 61 72 65 CD CC 4C 40 9A 99 89 40 00 00 C0 40 uare..L@...@ @  000080: 07 00 43 53 74 72 6F 6B 65 07 00 01 00 00 00 02 ..CStroke.......  000090: 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00 06 ................  0000A0: 00 00 00 07 00 00 00 09 00 43 54 72 69 61 6E 67 CTriang  0000B0: 6C 65 00 00 00 00 00 00 00 00 00 00 60 40 00 00 le..........`@..  0000C0: 00 00 00 00 00 00 9A 99 A9 40 05 00 43 52 65 63 .........@..CRec  0000D0: 74 66 66 1E 41 66 66 1E 41 9A 99 99 40 CD CC 1C tff.Aff.A...@...  0000E0: 41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A............... |

我發現，如果 *CObList* 之中有同一 class 所生出的許多個 objects （例如上例重複了兩個 *CRect* objects 和 *CTriangle* objects），它們的 class 名稱會被一再寫入檔案之中。資料量如果不大，倒也還好。但如果一張工程圖有 1,000,000 個 *CStroke* object，光是記錄重複的 class 名稱就用掉了 (1000000-1)\*strlen("*CDWordArray*")

= 10999989 bytes，也就是浪費了 10MB 的檔案空間。『沒關係』，也許你說『反正硬碟價格現在 1MB 才兩塊錢』。但如果這個檔案要上網傳遞呢？這筆賬就不能夠忽略了。

為了不要把重複的 class 名稱寫入檔案之中， 在 *CArchive::WriteClass()* 和

**268**

第５章 Polymorphism（多型）的應用

*CArchive::ReadClass()* 函式中必須有管理的動作。這可能需要一個 hash table 做為比對與儲存 class 名稱之用，動作複雜得多。商業化的 MFC 的確做到了，本章的範例程式則...唔...我想還是不要好了，太過複雜又偏離主題，會導至消化不良。

## 結語

千萬不要因為在本章中看到很多的 Ｍ- Ｆ- Ｃ 三個字， 就以為本章技術和Windows 平台有什麼糾葛不清的關係，並因而決定在現階段放棄學習這一章。不，不是這樣，這一章固然是向 Microsoft 的 MFC 工作小組學習，但我已經砍掉了所有與 Windows 平台有關的東西，改以 ANSI C runtime function 來完成底層動作。這些程式碼經過重新編譯後，可以在任何作業系統的 console mode 上執行（唯一要注意的是，你所使用的編譯器是否支援 # 和 ## 前置處理器）。

我們可以從實作 RTTI、Dynamic Creation 以及 Persistence 三大基礎建設的技術之中，獲得非常非常寶貴的經驗。

細細體會本章精心篩選並改編的程式碼，一定能使你對物件導向有更深一層的認識。

程式碼列表（ 本例可以一字不改以 VC++ 或 BC++ 或 G++ 編譯完成）

下面是本章所有程式的原始碼完整列表。

**MFCLITE.H**

#0001 //

#0002 // 檔名：mfclite.h

#0003 // 用途：模擬 MFC framework 的三大重要特質#0004 // 1. Run Time Type Identification #0005 // 2. Dynamic Creation

**269**

多型與虛擬（*Polymorphism in C++*）

#0006 // 3. Persistence（Serialization） #0007 // 本模組供應之 classes：

#0008 // CRuntimeClass, CObject, CFile, CArchive, CDWordArray, #0009 // CObList，及其他純只為模擬 MFC 繼承架構之 empty classes. #0010 // 本模組供應之 macros：

#0011 // DECLARE\_DYNAMIC, IMPLEMENT\_DYNAMIC #0012 // DECLARE\_DYNCREATE, IMPLEMENT\_DYNCREATE #0013 // DECLARE\_SERIAL, IMPLEMENT\_SERIAL

#0014 // 程式碼來源：修剪 MFC source 並略加更改而得。

#0015 // 修改部份：檔案之存取改用 ANSI C 標準函式

#0016 // （MFC 原採用 Win32 API 函式）

#0017 //

#0018

#0019 #ifndef \_MFCLITE\_

#0020 #define \_MFCLITE\_

#0021

#0022 #include <assert.h> // for assert()

#0023 #include <stdio.h> // for file stream's FILE\* #0024

#0025 #define max(a,b) (((a) > (b)) ? (a) : (b))

#0026 #define min(a,b) (((a) < (b)) ? (a) : (b)) #0027

#0028 #define TRACE1 printf #0029 #define UINT unsigned int #0030 #define BOOL int

#0031 #define DWORD unsigned long #0032 #define WORD unsigned short #0033 #define BYTE unsigned char #0034 #define LONG long

#0035 #define TRUE 1

#0036 #define FALSE 0 #0037 #define HFILE FILE\* #0038

#0039 // abstract iteration position #0040 struct POSITION { };

#0041 typedef POSITION\* **POSITION**; #0042

#0043 // forward class declaration #0044 class CArchive;

#0045 class CObject;

#0046

#0047 struct **CRuntimeClass**

#0048 {

#0049 char\* m\_lpszClassName;

#0050 int m\_nObjectSize;

#0051 UINT m\_wSchema; // schema number of the loaded class

#### 270

第５章 Polymorphism（多型）的應用

#0052 CObject\* (\* m\_pfnCreateObject)(); #0053 CRuntimeClass\* m\_pBaseClass;

#0054

#0055 // CRuntimeClass objects linked together in simple list #0056 CRuntimeClass\* m\_pNextClass; // next node

#0057 static CRuntimeClass\* pFirstClass; // start node #0058

#0059 // for Dynamic Creation #0060 CObject\* CreateObject();

#0061

#0062 // for Persistence

#0063 void Store(CArchive& ar) const;

#0064 static CRuntimeClass\* Load(CArchive& ar, UINT\* pwSchemaNum); #0065 };

#0066

#0067 struct **AFX\_CLASSINIT**

#0068 { AFX\_CLASSINIT(CRuntimeClass\* pNewClass); }; #0069

#0070 #define **RUNTIME\_CLASS**(class\_name) \ #0071 (&class\_name::class##class\_name)

#0072

#0073 #define **DECLARE\_DYNAMIC**(class\_name) \ #0074 public: \

#0075 static CRuntimeClass class##class\_name; \

#0076 virtual CRuntimeClass\* GetRuntimeClass() const; #0077

#0078 #define **DECLARE\_DYNCREATE**(class\_name) \ #0079 DECLARE\_DYNAMIC(class\_name) \

#0080 static CObject\* CreateObject(); #0081

#0082 #define **DECLARE\_SERIAL**(class\_name) \ #0083 DECLARE\_DYNCREATE(class\_name) \

#0084 friend CArchive& operator>>(CArchive& ar, class\_name\* &pOb); #0085

#0086 #define **\_IMPLEMENT\_RUNTIMECLASS**(class\_name, base\_class\_name, wSchema, pfnNew) \

#0087 static char \_lpsz##class\_name[] = #class\_name; \ #0088 CRuntimeClass class\_name::class##class\_name = { \

#0089 \_lpsz##class\_name, sizeof(class\_name), wSchema, pfnNew, \ #0090 RUNTIME\_CLASS(base\_class\_name), NULL }; \

#0091 static AFX\_CLASSINIT \_init\_##class\_name(&class\_name::class##class\_name); \

#0092 CRuntimeClass\* class\_name::GetRuntimeClass() const \ #0093 { return &class\_name::class##class\_name; } \ #0094

#0095 #define **IMPLEMENT\_DYNAMIC**(class\_name, base\_class\_name) \

#0096 \_IMPLEMENT\_RUNTIMECLASS(class\_name, base\_class\_name, 0xFFFF, NULL) #0097

#### 271

多型與虛擬（*Polymorphism in C++*）

#0098 #define **IMPLEMENT\_DYNCREATE**(class\_name, base\_class\_name) \ #0099 CObject\* class\_name::CreateObject() \

#0100 { return new class\_name; } \

#0101 \_IMPLEMENT\_RUNTIMECLASS(class\_name, base\_class\_name, 0xFFFF, \ #0102 class\_name::CreateObject)

#0103

#0104 #define **IMPLEMENT\_SERIAL**(class\_name, base\_class\_name, wSchema) \ #0105 CObject\* class\_name::CreateObject() \

#0106 { return new class\_name; } \

#0107 \_IMPLEMENT\_RUNTIMECLASS(class\_name, base\_class\_name, wSchema, \ #0108 class\_name::CreateObject) \

#0109 CArchive& operator>>(CArchive& ar, class\_name\* &pOb) \

#0110 { pOb = (class\_name\*)ar.ReadObject(RUNTIME\_CLASS(class\_name)); return ar; }

#0111

#0112

#0113 //--- MFC Lite classes... #0114

#0115 class **CObject**

#0116 {

#0117 public:

#0118 virtual CRuntimeClass\* GetRuntimeClass() const; #0119 BOOL IsKindOf(const CRuntimeClass\* pClass) const; #0120 virtual void Serialize(CArchive&) { };

#0121

#0122 public:

#0123 static CRuntimeClass classCObject; #0124 };

#0125

#0126 class **CCmdTarget** : public CObject #0127 {

#0128 DECLARE\_DYNAMIC(CCmdTarget)

#0129 };

#0130

#0131 class **CWinThread** : public CCmdTarget #0132 {

#0133 DECLARE\_DYNAMIC(CWinThread)

#0134 };

#0135

#0136 class CWnd;

#0137 class **CWinApp** : public CWinThread #0138 {

#0139 DECLARE\_DYNAMIC(CWinApp)

#0140 };

#0141

#0142

#0143 class **CDocument** : public CCmdTarget

#### 272

第５章 Polymorphism（多型）的應用

#0144 {

#0145 DECLARE\_DYNAMIC(CDocument)

#0146 };

#0147

#0148 class **CWnd** : public CCmdTarget #0149 {

#0150 DECLARE\_DYNCREATE(CWnd)

#0151 };

#0152

#0153 class **CFrameWnd** : public CWnd #0154 {

#0155 DECLARE\_DYNCREATE(CFrameWnd)

#0156 };

#0157

#0158 class **CView** : public CWnd #0159 {

#0160 DECLARE\_DYNAMIC(CView)

#0161 };

#0162

#0163 class **CFile** : public CObject #0164 {

#0165 DECLARE\_DYNAMIC(CFile)

#0166

#0167 public:

#0168 // Flag values #0169 enum OpenFlags {

#0170 modeRead = 0x0000,

#0171 modeWrite = 0x0001, }; #0172

#0173 CFile(char\* lpszFileName, UINT nOpenFlags);

#0174 virtual BOOL Open(char\* lpszFileName, UINT nOpenFlags); #0175 virtual void Close();

#0176 virtual UINT Read(void\* lpBuf, UINT nCount);

#0177 virtual void Write(const void\* lpBuf, UINT nCount); #0178

#0179 HFILE m\_hFile; // associated file #0180

#0181 public:

#0182 virtual ~CFile();

#0183

#0184 protected:

#0185 char m\_strFileName[40]; // file name #0186 };

#0187

#0188 class **CArchive**

#0189 {

#### 273

多型與虛擬（*Polymorphism in C++*）

#0190 public:

#0191 // Flag values

#0192 enum Mode { store = 0, load = 1 }; #0193

#0194 CArchive(CFile\* pFile, UINT nMode); #0195 ~CArchive();

#0196

#0197 BOOL IsLoading() const; #0198 BOOL IsStoring() const; #0199

#0200 UINT Read(void\* lpBuf, UINT nMax);

#0201 void Write(const void\* lpBuf, UINT nMax); #0202 void Close();

#0203

#0204 // Object I/O is pointer based to avoid added construction overhead. #0205 // Use the Serialize member function directly for embedded objects. #0206 friend CArchive& operator<<(CArchive& ar, const CObject\* pOb);

#0207 friend CArchive& operator>>(CArchive& ar, CObject\*& pOb);

#0208 friend CArchive& operator>>(CArchive& ar, const CObject\*& pOb); #0209

#0210 // insertion operations

#0211 CArchive& operator<<(BYTE by); #0212 CArchive& operator<<(WORD w); #0213 CArchive& operator<<(LONG l); #0214 CArchive& operator<<(DWORD dw); #0215 CArchive& operator<<(float f); #0216 CArchive& operator<<(double d); #0217

#0218 CArchive& operator<<(int i); #0219 CArchive& operator<<(short w); #0220 CArchive& operator<<(char ch); #0221 CArchive& operator<<(unsigned u); #0222

#0223 // extraction operations

#0224 CArchive& operator>>(BYTE& by); #0225 CArchive& operator>>(WORD& w); #0226 CArchive& operator>>(DWORD& dw); #0227 CArchive& operator>>(LONG& l); #0228 CArchive& operator>>(float& f); #0229 CArchive& operator>>(double& d); #0230

#0231 CArchive& operator>>(int& i); #0232 CArchive& operator>>(short& w); #0233 CArchive& operator>>(char& ch); #0234 CArchive& operator>>(unsigned& u); #0235

#### 274

第５章 Polymorphism（多型）的應用

#0236 // object read/write

#0237 CObject\* ReadObject(const CRuntimeClass\* pClass); #0238 void WriteObject(const CObject\* pOb);

#0239 void WriteClass(const CRuntimeClass\* pClassRef); #0240 CRuntimeClass\* ReadClass(const CRuntimeClass\* #0241 pClassRefRequested = NULL); #0242 DWORD ReadCount();

#0243 void WriteCount(DWORD dwCount); #0244

#0245 protected:

#0246 BOOL m\_nMode; // open mode

#0247 CFile\* m\_pFile; // associated CFile object #0248 };

#0249

#0250 class **CDWordArray** : public CObject #0251 {

#0252 DECLARE\_SERIAL(CDWordArray)

#0253

#0254 public:

#0255 CDWordArray();

#0256 ~CDWordArray();

#0257

#0258 void SetSize(int nNewSize, int nGrowBy = -1); #0259 int GetSize() const;

#0260

#0261 DWORD GetAt(int nIndex) const; #0262 DWORD& ElementAt(int nIndex); #0263

#0264 // Potentially growing the array

#0265 void SetAtGrow(int nIndex, DWORD newElement); #0266 int Add(DWORD newElement);

#0267

#0268 // overloaded operator helpers #0269 DWORD operator[](int nIndex) const; #0270 DWORD& operator[](int nIndex); #0271

#0272 void Serialize(CArchive&);

#0273

#0274 protected:

#0275 DWORD\* m\_pData; // the actual array of data #0276 int m\_nSize; // # of elements (upperBound - 1) #0277 int m\_nMaxSize; // max allocated

#0278 int m\_nGrowBy; // grow amount #0279 };

#0280

#0281 class **CObList** : public CObject

#### 275

多型與虛擬（*Polymorphism in C++*）

#0282 {

#0283 DECLARE\_SERIAL(CObList)

#0284

#0285 protected:

#0286 struct CNode

#0287 {

#0288 CNode\* pNext;

#0289 CNode\* pPrev;

#0290 CObject\* data;

#0291 };

#0292

#0293 public:

#0294 CObList();

#0295 ~CObList();

#0296

#0297 // count of elements #0298 int GetCount() const; #0299 BOOL IsEmpty() const; #0300

#0301 POSITION AddTail(CObject\* newElement); #0302 void RemoveAll();

#0303

#0304 // iteration

#0305 POSITION GetHeadPosition() const;

#0306 CObject\* GetNext(POSITION& rPosition) const; // return \*Position++ #0307

#0308 void Serialize(CArchive&);

#0309

#0310 protected:

#0311 CNode\* NewNode(CNode\*, CNode\*); #0312

#0313 CNode\* m\_pNodeHead;

#0314 CNode\* m\_pNodeTail;

#0315 int m\_nCount;

#0316 };

#0317

#0318 //--- inline function implementation #0319

#0320 inline int CDWordArray::Add(DWORD newElement) #0321 { int nIndex = m\_nSize;

#0322 SetAtGrow(nIndex, newElement);

#0323 return nIndex; } #0324

#0325 inline DWORD CDWordArray::GetAt(int nIndex) const #0326 { assert(nIndex >= 0 && nIndex < m\_nSize); #0327 return m\_pData[nIndex]; }

#### 276

第５章 Polymorphism（多型）的應用

|  |  |  |
| --- | --- | --- |
| #0328 |  | |
| #0329 | inline | DWORD& CDWordArray::ElementAt(int nIndex) |
| #0330 |  | { assert(nIndex >= 0 && nIndex < m\_nSize); |
| #0331 |  | return m\_pData[nIndex]; } |
| #0332 |  |  |
| #0333 | inline | DWORD CDWordArray::operator[](int nIndex) const |
| #0334 |  | { return GetAt(nIndex); } |
| #0335 |  |  |
| #0336 | inline | DWORD& CDWordArray::operator[](int nIndex) |
| #0337 |  | { return ElementAt(nIndex); } |
| #0338 |  |  |
| #0339 | inline | int CDWordArray::GetSize() const |
| #0340 |  | { return m\_nSize; } |
| #0341 |  |  |
| #0342 | inline | BOOL CArchive::IsStoring() const |
| #0343 |  | { return (m\_nMode & CArchive::load) == 0; } |
| #0344 |  |  |
| #0345 | inline | BOOL CArchive::IsLoading() const |
| #0346 |  | { return (m\_nMode & CArchive::load) != 0; } |
| #0347 |  |  |
| #0348 | inline | CArchive& CArchive::operator<<(int i) |
| #0349 |  | { return CArchive::operator<<((LONG)i); } |
| #0350 | inline | CArchive& CArchive::operator<<(unsigned u) |
| #0351 |  | { return CArchive::operator<<((LONG)u); } |
| #0352 | inline | CArchive& CArchive::operator<<(short w) |
| #0353 |  | { return CArchive::operator<<((WORD)w); } |
| #0354 | inline | CArchive& CArchive::operator<<(char ch) |
| #0355 |  | { return CArchive::operator<<((BYTE)ch); } |
| #0356 |  |  |
| #0357 | inline | CArchive& CArchive::operator<<(BYTE by) |
| #0358 |  | { m\_pFile->Write(&by, sizeof(BYTE)); return \*this; } |
| #0359 | inline | CArchive& CArchive::operator<<(WORD w) |
| #0360 |  | { m\_pFile->Write(&w, sizeof(WORD)); return \*this; } |
| #0361 | inline | CArchive& CArchive::operator<<(LONG l) |
| #0362 |  | { m\_pFile->Write(&l, sizeof(LONG)); return \*this; } |
| #0363 | inline | CArchive& CArchive::operator<<(DWORD dw) |
| #0364 |  | { m\_pFile->Write(&dw, sizeof(DWORD)); return \*this; } |
| #0365 | inline | CArchive& CArchive::operator<<(float f) |
| #0366 |  | { m\_pFile->Write(&f, sizeof(float)); return \*this; } |
| #0367 | inline | CArchive& CArchive::operator<<(double d) |
| #0368 |  | { m\_pFile->Write(&d, sizeof(double)); return \*this; } |
| #0369 |  |  |
| #0370 | inline | CArchive& CArchive::operator>>(int& i) |
| #0371 |  | { return CArchive::operator>>((LONG&)i); } |
| #0372 | inline | CArchive& CArchive::operator>>(unsigned& u) |
| #0373 |  | { return CArchive::operator>>((LONG&)u); } |

#### 277

多型與虛擬（*Polymorphism in C++*）

#0374 inline CArchive& CArchive::operator>>(short& w) #0375 { return CArchive::operator>>((WORD&)w); } #0376 inline CArchive& CArchive::operator>>(char& ch) #0377 { return CArchive::operator>>((BYTE&)ch); } #0378

#0379 inline CArchive& CArchive::operator>>(BYTE& by)

#0380 { m\_pFile->Read(&by, sizeof(BYTE)); return \*this; } #0381 inline CArchive& CArchive::operator>>(WORD& w)

#0382 { m\_pFile->Read(&w, sizeof(WORD)); return \*this; } #0383 inline CArchive& CArchive::operator>>(DWORD& dw)

#0384 { m\_pFile->Read(&dw, sizeof(DWORD)); return \*this; } #0385 inline CArchive& CArchive::operator>>(float& f)

#0386 { m\_pFile->Read(&f, sizeof(float)); return \*this; } #0387 inline CArchive& CArchive::operator>>(double& d)

#0388 { m\_pFile->Read(&d, sizeof(double)); return \*this; } #0389 inline CArchive& CArchive::operator>>(LONG& l)

#0390 { m\_pFile->Read(&l, sizeof(LONG)); return \*this; } #0391

#0392 inline CArchive& operator<<(CArchive& ar, const CObject\* pOb) #0393 { ar.WriteObject(pOb); return ar; }

#0394 inline CArchive& operator>>(CArchive& ar, CObject\*& pOb) #0395 { pOb = ar.ReadObject(NULL); return ar; }

#0396 inline CArchive& operator>>(CArchive& ar, const CObject\*& pOb) #0397 { pOb = ar.ReadObject(NULL); return ar; }

#0398

#0399 inline int CObList::GetCount() const #0400 { return m\_nCount; }

#0401

#0402 inline BOOL CObList::IsEmpty() const #0403 { return m\_nCount == 0; } #0404

#0405 inline POSITION CObList::GetHeadPosition() const #0406 { return (POSITION) m\_pNodeHead; }

#0407

#0408 inline CObject\* CObList::GetNext(POSITION& rPosition) const #0409 // return \*Position++

#0410 { CNode\* pNode = (CNode\*) rPosition; #0411 rPosition = (POSITION) pNode->pNext; #0412 return pNode->data; }

#0413

#0414 #endif /\* \_MFCLITE\_ \*/

#### 278

第５章 Polymorphism（多型）的應用

**MFCLITE.CPP**

#0001 //

#0002 // 檔名：mfclite.cpp #0003 // 用途：實作以下 classes：

#0004 // CRuntimeClass, CObject, 及其他純為展示用的 empty classes... #0005 // CFile, CArchive, CDWordArray, CObList

#0006 // 程式碼來源：修剪 MFC source 並略加更改而得。

#0007 // 修改部份：檔案之存取改用 ANSI C 標準函式

#0008 // （MFC 原採用 Win32 API 函式）

#0009 //

#0010

#0011 #include <stdio.h> // fopen, fclose, fread, fwrite... #0012 #include <memory.h> // memset, memcpy

#0013 #include <string.h> // strlen, strcmp #0014 #include "mfclite.h"

#0015

#0016 IMPLEMENT\_DYNAMIC(CCmdTarget, CObject)

#0017 IMPLEMENT\_DYNAMIC(CWinThread, CCmdTarget)

#0018 IMPLEMENT\_DYNAMIC(CWinApp, CWinThread)

#0019 IMPLEMENT\_DYNAMIC(CDocument, CCmdTarget)

#0020 IMPLEMENT\_DYNCREATE(CWnd, CCmdTarget)

#0021 IMPLEMENT\_DYNAMIC(CView, CWnd)

#0022 IMPLEMENT\_DYNCREATE(CFrameWnd, CWnd)

#0023

#0024 IMPLEMENT\_DYNAMIC(CFile, CObject)

#0025 IMPLEMENT\_SERIAL(CObList, CObject, 0)

#0026 IMPLEMENT\_SERIAL(CDWordArray, CObject, 0)

#0027

#0028 // static members initialization #0029 static char szCObject[] = "CObject";

#0030 struct CRuntimeClass CObject::classCObject =

#0031 { szCObject, sizeof(CObject), 0xFFFF, NULL, NULL, NULL }; #0032 static AFX\_CLASSINIT \_init\_CObject(&CObject::classCObject); #0033 CRuntimeClass\* CRuntimeClass::pFirstClass = NULL;

#0034

#0035 AFX\_CLASSINIT::AFX\_CLASSINIT(CRuntimeClass\* pNewClass)

#0036 {

#0037 pNewClass->m\_pNextClass = CRuntimeClass::pFirstClass; #0038 CRuntimeClass::pFirstClass = pNewClass;

#0039 }

#0040

#0041 //--- CRuntimeClass Implementation --- #0042

#### 279

多型與虛擬（*Polymorphism in C++*）

#0043 CObject\* CRuntimeClass::CreateObject()

#0044 {

#0045 if (m\_pfnCreateObject == NULL) #0046 {

#0047 TRACE1("Error: class not DECLARE\_DYNCREATE or \n" #0048 "DECLARE\_SERIAL : %s\n", m\_lpszClassName); #0049 return NULL;

#0050 }

#0051

#0052 CObject\* pObject = NULL;

#0053 pObject = (\*m\_pfnCreateObject)(); #0054

#0055 return pObject;

#0056 }

#0057

#0058 CRuntimeClass\* CRuntimeClass::Load(CArchive& ar, #0059 UINT\* pwSchemaNum)

#0060 // loads a runtime class description #0061 {

#0062 WORD nLen;

#0063 char szClassName[64];

#0064 CRuntimeClass\* pClass;

#0065

#0066 ar >> nLen;

#0067 ar.Read(szClassName, nLen\*sizeof(char)); #0068 szClassName[nLen] = '\0';

#0069

#0070 for (pClass = pFirstClass; #0071 pClass != NULL;

#0072 pClass = pClass->m\_pNextClass) #0073 {

#0074 if (::strcmp(szClassName, pClass->m\_lpszClassName) == 0)

#0075 return pClass;

#0076 }

#0077

#0078 TRACE1("Error: Class not found: %s \n", szClassName); #0079 return NULL; // not found

#0080 }

#0081

#0082 void CRuntimeClass::Store(CArchive& ar) const #0083 // stores a runtime class description

#0084 {

#0085 WORD nLen = (WORD)::strlen(m\_lpszClassName); #0086 ar << nLen;

#0087 ar.Write(m\_lpszClassName, nLen\*sizeof(char));

#0088 }

#### 280

第５章 Polymorphism（多型）的應用

#0089

#0090 //--- CObject Implementation --- #0091

#0092 CRuntimeClass\* CObject::GetRuntimeClass() const #0093 {

#0094 return &CObject::classCObject;

#0095 }

#0096

#0097 BOOL CObject::IsKindOf(const CRuntimeClass\* pClass) const #0098 {

#0099 CRuntimeClass\* pClassThis = GetRuntimeClass(); #0100 while (pClassThis != NULL)

#0101 {

#0102 if (pClassThis == pClass) #0103 return TRUE;

#0104 pClassThis = pClassThis->m\_pBaseClass; #0105 }

#0106 return FALSE; // walked to the top, no match #0107 }

#0108

#0109 //--- CFile Implementation --- #0110

#0111 CFile::CFile(char\* lpszFileName, UINT nOpenFlags) #0112 {

#0113 Open(lpszFileName, nOpenFlags);

#0114 }

#0115

#0116 UINT CFile::Read(void\* lpBuf, UINT nCount) #0117 {

#0118 if (nCount == 0)

#0119 return 0; // avoid Win32 "null-read" #0120

#0121 assert(lpBuf != NULL); #0122

#0123 if (nCount != ::fread(lpBuf, 1, nCount, m\_hFile)) #0124 assert(FALSE); // fread() fail

#0125

#0126 return (UINT)nCount;

#0127 }

#0128

#0129 void CFile::Write(const void\* lpBuf, UINT nCount) #0130 {

#0131 if (nCount == 0)

#0132 return; // avoid Win32 "null-write" option #0133

#0134 assert(lpBuf != NULL);

#### 281

多型與虛擬（*Polymorphism in C++*）

#0135

#0136 DWORD nWritten = ::fwrite(lpBuf, 1, nCount, m\_hFile); #0137 assert(nWritten == nCount);

#0138 }

#0139

#0140 CFile::~CFile()

#0141 {

#0142 Close();

#0143 }

#0144

#0145 void CFile::Close()

#0146 {

#0147 ::fclose(m\_hFile); #0148 m\_hFile = NULL; #0149 }

#0150

#0151

#0152 BOOL CFile::Open(char\* lpszFileName, UINT nOpenFlags) #0153 {

#0154 // attempt file creation #0155 switch (nOpenFlags)

#0156 {

#0157 case modeRead:

#0158 m\_hFile = ::fopen(lpszFileName, "rt"); #0159 break;

#0160 case modeWrite:

#0161 m\_hFile = ::fopen(lpszFileName, "wt"); #0162 break;

#0163 default:

#0164 assert(FALSE); // invalid share mode #0165 }

#0166

#0167 assert(m\_hFile != NULL); #0168 return TRUE;

#0169 }

#0170

#0171 //--- CDwordArray Implementation --- #0172

#0173 CDWordArray::CDWordArray()

#0174 {

#0175 m\_pData = NULL;

#0176 m\_nSize = m\_nMaxSize = m\_nGrowBy = 0; #0177 }

#0178

#0179 CDWordArray::~CDWordArray()

#0180 {

#### 282

第５章 Polymorphism（多型）的應用

#0181 delete[] (BYTE\*)m\_pData;

#0182 }

#0183

#0184 void CDWordArray::SetSize(int nNewSize, int nGrowBy) #0185 {

#0186 assert(nNewSize >= 0);

#0187

#0188 if (nGrowBy != -1)

#0189 m\_nGrowBy = nGrowBy; // set new size #0190

#0191 if (nNewSize == 0)

#0192 {

#0193 // shrink to nothing #0194 delete[] (BYTE\*)m\_pData; #0195 m\_pData = NULL;

#0196 m\_nSize = m\_nMaxSize = 0;

#0197 }

#0198 else if (m\_pData == NULL) #0199 {

#0200 // create one with exact size

#0201 m\_pData = (DWORD\*) new BYTE[nNewSize \* sizeof(DWORD)]; #0202 ::memset(m\_pData, 0, nNewSize \* sizeof(DWORD));

#0203 m\_nSize = m\_nMaxSize = nNewSize; #0204 }

#0205 else if (nNewSize <= m\_nMaxSize) #0206 {

#0207 // it fits

#0208 if (nNewSize > m\_nSize) #0209 {

#0210 // initialize the new elements #0211 ::memset(&m\_pData[m\_nSize], 0,

#0212 (nNewSize-m\_nSize) \* sizeof(DWORD)); #0213 }

#0214 m\_nSize = nNewSize; #0215 }

#0216 else

#0217 {

#0218 // otherwise, grow array #0219 int nGrowBy = m\_nGrowBy; #0220 if (nGrowBy == 0)

#0221 {

#0222 // heuristically determine growth when nGrowBy == 0 #0223 // (this avoids heap fragmentation in many situations) #0224 nGrowBy = min(1024, max(4, m\_nSize / 8));

#0225 }

#0226 int nNewMax;

#### 283

多型與虛擬（*Polymorphism in C++*）

#0227 if (nNewSize < m\_nMaxSize + nGrowBy)

#0228 nNewMax = m\_nMaxSize + nGrowBy; // granularity #0229 else

#0230 nNewMax = nNewSize; // no slush #0231

#0232 assert(nNewMax >= m\_nMaxSize); // no wrap around

#0233 DWORD\* pNewData = (DWORD\*) new BYTE[nNewMax \* sizeof(DWORD)]; #0234

#0235 // copy new data from old

#0236 ::memcpy(pNewData, m\_pData, m\_nSize \* sizeof(DWORD)); #0237

#0238 // construct remaining elements #0239 assert(nNewSize > m\_nSize); #0240

#0241 ::memset(&pNewData[m\_nSize], 0,

#0242 (nNewSize-m\_nSize) \* sizeof(DWORD)); #0243

#0244 // get rid of old stuff (note: no destructors called) #0245 delete[] (BYTE\*)m\_pData;

#0246 m\_pData = pNewData; #0247 m\_nSize = nNewSize; #0248 m\_nMaxSize = nNewMax; #0249 }

#0250 }

#0251

#0252 void CDWordArray::SetAtGrow(int nIndex, DWORD newElement) #0253 {

#0254 assert(nIndex >= 0);

#0255

#0256 if (nIndex >= m\_nSize) #0257 SetSize(nIndex+1);

#0258 m\_pData[nIndex] = newElement; #0259 }

#0260

#0261 void CDWordArray::Serialize(CArchive& ar) #0262 {

#0263 CObject::Serialize(ar);

#0264

#0265 if (ar.IsStoring())

#0266 {

#0267 ar.WriteCount(m\_nSize);

#0268 ar.Write(m\_pData, m\_nSize \* sizeof(DWORD)); #0269 }

#0270 else

#0271 {

#0272 DWORD nOldSize = ar.ReadCount();

#### 284

第５章 Polymorphism（多型）的應用

#0273 SetSize(nOldSize);

#0274 ar.Read(m\_pData, m\_nSize \* sizeof(DWORD)); #0275 }

#0276 }

#0277

#0278 //--- CArchive Implementation --- #0279

#0280 CArchive::CArchive(CFile\* pFile, UINT nMode) #0281 {

#0282 // initialize members not dependent on allocated buffer #0283 m\_nMode = nMode;

#0284 m\_pFile = pFile; #0285 }

#0286

#0287 CArchive::~CArchive()

#0288 {

#0289 if (m\_pFile != NULL) #0290 Close();

#0291 }

#0292

#0293 void CArchive::Close()

#0294 {

#0295 m\_pFile = NULL; #0296 }

#0297

#0298 UINT CArchive::Read(void\* lpBuf, UINT nMax) #0299 {

#0300 if (nMax == 0)

#0301 return 0;

#0302

#0303 assert(lpBuf != NULL); #0304 assert(IsLoading());

#0305

#0306 return m\_pFile->Read(lpBuf, nMax); #0307 }

#0308

#0309 void CArchive::Write(const void\* lpBuf, UINT nMax) #0310 {

#0311 if (nMax == 0)

#0312 return;

#0313

#0314 assert(lpBuf != NULL); #0315 assert(IsStoring());

#0316

#0317 m\_pFile->Write(lpBuf, nMax);

#0318 }

#### 285

多型與虛擬（*Polymorphism in C++*）

#0319

#0320 void CArchive::WriteCount(DWORD dwCount) #0321 {

#0322 if (dwCount < 0xFFFF) #0323 \*this << (WORD)dwCount; #0324 else

#0325 {

#0326 \*this << (WORD)0xFFFF;

#0327 \*this << dwCount; #0328 }

#0329 }

#0330

#0331 DWORD CArchive::ReadCount()

#0332 {

#0333 WORD wCount; #0334 \*this >> wCount;

#0335 if (wCount != 0xFFFF) #0336 return wCount;

#0337

#0338 DWORD dwCount; #0339 \*this >> dwCount; #0340 return dwCount;

#0341 }

#0342

#0343 void CArchive::WriteObject(const CObject\* pOb) #0344 {

#0345 // write class of object first

#0346 CRuntimeClass\* pClassRef = pOb->GetRuntimeClass(); #0347 WriteClass(pClassRef);

#0348

#0349 // cause the object to serialize itself #0350 ((CObject\*)pOb)->Serialize(\*this);

#0351 }

#0352

#0353 CObject\* CArchive::ReadObject(const CRuntimeClass\* pClassRefRequested) #0354 {

#0355 // attempt to load next stream as CRuntimeClass

#0356 CRuntimeClass\* pClassRef = ReadClass(pClassRefRequested); #0357

#0358 // check to see if tag to already loaded object #0359 CObject\* pOb;

#0360 if (pClassRef == NULL) #0361 {

#0362 TRACE1("bad class\n");

#0363 }

#0364 else

#### 286

第５章 Polymorphism（多型）的應用

|  |  |  |
| --- | --- | --- |
| #0365  #0366 | { | // allocate a new object based on the class acquired |
| #0367 |  | pOb = pClassRef->CreateObject(); |
| #0368 |  | if (pOb == NULL) |
| #0369 |  | TRACE1("Dynamic Creation Fail\n"); |
| #0370 |  |  |
| #0371 |  | pOb->Serialize(\*this); |
| #0372 | } |  |
| #0373 |  |  |

#0374 return pOb;

#0375 }

#0376

#0377 void CArchive::WriteClass(const CRuntimeClass\* pClassRef) #0378 {

#0379 assert(pClassRef != NULL);

#0380 assert(IsStoring()); // proper direction #0381

#0382 // store new class

#0383 pClassRef->Store(\*this);

#0384 }

#0385

#0386 CRuntimeClass\* CArchive::ReadClass(const CRuntimeClass\* pClassRefRequested)

#0387 {

#0388 assert(IsLoading()); // proper direction #0389

#0390 if (pClassRefRequested != NULL &&

#0391 pClassRefRequested->m\_wSchema == 0xFFFF) #0392 {

#0393 TRACE1("Warning: Cannot call ReadClass/ReadObject for %hs.\n", #0394 pClassRefRequested->m\_lpszClassName);

#0395 }

#0396

#0397 CRuntimeClass\* pClassRef;

#0398 UINT nSchema;

#0399 // new object follows a new class id

#0400 if ((pClassRef = CRuntimeClass::Load(\*this, &nSchema)) == NULL) #0401 TRACE1("bad class\n");

#0402

#0403 // return the resulting CRuntimeClass\* #0404 return pClassRef;

#0405 }

#0406

#0407 //--- CObList Implementation --- #0408

#0409 CObList::CObList()

#0410 {

#### 287

多型與虛擬（*Polymorphism in C++*）

#0411 m\_nCount = 0;

#0412 m\_pNodeHead = m\_pNodeTail = NULL; #0413 }

#0414

#0415 CObList::~CObList()

#0416 {

#0417 RemoveAll();

#0418 assert(m\_nCount == 0);

#0419 }

#0420

#0421 void CObList::RemoveAll()

#0422 {

#0423 // destroy elements #0424 CNode\* p = m\_pNodeHead; #0425 while(p != NULL)

#0426 {

#0427 BYTE\* bytes = (BYTE\*) p; #0428 CNode\* pNext = p->pNext; #0429 delete[] bytes;

#0430 p = pNext;

#0431 }

#0432

#0433 m\_nCount = 0;

#0434 m\_pNodeHead = m\_pNodeTail = NULL; #0435 }

#0436

#0437 POSITION CObList::AddTail(CObject\* newElement) #0438 {

#0439 CNode\* pNewNode = NewNode(m\_pNodeTail, NULL); #0440 pNewNode->data = newElement;

#0441 if (m\_pNodeTail != NULL)

#0442 m\_pNodeTail->pNext = pNewNode; #0443 else

#0444 m\_pNodeHead = pNewNode; #0445 m\_pNodeTail = pNewNode; #0446 return (POSITION) pNewNode; #0447 }

#0448

#0449 CObList::CNode\*

#0450 CObList::NewNode(CObList::CNode\* pPrev,

#0451 CObList::CNode\* pNext)

#0452 {

#0453 CObList::CNode\* pNode = (CNode\*) new BYTE[sizeof(CNode)]; #0454 pNode->pPrev = pPrev;

#0455 pNode->pNext = pNext; #0456 m\_nCount++;

#### 288

第５章 Polymorphism（多型）的應用

#0457 assert(m\_nCount > 0); // make sure we don't overflow #0458

#0459 pNode->data = 0; // start with zero #0460 return pNode;

#0461 }

#0462

#0463 void CObList::Serialize(CArchive& ar) #0464 {

#0465 CObject::Serialize(ar);

#0466

#0467 if (ar.IsStoring())

#0468 {

#0469 ar.WriteCount(m\_nCount);

#0470 for (CNode\* pNode = m\_pNodeHead; #0471 pNode != NULL;

#0472 pNode = pNode->pNext) #0473 {

#0474 ar << pNode->data;

#0475 }

#0476 }

#0477 else

#0478 {

#0479 DWORD nNewCount = ar.ReadCount(); #0480 CObject\* newData;

#0481 while (nNewCount--)

#0482 {

#0483 ar >> newData;

#0484 AddTail(newData);

#0485 }

#0486 }

#0487 }

**SHAPE.H**

#0001 //

#0002 // 檔名：shape.h

#0003 // 本模組供應之 classes（Shape hierarchy）：

#0004 // CShape, CSquare, CRect, CCircle, CEllipse, CTriangle #0005 // 注意：為使用「MFCLITE 三大特質」，必須含入 mfclite.h

#0006 //

#0007

#0008 #ifndef \_SHAPE\_

#0009 #define \_SHAPE\_

#0010

#### 289

多型與虛擬（*Polymorphism in C++*）

#0011 #include "mfclite.h" // MFC Lite module #0012

#0013 class **CShape** : public CObject #0014 {

#0015 DECLARE\_DYNCREATE(CShape)

#0016 public:

#0017 virtual void display() { }

#0018 virtual void Serialize(CArchive&) { }; #0019 };

#0020

#0021 class **CSquare** : public CShape #0022 {

#0023 DECLARE\_SERIAL(CSquare)

#0024 public:

#0025 virtual void display();

#0026 virtual void Serialize(CArchive&);

#0027 CSquare(float xval=0.0, float yval=0.0, float hval=0.0) #0028 {

#0029 x = xval;

#0030 y = yval;

#0031 height = hval;

#0032 }

#0033

#0034 protected:

#0035 float x, y; #0036 float height;

#0037 };

#0038

#0039 class **CRect** : public CSquare #0040 {

#0041 DECLARE\_SERIAL(CRect)

#0042 public:

#0043 virtual void display();

#0044 virtual void Serialize(CArchive&); #0045 CRect(float xval=0.0, float yval=0.0, #0046 float hval=0.0, float wval=0.0) #0047 : CSquare(xval, yval, hval)

#0048 {

#0049 width = wval;

#0050 }

#0051

#0052 protected:

#0053 float width;

#0054 };

#0055

#0056 class **CCircle** : public CShape

#### 290

第５章 Polymorphism（多型）的應用

#0057 {

#0058 DECLARE\_SERIAL(CCircle)

#0059 public:

#0060 virtual void display();

#0061 virtual void Serialize(CArchive&);

#0062 CCircle(float xval=0.0, float yval=0.0, float rval=0.0) #0063 {

#0064 x = xval;

#0065 y = yval;

#0066 r = rval;

#0067 }

#0068

#0069 protected:

#0070 float x, y; #0071 float r;

#0072 };

#0073

#0074 class **CEllipse** : public CCircle #0075 {

#0076 DECLARE\_SERIAL(CEllipse)

#0077 public:

#0078 virtual void display();

#0079 virtual void Serialize(CArchive&);

#0080 CEllipse(float xval=0.0, float yval=0.0, #0081 float rval=0.0, float r2val=0.0) #0082 : CCircle(xval, yval, rval)

#0083 {

#0084 r2 = r2val;

#0085 }

#0086

#0087 protected:

#0088 float r2;

#0089 };

#0090

#0091 class **CTriangle** : public CShape #0092 {

#0093 DECLARE\_SERIAL(CTriangle)

#0094 public:

#0095 virtual void display();

#0096 virtual void Serialize(CArchive&);

#0097 CTriangle(float x1val=0.0, float y1val=0.0, #0098 float x2val=0.0, float y2val=0.0,

#0099 float x3val=0.0, float y3val=0.0) #0100 {

#0101 x1 = x1val; y1 = y1val; #0102 x2 = x2val; y2 = y2val;

#### 291

多型與虛擬（*Polymorphism in C++*）

#0103 x3 = x3val; y3 = y3val; #0104 }

#0105

#0106 protected:

#0107 float x1, y1, x2, y2, x3, y3; #0108 };

#0109

#0110 class **CStroke** : public CShape #0111 {

#0112 DECLARE\_SERIAL(CStroke)

#0113 public:

#0114 virtual void display();

#0115 virtual void Serialize(CArchive&); #0116

#0117 public:

#0118 CDWordArray aDArray;

#0119 };

#0120

#0121 #endif /\* \_SHAPE\_ \*/

**SHAPE.CPP**

#0001 //

#0002 // 檔名：shape.cpp

#0003 // 用途：實作以下 classes：

#0004 // CShape, CSquare, CRect, CCircle, CEllipse, CTriangle #0005 //

#0006

#0007 #include <iostream.h>

#0008 #include "shape.h"

#0009

#0010 IMPLEMENT\_DYNCREATE(CShape, CObject)

#0011 IMPLEMENT\_SERIAL(CSquare, CShape, 0)

#0012 IMPLEMENT\_SERIAL(CRect, CSquare, 0)

#0013 IMPLEMENT\_SERIAL(CCircle, CShape, 0)

#0014 IMPLEMENT\_SERIAL(CEllipse, CCircle, 0)

#0015 IMPLEMENT\_SERIAL(CTriangle, CShape, 0)

#0016 IMPLEMENT\_SERIAL(CStroke, CShape, 0)

#0017

#0018 void CSquare::display()

#0019 {

#0020 cout << "CSquare:";

#0021 cout << " x=" << x << ", y=" << y;

#0022 cout << ", height=" << height << endl;

#### 292

第５章 Polymorphism（多型）的應用

#0023 }

#0024

#0025 void CRect::display()

#0026 {

#0027 cout << "CRect:";

#0028 cout << " x=" << x << ", y=" << y;

#0029 cout << ", height=" << height

#0030 << ", width=" << width << endl; #0031 }

#0032

#0033 void CCircle::display()

#0034 {

#0035 cout << "CCircle:";

#0036 cout << " x=" << x << ", y=" << y #0037 << " ,r=" << r << endl; #0038 }

#0039

#0040 void CEllipse::display()

#0041 {

#0042 cout << "CEllipse:";

#0043 cout << " x=" << x << ", y=" << y;

#0044 cout << ", r=" << r << ", r2=" << r2 << endl; #0045 }

#0046

#0047 void CTriangle::display()

#0048 {

#0049 cout << "CTriangle:";

#0050 cout << " x1=" << x1 << ", y1=" << y1; #0051 cout << " x2=" << x2 << ", y2=" << y2;

#0052 cout << " x3=" << x3 << ", y3=" << y3 << endl; #0053 }

#0054

#0055 void CStroke::display()

#0056 {

#0057 cout << "CStroke:"; #0058

#0059 CDWordArray\* pDArray = (CDWordArray\*)&(this->aDArray); #0060 for (int i = 0; i< pDArray->GetSize(); i++)

#0061 cout << " " << (\*pDArray)[i]; #0062 cout << endl;

#0063 }

#0064

#0065 void CSquare::Serialize(CArchive& ar) #0066 {

#0067 CObject::Serialize(ar);

#0068

#### 293

多型與虛擬（*Polymorphism in C++*）

#0069 if (ar.IsStoring())

#0070 {

#0071 ar << x << y << height; #0072 }

#0073 else

#0074 {

#0075 ar >> x >> y >> height; #0076 }

#0077 }

#0078

#0079 void CRect::Serialize(CArchive& ar) #0080 {

#0081 CObject::Serialize(ar);

#0082

#0083 if (ar.IsStoring())

#0084 {

#0085 ar << x << y << height << width; #0086 }

#0087 else

#0088 {

#0089 ar >> x >> y >> height >> width; #0090 }

#0091 }

#0092

#0093 void CCircle::Serialize(CArchive& ar) #0094 {

#0095 CObject::Serialize(ar);

#0096

#0097 if (ar.IsStoring())

#0098 {

#0099 ar << x << y << r; #0100 }

#0101 else

#0102 {

#0103 ar >> x >> y >> r; #0104 }

#0105 }

#0106

#0107 void CEllipse::Serialize(CArchive& ar) #0108 {

#0109 CObject::Serialize(ar);

#0110

#0111 if (ar.IsStoring())

#0112 {

#0113 ar << x << y << r << r2; #0114 }

#### 294

第５章 Polymorphism（多型）的應用

#0115 else

#0116 {

#0117 ar >> x >> y >> r >> r2; #0118 }

#0119 }

#0120

#0121 void CTriangle::Serialize(CArchive& ar) #0122 {

#0123 CObject::Serialize(ar);

#0124

#0125 if (ar.IsStoring())

#0126 {

#0127 ar << x1 << y1 << x2 << y2 << x3 << y3; #0128 }

#0129 else

#0130 {

#0131 ar >> x1 >> y1 >> x2 >> y2 >> x3 >> y3; #0132 }

#0133 }

#0134

#0135 void CStroke::Serialize(CArchive& ar) #0136 {

#0137 CObject::Serialize(ar);

#0138

#0139 if (ar.IsStoring())

#0140 {

#0141 aDArray.Serialize(ar);

#0142 }

#0143 else

#0144 {

#0145 aDArray.Serialize(ar);

#0146 }

#0147 }

**TEST.CPP**

#0001 //

#0002 // 檔名：test.cpp

#0003 // 用途：測試 MFCLITE 模組所提供的三大基礎服務

#0004 // 1. 以 COblist 容納各種 shape objects。

#0005 // 然後輸出到檔案 output.dat 中，驗證 persistence。#0006 // 2. 從檔案 output.dat 中讀出資料並顯示，

#0007 // 驗證 persistence 和 dynamic creation。

#0008 // 3. 呼叫 IsKindOf 及利用 GetRuntimeClass 輸出 class

#### 295

多型與虛擬（*Polymorphism in C++*）

|  |  |  |
| --- | --- | --- |
| #0009 | // | 名稱，以直接驗證 RTTI。上述 1, 2 的成功實已 |
| #0010 | // | 間接驗證了 RTTI 的成功！ |

#0011 //

#0012

#0013 #include <iostream.h>

#0014 #include "mfclite.h" // MFC Lite Module #0015 #include "shape.h" // Shape Hierarchy #0016

#0017 void main()

#0018 {

#0019 // examine persistence(write to file)

|  |  |  |
| --- | --- | --- |
| #0020 | { |  |
| #0021 |  | CShape\* pShape[8]; |
| #0022 |  |  |
| #0023 |  | // prepare raw data |
| #0024 |  | pShape[0] = new CEllipse(3.0, 3.0, 7.0, 21.0); |
| #0025 |  | pShape[1] = new CCircle(5.0, 5.0, 7.0); |
| #0026 |  | pShape[2] = new CTriangle(0.0, 0.0, 1.0, 0.0, 0.0, 1.0); |
| #0027 |  | pShape[3] = new CRect(5.6, 6.8, 3.0, 9.0); |
| #0028 |  | pShape[4] = new CSquare(3.2, 4.3, 6.0); |
| #0029 |  | pShape[5] = new CStroke; |
| #0030 |  | CStroke\* pStroke = dynamic\_cast<CStroke\*>(pShape[5]); |
| #0031 |  | assert(pStroke); |
| #0032 |  | pStroke->aDArray.Add(1); |
| #0033 |  | pStroke->aDArray.Add(2); |
| #0034 |  | pStroke->aDArray.Add(3); |
| #0035 |  | pStroke->aDArray.Add(4); |
| #0036 |  | pStroke->aDArray.Add(5); |
| #0037 |  | pStroke->aDArray.Add(6); |
| #0038 |  | pStroke->aDArray.Add(7); |
| #0039 |  | pShape[6] = new CTriangle(0.0, 0.0, 3.5, 0.0, 0.0, 5.3); |
| #0040 |  | pShape[7] = new CRect(9.9, 9.9, 4.8, 9.8); |
| #0041 |  |  |
| #0042 |  | CObList myList; // my container |
| #0043 |  |  |
| #0044 |  | for (int i=0; i<8; i++) // prepare container's data |
| #0045 |  | myList.AddTail(pShape[i]); |
| #0046 |  |  |
| #0047 |  | // retrieval all data |
| #0048 |  | POSITION pos = myList.GetHeadPosition(); |
| #0049 |  | while (pos != NULL) { |
| #0050 |  | CShape\* pS = (CShape\*)myList.GetNext(pos); |
| #0051 |  | pS->display(); // show it. See p.266 |
| #0052 |  | } |
| #0053 |  |  |
| #0054 |  | CFile\* pFile = new CFile("output.dat", CFile::modeWrite); |

#### 296

第５章 Polymorphism（多型）的應用

#0055 CArchive ar(pFile, CArchive::store); #0056

#0057 myList.Serialize(ar); // persistence, write to file #0058

#0059 ar.Close();

#0060 pFile->Close();

#0061 }

#0062

|  |  |  |
| --- | --- | --- |
| #0063 // examine persistence(read from file) and dynamic creation | | |
| #0064 | { |  |
| #0065 |  | CFile\* pFile = new CFile("output.dat", CFile::modeRead); |
| #0066 |  | CArchive ar(pFile, CArchive::load); |
| #0067 |  | CObList myNewList; // another container |
| #0068 |  | myNewList.Serialize(ar); // persistence, read from file |
| #0069 |  | ar.Close(); |
| #0070 |  | pFile->Close(); |
| #0071 |  |  |
| #0072 |  | // retrieval all data |
| #0073 |  | POSITION pos = myNewList.GetHeadPosition(); |
| #0074 |  | while (pos != NULL) { |
| #0075 |  | CShape\* pS = (CShape\*)myNewList.GetNext(pos); |
| #0076 |  | pS->display(); // show it, see p.267 |
| #0077 |  | } |
| #0078 | } |  |
| #0079 |  |  |
| #0080 // examine IsKindOf() and RTTI (class name query) | | |
| #0081 | { |  |
| #0082 |  | CDocument\* pDoc = new CDocument; |
| #0083 |  | CView\* pView = new CView; |
| #0084 |  | CRect\* pRect = new CRect; |
| #0085 |  |  |
| #0086 |  | cout << pDoc->IsKindOf(RUNTIME\_CLASS(CDocument)) << endl; // 1 |
| #0087 |  | cout << pDoc->IsKindOf(RUNTIME\_CLASS(CObject)) << endl; // 1 |
| #0088 |  | cout << pDoc->IsKindOf(RUNTIME\_CLASS(CCmdTarget)) << endl; // 1 |
| #0089 |  | cout << pDoc->IsKindOf(RUNTIME\_CLASS(CView)) << endl; // 0 |
| #0090 |  | cout << pView->IsKindOf(RUNTIME\_CLASS(CObject)) << endl; // 1 |
| #0091 |  | cout << pView->IsKindOf(RUNTIME\_CLASS(CShape)) << endl; // 0 |
| #0092 |  | cout << pRect->IsKindOf(RUNTIME\_CLASS(CRect)) << endl; // 1 |
| #0093 |  | cout << pRect->IsKindOf(RUNTIME\_CLASS(CSquare)) << endl; // 1 |
| #0094 |  | cout << pRect->IsKindOf(RUNTIME\_CLASS(CShape)) << endl; // 1 |
| #0095 |  | cout << pRect->IsKindOf(RUNTIME\_CLASS(CObject)) << endl; // 1 |
| #0096 |  | cout << pRect->IsKindOf(RUNTIME\_CLASS(CView)) << endl; // 0 |
| #0097 |  |  |
| #0098 |  | cout << pDoc->GetRuntimeClass()->m\_lpszClassName << endl; // CDocument |
| #0099 |  | cout << pView->GetRuntimeClass()->m\_lpszClassName << endl; // CView |
| #0100 |  | cout << pRect->GetRuntimeClass()->m\_lpszClassName << endl; // CRect |

#### 297

多型與虛擬（*Polymorphism in C++*）

#0101 }

#0102 }

註：理論上，這個 MFC-Lite class library 應該可以使用任何編譯器建造完成（因為其中並沒有使用任何特定編譯器所提供之特定功能），並應該可以在任何作業系統上執行（因為凡是作業系統層面的動作，如記憶體配置和檔案存取，MFC-Lite 皆使用標準的 C runtime library）。我已在 Visual C++、Borland C++Builder、GNU C++ 等編譯平台上編譯無誤，並在 Windows console mode、MS-DOS、FreeBSD、Solaris（SunOS）等作業平台上執行無誤。

當然，我想你一定不會犯下這樣的錯誤：在Ａ作業平台上編譯完成後，直接把可執行檔拷貝到Ｂ作業平台去執行。

任何人如果在其他平台（如 OS/2、Mac）上編譯並執行成功（或不成功），盼您

能以 email 告訴我您的寶貴經驗，謝謝。我自己也會設法在不同平台上多做嘗試。

#### 298

第６章 從 C++ Object Model 邁向 Component Object Model

#### 第６章

從 COM 邁向 COM

##### From C++ Object Model to Component Object Model

章名下得很有趣！第一個 COM 是指 C++ Object Model，第二個 COM 是指

Component Object Model。

什麼是 C++ Object Model？這個主題泛指 C++ objects 的記憶體佈局、虛擬機制的形成、C++ 語意的實作技術...。我已經在第２章描述了其中最重要的部份。

C++ 提供的三大物件導向特性：封裝（Encapsulation）、繼承（Inheritance）、多型（Polymorphism，或虛擬 Virtual），帶給我們無限美好的可能與巨大的彈性。如果你把 C++ classes 包裝在一個 library 之中，理想上它就應該像是一個黑盒子，外界只要從說明書中知道其功能和使用介面，就可以運用它，不必知道黑盒子的內部構造與行為。但這只是一種理想，在程度上，目前的 C++ Object Model 有其未至之處：

1. C++ classes 可以跨編譯平台嗎？ Ａ牌編譯器製作的 class library 可否由Ｂ牌編譯器製作的 client 程式使用？
2. C++ classes 可以跨作業平台嗎？ Ａ環境下的製作的 class library 可否

#### 299

多型與虛擬（*Polymorphism in C++*）

由Ｂ環境下的 client 程式使用？

1. C++ classes 可以跨語言平台嗎？ 使用Ａ語言所製作的 class library 可否由以Ｂ語言完成的 client 程式使用？
2. C++ classes 可以跨網路傳輸嗎？ 我指的不是上傳（ upload） 或下傳

（ download）而已，而是指支援分散式（ distributed）電算架構的能力。

這些答案都是否定的，原因在於 C++ Object Model 沒有所謂的二進位標準（binary level standard）。為了達到這些理想，我們需要另一個 Object Model，把 object 封裝得更好。

#### 三套 Object Model

這樣的 Object Model 目前市場上主要有三套：

* Component Object Model（ COM）： 微軟主推的一套規格。
* Common Object Request Broker Architecture （ CORBA ） ： Object Management Group（ OMG） 於 1992 開始主推的一套規格。
* System Object Model（ SOM）： IBM 根據 CORBA 實作出來的一個和程式語言無關的架構。

它們都企圖定義出分散式、二進位層面的物件模型，並且與任何程式語言無關。這些努力將把物件模型經由更多的間接性推往更高的彈性，但也勢必賠上更多的效率（包括空間與時間）。當然，效率的降低只要是在可以接受的範圍內，就不成為問題。以軟體的長遠角度來看，規格的完整度與未來的彈性空間，遠比短程的效率需求來得重要。

這一章主要為你介紹 Component Object Model（COM）。告訴你為什麼需要它， 如何設計它，以及如何實作它。COM 的範圍非常非常廣泛，已經成為 Microsoft 所有技術的基石。本章如果能夠成為你邁向 COM 的第一個踏腳石，我的目的就達到了。而我相信，通過了第２章（C++ Object Model）考驗的你，可以輕易從本章幾個具體而微的實例中，知道 COM 的 why 與 how，並徹底瞭解 COM 之中

#### 300

第６章 從 C++ Object Model 邁向 Component Object Model

最基礎的建設：*IUnknown*。

這薄薄的一章當然不夠讓你一窺 COM 全貌。有了技術基礎之後，你需要其他專論 COM 的書籍。本章最後我會給你一個閱讀導引。

參考資料

本章文字中的程式片段，取材自 Don Box 的 ***Essential COM*** 1 第一章。該書第一章對於 COM 有很好的導引，但僅止於片段片段的說理，沒有展現完整實作碼

（且其片段碼之中含有少量錯誤）。事實上，程式的完整實作過程，需要處理許多細微之處。我將在本章的每一個階段呈現給你完整的、具體而微的程式碼。這些範例程式都是Win32 Console Mode（DOS-like）程式，其目標包括：

1. 如何把 C++ classes 包裝在 DLL 之中， 用以協助 client 程式。
2. 如何把 C++ classes 分離出 interface。這是通往 COM 的第一步。
3. 如何架構美好的 interface 性質， 完成 COM 最基礎的 *IUnknown* 設計。

## C++ Object Model 所面臨的問題

#### 問題現場之一： 新舊雜陳的 DLLs

我目前絕大部份時間使用 Windows 95 作業系統。下面是我的 Windows 目錄中的部份檔案列表：

Volume in drive D is MS-CWIN95 Volume Serial Number is 2518-18E1 Directory of D:\WIN95\SYSTEM

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MFC250 | DLL | 322,384 | 04-26-95 | 23:15 |
| MFC30 | DLL | 322,832 | 11-07-95 | 0:00 |
| MFC40 | DLL | 924,432 | 10-15-96 | 10:40 |

1 ***Essential COM***（Don Box / Addison Wesley / 1998），中譯本名為 COM 本質論（侯俊傑譯

/ 碁峰 / 1998），是一本把 COM 的來龍去脈講解得極為精譬的好書，層次很深。

#### 301

多型與虛擬（*Polymorphism in C++*）

MFC42 DLL 941,840 03-29-97 16:43

它們分別是 MFC 2.5, MFC 3.0, MFC 4.0, MFC 4.2 的動態聯結函式庫（DLLs）。既然稱為動態聯結，為什麼我還要保留不同的版本呢？這豈不是有點失了意義 嗎？

#### 問題現場之二： 不定時炸彈

於是我狠下心來把所有舊版的 MFC DLL 殺掉。幾個日子下來平安無事。直到有一天我執行一個向來可以順利執行的程式，竟然使系統崩潰（crash）。

#### 問題現場之三：

在我的程式設計生涯中，寫了幾個很不錯的 classes，我把它們包裝成 DLL，希望好東西讓好朋友分享。可惜我的好朋友使用 B 牌編譯器，而我使用 M 牌編譯器，他的程式老無法正確地動態聯結上我的 class library（DLL），致使他必須向我要我的 library source code。

#### 問題現場之四：

我還有一些好朋友根本不使用 C++ 語言，他們使用 Java、Visual Basic、Delphi

（Pascal）。把這些好東西（C++ classes library）讓他們分享的念頭，我可是從來都沒有起過，因為我知道那根本是癡人說夢，緣木求魚。

前兩個問題直接關係到軟體使用者，後兩個問題則關係到軟體開發者。我可不想以 source level 相容為滿足，也不妄想 C++ 語言會統一全世界的 programmers。那麼，該如何滿足上述幾項都和 reusability（可重複使用性）有關的需求呢？

讓我先從動態聯結說起。

#### 302

第６章 從 C++ Object Model 邁向 Component Object Model

## 所謂動態聯結

所謂動態聯結，是指 library 的主體（程式碼）並不與呼叫端（client）聯結為一體，而是在 client 載入時期由系統載入器將必要的呼叫關係串接好（術語上稱為fixup），此即謂之「動態（執行時期）」聯結，而非「靜態（編譯時期）」聯結。在編譯時期，client 程式只與 DLL 的一個「用以描述開放介面」的小型 library

（所謂的 import library）做靜態聯結。import library 的體積很小，對於磁碟空間沒有什麼大影響，更不必載入記憶體中，耗費記憶體空間。

如此一來，當 DLL 版本有所更新，只要其對外介面（例如 classes 名稱、member functions 型態、global function 型態）不變，client 程式並不需要重新與之聯結； 當 client 端版本有所更新，也與 DLL 無涉。因此，DLL 的持續演進，對 client 程式而言有著「部份抽換更新」的好處。

DLL 的另一個好處是，library 本體只需存在一份就好，未執行前節省了磁碟空間，執行後則節省了記憶體空間。此外，DLL 一旦修改，應該就有「一呼百應」的效果，因為所有的 clients 都映射到唯一一份 library 本體（這個論點其實是有問題的，請繼續往下看）。

那麼，為什麼我的電腦裡頭要放前後四個版本的 MFC DLLs？不是才說「只要介面不變，DLL 的演化就與 client 無關」嗎？C++ 既然提供了資料封裝的特性， 如果 library 的版本更新只是增加了一些封裝過的資料並改善函式的運作效率， 應該完全不影響 client 才是呀！

**303**

多型與虛擬（*Polymorphism in C++*）

Application A

FastString.obj

靜態聯結：整個 *FastString* 模組會內嵌到應用程式之中，至使應用程式體積暴漲，但獨立。

Application C

FastString.obj

圖 6.1a 靜態聯結示意圖

Application B

FastString.obj

FastString import library

Application A

FastString.DLL

Application B

Application C

動態聯結：只有一個 *FastString* import library（體積很小）會被內嵌到應用程式之中。程式執行時需要 DLL 在場服務。

另一種動態聯結的方式是使用*LoadLibrary()* 和 *GetProcAddress()*，那麼就甚至不再需要 DLL 的 import library。

FastString import library

FastString import library

圖 6.1b 動態聯結示意圖

#### 具體實例

假設廠商Ａ出品一套 class library，其中對於字串的搜尋有一個獨特的演算法。圖

6.2 是這個 class library 的原始碼。請注意，為了在 DLL 之中輸出一個 class， 必須在 class 宣告時加上修飾詞如下：

class \_declspec(dllexport) FastString

#### 304

第６章 從 C++ Object Model 邁向 Component Object Model

如果沒有這麼做，編譯器無法為你在對應的 import library（.LIB）中記錄 class

*FastString* 的相關資訊，也就無從開放（export，匯出）這個 class。

圖 6.2 FastSting DLL（ 示範如何在 DLL 中開放 C++ classes） ：

#### faststr.mak

#0001 # filename : faststr.mak

#0002 # make file for COM sample faststr v1

#0003 # usage : nmake faststr.mak (any VC++ version) #0004

#0005 all: faststr.dll client.exe #0006

#0007 faststr.obj: faststr.cpp faststr.h #0008 cl -c faststr.cpp

#0009

#0010 faststr.dll: faststr.obj

#0011 link -DLL -OUT:faststr.dll faststr.obj #0012

#0013 client.exe : client.cpp faststr.lib #0014 cl client.cpp faststr.lib

#### faststr.h

#0001 class \_declspec(dllexport) FastString #0002 {

#0003 char \*m\_psz;

#0004 public:

#0005 FastString(const char \*psz); #0006 ~FastString(void);

#0007 void Delete(void); // deletes this instance #0008 int Length(void) const; // returns # of characters #0009 int Find(const char \*psz) const; // returns offset #0010 };

#### faststr.cpp

#0001 #include <string.h>

#0002 #include <iostream.h>

#0003 #include "faststr.h" #0004

#0005 // FastString's member functions

#0006 FastString::FastString(const char \*psz) #0007 : m\_psz(new char[strlen(psz) + 1]) {

#### 305

多型與虛擬（*Polymorphism in C++*）

#0008 strcpy(m\_psz, psz);

#0009 }

#0010

#0011 FastString::~FastString(void) {

#0012 delete[] m\_psz;

#0013 cout << "FastString::~FastString" << endl; #0014 }

#0015

#0016 void FastString::Delete(void) { #0017 delete this;

#0018 cout << "FastString::Delete" << endl; #0019 }

#0020

#0021 int FastString::Length(void) const { #0022 return strlen(m\_psz);

#0023 }

#0024

#0025 int FastString::Find(const char \*psz) const { #0026 // O(1) 搜尋碼，略。

#0027 return 1;

#0028 }

#### client.cpp

#0001 #include <iostream.h>

#0002 #include "faststr.h" #0003

#0004 int func()

#0005 {

#0006 int n = -1;

#0007 FastString fs("Hi Bob!"); #0008 n = fs.Find("ob");

#0009 n = fs.Length();

#0010 cout << "Hi Bob! Length = " << n << endl;

#0011 fs.Delete();

#0012 return n;

#0013 }

#0014

#0015 void main()

#0016 {

#0017 cout << "sizeof(FastString)=" << sizeof(FastString) << endl; #0018 int i = func();

#0019 }

#### 306

第６章 從 C++ Object Model 邁向 Component Object Model

#### 執行結果：

sizeof(FastString)=4 Hi Bob! Length = 7

FastString::~FastString FastString::Delete FastString::~FastString

你是否已經發現，程式執行了兩次 destructor？一次發生在 client.cpp L11 呼叫*FastString::Delete()* ， 後者呼叫了 *FastString::~FastString()* 。 另一次發生在client.cpp L13 離開 function scope 時，自動呼叫 *FastString::~FastString()*（當然是編譯器暗中做的手腳）。

對 destructor 呼叫兩次是沒有必要的。*FastString* 在這裡其實並不需要設計一個

*Delete()* 函式，這麼做是為了在更複雜的架構中埋一個伏筆。稍後我會再談到它。

顯然，以 DLL 形式來製作 class library，好處多多：

1. 節省磁碟空間。
2. 節省多工環境下的記憶體空間。
3. 更新 DLLs 即相當於部份抽換 client 程式。
4. 更新一個 DLL 即更新了其所有 clients 程式。但實際上並沒有這麼理想！

#### 307

多型與虛擬（*Polymorphism in C++*）

C++ 缺乏二進位標準（ binary level standard）

問題在於 C++ 缺乏一個二進位標準。C++ Standard 只負責規範「什麼樣的語意

（semantics）會在執行時期造成什麼效果」，卻沒有規範這樣的效果應該如何實作出來。這也就是為什麼我在本書第２章多次提到，vptr 的位置、vtbl 的佈局、virtual functions 的喚起、*this* 指標的移動... 在不同的編譯器上可能有不同的實作方式。

缺乏 binary level 標準，造成的第一個問題是：各家編譯器廠商各有自己的 name mangling 作法（請參考第１章 p.14），DLLs 如果以Ａ編譯器完成，clients 就不能夠使用Ａ以外的編譯器。雖然我們可以使用 extern "C" 來抑制 name mangling

（請參考第１章 p.15），但那只對 global functions 有效，而 class library 匯出的卻是 classes。

第二個問題是：class library 所供應的是 binary level code，而各家編譯器可能以專屬（特別的）方法完成某些 C++ 性質，以維持其高效率及高競爭力（例如第

２章所提的 Microsoft vcall thunk），這使得「語言特性無法跨越 DLL 的邊界」。

第三個問題是：C++ 語言的封裝其實是屬於 source level 層面。換句話說由於

C++ 物件模型沒有一定的標準，「封裝」在 binary level 其實談不上「封裝」。

以先前的 *FastString* 為例，如果 DLL 廠商發現了一個更快的字串搜尋法，但必須因此在 class *FastString* 中增加一個 data member：

class FastString {

const int m\_cch; // count of characters char \*m\_psz;

public:

... // 與前同

};

並修改兩個 member functions：

#### 308

第６章 從 C++ Object Model 邁向 Component Object Model

FastString::FastString(const char \*psz)

: m\_cch(strlen(psz)), m\_psz(new char[m\_cch + 1]) { 2

strcpy(m\_psz, psz);

}

int FastString::Length(void) const {

return m\_cch; // return cached length

}

其中的 *m\_cch* 是一個被封裝的 private member。

經過一番努力，新的 DLL 出品了，名為 FastString 2.0，並延用舊檔名。接著 Client

程式開發廠商獲得了新版 DLL，也開發出 App 2.0（並延用舊檔名），然後把 App

2.0 連同 FastString 2.0 賣給客戶。客戶的電腦之中原本有著 App 1.0 和FastString 1.0，以及使用 FastString 1.0 的其他 client 程式。現在客戶將 App 2.0 安裝妥當，安裝過程中由於檔名相同，FastString 2.0 覆蓋掉了 FastString 1.0。安裝後新的 App 2.0 當然可以愉快執行，其他那些使用 FastString 1.0 的程式卻可能在暗夜中哭泣（也連帶讓客戶哭泣）。

為什麼？新增的 *m\_cch* 不是封裝起來了嗎？不，如果你瞭解 C++ 物件模型（本書第２章），你就知道沒有！FastString 1.0 的 object 大小是 4 bytes，FastString 2.0的 object 大小是 8 bytes，而 1.0 版和 2.0 版的所有 *FastString* member functions 都假設 client 會為 *FastString* objects 配置它們各自以為的大小。那麼，當 1.0 版的 client 程式（並未與 FastString 2.0 的 import library 重新聯結過）配置出一個*FastString* object，其大小依然是 4 bytes。而當它呼喚 *FastString* member functions 準備處理該 object 時，由於目前是 FastString 2.0 當家服務，那些 member functions 會把 object 大小認定為 8 bytes。這種認知上的誤差，使得當某個*FastString* member functions 要存取其 object 的最後 4 個 bytes 時，存取到的卻

2 由於我們要求 *m\_cch* 比 *m\_psz* 更早初始化（因為 *m\_psz* 的初始化需要 *m\_cch* 值），所以在 *FastString* 的宣告中必須把 *m\_cch* 安排在 *m\_psz* 的前面，以免造成錯亂。請參考第１ 章 p.52 的「Initialization List」一節。

#### 309

多型與虛擬（*Polymorphism in C++*）

是屬於他人的記憶體。這種情況下沒有當機，算你幸運3！

這就是為什麼我（以及許多許多程式工作者）的電腦上不敢只留下最新版的 MFC DLL，而必須保留舊版本的原因，因為我實在不知道目前已存在或將來要置入於我的電腦裡頭的應用軟體中，有多少個軟體需要的是舊版 MFC DLL 而非新版MFC DLL！而軟體公司也束手無策，似乎只能以不同的 DLL 檔名來化解可能當機的危機！

*FastString* DLL version 2.0

sizeof(*FastString*) == 8

Application A version 2.0

*FastString* import library

Application B version 1.0

*FastString* import library

sizeof(*FastString*) == 4

###### 4 bytes

4 bytes

Application C version 1.0

*FastString* import library

sizeof(*FastString*) == 4

*FastString* 2.0 的每一個 member functions 都認為它們所要處理的 object 大小是 8 bytes。但 Application B 1.0 或 Application C 1.0 所產生的 object，大小只有 4 bytes。這種對於 object 大小的認知差異便有可能在 object 的處理過程中使系統當掉（crash）。

1. 如果那塊記憶體是無人地帶，或是無關痛癢的地帶，那就只會造成錯誤資料而不會當機。

#### 310

第６章 從 C++ Object Model 邁向 Component Object Model

DLL 和 client 之間的耦合（ coupling） 關係

問題的徵結在於，client 程式必須知道 object 佈局，才能正確配置並使用之。於是， client 和 object DLL 之間形成了一種糾葛攀纏的關係（ 術語稱之為coupling，緊密耦合），不能夠彼此 100% 獨立。這種耦合關係並非全無好處， 它可以使編譯器放心大膽地製造出效率極高的碼（以專用的、私人的、獨密的作法）。但 client 與 object 的耦合關係，對於著眼在「打破編譯器之間的藩籬， 進而打破程式語言之間的藩籬」的所謂 components（軟體元件）而言，卻是個必須克服的障礙。

Components 是什麼？以 source level 的角度看，它就像是 C++ 的 classes。有著自我圓融、自我滿足的資料與動作（函式）。號稱「軟體元件」或「軟體 IC」的components，必須能夠被不同編譯器所完成的 clients 程式或甚至不同語言所完成的 clients 程式使用。它甚至應該具備網路傳輸能力，以協助達成 client-server 分散式電算架構。也就是說，如果甲電腦上執行的Ａ程式需要 10 個 components， Ａ程式可以透過網路分別到 10 台伺服器取得這 10 個 components，讓自己順利執行起來。這 10 個 components 依舊在各自的伺服器上運作，並沒有被抓到甲電腦4；網路上傳輸的是對 components 所下的命令（event 或 message），以及components 的執行結果。

1. 如果將 components 從伺服端抓到 local 端，那就只是 download 動作，沒什麼值得討論的了。

#### 311

多型與虛擬（*Polymorphism in C++*）

抽取一個介面（ interface）

製作components，就必須先解決前述的binary level 封裝問題。要解決這個封裝問題，有一個辦法，就是不要在 client 端發生 object instantiation 動作，而是以一道防火巷阻隔起來。讓 object instantiation 動作永遠發生在 DLL 自身之內，就可以永保版本的正確性，不會再發生 object 大小的認知差異。

DLL 應該只開放單一窗口，以 global function 的姿態出現，並以 extern "C" 修飾之，將編譯器對該函式的 name mangling 動作抑制下來。這個global function 的任務就是 *new* 出一個object。由於各家 C++ 編譯器對於 global function 的處理方式，基本上延用 C 的規格，所以只要函式的供應者（DLL）和函式的使用者

（client 端）指定相同的呼叫習慣（calling convention 5），在參數的傳遞次序和

堆疊的處理便能夠獲得一致。

此外，DLL 中各個 classes 的虛擬機制，與 client 端編譯器所採用的虛擬機制也必須一致。幸運的是市面上幾乎所有 C++ 編譯器都採用 vptr 和 vtbl 來設計虛擬機制（第２章）。雖然 vtbl 的佈局有互不相容的兩大主流6，算是一個壞消息， 然而在某個特定平台上通常某一技術會成為主流7，這又似乎是個好消息。

class *FastString* 原本的所有 public member functions，現在都可以提昇到更上層的一個所謂的 interface class 來，成為一個個 virtual functions。為了保持 interface 的抽象性，同時也為了完全不給予 client 端有關於 object 的任何聯想或耦合機會，interface 不應該有 data members，它的所有 member functions 都應該是 pure

1. 標準的呼叫習慣（calling convention）有三種：pascal、c、stdcall。在「參數的傳遞次序」和

「堆疊處理的責任歸屬」上各不相同。

1. vtbl 佈局的兩大主流技術是所謂的 cfront 技術和 adjustor thunk 技術。
2. Don Box（***Essential COM*** 作者） 曾說 Win32 平台上風行的是 adjustor thunk 技術，Solaris 上風行的是 cfront 技術。

#### 312

第６章 從 C++ Object Model 邁向 Component Object Model

virtual。這些 pure virtual functions 既然都打算開放給外界呼叫使用，以 C++ struct 來製作 interface 將是很理想的，因為C++ struct 中各個 members 的預設存取層級（access level）就是public：

struct declspec(dllexport) **IFastString**

{

virtual void Delete(void) **= 0**; virtual int Length(void) const **= 0**;

virtual int Find(const char \*psz) const **= 0**;

};

class **FastString** : public **IFastString**

{

...

};

DLL 所開放的 global function，主要目的就是讓 *FastString* object 的誕生行為能夠發生在 DLL 自身內，然後把該 object 指標傳回給client：

// part of DLL

extern "C" declspec(dllexport)

**IFastString\*** \_stdcall CreateFastString (const char \*psz) { return new FastString(psz);

}

這個指標必須被轉型為 interface 指標（本例為 *IFastString*\*），這又是多型的一種應用，使外界根據這個指標只能獲得 *IFastString* 的權力範圍，絕對無法存取到*FastString* 的兩個 data members ： *m\_cch* 和 *m\_psz* 。將 *FastString*\* 轉型為*IFastString*\*，是一種「向上轉型（upcast）」，也就是說是一種 implicit cast，絕對沒有危險（請參考第３章 p.168），所以不必藉任何轉型運算子之助就可以直接辦到。

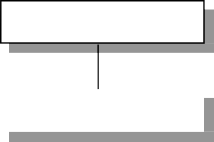
現在我來為 class *FastString* 設計一個可以將資料嚴密封裝的對外介面（所謂的

interface）。圖 6.3 是架構示意，圖 6.4 是模組結構，圖 6.5 是程式碼。

#### 313

多型與虛擬（*Polymorphism in C++*）

Type Hierarchy

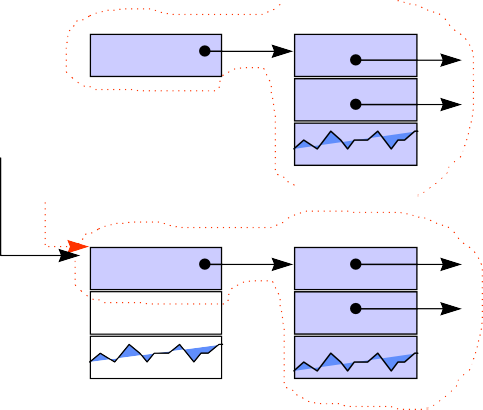


FastString

IFastString

**Binary Layout**

*IFastFasting* object



vtbl

vptr

(FastString\*) pfs

(IFastString\*) pfs

vtbl

vptr

*IFastFasting* subobject in *FastString* object

Application #include “ifaststr.h”

...

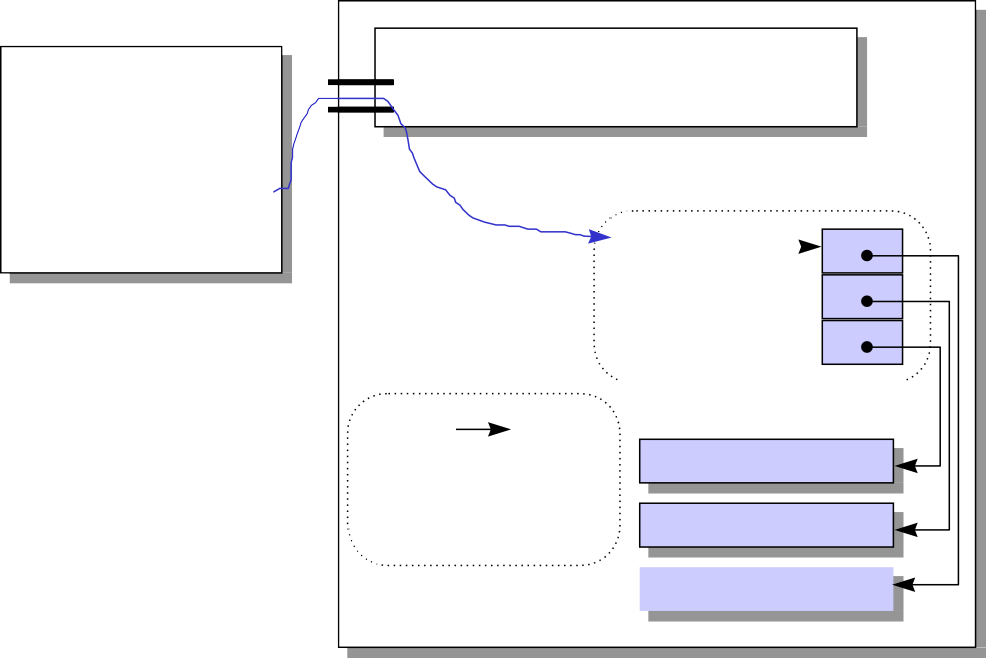
...

IFastSting\* pfs = CreateFastString(“hello”); pfs->Length();

...

DLL

{return new FastString(psz); }



IFastString\* CreateFastString(const char\* psz)

vptr

FastString::Find( )

class FastString : public IFastString

IFastString\*

|  |  |
| --- | --- |
| vptr |  |
|  |
| m\_cch |
| m\_psz |

由於傳回的 *FastString*\* 指標被轉型為 *IFastString*\*（這又是一種多型應用），使得 client 端只能夠透過*IFastString*\* 指標存取其範疇

（scope）內的 members，也就是本例的灰色部份 -- 都是一些 member functions。

struct IFastString

|  |
| --- |
| null |
| null |
| null |

FastString::Delete( ) FastString::Length( )

圖 6.3 經過 interface 隔離之後的 FastString DLL 和 client 程式

#### 314

第６章 從 C++ Object Model 邁向 Component Object Model

end user

application vendor

component vendor



so o

included to

included to

build to

included to

to

build to

dynamic link

dynamic link

sold to

sold to

sold

ld t

IFastString.h

IFastString.h

FastString.dll

FastString.dll

FastString.dll

Client.exe

Client.exe

FastString.cpp

FastString.h

client.cpp

圖 6.4 FastString 的模組結構

圖 6.5 經過 interface 隔離之後的 FastString DLL 和 client 程式碼：

#### faststr.mak

#0001 # filename : faststr.mak

#0002 # make file for COM sample faststr v2

#0003 # usage : nmake faststr.mak (any VC++ version) #0004

#0005 all: faststr.dll client.exe #0006

#0007 faststr.obj: faststr.cpp faststr.h ifaststr.h #0008 cl -c faststr.cpp

#0009

#0010 faststr.dll: faststr.obj

#0011 link -DLL -OUT:faststr.dll faststr.obj #0012

#0013 client.exe : client.cpp ifaststr.h faststr.lib #0014 cl client.cpp faststr.lib

#### 315

多型與虛擬（*Polymorphism in C++*）

#### ifaststr.h

#0001 #define **EXPORT** declspec(dllexport) #0002 #define **API** \_stdcall

#0003

#0004 **struct** EXPORT IFastString

#0005 {

#0006 virtual void Delete(void) **= 0**; #0007 virtual int Length(void) const **= 0**;

#0008 virtual int Find(const char \*psz) const **= 0**; #0009 };

#0010

#0011 extern "C" EXPORT

#0012 **IFastString\*** API CreateFastString(const char \*psz);

#### faststr.h

#0001 #include "ifaststr.h"

#0002

#0003 class FastString : **public IFastString**

#0004 {

#0005 const int m\_cch; // count of characters #0006 char \*m\_psz;

#0007 public:

#0008 FastString(const char \*psz); #0009 ~FastString(void);

#0010 void Delete(void); // deletes this instance

#0011 int Length(void) const; // returns # of characters #0012 int Find(const char \*psz) const; // returns offset #0013 };

#### faststr.cpp

#0001 #include <string.h>

#0002 #include <iostream.h>

#0003 #include "faststr.h"

#0004

#0005 // global function

#0006 extern "C" EXPORT

#0007 **IFastString\*** API CreateFastString (const char \*psz) { #0008 return new FastString(psz);

#0009 }

#0010

#0011 // FastString's member functions

#0012 FastString::FastString(const char \*psz)

#0013 : m\_cch(strlen(psz)), m\_psz(new char[m\_cch + 1]) {

#### 316

第６章 從 C++ Object Model 邁向 Component Object Model

#0014 strcpy(m\_psz, psz);

#0015 }

#0016

#0017 void FastString::Delete(void) { #0018 delete this;

#0019 cout << "FastString::Delete" << endl; #0020 }

#0021

#0022 FastString::~FastString(void) {

#0023 delete[] m\_psz;

#0024 cout << "FastString::~FastString" << endl; #0025 }

#0026

#0027 int FastString::Length(void) const { #0028 return m\_cch;

#0029 }

#0030

#0031 int FastString::Find(const char \*psz) const { #0032 // O(1) 搜尋碼，略。

#0033 return 1;

#0034 }

#### client.cpp

#0001 #include <iostream.h>

#0002 #include "ifaststr.h" // note: not faststr.h

#0003

#0004 int func()

#0005 {

#0006 int n = -1;

#0007 **IFastString** \*pfs = CreateFastString("Polymorphism in C++"); #0008 if (pfs) {

#0009 n = pfs->Find("C++");

#0010 n = pfs->Length();

#0011 cout << "Length = " << n << endl; #0012 pfs->Delete();

#0013 }

#0014 return n;

#0015 }

#0016

#0017 void main()

#0018 {

#0019 cout << "sizeof(IFastString)=" << sizeof(IFastString) << endl; #0020 int i = func();

#0021 }

#### 317

多型與虛擬（*Polymorphism in C++*）

#### 執行結果：

sizeof(IFastString)=4 Length = 19 FastString::~FastString FastString::Delete

請注意，在此架構（interface 的阻絕）之下，你（client 設計者）絕對不可能獲知 *FastString* 的大小，因為你（client 設計者）根本無從獲得 *FastString* 的宣告， 那是 component 開發廠商的原始碼的一部份。你唯一能獲得（買得）的是*FastString* 的 interface（*IFastString*）的宣告，其 object 大小是 4 bytes。是的， 任何 components 的 interface 大小永遠是 4 bytes，因為它只內含一個 vptr。

再請注意 *FastString* 的 destructor 的執行次數。 先前 p.307 我曾提過*FastString::Delete()* 是一個伏筆，當時並沒有必要如此設計。本例之中 client.cpp L12 呼叫 *Delete()*，導至 *IFastString::Delete()* 被喚起。然而此時的 *pfs* 指向一個貨真價實的 *FastString* object，而 *Delete()* 又是個 virtual function，所以喚起的其實是 *FastString::Delete()*（我在第４章談到很多這種情況），其內則是以 *delete* 運算子刪除了 *this* 指標所指物件，於是引發 *FastString::~FastString()*。

然而事情似乎被過度複雜化了。試著思考一下，不要設計 *Delete()*，而在 client.cpp

中這麼做，會如何？：

#0007 IFastString \*pfs = CreateFastString("Polymorphism in C++"); #0008 if (pfs) {

...

#0012 delete pfs;

答案當然是喚起 *pfs* 的 destructor，也就是 *IFastString::~IFastStrting()*。但別忘了C++ Standard 保證 *delete* 的 C++ 語意會喚起從最外圍到最內圍（從繼承體系的最尾端到最頂端）的 destructor，而 *pfs* 如今指向一個貨真價實的 *FastString* object，所以編譯器會企圖先喚起外圍的 *FastString::~FastStrting()*。

可是如果 *IFastString* 的 destructor 不是個 virtual function，編譯器就沒有辦法透

#### 318

第６章 從 C++ Object Model 邁向 Component Object Model

過虛擬機制找到外圍（本例為 *FastString*）的 destructor，也就沒有機會做這個動作（像 faststr.cpp L23 那樣）：

delete[] m\_psz; // m\_psz belong to FastString object

於是 *FastString* object 中的 *m\_psz* 所代表的記憶體空間會遺失掉，造成 memory leak。

那就為 *IFastString* 設計一個 virtual destructor（第１章 p.59） 不就好了嘛！話雖不錯，不幸的是各家 C++ 編譯器對於 virtual destructor 在 vtbl 中的位置並沒有口徑一致。為了保持 component 之於編譯器的獨立性，我們只好在 interface 中捨棄 virtual destructor 的作法，另外設計一個 virtual *Delete()* 函式，並讓 derived class 改寫 *Delete()*，在其中 *delete* 它自己。一如你現在所看到的樣子。

#### 執行時期載入（ Runtime Loading）

為了將 client 和 DLL 的耦合關係進一步砍斷，我們甚至可以不必將 client 與DLL import library 做靜態聯結，而是等到執行時期才利用系統呼叫，將 DLL 載入，並取得其對外窗口（那個 global 函式）。以 Windows 系統為例，系統函式*LoadLibrary()* 可以載入 DLL、*GetProcAddress()* 可以取得某特定函式的位址。我們可以在 client 程式中增加一個函式，有著類似下面的內容（注意，此例並不完全吻合本章實例；本章要求取得的是 *CreateFastString()* 函式位址，下面所取的則是一般比較通用的 *CreateInstance()*，函式名稱及型態都不相同）：

#### 319

多型與虛擬（*Polymorphism in C++*）

typedef void (\***CREATEFUNCPTR**)();

void CallCreateInstance(char\* name)

{

// Load dynamic link library into process.

HINSTANCE hComponent = **::LoadLibrary**(name) ; if (hComponent == NULL)

{

cout << "Error: Cannot load component." << endl ; return;

}

// Get address for CreateInstance function.

CREATEFUNCPTR CreateInstance

= (CREATEFUNCPTR)**::GetProcAddress**(hComponent, "**CreateInstance**") ;

if (CreateInstance == NULL)

{

cout << "Error: Cannot find CreateInstance()" << endl; return;

}

CreateInstance();

}

有了這樣的安排，client 程式甚至可以在執行時期讓使用者選擇載入哪一個DLL。當然，這裡既然用到了系統呼叫，就與特定執行平台扯上關係了。上述程式碼既然使用了兩個 Win32 API 函式，我們因此必須含入 windows.h。

#### 320

第６章 從 C++ Object Model 邁向 Component Object Model

## Classes 的擴展性

截至目前，一切都好。我們能夠製作一個「封裝嚴密，因而可在任何編譯平台上製作，並移植到任何作業平台上執行」的 class library（DLL）。DLL 可以持續演化進步 -- 只要它和 client 間的「契約」沒有改變。所謂契約就是上述的interface，也就是那個由 pure virtual functions 所組成的 struct。

但是如果interface 曾經發佈，你的 components（DLL）也有了一些客戶，那麼之後的任何對於 interface 的改變，都會危害到原來的客戶（client 程式）。

那麼，難道所謂的 DLL 演化，就只是不斷地改善效率，沒有辦法開發新功能嗎

（新功能需要新的介面）？也許你藝高人膽大，對 vtbl 的全盤瞭解使你膽敢把新功能加到原功能的後面：

class IFastString { public:

// version 1.0

virtual void Delete(void) = 0; virtual int Length(void) const = 0;

virtual int Find(const char \*psz) const = 0;

// version 2.0

virtual int FindN(const char \*psz, int n) = 0;

};

因為你正確地知道，上述新的 *IFastString* 宣告，使 object 的佈局改變如下：

class IFastString class FastString : public IFastString

vptr

m\_psz

m\_cch

vptr

|  |
| --- |
| Delete (null) |
| Length (null) |
| Find (null) |
| FindN (null) |

|  |
| --- |
| FastString::Delete |
| FastString::Length |
| FastString::Find |
| FastString::FindN |

**321**

多型與虛擬（*Polymorphism in C++*）

當新版 DLL 為舊版 client 程式提供服務時，上述架構沒有問題。多出一個*FindN()* 函式介面，對舊版 client 程式並沒有影響，因為舊版 client 程式根本對它一無所知，也沒有呼叫它。但如果是舊版 DLL 為新版 client 程式提供服務呢？ 新版 client 程式一呼叫 *FindN()*，就會導至系統當機（crash），因為舊版 DLL 根本沒有 *FindN()* 的存在。

因此，為了保險起見，千萬不要對 interface 有一絲一毫的改變。

#### 擴展辦法之一： 單一繼承並增加繼承深度

但是「讓 component 的功能擴充」的需求仍然存在。一個解決方法是讓 interface

繼承另一個 interface，也就是繼續使用單一繼承並加深其深度。

class IFastString2 : public IFastString { public:

|  |  |
| --- | --- |
| IFastString | |
|  |  |
| IFastString2 | |

// version 2.0

virtual int FindN(const char \*psz, int n) = 0;

};

於是，當 client 端在程式進行之中面對一個 *IFastString*\* 指標時，必須使用*dynamic\_cast* 運算子（第３章）判斷是否能夠安全地將它轉型為 *IFastString2*\*， 然後才能安心使用 *FindN()*：

int Func(IFastString \*pfs)

{

IFastString2 \*pfs2 = dynamic\_cast<IFastString2\*>(pfs); assert(pfs2);

return pfs2->FindN("JJHou", 16);

}

#### 擴展辦法之二： 多重繼承

第二個解決方法是讓 *FastString* 同時繼承兩個（或多個）彼此無關的 interfaces， 也就是使用多重繼承。例如我們新增一個 interface 如下：

#### 322

第６章 從 C++ Object Model 邁向 Component Object Model

class IPersistentObject

{

public:

virtual void Delete(void) = 0;

virtual bool Load(const char \*pszFileName) = 0; virtual bool Save(const char \*pszFileName) = 0;

};

class FastString : public IFastString,

public IPersistentObject

{

private:

FastString

IFastString

IPersistentObject

int m\_cch; // count of characters char \*m\_psz;

public:

FastString(const char \*psz);

~FastString(void);

// common methods

void Delete(void); // deletes this instance

// IFastString methods

int Length(void) const; // returns # of characters int Find(const char \*psz) const; // returns offset

// IPersistentObject methods

bool Load(const char \*pszFileName); bool Save(const char \*pszFileName);

};

當 client 端在程式進行之中面對一個 *IFastString*\* 指標時， 必須使用*dynamic*\_*cast* 運 算 子 （ 第 ３ 章 ） 判 斷 是 否 能 夠 安 全 地 將 它 轉 型 為*IPersistentObject*\*，然後才能安心使用 *Load()* 和 *Save()* 兩函式（注意：這是所謂的橫向轉型，第３章p.177 提過）：

bool Func(IFastString \*pfs, const char \*pszFN)

{

bool bResult = false; IPersistentObject \*ppo =

dynamic\_cast<IPersistentObject\*>(pfs); assert(ppo);

return ppo->Save(pszFN);

}

#### 323

多型與虛擬（*Polymorphism in C++*）

我們在這裡體驗了 *dynamic*\_*cast* 運算子的甜頭。然而對於 components 而言，這個甜頭猶如包了糖衣的毒藥，因為 *dynamic*\_*cast* 屬於 RTTI 的機能之一，而 C++ Standard 只規範了 RTTI 的語法和語意，並未規範 RTTI 的實作方法。目前已知各 C++ 編譯器廠商對於 RTTI 是各顯神通，沒有標準可言。那麼，欲繼續保持

「和任何編譯平台無關」的性質，唯一可行的方法就是，自己完成一個類似的函式。假設名之為 *Dynamic\_Cast()*。現在，讓我們把整個 interface 架構重新劃分得更模組化一些（圖 6.6 可以表現出這個架構）：

class IExtensibleObject

{

public:

virtual void \*Dynamic\_Cast(const char \*pszType) = 0; virtual void Delete(void) = 0;

};

class IPersistentObject : public IExtensibleObject

{

public:

virtual bool Load(const char \*pszFileName) = 0; virtual bool Save(const char \*pszFileName) = 0;

};

class IFastString : public IExtensibleObject

{

public:

virtual int Length(void) const = 0;

virtual int Find(const char \*psz) const = 0;

};

class FastString : public IFastString,

public IPersistentObject

{

int m\_cch; // count of characters char \*m\_psz;

public:

FastString(const char \*psz);

~FastString(void);

// IExtensibleObject methods

void \*Dynamic\_Cast(const char \*pszType);

#### 324

第６章 從 C++ Object Model 邁向 Component Object Model

void Delete(void); // deletes this instance

// IFastString methods

int Length(void) const; // returns # of characters int Find(const char \*psz) const; // returns offset

// IPersistentObject methods

bool Load(const char \*pszFileName); bool Save(const char \*pszFileName);

};

Type Hierarchy Binary Layout

FastString

IPersistentObject

IFastString

IExtensibleObject

IExtensibleObject

m\_psz

m\_cch

vptr

vptr

|  |
| --- |
| FastString::Dynamic\_Cast |
| FastString::Delete |
| FastString::Length |
| FastString::Find |

圖 6.6 加入更多 interfaces 之後的 FastString 架構及佈局

|  |
| --- |
| FastString::Dynamic\_Cast |
| FastString::Delete |
| FastString::Load |
| FastString::Save |

#### 325

多型與虛擬（*Polymorphism in C++*）

*Dynamic\_Cast()* 的實作並不困難，只要利用 *static*\_*cast* 運算子來限制 *this* 指標的權力範圍（scope）即可模擬出來：

void \*FastString::Dynamic\_Cast(const char \*pszType)

{

if (strcmp(pszType, "IFastString") == 0) return static\_cast<IFastString\*>(this);

else if (strcmp(pszType, "IPersistentObject") == 0) return static\_cast<IPersistentObject\*>(this);

else if (strcmp(pszType, "IExtensibleObject") == 0) return static\_cast<IFastString\*>(this);

else

return 0; // 外界詢問的是一個未支援的 interface

}

為什麼棄「*dynamic\_cast* 之直接」而就「*static*\_*cast* 的模擬」？原因一如我才剛說過的， 各家編譯器對於 *dynamic*\_*cast* 運算子的作法是各顯神通， 對於*static*\_*cast* 運算子的作法則是相當一致（因為它並不牽扯虛擬機制），那麼我們的程式碼就不會被牽制於某個特定的編譯平台上了。

請注意，當 client 端詢問共同的 base interface *IExtensibleObject* 時，上述實作碼將指標轉型為 *IFastString*。這是因為下面這個指令曖昧不明（第３章 p.175 曾提過）：

return static\_cast<IExtensibleObject\*>(this)

原因是 *IFastString* 和 *IPersistentObject* 兩者都衍生自 *IExtensibleObject*。如果*IExtensibleObject* 是它們的一個虛擬基礎類別，那麼這個轉型動作就不再曖昧不明，上述指令也將通過編譯。然而，虛擬基礎類別會大幅度增加「物件執行時期的複雜度」，而且各家編譯器對於虛擬繼承有不同的作法，完全沒有標準可言。所以 COM 並不使用虛擬繼承。

#### 326

第６章 從 C++ Object Model 邁向 Component Object Model

## 物件生命的管理

由於 object 指標可能在 client 程式中因應各種場合的需求而一再一再地被施以*Dynamic\_Cast()* 動作，轉型為不同的 base class type（也就是各種 interface type）， 所以以下畫面就會常常出現在 client 程式中：

int func(void)

{

**IFastString**\* pfs = 0;

**IPersistentObject**\* ppo = 0;

pfs = CreateFastString("hello everybody"); if (pfs)

{

ppo = (IPersistentObject\*)pfs->Dynamic\_Cast("IPersistentObject"); if (!ppo)

pfs->Delete(); else {

ppo->Save("C:\\autoexec.bat");

ppo->Delete();

}

}

}

雖 然 上 例 最 初 是 以 *IFastString*\* 繫結到 object 身上， 最後卻是經由*IPersistentObject\** 來呼叫 *Delete()* 函式。沒錯，C++ 虛擬機制會自動處理這件事情（因為所有的 *IExtensibleObject* derived vtbls 都將指向 *Delete()* 函式的唯一一份實作碼，請參考第２章），然而把「追蹤哪一個指標被關聯到哪一個 object， 並確定針對每個 object 只呼叫 *Delete()* 一次」的責任放在 client 程式身上，頗為殘忍。在稍微大一點的 client 程式中，這個任務十分複雜，會把 client 開發人員搞得人仰馬翻。

最理想的作法就是把 object 生命管理的責任交給 components。我們可以讓component 本身維護一個 reference count（參用計數），每當有一個 interface 指標被複製，它就 +1，每當有一個 interface 指標被摧毀，它就 -1，當它變成 0， component 就把指標 *delete* 掉。

**327**

多型與虛擬（*Polymorphism in C++*）

於是 *IExtensibleObject* 的定義必須從原本的：

class IExtensibleObject

{

public:

virtual void \*Dynamic\_Cast(const char\* pszType) = 0;

virtual void Delete(void) = 0;

};

改變為：

class IExtensibleObject

{

public:

virtual void \*Dynamic\_Cast(const char\* pszType) = 0;

virtual void DuplicatePointer(void) = 0; virtual void DestroyPointer(void) = 0;

};

這兩個函式實作如下：

class FastString : public IFastString,

public IPersistentObject

{

int m\_cPtrs; // count of outstanding ptrs

...

public:

// 將指標計數器設為 0

FastString(const char \*psz) : m\_cPtrs(0) { }

void DuplicatePointer(void)

{

++m\_cPtrs; // 表示「指標被複製了」

}

void DestroyPointer(void)

{

// 當最後一個指標被摧毀，就刪除此一 object！ if (--m\_cPtrs == 0)

delete this;

}

...

};

#### 328

第６章 從 C++ Object Model 邁向 Component Object Model

為了在岔斷式多緒環境（preemptive multi-threading environment）之中保證上述reference count 的遞增和遞減動作絕對不會被打斷（被打斷的話就有可能糟糕）， 我們可以使用作業系統所提供的特殊系統函式來取代 ++ 和 -- 運算子。以Windows 作業系統為例，我們可以使用兩個 Win32 函式 *InterlockedIncrement()* 和 *InterlockedDecrement()*，將上述兩個函式重新定義為：

void DuplicatePointer(void)

{

InterlockedIncrement(&m\_cPtrs);

}

void DestroyPointer(void)

{

if (InterlockedDecrement(&m\_cPtrs) == 0) delete this;

}

現在，*IExtensibleObject* 的所有 user 都必須遵循兩項遊戲規則：

1. 當一個 interface 指標被複製， 就得呼叫 *DuplicatePointer()*。
2. 當一個 interface 指標不再有用， 就得呼叫 *DestroyPointer()*。

為了支援這些函式， 所有處理或管理 interface 指標的程式碼，都必須固守*DupliatePointer* / *DestroyPointer* 這兩個簡單規則。因此，*FastString* 也有兩個函式需要配合修改如下：

IFastString\* CreateFastString(const char \*psz)

{

IFastString \*pfsResult = new FastString(psz);

if (pfsResult)

pfsResult->DuplicatePointer();

return pfsResult;

}

#### 329

多型與虛擬（*Polymorphism in C++*）

void \*FastString::Dynamic\_Cast(const char \*pszType)

{

void \*pvResult = 0;

if (strcmp(pszType, "IFastString") == 0) pvResult = static\_cast<IFastString\*>(this);

else if (strcmp(pszType, "IPersistentObject") == 0) pvResult = static\_cast<IPersistentObject\*>(this); else if (strcmp(pszType, "IExtensibleObject") == 0)

pvResult = static\_cast<IFastString\*>(this); else

return 0; // client 所詢問的，是一個未被支援的 interface

// pvResult 現在內含一個被複製的指標，所以

// 我們必須在 return 之前呼叫 DuplicatePointer

((IExtensibleObject\*)pvResult)->DuplicatePointer();

return pvResult;

}

現在，client 程式可以變得更清爽而一致得多，並且在愈複雜的程式中愈能顯示其成效：

void func(void)

{

IFastString \*pfs = 0; IPersistentObject \*ppo = 0;

pfs = CreateFastString("Feed BOB"); if (pfs)

{

ppo = (IPersistentObject \*)pfs->Dynamic\_Cast("IPersistentObject"); if (ppo)

{

ppo->Save("C:\\autoexec.bat"); ppo->DestroyPointer();

}

pfs->DestroyPointer();

}

}

你看到了，client 只要遵循兩個簡單的規則，就可以讓 components 自己管理自己的生命。

#### 330

第６章 從 C++ Object Model 邁向 Component Object Model

## 程式總整理，IUnknown 現身

現在我把整個 *FastString* 做個整理，並且把幾個名稱改頭換面一下：

* 把 *IExtensibleObject* 改為 *IUnknown*
* 把 *Dynamic*\_*Cast* 改名為 *QueryInterface*
* 把 *DuplicatePointer* 改名為 *AddRef*
* 把 *DestroyPointer* 改名為 *Release*

可能有不少讀者已經看過一些些 COM 程式，或讀過一些些 COM 書籍，這可不就恍然大悟了：原來 COM 中最基礎的 interface：*IUnknown*，以及它的三個member functions：*QueryInterface*、*AddRef*、*Release*，就是這麼回事。

完整程式列於圖 6.7。請注意，為了讓這個 component 適用於多緒環境，我在編譯時加上 -MT 選項，聯結時使用 libcmt.lib，這都是 Visual C++ 編譯器上的設定。此外，我並沒有像前面所說的那樣，使用 Win32 API 來確保reference count 的增減動作不會在岔斷式多緒多工環境中被干擾，不過你可以試著這麼做。

我所提供的這個程式裡頭，許多函式並沒有實際作用，常常只是個空函式或一些無關痛癢的文字輸出動作。我的目的只在於向你展示 COM 的 component 和application 的架構而已。

**COM 只是一份規格**

我想你已經十分清楚了，COM 是一個規格，而 C++ 的 abstract class 與 virtual function 的實作方式最貼近這份規格。

#### 331

多型與虛擬（*Polymorphism in C++*）

## 接下去的路

這一章只是個開始，告訴你為什麼需要發展出 COM，告訴你 interface 是怎麼回事，*IUnknown* 是怎麼回事，*QueryInterface()* 做些什麼動作，*AddRef()* 和 *Release()* 又做些什麼動作。但你切莫以為這就是一個標準的 COM component；本章例子雖然展現了 COM 的精神，距離真正的 COM，卻還有一段距離。不過，有了這些基礎，我相信現在你有足夠的能力去看一些相關「大書」了。

下面是兩本極佳的 COM 書籍，給你做為參考。搭配著看，可得最大效益。

* ***Essential COM***（ Don Box / Addison Wesley / 1998； 7 章， 440 頁） 中譯本： COM 本質論（ 侯俊傑譯/碁峰/1998）
  1. COM as a Better C++
  2. Interfaces
  3. Classes
  4. Objects
  5. Apartments
  6. Applications
  7. Miscellanea
     1. The Evolution of Objects
     2. Selected Code Fragements

這是一本能夠把 COM 來龍去脈說得一清二楚的好書。第一章可以讓你清楚COM 的設計動機，包括interface、reference counting、*CreateObject*...。第二章介紹 COM interfaces。作者在前一章步步經營演化而來的一個 virtual base class 此時搖身一變成為功能對等的 COM *IUnknown*。這一章介紹了IDL（一種 interface 描述語言）、MIDL（微軟的一套 IDL 編譯器）、GUID（保證絕不重複的一個 128-bit數字）、HRESULT（任何平台通用的一種回返值型態）、*IUnknown*（所有 COM interfaces 的基礎）。

第三章開始，本書往深澀的方向走去，比較好的方式是穿插著看 ***Inside COM***。

**332**

第６章 從 C++ Object Model 邁向 Component Object Model

本書用詞比較學術化（我的意思是艱澀）。這會使不習慣的讀者在攻擊發起時進度受阻。閱讀時請特別留心前兩章，它們是學習 COM 的關鍵。

* ***Inside COM***（ Dale Rogerson / Microsoft Press / 1996； 12 章， 376 頁）
  + - 1. Components
      2. The Interfaces
      3. QueryInterface
      4. Reference Counting
      5. Dynamic Linking
      6. HRESULTs, GUIDs, the Registry, and Other Details
      7. The Class Factory
      8. Component Reuse : Containment and Aggregation
      9. Making It Easier
      10. Servers in EXEs
      11. Dispatch Interfaces and Automation
      12. Multiple Threads
      13. Putting It All Together

這本書文字淺顯，義理精確，前進速度適中，每一個階段附有一個完整的程式。只可惜一開始未能像 ***Essential COM*** 一樣地從 COM 起源開始分析。不過，我相信現在你已經有能力快樂地閱讀它了。

Dale Rogerson 的這本書，每一章每一節的前後轉承文字說明接續得非常不錯，每一個句子的語意也很清晰（不會用一大串一長串的子句連接詞），讓我們一路讀來順暢快意，不需要多花心思在其文章架構，可以把全付精神用來對付 COM 技術。

#### 333

多型與虛擬（*Polymorphism in C++*）

圖 6.7 FastString 第三個版本。模擬 COM / IUnknown 架構：

#### faststr.mak

#0001 # filename : faststr.mak

#0002 # make file for COM sample faststr v4

#0003 # usage : nmake faststr.mak (any VC++ version) #0004

#0005 all: faststr.dll client.exe #0006

#0007 faststr.obj: faststr.cpp faststr.h ifaststr.h #0008 cl -c -MT faststr.cpp

#0009

#0010 faststr.dll: faststr.obj

#0011 link -DLL -OUT:faststr.dll faststr.obj libcmt.lib #0012

#0013 client.exe : client.cpp ifaststr.h faststr.lib #0014 cl client.cpp faststr.lib

#### ifaststr.h

#0001 // ifaststring.h

#0002

#0003 #define **EXPORT** declspec(dllexport) #0004 #define **API** \_stdcall

#0005

#0006 struct EXPORT IUnknown { #0007 virtual void \*QueryInterface(const char \*pszType) = 0; #0008 virtual void AddRef(void) = 0;

#0009 virtual void Release(void) = 0;

#0010 };

#0011

#0012 struct EXPORT IPersistentObject : public IUnknown { #0013 virtual int Load(const char \*pszFileName) = 0;

#0014 virtual int Save(const char \*pszFileName) = 0; #0015 };

#0016

#0017 struct EXPORT IFastString : public IUnknown { #0018 virtual int Length(void) const = 0;

#0019 virtual int Find(const char \*psz) const = 0; #0020 };

#0021

#0022 extern "C" EXPORT

#0023 IFastString \* API CreateFastString(const char \*psz);

#### 334

第６章 從 C++ Object Model 邁向 Component Object Model

#### faststr.h

#0001 // faststring.h ///////////////////////////

#0002 #include "ifaststr.h" #0003

#0004 class FastString : public IFastString, #0005 public IPersistentObject

#0006 {

#0007 int m\_cPtrs; // count of outstanding ptrs #0008 int m\_cch; // count of characters

#0009 char \*m\_psz;

#0010 public:

#0011 FastString(const char \*psz); #0012 ~FastString(void);

#0013

#0014 // IUnknown methods

#0015 void \*QueryInterface(const char \*pszType); #0016 void AddRef(void) { ++m\_cPtrs; }

#0017 void Release(void)

#0018 { if (--m\_cPtrs == 0) delete this; } #0019

#0020 // IFastString methods

#0021 int Length(void) const; // returns # of characters #0022 int Find(const char \*psz) const; // returns offset #0023

#0024 // IPersistentObject methods

#0025 int Load(const char \*pszFileName); #0026 int Save(const char \*pszFileName); #0027 };

#### faststr.cpp

#0001 // faststring.cpp (part of DLL) //////////// #0002 #include <string.h>

#0003 #include <iostream.h>

#0004 #include "faststr.h" #0005

#0006 // global function #0007 extern "C" EXPORT

#0008 IFastString\* API CreateFastString (const char \*psz) #0009 {

#0010 IFastString \*pfs = new FastString(psz); #0011 if (pfs)

#0012 pfs->AddRef();

#0013 return pfs;

#### 335

多型與虛擬（*Polymorphism in C++*）

#0014 }

#0015

#0016 // FastString's member functions

#0017 FastString::FastString(const char \*psz) #0018 : m\_cPtrs(0), m\_cch(strlen(psz)),

#0019 m\_psz(new char[m\_cch + 1])

#0020 {

#0021 strcpy(m\_psz, psz);

#0022 }

#0023

#0024 FastString::~FastString(void)

#0025 {

#0026 delete[] m\_psz;

#0027 // cout << "FastString::~FastString" << endl; #0028 }

#0029

#0030 int FastString::Length(void) const #0031 {

#0032 return m\_cch;

#0033 }

#0034

#0035 int FastString::Find(const char \*psz) const #0036 {

#0037 // O(1) 搜尋碼，略。

#0038 // cout << "FastString::Find" << endl; #0039 return 1;

#0040 }

#0041

#0042 int FastString::Load(const char \*pszFileName) #0043 {

#0044 // cout << "FastString::Load" << endl; #0045 return 1;

#0046 }

#0047

#0048 int FastString::Save(const char \*pszFileName) #0049 {

#0050 // cout << "FastString::Save" << endl; #0051 return 1;

#0052 }

#0053

#0054 void \*FastString::QueryInterface(const char \*pszType) #0055 {

#0056 void \*pvResult = 0;

#0057 if (strcmp(pszType, "IFastString") == 0) #0058 pvResult = static\_cast<IFastString\*>(this);

#0059 else if (strcmp(pszType, "IPersistentObject") == 0)

#### 336

第６章 從 C++ Object Model 邁向 Component Object Model

#0060 pvResult = static\_cast<IPersistentObject\*>(this); #0061 else if (strcmp(pszType, "IUnknown") == 0)

#0062 pvResult = static\_cast<IFastString\*>(this); #0063 else

#0064 return 0; // 外界詢問的是一個未支援的 interface

#0065

#0066 ((IUnknown\*)pvResult)->AddRef();

#0067 return pvResult;

#0068 }

#### client.cpp

#0001 #include <iostream.h>

#0002 #include "ifaststr.h" #0003

#0004 void func()

#0005 {

#0006 IFastString \*pfs = CreateFastString("Polymorphism in C++"); #0007 IPersistentObject \*ppo = 0;

#0008 if (pfs) { #0009 int n = -1;

#0010 n = pfs->Find("C++");

#0011 n = pfs->Length();

#0012 cout << "Length = " << n << endl; #0013

#0014 ppo= (IPersistentObject\*)pfs->QueryInterface("IPersistentObject"); #0015 if (ppo) {

#0016 ppo->Save("C:\\autoexec.bat");

#0017 ppo->Release();

#0018 }

#0019 pfs->Release();

#0020 }

#0021 }

#0022

#0023 void main()

#0024 {

#0025 func();

#0026 }

#### 337

多型與虛擬（*Polymorphism in C++*）

#### 338

附錄A 參考書目

**附錄Ａ**

# 參考書目

以下是我寫作本書時所參考的資料。所列書籍同時也是我所極力推薦的作品，它們都是各自領域的最佳書籍。

* ***C++ Primer 3/e*** Stanley B. Lippman/Addison Wesley/1998
* ***Inside The C++ Object Model*** Stanley B. Lippman/Addison Wesley/1996

中譯本：深度探索 C++ 物件模型（侯俊傑譯/碁峰/1998）

* ***Essential COM*** Don Box/Addison Wesley/1998

中譯本：COM 本質論（侯俊傑譯/碁峰/1998）

* ***Inside COM*** Dale Rogerson/Microsoft Press/1996
* **深入淺出 MFC *2/e*** 侯俊傑/ 松崗/ 1997
* MFC source code Microsoft Corp.

**339**

多型與虛擬（*Polymorphism in C++*）

#### 340

附錄B C++ 的沉迷與愛戀

**附錄 B**

# C++ 的

沉迷與愛戀

侯捷 [jjhou@ccca.nctu.edu.tw](mailto:jjhou@ccca.nctu.edu.tw) 1998.09.28 第一次發表於清大資訊BBS（楓橋驛站）電腦書訊版 140.114.87.5

每年的 09/28 於我都是一個特殊的日子 -- 不只是因為教師節。今年很特殊地沒有普天同慶，那麼我就寫篇文章自己慶祝一下好了。

我於今年七月發表了一本著作 多型與虛擬 和一本譯作 深度探索 C++ 物件模型，獲得很大的迴響。這些作品都不是針對 C++ 的完全初學者所寫，但從初階到高階為數眾多的 C++ guy，熱情地表達了他們對這些主題的喜悅。

在許多來信中，我看到一些有趣的現象，也感受到一些值得整理下來的想法。所以，根據我個人的學習過往、我的教學經驗、以及週遭朋友的心得交流，寫下這篇文章，或可為後學者戒。

**341**

多型與虛擬（*Polymorphism in C++*）

#### 多型與虛擬 序言節錄

首先讓我節錄 多型與虛擬 一書序言：

一般而言 C++ 是一個難學易用的語言。C++ 的難學，初始在於其重重的佈幕， 佈幕之中編譯器對我們的程式碼做了太多的手腳，使我們慣於循序思考的工程腦袋一無所措。及長，又面臨新的思維模式，使我們必須扭轉慣常的思考習慣。

C++ 的易用則在於其巨大的彈性，能夠以多型（polymorphism）、虛擬（virtual）、模板（template）、泛型（generalization）等種種型式，讓既有的碼去處理未知的、未來的資料型態。

當然，易用必須先能用。用不好或不能用的話，「寫 C++ 程式」最後就成了只

是「使用 C++ 編譯器」，這是大家常拿來彼此調侃的笑話。

在「難學」的背景下，「易用」是使我們依然前仆後繼的動力。愈來愈多的大學資訊科系把 C++ 開在大一課程，這雖然說明 C++ 是多麼地重要，可也苦了資訊新兵們。

其實「難學」的最大癥結在於，很難得有一本書，能夠一針見血地指出多型與虛擬的重要性；在我們粗具語法基礎之後，直接把我們導引到最核心最重要的思想， 並且在建立這個思想的過程中，提供足夠的必要基礎。

#### 困難度之一

「C++ 是個難學易用的語言」，這句話相信很多人心有戚戚。C++ 的學習難度， 一在於語言本身太多的「幕」，一在於 "paradigm shift" （思考模式的移轉）。

傳統循序語言如 C, Pascal, Basic, Fortran...，除了模樣看起來稍有不同，基本上都是函式 call 來 call 去，大同小異，很容易掌握。你想做的動作，在 code 中都看得一清二楚。你所看不到的，犖犖大者也不過就是編譯器為你的函式加上用以

#### 342

附錄B C++ 的沉迷與愛戀

處理堆疊的一小段碼（prologue 和 epilogue），這一小段碼基本上做的是雜務， 你沒看到也沒有關係（更好），並不影響你對程式邏輯的思考。

C++ 不一樣，C++ 有太多和程式邏輯息息相關的動作是編譯器為我們加上去的。換句話說 C++ 編譯器為我們「加碼」。如果不識清這一節，學習C++ 有如霧裡看花，霧非霧，花非花。

編譯器為我們的 C++ 程式加了什麼碼呢？很多！物件誕生時 ctor 會被喚起，物件死亡時 dtor 會被喚起，這都是加碼的結果。ctor 中設定 vtpr 和 vtbl，這也是加碼的結果。new 單一物件時會產生 memory block cookie，new 物件陣列時會產生一個內部結構記錄著 object size 和 class ctor...，這也都是布幕後的工作。可以說，程式碼中看不到而卻必須完成的所有與程式邏輯有關的動作，統統都是 C++ 編譯器加碼後的結果。

當「繼承」發生，整個情況變得稍微複雜起來。「多重繼承」又更複雜一些，「虛擬繼承」再更複雜一些。

這些布幕後的主題，統可歸類為所謂的 C++ object model（物件模型）。如果不知道這些底層機制，你就只能夠把 make destructors virtual in base classes

（*Effective C++*, item14）或 never treat arrays polymorphically （*More Effective C++*, item 3）這類規則硬背下來，卻不明白它的道理。

用一樣東西，卻不明白它的道理，林語堂如是說：『不高明』。只知道 how，不知道 why，侯捷如是說：『不高明』。

#### 困難度之二

C++ 的第二個學習難度在於 "paradigm shift"（思考模式的移轉）。別說自己設計classes 了，光使用別人的 classes，就都是一種思考模式和行為模式的移轉。MFC 或 OWL 或VCL 的使用者必然甚能夠領略並體會我的意思。

#### 343

多型與虛擬（*Polymorphism in C++*）

使用所謂的 application framework（一種大型的、凝聚性強的、有著物件導向公共基礎建設的 class library），你的碼和 framework 之間究竟是怎樣的關係呢？ framework 提供的一大堆可改寫的虛擬函式的意義與價值究竟在哪裡呢？為什麼framework 所設計的種種美好性質以及各式各樣的演算法竟然可以施行於我們自己設計的 class types 身上呢？framework 被設計時，並不知道我們的存在呀！

這正是物件導向中的多型（polymorphism）的威力。

稍早所說的 C++ 物件模型，偏屬程式設計的低層面；這裡所說的思考模式移轉， 則是程式設計的高層面。能夠把新思維模式的威力發揮得最淋漓盡致的，當推物件導向的 polymorphism（多型）和 generalization（泛型）。如果你沒有使用這兩項特性，等於入 C++ 寶山而空手返。

#### 反覆焠鍊， 循環震盪

想像 C++ 是一把用來解決程式問題的刀，要它堅軔，要它鋒利，就必須經過多次的回火，在高熱和驟冷之間焠鍊。

初學 C++ 語法（syntax）之後，你應該儘快嘗試體驗 polymorphism （大致而言也就是虛擬函式的運用）。等到對 OOP 的精神有了大局掌控的能力，但對 C++ 的許多小細節不甚清楚，就是回到C++ 物件模型焠鍊的時機。

成長，是在高階（polymorphism）和低階（object model）之間反覆震盪，才能夠震盪到更高的位階，而不是平平庸庸於中階（C++ syntax）的一灘死水。

#### 不要沉淪於 C++ syntax

100 個人跟我說他懂 C++/OOP，只有 10% 不到可以讓我認為他沒有胡吹大氣。太多的人，上嘛上不到 polymorphism，下嘛又下不到object model。就這樣不上不下地卡在 C++ 語法層面。大一學了 C++，到大四快畢業了，連 virtual functions 是怎麼回事都期期艾艾支支吾吾說不出個道理。

#### 344

附錄B C++ 的沉迷與愛戀

有時候我覺得，太苛責同學也於心不忍，因為同學們事實上處於一種無知的狀態， 既不知道 C++/OOP 該怎麼學，也不知道哪些書可以教他們正確地學。所以，苛責同學，勿寧求全老師。

眾所週知，大學教授泰半是動口不動手，普遍的心態是「論文第一，升等為要。程式語言？哎，末流！」。「末流」課程通常由教授們輪流教，誰倒楣誰來教； 於是就有「下學期要教 C++ 語言了，這學期寒（暑）假趕快去要本書來惡補」的情況發生。偏偏程式語言這東西，只動口不管用，一定要動手，而且要常動手。如果老師自己沒有摸到C++/OOP 的精神，學生又能學到什麼？

有些學校資訊系並不教特定的程式語言，老師們的態度是「語言是一種自己學就好了的東西嘛，拿到大學殿堂來，哎，不入流」！於是應該為學生打好實務基礎的時間，卻拿來天馬行空騰雲駕霧，大談抽象意念。飽讀經書的老師們可能忽略了，技術底子不夠的學生最需要的不是形而上的道，而是形而下的器。

我們是先能夠欣賞具象畫，還是先能夠欣賞抽象畫？我們不都是先對著畢卡索的畫大嘆「這是什麼東西」，直到自己的藝術涵養夠豐富了、人生閱練更飽滿了、能夠舉一隅以三隅反了、能夠接觸類旁通左右逢源了，才轉而開始欣賞或甚至進入畢卡索的抽象意境嗎？

老師們各有專長，要老師們來教非彼專長的大班課、基礎課，我又覺得似乎也太為難了。那麼，苛責老師，不如責怪學校當局。如果學校當局能夠聘請經驗十足又有教學熱誠的工程師來教這類實務學科，不是三方皆大歡喜嗎？別說什麼制度僵化啦、難以突破啦，大學是高度自治區，禮聘幾位兼任老師，不全都是系上的權責範圍內嗎？

學子們在課程上學不到東西，就只好閉門自修。但是，循序性（sequential）語言尚有自修學會的可能，物件導向語言嘛，以大學生的程度來講，我認為自修實在困難，通常修出個四不像、半瓶水。現實所見，比比皆是。

#### 345

多型與虛擬（*Polymorphism in C++*）

#### 不要沉迷於 C++ semantics 和 C++ object model

對於底層知識有濃厚興趣的朋友，下探到 object model 領域，一定會非常開心地在 object size、object layout、vptr/vtbl、以及許多佈幕後的技術之間玩將起來。瞭解這些東西當然是好的，但是由於一探究竟得其奧秘的快感與成就感，使得一些朋友們在這個層面裡「玩」起來了；小地方玩得很精，玩得不亦樂乎，玩得忽略了 C++/OOP 的最終目標。

最終目標是 polymorphism！

我要說，在 C++ syntax 以及相對低階的 C++ semantics 或甚至更低階的 C++ object model 裡，不要玩得太過火。過猶不及，會傷身的。C++ 經典名書內附的習題，有些在我看來頗有玩得過火的味道。

Programming 應該是一種天馬行空的想像力與創意的組合；如果你能夠自己想題目，譬如說實作一個天體運行的 class 體系、或是實作一個生物分類（界門綱目科屬種）體系，不是很有趣嗎？準備資料的過程中，查查百科全書，你也因此查到了太陽系九大行星的幾何資料，哈雷慧星的軌道週期，或是黑面琵鷺的「界門綱目科屬種」英文名稱，這難道不比鑽研那些純粹頭腦體操的題目有趣嗎？

固然，在科學與工程的領域裡，無技術無以為立，但別把自己弄得過於僵化，過於匠氣。僵化與匠氣是我們教育體系的最大沉痾。到了高專層次，敗象顯露無遺。

#### 名書推薦

如果沒有介紹幾本好書，我就是為德不卒。

C++ 相關書籍，如天上繁星，如過江之鯽。廣博如四庫全書者有之（*The C++ Programming Language*、*C++ Primer*），深奧如重山複水者有之（*Inside The C++ Object Model*），獨沽一味者有之（*C++ Programming Style*、*More Effective C++*），獨樹一幟者有之（*The Design and Evolution of C++*），另闢蹊徑者亦有之（*STL*

#### 346

附錄B C++ 的沉迷與愛戀

*tutorial Reference Guide*）。

以下是我認為你應該要擁有的書籍。有趣的是，我才在自己班上做了一個調查（我教的是物件導向 Windows 程式設計，學生應該要有良好的 C++/OO 基礎），擁有以下 1~5 本書的人舉手。舉手人數都很少，而且老是那幾位（最高記錄是擁有四本）。這讓我感覺，強者恆強，弱者恆弱。悲夫！

1. *C++ Primer* (3/e), Lippman/A.W./1998
2. *The C++ Programming Language* (3/e), Bjarne/A.W./1997

這兩本書是 C++ 經典百科。就內容水平而言，同為瑜亮。普遍的印象是，第一本較易接受，第二本澀味稍重。第二本書作者 Bjarne 是 C++ 語言的創造者，所以有其權威性。我認識的多位 C++/OOP 高手，都是兩書齊具。

1. *Inside The C++ Object Model,* Lippman/A.W./1996

此書講解 C++ object model，上窮碧落下黃泉。內容很好，層次也高，可惜原文書大大小小的文字錯誤繁如晨星，閱讀時需小心。

1. *Effective C++,* Meyers/A.W./1992
2. *More Effective C++,* Meyers/A.W./1996

這兩本書以條列方式講解 C++ programming 的重要觀念，使你的程式更穩健更有效率。書中許多觀念涉及 C++ object model，與 **3.** 書混合看，如魚得水。

1. *Polymorphism in C++* 多型與虛擬 侯俊傑/松崗/1998

此書在語法粗具的基礎上，直接把讀者導引到物件導向最核心最重要的觀念， 並且在建立這個觀念的過程中，提供足夠的必要基礎。

這裡只列出一本中文書，是因為這方面的中文書我看得少，英文書看得多。「恐有遺珠之憾」這類「八方得體」的話，還是說一下好了 。

注意，這些都只是強本固元用來紮基礎的書籍而已，要觀摩大型程式經驗，還有

#### 347

多型與虛擬（*Polymorphism in C++*）

諸如 *Large Scale C++ Software Design*（John Lakos/A.W./1996）可以閱讀。

OO 的世界，不止OOP，還有 OOA/OOD，那又是一缸子的學問和一缸子的書。 

#### 348