

C++ 設計新思維

泛型編程與設計範式之應用

Modern C++ Design

Generic Programming
and Design Patterns Applied

Andrei Alexandrescu 著

侯捷 / 於春景 合譯

— |

| —

— |

| —

前衛的意義

侯捷譯序

一般人對 C++ templates 的粗淺印象，大約停留在「容器 (containers)」的製作上。稍有研究則會發現，templates 衍生出來的 C++ Generic Programming (泛型編程) 技術，在 C++ 標準程式庫中已經遍地開花結果。以 STL 為重要骨幹的 C++ 標準程式庫，將 templates 廣泛運用於容器 (containers)、演算法 (algorithms)、仿函式 (functors)、配接器 (adapters)、配置器 (allocators)、迭代器 (iterators) 上頭，無處不在，無役不與，乃至於原有的 class-based iostream 都被改寫為 template-based iostream。

徹底研究過 STL 源碼 (SGI 版本) 的我，原以為從此所有 C++ templates 技法都將不出我的理解與經驗之外。但是《*Modern C++ Design*》在在打破了我的想法與自信。這本書所談的 template 技巧，以及據以實作出來的 Loki 程式庫，讓我瞠目結舌，陷入沉思…與…呃…恍惚☹。

本書分為兩大部分。首先 (第一篇) 是基礎技術的討論，包括 template template parameters (請別懷疑，我並沒有多寫一個字)、policies-based design、compile-time programming、recursive templates, typelists。每一項技術都讓人聞所未聞，見所未見。

第二部分 (第二篇) 是 Loki 程式庫的產品設計與實作，包括 Small-Object Allocation¹, Generalization Functors, Singleton, Smart Pointers, Object Factories, Abstract Factory, Visitor, Multimethods。對設計範式² (design patterns) 稍有涉獵的讀者馬上可以看出，這一部分主題都是知名的範式。換言之，作者 Andrei 嘗試以 templates-based, policies-based 手法，運用第一篇完成的基礎建設，將上述範式具體實現出來，使任何人能夠輕鬆地在 Loki 程式庫的基礎上，享受設計範式所帶來的優雅架構。

¹ Small-Object Allocation 屬於底層服務的「無名英雄」，故而在章節組織上仍被劃入第一篇。

² patterns 一詞，臺灣大陸兩地共出現三種譯法：(1) 範式 (2) 樣式 (3) 模式。我個人最喜歡「範式」，足以說明 patterns 的「典範」意味。因此本書以「範式」稱 patterns。顧及大陸術語習慣，簡體版以「模式」稱 patterns。本書所有 patterns 都保留英文名稱並以特殊字型標示，例如 *Object Factories*, *Visitors*...

設計範式 (Design Patterns) 究竟能不能被做成「易開罐」讓人隨時隨地喝上一口，增強體力？顯然範式社群 (patterns community) 中有些人不這麼認為 — 見稍後 Scott Meyers 序文描述。我以為，論斷事物不由本質，儘好口舌之辯的人，不足取也。Andrei 所拓展的天地，Loki 所達到的高度，不會因為它叫什麼名字而有差異，也不會因為任何人加諸它身上的什麼文字包裝或批評或解釋或討好，而有不同。它，已經在那兒了。

本書涉足無人履踏之境，不但將 C++ templates 和 generics programming 技術做了史無前例的推進，又與 design patterns 達成巧妙的結合。本書所談的技術，所完成的實際產品，究竟是狂熱激進的象牙塔鑽研？抑或高度實用的嶄新設計思維？做為一個技術先鋒，Loki 的現實價值與未來，唯賴你的判斷，和時間的篩選。

然而我一定要多說一句，算是對「唯實用論」的朋友們一些忠告。由來技術的推演，並不只是問一句「它有用嗎」或「它現在有用嗎」可以論斷價值的。牛頓發表萬有引力公式，並不知道三百年後人們用來計算軌道、登陸月球。即使在講述「STL 運用」的課堂上，都還有人覺得太前衛，期盼卻焦躁不安，遑論「STL 設計思維和內部實作」這種課，遑論 Loki 這般前衛技術。很多人的焦慮是：我這麼學這麼做這麼寫這麼用，同儕大概看不懂吧，大概跟不上吧。此固值得關注，但個人的成長千萬別被群體的慣性絆住腳步³。我們曾經鄙夷的別人的「無謂」前衛，可能只因我們故步自封，陷自己於一成不變的行為模式；或因為我們只看到自家井口的天空。當然，也可能某些前衛思想和技術，確實超越了龐大笨重遲緩的現實世界的接受度。你有選擇。做為一位理性思考者，身在單純可愛的技術圈內，請不要妄評先鋒，因為他實在站在遠比你（我）高得太多的山巔上。不當的言語和文字並不能為你（我）推砌樓台使與同高。

深度 + 廣度，古典 + 前衛，理論 + 應用，實驗室 + 工廠，才能構築一個不斷進步的世界。

侯捷 2003/01/08 於 臺灣·新竹

jjhou@ccca.nctu.edu.tw

<http://www.jjhou.com> (繁)

<http://jjhou.csdn.net> (簡)

P.S. 本書譯稿由我和於春景先生共同完成。春景負責初譯，我負責其餘一切。春景技術到位，譯筆極好，初譯稿便有極佳品質，減輕我的許多負擔。循此以往必成為第一流 IT 技術譯家。我很高興和他共同完成這部作品。本書由我定稿，責任在我身上，勘誤表由我負責。本書同步發行繁體版和簡體版；基於兩岸計算機術語的差異，簡體版由春景負責必要轉換。

P.S. 本書初譯稿前三章，邱銘彰先生出力甚多，特此致謝。

P.S. STL, Boost, Loki, ACE...等程式庫的發展，為 C++ 領域挹注了極大活力和競爭力，也使泛型技術在 C++ 領域有極耀眼的發展。這是 C++ 社群近年來最令人興奮的事。如果你在 C++ 環境下工作，也許這值得你密切關注。

³ 從萬有引力觀之，微小粒子難逃巨大質量團的吸滯（除非小粒子擁有高能量）。映照人生，這或許是一種悲哀。不過總會有那麼一些高能粒子逸脫出來 — 值得我們轉悲為喜，懷抱希望。

譯序

by 於春景

三年前，當我第一次接觸 `template` 的時候，我認為那只不過是一位「戴上了新帽子」的舊朋友：在熟悉的 `class` 或 `function` 的頭頂上，你只需扣上那頂古怪的尖角帽 — 添上一句 `template <class T1, ...>` — 然後將熟悉的資料型別替換為 `T1, T2...`，一個 `template` 就搖身而至！嗯，我得承認，戴上了帽子的 `template` 的確是個出色的程式碼生成器，好似具有滋生程式碼「魔法」的 `macro`，但畢竟還不能成其為「戴上了帽子的魔術師」。

後來，我開始學習 GP（generic programming）和 STL（standard template library）。我不禁啞然。在 GP 領域，`template` 竟扮演著如此重要的角色，以至於成為 C++ GP 的基石。在 GP 最重要的商業實作品 STL 中，`template` 向我們展示其無與倫比的功效。回想起自己當初對 `template` 的比喻，啞然失笑之餘，我驚嘆 `template` 在 GP 和 STL 中將自己的能力發揮到了「極致」。

然而，這一次，《*Modern C++ Design*》又讓我默然。我不得不承認，Andrei Alexandrescu 的這部著作（及其 Loki library）帶給我的，是對 `template` 和 GP 技術又一次震撼般的認識！

這種震撼感受，源於技術層面，觸及設計範疇。`template` 的技術核心在於編譯期動態機制。和運行期多型（runtime polymorphism）相比，這種動態機制提供的編譯期多型特性，給了程式運行期無可比擬的效率優勢。本書中，Andrei 對 `template` 編譯期動態機制的運用可謂淋漓盡致。以 `template` 打造而成的 `typelist`、`small-object allocator`、`smart pointer` 不僅具有強大功能，而且體現了無限的擴充性；將 `template` 技術大膽地運用到 design patterns 中，更為 design patterns 的實現提供了靈活、可復用的泛型組件。

在這些令人目眩的實作技術之後，蘊涵著 Andrei 倡導並使用的 policy-based 設計技術。利用這一耳目一新的設計思想，用戶程式碼不再僅僅是技術實作上的細節，你甚至可以讓程式碼在編譯期作出設計方案的選擇！這種將「設計概念」和「`template` 編譯期多型」結合起來的設計思維，將 C++ 程式設計技術提昇到了新的高度，足以振聾發聵。

也許只有時間才能證實，Andrei 為我們展示的，或許是 C++ 程式設計技術的一次革命；在 C++ 的歷史上，《*Modern C++ Design*》將是一部重要的著作。Andrei 對 template、generic programming 技術、以及 template 在 design patterns 中的運用等課題所作的深入闡釋和大膽實踐，可謂前無古人。

遺憾的是，在當今主流 C++ 編譯器上，Loki 很難順利通過編譯。例如面對 "template template parameter" 的「難題」，很多編譯器毫無招架之力。應該說，這並不是 Andrei 和 Loki 過於超前，而是 C++ 編譯器應當迅速跟進。這意味作為 C++ 程式員的你，也應當迅速跟進！

作為 C++ 程式員的我，已從此書獲益良多。這是一部讓我在翻譯過程中毫不感到倦怠的巨著。它時時引發我思索，給我以啓迪，並讓我重拾研習 C++ 的快樂。這得感謝 Andrei。在這樣一部講述高級技術的專著中，Andrei 的講解細緻深入，條理得當，語言卻又極為簡明清晰。我期望中文版保留了這一特色。

除了作者之外，在翻譯本書的過程中，給我更多教益的還有侯捷先生。我的初譯稿便是在先生不斷的鼓勵和指導下完成的。先生謙和的人品和技術上的深邃見解，令我欽佩和謹記。還要感謝周筠女士，我的每一本譯作都離不開您的參與和悉心幫助，本書也不例外。最後，感謝所有關心我的朋友，願你們也像我一樣喜愛這本書。

於春景 2002/12/15

深圳蛇口，海上世界

billyu@lostmouse.net

目錄

Contents

譯序 by 侯捷	i
譯序 by 於春景	iii
目錄	v
序言 by Scott Meyers	xi
序言 by John Vlissides	xv
前言	xvii
致謝	xxi
第一篇 技術 (Techniques)	1
第 1 章 以 Policy 為基礎的 Class 設計 (Policy-Based Class Design)	3
1.1 軟體設計的多樣性 (Multiplicity)	3
1.2 全功能型 (Do-It-All) 介面的失敗	4
1.3 多重繼承 (Multiple Inheritance) 是救世主？	5
1.4 Templates 帶來曙光	6
1.5 Policies 和 Policy Classes	7
1.6 更豐富的 Policies	12
1.7 Policy Classes 的解構式 (Destructors)	12
1.8 透過不完全具現化 (Incomplete Instantiation) 而獲得的選擇性機能 (Optional Functionality)	13
1.9 結合 Policy Classes	14
1.10 以 Policy Classes 訂製結構	16
1.11 Policies 的相容性	17
1.12 將一個 Class 分解為一堆 Policies	19
1.13 摘要	20

第 2 章	技術 (Techniques)	23
2.1	編譯期 (Compile-Time) Assertions	23
2.2	Partial Template Specialization (模板偏特化)	26
2.3	區域類別 (Local Classes)	28
2.4	常整數映射為型別 (Mapping Integral Constants to Types)	29
2.5	型別對型別的映射 (Type-to-Type Mapping)	31
2.6	型別選擇 (Type Selection)	33
2.7	編譯期間偵測可轉換性 (Convertibility) 和繼承性 (Inheritance)	34
2.8	type_info 的一個外覆類別 (Wrapper)	37
2.9	NullType 和 EmptyType	39
2.10	Type Traits	40
2.11	摘要	46
第 3 章	Typelists	49
3.1	Typelists 的必要性	49
3.2	定義 Typelists	51
3.3	將 Typelist 的生成線性化 (linearizing)	52
3.4	計算長度	53
3.5	間奏曲	54
3.6	索引式存取 (Indexed Access)	55
3.7	搜尋 Typelists	56
3.8	附加元素至 Typelists	57
3.9	移除 Typelist 中的某個元素	58
3.10	移除重複元素 (Erasing Duplicates)	59
3.11	取代 Typelist 中的某個元素	60
3.12	為 Typelists 局部更換次序 (Partially Ordering)	61
3.13	運用 Typelists 自動產生 Classes	64
3.14	摘要	74
3.15	Typelist 要點概覽	75
第 4 章	小型物件配置技術 (Small-Object Allocation)	77
4.1	預設的 Free Store 配置器	78
4.2	記憶體配置器的工作方式	78
4.3	小型物件配置器 (Small-Object Allocator)	80
4.4	Chunks (大塊記憶體)	81
4.5	大小一致 (Fixed-Size) 的配置器	84
4.6	SmallObjAllocator Class	87
4.7	帽子下的戲法	89

4.8	簡單，複雜，終究還是簡單	92
4.9	使用細節	93
4.10	摘要	94
4.11	小型物件配置器 (Small-Object Allocator) 要點概覽	94
第二篇	組件 (Components)	97
第 5 章	泛化仿函式 (Generalized Functors)	99
5.1	Command 設計範式	100
5.2	真實世界中的 Command	102
5.3	C++ 中的可呼叫體 (Callable Entities)	103
5.4	Functor Class Template 骨幹	104
5.5	實現「轉發式」(Forwarding) Functor::operator()	108
5.6	處理仿函式	110
5.7	做一個，送一個	112
5.8	引數 (Argument) 和回返型別 (Return Type) 的轉換	114
5.9	處理 pointer to member function (成員函式指標)	115
5.10	繫結 (Binding)	119
5.11	將請求串接起來 (Chaining Requests)	122
5.12	現實世界中的問題之 1：轉發式函式的成本	122
5.13	現實世界中的問題之 2：Heap 配置	124
5.14	透過 Functor 實現 Undo 和 Redo	125
5.15	摘要	126
5.16	Functor 要點概覽	126
第 6 章	Singletons (單件) 實作技術	129
6.1	靜態資料 + 靜態函式 != Singleton	130
6.2	用以支援 Singleton 的一些 C++ 基本手法	131
6.3	實施「Singleton 的唯一性」	132
6.4	摧毀 Singleton	133
6.5	Dead (失效的) Reference 問題	135
6.6	解決 Dead Reference 問題 (I)：Phoenix Singleton	137
6.7	解決 Dead Reference 問題 (II)：帶壽命的 Singletons	139
6.8	實現「帶壽命的 Singletons」	142
6.9	生活在多緒世界	145
6.10	將一切組裝起來	148
6.11	使用 SingletonHolder	153
6.12	摘要	155
6.13	SingletonHolder Class Template 要點概覽	155

第 7 章	Smart Pointers (精靈指標)	157
7.1	Smart Pointers 基礎	157
7.2	交易	158
7.3	Smart Pointers 的儲存	160
7.4	Smart Pointer 的成員函式	161
7.5	擁有權 (Ownership) 管理策略	163
7.6	Address-of (取址) 運算子	170
7.7	隱式轉換 (Implicit Conversion) 至原始指標型別	171
7.8	相等性 (Equality) 和不等性 (Inequality)	173
7.9	次序比較 (Ordering Comparisons)	178
7.10	檢測及錯誤報告 (Checking and Error Reporting)	181
7.11	Smart Pointers to const 和 const Smart Pointers	182
7.12	Arrays	183
7.13	Smart Pointers 和多緒 (Multithreading)	184
7.14	將一切組裝起來	187
7.15	摘要	194
7.16	SmartPtr 要點概覽	194
第 8 章	Object Factories (物件工廠)	197
8.1	為什麼需要 Object Factories	198
8.2	Object Factories in C++ : Classes 和 Objects	200
8.3	實現一個 Object Factory	201
8.4	型別標識符號 (Type Identifiers)	206
8.5	泛化 (Generalization)	207
8.6	細節瑣務	210
8.7	Clone Factories (複製工廠、翻製工廠、克隆工廠)	211
8.8	透過其他泛型組件來使用 Object Factories	215
8.9	摘要	216
8.10	Factory Class Template 要點概覽	216
8.11	CloneFactory Class Template 要點概覽	217
第 9 章	Abstract Factory (抽象工廠)	219
9.1	Abstract Factory 扮演的架構角色 (Architectural role)	219
9.2	一個泛化的 Abstract Factory 介面	223
9.3	實作出 AbstractFactory	226
9.4	一個 Prototype-Based Abstract Factory 實作品	228
9.5	摘要	233
9.6	AbstractFactory 和 ConcreteFactory 要點概覽	233

第 10 章	Visitor (訪問者、視察者)	235
10.1	Visitor 基本原理	235
10.2	重載 (Overloading) : Catch-All 函式	242
10.3	一份更加精鍊的實作品 : Acyclic Visitor	243
10.4	Visitor 之泛型實作	248
10.5	再論 "Cyclic" Visitor	255
10.6	變化手段	258
10.7	摘要	260
10.8	Visitor 泛型組件要點概覽	261
第 11 章	Multimethods	263
11.1	什麼是 Multimethods?	264
11.2	何時需要 Multimethods ?	264
11.3	Double Switch-on-Type : 暴力法	265
11.4	將暴力法自動化	268
11.5	暴力式 Dispatcher 的對稱性	273
11.6	對數型 (Logarithmic) Double Dispatcher	276
11.7	FnDispatcher 和對稱性	282
11.8	Double Dispatch (雙重分派) 至仿函式 (Functors)	282
11.9	引數的轉型 : static_cast 或 dynamic_cast ?	285
11.10	常數時間的 Multimethods : 原始速度 (Raw Speed)	290
11.11	將 BasicDispatcher 和 BasicFastDispatcher 當做 Policies	293
11.12	展望	294
11.13	摘要	296
11.14	Double Dispatcher 要點概覽	297
附錄	一個超迷你的多緒程式庫 (A Minimalist Multithreading Library)	301
A.1	多緒的反思	302
A.2	Loki 的作法	303
A.3	整數型別上的原子操作 (Atomic Operations)	303
A.4	Mutexes (互斥器)	305
A.5	物件導向編程中的鎖定語意 (Locking Semantics)	306
A.6	可有可無的 (Optional) volatile 飾詞	308
A.7	Semaphores, Events 和其他好東西	309
A.8	摘要	309
	參考書目 (Bibliography)	311
	索引 (Index)	313

序言

by Scott Meyers

1991 年，我寫下《*Effective C++*》第一版。那本書幾乎沒有討論 `template`，因為它剛剛才被加入語言之中，我對它幾乎一無所知。爲了書中包含的一點點 `template` 程式碼，我曾透過電子郵件請別人驗證，因為我手上的編譯器都沒有提供對 `template` 的支援。

1995 年，我寫下《*More Effective C++*》。又一次，我幾乎沒有講述 `template`。這一次阻止我的，既不是對 `template` 知識的缺乏（在那本書的初稿中，我曾打算以一整章講述 `template`），也不是我的編譯器在這方面有所缺陷。真正的理由是我擔心，C++ 社群對 `template` 的理解即將經歷一次巨大的變化，我對它所說的任何事情，也許很快就會被認爲是陳舊的、膚淺的、甚至完全錯誤的。

我的擔心出於兩個原因。第一個原因和 John Barton 及 Lee Nackman 在 *C++ Report* 1995 年 1 月的一篇專欄文章有關。這篇文章討論的是：如何經由 `template` 執行型別安全的維度分析，同時做到運行期零成本。我自己也曾在這個問題上花了不少時間，而且我知道很多人也在尋找解答，但沒有人成功。Barton 和 Nackman 的創新解法讓我認識到，`template` 在太多的地方有用，不只是用來生成「T 容器」。

以下是他們的設計示例。這段程式碼對兩個物理量作乘法運算，而這兩個物理量具有任意維數的型別：

```
template<int m1, int l1, int t1, int m2, int l2, int t2>
Physical<m1+m2, l1+l2, t1+t2> operator*(Physical<m1, l1, t1> lhs,
                                         Physical<m2, l2, t2> rhs)
{
    return Physical<m1+m2, l1+l2, t1+t2>::unit*lhs.value()*rhs.value();
}
```

即使我沒有說明這段程式碼，有一點很清楚：這個 `function template` 有 6 個參數，但沒有一個是型別！`template` 的這種用法對我來說是頭一次見到，我確實有點目眩。

不久之後，我開始閱讀 STL。在 Alexander Stepanov 精巧的程式庫設計中，容器（containers）對演算法（algorithms）一無所知，演算法亦對容器一無所知；迭代器（iterator）的行爲像指標

（但卻有可能是物件）；容器和演算法像接受函式指標一樣地接受函式物件（function object）；用戶可以擴充程式庫，但不必繼承其中任何 base class，也不必重新定義任何 virtual function。這一切都讓我覺得——就像當初我看到 Barton 和 Nackman 的成果那樣——我對 template 幾乎一無所知。

所以，在《*More Effective C++*》中，我幾乎沒有提到 template。我還能怎樣？我對 template 的認識還停留在「T 容器」階段，而 Barton、Nackman、Stepanov，還有其他人都已證明，那種用法只不過剛剛觸到 template 的皮毛而已。

1998 年，Andrei Alexandrescu 和我開始了電子郵件交流；不久之後我意識到，我得再次修正我對 template 的認識。Barton、Nackman、Stepanov 讓我感到震驚的是：template 可以「做什麼」；而 Andrei 的成果最初給我的印象是：template「如何」完成它所做的事情。

在 Andrei 協助推廣的很多工具中，有這樣一個最簡單的東西；當我向人們介紹 Andrei 的工作時，我也一直將這當作一個例子。這就是 CTAAssert template，作用和 assert 巨集類似，但施行於「可在編譯期間被核定（evaluated）」的條件句中。以下便是 CTAAssert template：

```
template<bool> struct CTAAssert;  
template<> struct CTAAssert<true> {};
```

僅此而已。請注意，這個 CTAAssert 從來沒被定義。請注意，它有一個針對 true（而非 false）的特化體。在這個設計中，「缺少」的東西至少和提供的東西一樣重要。它讓你以一種新的方式看待 template，因為大部份「源碼」被刻意遺漏了。和我們大多數人以往的想法相比，這是一種極為不同的思維方式。（本書之中 Andrei 討論了一個更為複雜的 CompileTimeChecker template，而不是 CTAAssert）

後來，Andrei 將注意力轉移到 idioms（慣用手法）和 design patterns（設計範式，尤其是 GoF⁴ 範式）的開發上，提供了 template-based 實作品。這導致他和範式社群的一場短暫衝突，因為後者信奉一條基本原則：範式（patterns）無法以程式碼表述。一旦弄清 Andrei 是在致力於使範式的實現得以自動化，而非試圖將範式本身以程式碼來表述，反對聲音也就消失了。我很高興看到，Andrei 和 GoF 之一 John Vlissides 達成了合作；在 *C++ Report* 上，他們就 Andrei 的研究成果推出了兩個專欄。

在開發 templates 用以產生 idioms（慣用手法）和 design patterns（設計範式）實作品時，所有實作者需要面對的各種設計抉擇，Andrei 也都必須面對。程式碼應該做到多緒安全嗎？輔助儲存器應當來自 heap 或是 stack 抑或 static pool？提領 smart pointers 之前是否應該針對 null 進行檢查？程式關閉時如果 Singleton 的解構式試圖使用另一個已被摧毀的 Singleton，會發生什

⁴ "GoF" 意味 "Gang of Four"，指的是 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides，他們四人是範式（patterns）權威書籍《*Design Patterns: Elements of Resusable Object-Oriented Software*》（Addison-Wesley, 1995）的作者。

麼事？Andrei 的目標是：為用戶提供所有可能的設計選擇，但不強制任何東西。

Andrei 的方案是：將這種選擇以 *policy classes* 的形式封裝起來，允許客戶將 *policy classes* 當作 *template* 參數傳遞，同時為這種 *classes* 提供合理的預設值，使大多數客戶可以忽略這些參數。其結果令人瞠目結舌。例如本書的 *SmartPointer* *template* 只有 4 個 *policy* 參數，但它可以生成 300 多個不同的 *smart pointer* 型別，每一個都具有不同的行為特徵！滿足於 *smart pointer* 預設行為的程式員可以忽略 *policy* 參數，只需指定「*smart pointer* 所指物件」之型別，從而獲得精心製作的 *smart pointer class* 所帶來的好處。嗯，不費吹灰之力。

最後要說的是，本書敘述了三個不同的技術故事，每一個都引人入勝。首先，它就 C++ *template* 的能力和靈活性提供了新的見解 — 如果第三章的 *typelists* 沒有讓你感到振奮，一定是因為你暮氣沉沉）。第二，它標示了一個正交維度（*orthogonal dimensions*），告訴我們 *idioms* 和 *patterns* 的實現可以不同。對 *templates* 設計者和 *patterns* 實作者而言，這是十分重要的資訊，但是在大多數講述 *idioms* 或 *patterns* 的文獻中你都找不到這方面的研究。第三，*Loki*（本書介紹的 *template library*）源碼可以免費下載，所以你可以研究 Andrei 討論的 *idioms* 和 *patterns* 所對應的 *template* 實際作品。這些程式碼可以嚴格檢測你的編譯器對 *templates* 的支援程度，此外當你開始自己的 *template* 設計時，它還是無價的起點。當然，直接使用 *Loki* 也是完全可以的（而且完全合法），我知道 Andrei 也願意你運用他的成果。

就我所知，*template* 的世界還在變化，速度之快就像 1995 年我迴避寫它的時候一樣。從發展的速度來看，我可能永遠不會寫有關 *template* 的技術書籍。幸運的是一些人比我勇敢，Andrei 就是這樣一位先鋒。我想你會從此書得到很多收穫。我自己就得到了很多。

Scott Meyers
September 2000

序言

by John Vliissides

關於 C++，還有什麼沒有說到的？唔，很多，本書所談的一切幾乎都是。本書提供的是編程技術 — generic programming、template metaprogramming、OO programming、design patterns — 的融合。這些技術分開來可以有良好的理解，但對於它們之間的協作關係，我們才剛剛開始認識。這些協同作用為 C++ 打開了全新視野，而且不僅僅在編程方面，還在於軟體設計本身；對軟體分析和軟體架構來說，它也具有豐富的內涵。

Andrei 的泛型組件將抽象層次提昇到了新的高度，足以使 C++ 在各方面看起來像是一種設計規格（design specification）語言。但是，不同於專用的設計語言，你還保有 C++ 全部的表達性和對它的駕輕就熟。Andrei 向你展示如何根據設計思想 — singletons、visitors、proxies、abstract factories... — 來編寫程式。甚至你可以經由 template 參數改變實作選擇，而且幾乎沒有運行期開銷。你不必求助於新的開發工具，也不必學習晦澀難懂的方法學（methodology）。你需要的只是一個可靠的、新型的 C++ 編譯器，以及本書。

多年來，程式碼生成器（code generators）一直有類似承諾，但我自己的研究以及實踐經驗使我相信，最終，程式碼生成器無法匹敵。你會有「往返旅程（round-trip）」問題、「缺乏值得生成的程式碼」問題、「生成器不靈活」問題，「生出莫名其妙的程式碼」問題，當然還有「無法將自己的程式碼和該死的生成出來的程式碼整合在一起」的問題。這些問題中的任何一個都有可能成為絆腳石；而且，對大多數編程挑戰而言，這些絆腳石都使得「程式碼自動生成」不可能成為一種解決方案。

如果能獲得「程式碼自動生成」理論上的好處 — 快速、易開發、贅餘降低、錯誤更少 — 而又沒有它們的缺點，該有多好！這正是 Andrei 的作法所承諾的。在易於使用、可相互混合和匹配的 templates 中，泛型組件實現了出色的設計。它們完成的幾乎就是程式碼生成器的功能：產生供編譯器使用的規範程式碼（boilerplate code）。差別在於它們是在 C++ 之內（而非之外）完成這些功能。成果是「與應用程式碼的無縫整合」。同時你還是可以運用 C++ 語言的全部威力，對設計進行擴充、改寫、或者調整，從而符合你的需要。

無可否認，這裡的一些技術很複雜，因而難以領會，特別是第 3 章的 `template metaprogramming` 部份。但是一旦你掌握了它，你就奠定了泛型組件架構（`generic componentry`）的堅實基礎；後續章節中的各個泛型組件幾乎就是自己構造自己。事實上我認為第 3 章關於 `template metaprogramming` 的內容就值得本書價格，何況還有另外 10 個充滿見地、讓你獲益匪淺的章節。
"10" 其實是代表一個數量級。而我確信，你獲得的回報會比這個數量還多得多。

John Vlissides

IBM T.J. Watson Research

September 2000

前言

Preface

也許你正在書店裡捧著這本書，問自己該不該買下它。或者，你正在公司的圖書室裡，猶豫該不該花時間閱讀它。我知道你時間寶貴，所以我開門見山。如果你曾經問過自己：如何撰寫更高級的 C++ 程式？如何應付即使在很乾淨的設計中仍然像雪崩一樣發生的不相干細節？如何構建可復用組件，使得每次將這些組件應用到下一個程式時都無需對它們大動干戈？如果你曾這樣問過自己，那麼，本書正是爲你所寫。

想像這樣的情景。你剛從一次設計會議回來，帶著一些列印圖表，上面有你潦草寫下的註解。哦，物件之間傳遞的事件型別 (event type) 不再是 `char` 而是 `int` 了，於是你修改一行程式碼。指向 `Widget` 的 `smart pointers` 太慢了，得取消一些檢查措施，讓它們快一點，於是你又修改一行程式碼。另一個部門剛才添加 `Gadget class`，你的 *object factory* 必須支援它，於是你再次改動一行程式碼。

你修改了這個設計。編譯，連結，搞定。

且慢，場景有點問題，不是嗎？現實情形更可能是：你匆匆從會議中趕回來，因爲有一大堆工作要做。於是你開始地毯式搜索，並在程式碼上大動干戈：添加新的程式碼、引入臭蟲、消除臭蟲…，這就是程式員的生活不是嗎？本書也許不能保證你實現第一場景，但它朝著那個方向邁出堅實的一步。它對軟體設計師展示的 C++，宛如一種新語言。

傳統上，程式碼是軟體系統中最瑣碎、最複雜的環節。儘管歷史上出現了各種層次的編程語言，支持各種設計方法（譬如物件導向方法），但在藍圖和程式碼之間，總是橫亙著一條鴻溝。這是因爲，程式碼必須仔細關照具體實現和某些輔助性任務中極其細節的問題。因此，設計意圖往往被無盡的細節吞噬。

本書提供了一組可復用的設計產品 — 所謂「泛型組件」，以及設計這些組件所需要的技術。這些泛型組件爲用戶帶來的明顯好處，集中於程式庫方面，而處於更廣泛的系統架構空間中。本書提供的編程技術和實作品 (implementation) 所反映的任務和議題，傳統上落於設計範疇

之中，是編寫程式碼之前必須完成的東西。由於身處較高層次，泛型組件就有可能以一種不同尋常但簡潔、易於表達、易於維護的方式，將複雜的架構反映到程式碼中。

這裡結合了三個要素：設計範式 (design patterns)、泛型編程 (generic programming)、C++。結合這些要素後，我們獲得極高層次的可復用性，無論是橫向或縱向。從橫向空間來看，少量 library code 就可以實現組合性的、實質上具有無窮數量的結構和行為。從橫向空間來看，由於這些組件的通用性，它們可廣泛應用於各種程式中。

本書極大歸功於設計範式 (design patterns) — 面臨物件導向程式開發中的常見問題時，它是強有力的解決方案。設計範式是經過提煉的出色設計方法，對於很多情況下碰到的問題，它都是合理而可復用的解決方案。設計範式致力於提供深具啟發、易於表達和傳遞的設計詞彙。它們所描述的，除了問題 (problem) 之外，還有久經考驗的解法及其變化形式，以及選擇每一種方案所帶來的後果。設計範式超越了任何一種設計語言所能表達的東西 — 無論那種語言多麼高級。本書遵循並結合某些設計範式，提供的組件可以解決廣泛的具體問題。

泛型編程是一種典範 (paradigm)，專注於將型別 (type) 抽象化，形成功能需求方面的一個精細集合，並利用這些需求來實現演算法。由於演算法為其所操作的型別定義了嚴格、精細的介面，因此相同的演算法可以運用於廣泛的型別集 (a wide collection of types)。本書提供的實作品採取泛型編程技術，以最小代價獲得足以和手工精心編寫的程式碼相匹敵的專用性、高度簡潔和效率。

C++ 是本書使用的唯一工具。在本書中，你不會看到漂亮的視窗系統、複雜的網路程式庫或靈巧的日誌記錄 (logging) 機制。相反的，你會發現很多基礎組件；這些組件易於實現以上所有系統 (甚至更多)。C++ 具有實現這一切所需要的廣度，其底層的 C 記憶體模型保證了最原始效率 (raw performance)，對多型 (polymorphism) 的支援成就了物件導向技術，templates 則展現為一種令人難以置信的程式碼生成器。Templates 遍及本書所有程式碼，因為它們可以令用戶和程式庫之間保持最密切的協作。在遵循程式庫約束的基礎上，程式庫的用戶可以完全控制程式碼的生成方式。泛型組件庫的角色在於，它可以讓用戶指定的型別和行為，與泛型組件結合起來，形成合理的設計。由於所採技術之靜態特性，在結合和匹配相應組件時，產生的錯誤通常在編譯期便得以發現。

本書最明顯的意圖在於創建泛型組件，這些組件預先實現了設計模組，主要特點是靈活、通用、易用。泛型組件並不構成 framework。實際上它們採用的作法是互補性的；雖然 framework 定義了獨立的 classes，用來支援特定的物件模型，但泛型組件(s) 是輕量級設計工具，互相獨立，可自由組合和匹配。實現 frameworks 時泛型組件可帶來很大幫助。

本書讀者

本書預定的讀者主要分為兩類。第一類是富有經驗的 C++ 程式員，他們希望經由本書掌握最先進的程式庫編寫技術。本書提供了新而強大的 C++ 技術，這些技術具有驚人能力，有一些甚至令人匪夷所思。這些技術對於撰寫高級程式庫極有幫助。當然，希望更上層樓的中階 C++ 程式員也會發現本書十分有益，特別是如果他們願意付出一些毅力。本書雖然有時給出一些高難度的 C++ 程式碼，但都有詳盡說明。

本書預定的第二類讀者是繁忙的程式員，他們需要完成工作，但無法投入時間進行深入學習。他們可以快速略過最複雜的細節，把注意力放在如何使用本書提供的程式庫上。每一章都有一個導入說明，並以概覽 (Quick Facts) 結束。程式員們會發現這種安排方式為理解和使用本書組件提供了有益的參考。這些組件可以分開理解，它們功能強大但很安全，而且讓人樂於使用。

你需要扎實的 C++ 經驗，以及強烈的求知慾。你也需要對 templates 和 STL (Standard Template Library) 有一定的掌握。

如果你已經了解 design patterns (Gamma 等著, 1995)，那當然好，但並非必要。書中對於用到的 patterns 和 idioms (慣用手法) 都有詳細介紹。本書並不是 patterns 方面的專著，並不試圖完整闡述 patterns。由於 patterns 是程式庫設計者從實踐的角度提出的，所以即使那些曾經關注 patterns 的讀者也會發現，他們的視野如今有了更新 — 如果他們曾經受到束縛的話。

Loki

本書講述一個實際的 C++ 程式庫，稱為 Loki。Loki 是挪威神話中的智慧之神，同時也是一個淘氣鬼；作者希望，這個程式庫的創意和靈活會讓你想起那個有趣的挪威神話人物。程式庫中的所有元素都位於命名空間 (namespace) Loki 之內；此一名稱並未出現於書中範例程式上，因為那會為程式碼帶來非必要的縮排格式，並增加程式碼的數量。Loki 是免費的，你可以從 <http://www.awl.com/cseng/titles/0-201-70431-5> 下載它。

除了執行緒 (threading) 部分，Loki 完全以標準 C++ 寫成。唉，這也意味目前很多編譯器無法處理其中某些部份。我在 Metrowerks CodeWarrior Pro 6.0 和 Comeau C++ 4.2.38 上實作並測試 Loki，並且都在 Windows 平台上。KAI C++ 處理 Loki 程式碼好像也沒有問題。隨著供應商逐漸發行更新更好的編譯器，你將能夠運用 Loki 提供的所有功能。

本書提供的 Loki 程式碼和範例採用了一種很普及的寫碼標準，這一標準最早由 Herb Sutter 倡導。我相信你很快便能適應它。簡單地說：

- classes、functions、列舉型別 (enumerated type) 看起來像 likeThis。
(譯註：由於版面上的需要，中譯本的 functions 看起來像 likeThis)
- 變數和列舉元看起來像 likeThis。
- 成員變數看起來如 likeThis_。
- template 參數如果只可能是用戶自定型別，那麼它會被宣告為 class；如果還可能是基本型

別，那麼它會被宣告為 `typename`。

內容組織

本書由兩大篇組成：技術篇和組件篇。第一篇（1~4 章）講述的是，在泛型編程中 — 特別是在泛型組件的構造中 — 所運用的 C++ 技術。它展示了與 C++ 相關的大量功能和技術：policy-based 設計、partial template specialization、typelists、local classes 等等。你可以按部就班地閱讀本篇，然後回過頭來參考特定章節。

第二篇建立在第一篇的基礎上，實作出多個泛型組件。這些並非紙上談兵；他們是具有工業強度的組件，可應用於現實世界的應用程式中。C++ 開發者在日常工作中經常遇到的議題，例如 smart pointers、object factories、functor objects，在此都有深入的探討，並提供泛型實作。文中提供的實作品滿足了基本需要，解決了基本問題。本書並不講述這一塊那一塊程式碼做些什麼，它採行的方法是：討論問題，選擇設計決策，然後逐步實現這些設計決策。

第 1 章提供的是 policies，一種有助於產生靈活設計的 C++ 技巧。

第 2 章討論和泛型編程有關的通用 C++ 技巧。

第 3 章實作 typelists，一種功能強大、用於操縱型別的資料結構。

第 4 章介紹一個重要的輔助工具：小型物件配置器。

第 5 章介紹泛化仿函式的概念；在運用 *Command* 範式的設計中，它很有用處。

第 6 章講述 *Singleton* 物件。

第 7 章討論和實現了 *Smart Pointers*。

第 8 章講述 *generic Object Factories*。

第 9 章探討 *Abstract Factory* 設計範式，並提供一份實作品。

第 10 章以泛型方式實現了 *Visitor* 設計範式的幾個變型。

第 11 章實現了數個 *MutiMethod* 引擎；這些方案體現設計上的各種選擇。

「設計」涵蓋許多重要工作，C++ 程式員必須以規律的、標準的、合格的基礎和準則來對付。我個人認為 *Object Factories*（第 8 章）是所有高品質多型設計（polymorphic design）的基石。*Smart Pointers*（第 7 章）是大大小小許多 C++ 應用程式的重要組件。*Generalized Functors*（第 5 章）有極為寬廣的應用，一旦你擁有它，許多複雜的設計問題都能夠迎刃而解。其他更特殊的泛型組件，例如 *Visitor*（第 10 章）或 *MultiMethod*（第 11 章），也都有重要而合適的應用，並將語言的支援推向極致。

致謝

Acknowledgements

我要感謝我的父母，他們勤勞地度過了那段最爲漫長艱辛的歲月。

我要特別強調的是，這本書，連同我的大部份職業生涯，如果沒有 Scott Meyers，就都不會存在。自 1998 年於 C++ World Conference 結識 Scott 以來，他就一直幫助我，使我做得更多更好。Scott 第一個熱情地鼓勵我，讓我將我的早期想法付諸實踐。他將我引荐給 John Vlissides，促成了另一個具有豐碩成果的合作；他說動 Herb Sutter，讓我成爲 *C++ Report* 的專欄作家；他將我介紹給 Addison-Wesley 出版公司，實質上強迫著我開始這本書的寫作，而那時我對紐約的銷售人員一點都不了解。整本書的創作過程中，Scott 一直幫助我，給我審閱和建議，和我分享寫作的痛苦，但沒有得到任何好處。

多謝 John Vlissides，他不但提出深邃的見解，讓我相信我的方案中存在問題，還爲我提出了更好的方案。第 9 章之所以存在，正是因爲 John 堅持「事情可以做得更好」。

感謝 P.J. Plauger 和 Marc Briand，他們鼓勵我爲 *C/C++ User Journal* 撰寫文章，那時我以爲專欄作家是外星人。

感謝我的編輯 Debbie Lafferty，她給了我不斷的支持，並提出敏銳的建議。

我在 RealNetworks 的同事，特別是 Boris Jerkunica 和 Jim Knaack，給我很大的幫助；他們爲我營造了自由、競爭、向上的氣氛。我爲此感謝他們。

我也十分感激 comp.lang.c++.moderated 和 comp.std.c++ Usenet 新聞群組的所有參與者。這些朋友極大地促進了我對 C++ 的認識。

我還要把我的感謝獻給本書初稿審閱者：Mihail Antonescu, Bob Archer（書稿的最完整審閱者），Allen Broadman, Ionut Burete, Mirel Chirita, Steve Clamage, James O. Coplien, Doug Hazen, Kevlin Henney, John Hickin, Howard Hinnant, Sorin Jianu, Zoltan Kormos, James Kuyper, Lisa Lippincott, Jonathan H. Lundquist, Petru Marginenean, Patrick McKillen, Florin Mihaila, Sorin Oprea, John Potter,

Adrian Rapiteanu, Monica Rapiteanu, Brian Stanton, Adrian Steflea, Herb Sutter, John Torjo, Florin Trofin, Cristi Vlasceanu。他們投入了大量的精力閱讀初稿並提出建議。沒有他們，本書的品質將大打折扣。

感謝 Greg Comeau，他免費提供給我第一流的 C++ 編譯器。

最後，我非常感謝我的所有家人和朋友，感謝他們無盡的鼓勵和支持。

Part I

技術

Techniques

1

以 Policy 為基礎的 Class 設計

Policy-Based Class Design

這一章將介紹所謂 *policies* 和 *policy classes*，它們是一種重要的 *classes* 設計技術，能夠增加程式庫的彈性並提高復用性，這正是 **Loki** 的目標所在。簡言之，具備複雜功能的 *policy-based class* 是由許多小型 *classes*（稱為 **policies**）組成。每一個這樣的小型 *class* 都只負責單純如行為或結構的某一面（*behavioral or structural aspect*）。一如名稱所示，一個 *policy* 會針對特定主題建立一個介面。在「遵循 *policy* 介面」的前提下，你可以採用任何適當方法來實作 *policies*。

由於你可以混合並匹配各種 *policies*，所以藉由小量核心基礎組件（*core elementary components*）的組合，你可完成一個「行為集」（*behaviors set*）。

本書許多章節都用到了 *policies*，例如第 6 章的泛型類別 *SingletonHolder* 運用 *policies* 來管理物件壽命和多緒安全（*thread safety*）。第 7 章的 *SmartPtr* 幾乎全由 *policies* 建構出來。第 11 章的雙分派引擎（*double-dispatch engine*）運用 *policies* 決定各種取舍。第 9 章的泛型 *Abstract Factory*（Gamma et al. 1995）則運用 *policies* 來選擇不同的生成方法。

本章將說明如何運用 *policies* 來解決問題，並提供 *policy-based classes design* 的詳細內容。當你要把 *class* 分解為 *policies* 時，本章也會給你一些忠告。

1.1 軟體設計的多樣性 (Multiplicity)

軟體工程，也許比其他工程展現出更豐富的多樣性。你可以採用多種正確作法完成一件事，而對與錯之間存在無盡的細微差別。每一個新選擇都會開啓一個新世界。一旦你選擇了一個解決方案，會有一大堆變化隨之湧現，而且會在每個階段中持續不停地湧現，大至系統架構，小至程式片段。所謂軟體設計就是解域空間（*solution space*）中的一道選擇題。

讓我們考慮一個簡單的入門級程式雛型：一個 *Smart Pointer*（機靈指標，第 7 章）。這種 *class* 可被用於單緒或多緒之中，可以運用不同的 *ownership*（擁有權）策略，可以在安全與速度之間協調，可以支援或不支援「自動轉為內部指標」。以上這些特性都可以被自由組合起來，而所謂解答，就是最適合你的應用程式的那個方案。

設計的多樣性不斷困惑新手。遭遇一個軟體設計問題時，什麼是最好的解法？是 *Events*？還是

Objects ? Observers ? Callbacks ? Virtuals ? Templates ? 根據不同的規模和層次，許多不同的解法似乎一樣好。

專業軟體設計師跟新手的最大不同在於，前者知道什麼可以有效運作，什麼不可以。任何設計結構上的問題，都有許多合適解法，然而它們各有不同規格並且各有優缺點，對眼前的問題可能適合也可能不適合。白板上可接受的方案，不一定真有實用價值。

設計一個軟體系統很困難，因為它不斷要求你做抉擇。而程式設計猶如人生，抉擇是困難的。

好極了，經驗老練的設計師知道怎樣的抉擇會引領出好的設計。但對新手來說，每一個設計抉擇都開啓一扇未知世界之門。有經驗的設計者就像一名好棋手，可以看出好幾步棋之後的局勢。但這需要時間來學習。或許這就是為何編程（programming）才華可能在年輕時就展現出來，而軟體設計（software design）才華卻常需要更多時間才能成熟。

除了困惑新手，設計時「各種決定的組合」也常困擾程式庫（library）撰寫者。爲了將有用的設計實作出來，程式庫設計者必須將各種常見情況加以分類融合，並給出一個開放架構，如此一來應用程式員（application programmer）便能根據個別情況組合出他所需要的功能。

更確切地說，如何在程式庫中包裝出既富彈性又有優良設計的組件（components）？如何讓使用者自行裝配這些組件？如何在合理大小的程式碼中對抗「邪惡的」多樣性？這些正是本章乃至本書試圖回答的問題。

1.2 全功能型（Do-It-All）介面的失敗

「在一個全功能型介面下實作所有東西」的作法並不好。理由很多。

比較重要的負面影響包括智力上的負荷、體積大小的陡峭爬升、以及效率考量。龐大的 classes 不能視爲成功，因為它們會導致沉重的學習負荷，並且有「非必要之大規模」傾向，使得程式碼遠比手工製作還慢。

一個過於豐富的介面的最大問題，或許在於缺乏靜態型別安全性（static type safety）。系統架構的一個主要基本原則是：以「設計」實現某些「原則」（axioms），例如你不能產生兩個 *Singleton* 物件（第 6 章）或產生一個 "disjoint" 族系物件（第 9 章）。理想上，一個良好設計應該在編譯期強制表現出大部份 constraints（約束條件、規範）。

在一個「無所不包」的大型介面上厲行如此的 constraints 是很困難的。一般來說一旦你選擇了某一組 design constraints，大型介面內就只有某些子集能夠維持其有效語意。程式庫（library）的運用向來在「語法有效」和「語意有效」之間存在著縫隙。程式員可能寫出愈來愈多的概念，它們雖然「語法有效」，卻「語意無效」。

舉個例子，考慮以 thread-safety（多緒安全性）觀點來實作 *Singleton* 物件。如果程式庫完全包裝了執行緒，那麼一個特別的、不可移植的多緒系統就無法運用這個 *Singleton* 程式庫。如果這個程式庫允許其客戶使用未受保護的基本函式（primitive functions），那麼很可能程式員會

因為寫出一些「語法有效」但「語意無效」的程式而破壞了整個設計。

如果程式庫將不同的設計實作為各個小型 classes，每個 class 代表一個特定的罐裝解法，如何？例如在 smart pointer 例中你會看到 SingleThreadedSmartPtr, MultiThreadedSmartPtr, RefCountedSmartPtr, RefLinkedSmartPtr 等等。

這種作法的問題是會產生大量設計組合。例如上述提到的四個 classes 必然導致諸如 SingleThreadedRefCountedSmartPtr 這樣的組合。如果再加一個設計選項（例如支援型別轉換），會產生更多潛在組合。這最終會讓程式庫實作者和使用者受不了。很明顯地這並不是一個好方法。面對這麼多的潛在組合（近乎指數爬升），千萬別企圖使用暴力列舉法。

這樣的程式庫不只造成大量的智力負荷，也是極端的嚴刻而死板。一點點輕微不可測的訂製行為（例如試著以一個特定值為預先建構好的 smart pointers 設初值）都會造成整個精心製作的 library classes 沒有用處。

設計是為了解行 constraints（約束條件、規範）。因此，以設計為目標的程式庫必須幫助使用者精巧完成設計，以實現使用者自己的 constraints，而不是實現預先定義好的 constraints。罐裝設計不適用於以設計為目標用途的程式庫，就像魔術數字不適用於一般程式碼一樣。當然啦，一些「最普遍的、受推薦的」罐裝解法將受到大眾歡迎 — 只要客端程式員必要時能夠改變它們。

程式庫領域中的目前水平令人遺憾：低階的通用型程式庫和特化型程式庫大量存在，而用來直接輔助設計 — 這是最高階結構 — 的程式庫幾乎沒有。這種情況很矛盾，因為任何不那麼平淡無奇的應用程式都有自己的設計，所以一個以設計為目標用途的程式庫應該適用於絕大多數應用程式。

Frameworks（框架）試圖填補這樣的裂口，不過這種產品把應用程式限制在特定設計內，而不是讓使用者選擇並訂製（customize）設計。如果程式員需要實作出自己的設計，他們必須從一開始的 classes，functions...做起。

1.3 多重繼承 (Multiple Inheritance) 是救世主？

TemporarySecretary classes 繼承自 Secretary 和 Temporary¹，因此它同時擁有後兩者（秘書和臨時雇員）的特性，以及其他可能的更多特性。這導致一種想法：多重繼承 (Multiple Inheritance) 可能有助於處理「設計組合」 — 透過少量的、明確選擇後的 based classes。這麼一來使用者藉由繼承 BaseSmartPtr, MultiThreaded 和 RefCounted，便可製作出具有多緒能力、參用計數功能的精靈指標。不過，任何一位有經驗的 class 設計者都知道，這樣天真的設計方式其實無法運作。

¹ 這個例子來自 Bjarne Stroustrup 「支持多重繼承」的舊論點，出現於《The C++ Programming Language》第一版。從那個時候起，C++ 就再沒有提倡過多重繼承。

分析多重繼承的失敗原因，有助於產生更富彈性的設計，這可以對健全的設計方案提供一些有趣的想法。藉由多重繼承機制來組合多項功能，會產生如下問題：

1. 關於技術 (*Mechanics*)。目前並沒有一成不變即可套用的程式碼，可以在某種受控情況下將繼承而來的 classes 組合 (assemble) 起來。唯一可組合 `BaseSmartPtr`, `MultiThreaded` 和 `RefCounted` 的工具是語言提供的「多重繼承」機制：僅僅只是將被組合的 base classes 結合在一起並建立一組用來存取其成員的簡單規則。除非情況極為單純，否則結果難以讓人接受。大多數時候你得小心協調繼承而來的 classes 的運轉，讓它們得到所需的行為。
2. 關於型別資訊 (*Type information*)。Based classes 並沒有足夠的型別資訊來繼續完成它們的工作。例如，想像一下，你正試著藉由繼承一個 `DeepCopy` class 來為你的 smart pointer 實作出深層拷貝 (deep copy)。但 `DeepCopy` 應該具有怎樣的介面呢？它必須產生一個物件，而其型別目前未知。
3. 關於狀態處理 (*State manipulation*)。base classes 實作之各種行為必須操作相同的 state (譯註：意指資料)。這意味他們必須虛擬繼承一個持有該 state 的 base class。由於總是由 user classes 繼承 library classes (而非反向)，這會使設計更加複雜而且變得更沒有彈性。

雖然本質上是組合 (combinatorial)，但多重繼承無法單獨解決設計時的多樣性選擇。

1.4 Templates 帶來曙光

templates 是一種很適合「組合各種行為」的機制，主要因為它們是「依賴使用者提供的型別資訊」並且「在編譯期才產生」的程式碼。

和一般的 class 不同，class templates 可以不同的方式訂做。如果想要針對特定情況來設計 class，你可以在你的 class template 中特化其成員函式來因應。舉個例子，如果有一個 `SmartPtr<T>`，你可以針對 `SmartPtr<Widget>` 特化其任何成員函式。這可以為你在設計特定行為時提供良好粒度 (granularity)。

猶有進者，對於帶有多個參數的 class templates，你可以採用 partial template specialization (偏特化，第 2 章介紹)。它可以讓你根據部分參數來特化一個 class template。例如，下面是一個 template 定義：

```
template <class T, class U> class SmartPtr { ... };
```

你可以令 `SmartPtr<T, U>` 針對 `Widget` 及其他任意型別加以特化，定義如下：

```
template <class U> class SmartPtr<Widget, U> { ... };
```

由於 `template` 的編譯期特性以及「可互相組合」特性，使它在設計期非常引人注目。然而一旦你開始嘗試實作這些設計，你會遭遇一些不是那麼淺白的問題：

1. 你無法特化結構。單單使用 `templates`，你無法特化「`class` 的結構」（我的意思是其資料成員），你只能特化其成員函式。
2. 成員函式的特化並不能「依理擴張」。你可以對「單一 `template` 參數」的 `class template` 特化其成員函式，卻無法對著「多個 `template` 參數」的 `class template` 特化其個別成員函式。例如：

```
template <class T> class Widget
{
    void Fun() { .. generic implementation ... }
};
// OK: specialization of a member function of Widget
template <> void Widget<char>::Fun()    // 譯註：原文少了 void
{
    ... specialized implementation ...
}

template <class T, class U> class Gadget
{
    void Fun() { .. generic implementation ... }
};
// Error! Cannot partially specialize a member class of Gadget
// 譯註：因為這是 member function 的 Explicit specialization 並無 partial
// specialization 機制。注意這和 class templates 不同！參見 C++ Primer, 16.9 節
template <class U> void Gadget<char, U>::Fun()
{
    ... specialized implementation ...
}
```

3. 程式庫撰寫者不能夠提供多筆預設值。理想情況下 `class template` 的作者可以對每個成員函式提供一份預設實作品，卻不能對同一個成員函式提供多份預設實作品。

現在讓我們比較一下多重繼承和 `templates` 之間的缺點。有趣的是兩者互補。多重繼承欠缺技術（mechanics），`templates` 有豐富的技術。多重繼承缺乏型別資訊，而那東西在 `templates` 裡頭大量存在。`Templates` 的特化無法擴張（scales），多重繼承卻很容易擴張。你只能為 `template` 成員函式寫一份預設版本，但你可以寫數量無限的 `base classes`。

根據以上分析，如果我們將 `templates` 和多重繼承組合起來，將會產生非常彈性的裝置（device），應該很適合用來產生程式庫中的「設計元素」（design elements）。

1.5 Policies 和 Policy Classes

`Policies` 和 `Policy Classes` 有助於我們設計出安全又有效率且具高度彈性的「設計元素」。所謂 `policy`，用來定義一個 `class` 或 `class template` 的介面，該介面由下列項目之一或全部組成：內隱

型別定義（inner type definition）、成員函式和成員變數。

Policies 也被其他人用於 **traits**（Alexandrescu 2000a），不同的是後者比較重視行為而非型別。Policies 也讓人聯想到設計範式 *Strategy*（Gamma et al. 1995），只不過 policies 吃緊於編譯期（所謂 compile-time bound）。

舉個例子，讓我們定義一個 policy 用以生成物件：Creator policy 是個帶有位別 T 的 class template，它必須提供一個名為 Create 的函式給外界使用，此函式不接受引數，傳回一個 *pointer to T*。就語意而言，每當 Create() 被呼叫就必須傳回一個指標，指向新生的 T 物件。至於物件的精確生成模式（creation mode），留給 policy 實作品做為迴旋餘地。

讓我們來定義一個可實作出 Creator policy 的 class。產生物件的可行辦法之一就是運算式 new，另一個辦法是以 malloc() 加上 placement new 運算子（Meyers 1998b）。此外還可以採用複製（cloning）方式來產生新物件。下面是三種作法的實例呈現：

```
template <class T>
struct OpNewCreator
{
    static T* Create()
    {
        return new T;
    }
};

template <class T>
struct MallocCreator
{
    static T* Create()
    {
        void* buf = std::malloc(sizeof(T));
        if (!buf) return 0;
        return new(buf) T;
    }
};

template <class T>
struct PrototypeCreator
{
    PrototypeCreator(T* pObj = 0)
        : pPrototype_(pObj)
    {}
    T* Create()
    {
        return pPrototype_ ? pPrototype_->Clone() : 0;
    }
    T* GetPrototype() { return pPrototype_; }
    void SetPrototype(T* pObj) { pPrototype_ = pObj; }
private:
    T* pPrototype_;
};
```


任何一個 policy 都可以有無限多份實作品。實作出 policy 者便稱為 **policy classes**²，這東西並不意圖被單獨使用，它們主要用於繼承或被內含於其他 classes。

這裡有一個重要觀念：policies 介面和一般傳統的 classes 介面（純虛擬函式集）不同，它比較鬆散，因為 policies 是語法導向（syntax oriented）而非標記導向（signature oriented）。換句話說 **Creator** 明確定義的是「怎樣的語法構造符合其所規範的 class」，而非「必須實作出哪些函式」。例如 **Creator** policy 並沒有規範 `Create()` 必須是 `static` 還是 `virtual`，它只要求 class 必須定義出 `Create()`。此外 **Creator** 也只規定 `Create()` 應該（但非必須）傳回一個指向新物件的指標。因此 `Create()` 也許會傳回 0 或丟出異常——這都極有可能。

面對一個 policy，你可以實作出數個 policy classes。它們全都必須遵守 policy 所定義的介面。稍後你會看到一個例子，使用者選擇了一個 policy 並應用到較大結構中。

先前定義的三個 policy classes，各有不同的實作方式，甚至連介面也有些不同（例如 `PrototypeCreator` 多了兩個函式 `GetPrototype()` 和 `SetPrototype()`）。儘管如此，它們全都定義 `Create()` 並帶有必要的回返型別，所以它們都符合 **Creator** policy。

現在讓我們看看如何設計一個 class 得以利用 **Creator** policy，它以複合或繼承的方式使用先前所定義的三個 classes 之一，例如：

```
// Library code
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{ ... };
```

如果 class 採用一個或多個 policies，我們稱其為 **hosts** 或 **host classes**³。上例的 `WidgetManager` 便是「採用了一個 policy」的 **host class**。**Hosts** 負責把 policies 提供的結構和行為組成一個更複雜的結構和行為。

當客戶端將 `WidgetManager` template 具現化（instantiating）時，必須傳進一個他所期望的 policy：

```
// Application code
typedef WidgetManager< OpNewCreator<Widget> > MyWidgetMgr;
```

讓我們分析整個來龍去脈。無論何時，當一個 `MyWidgetMgr` 物件需要產生一個 `Widget` 物件，它便呼叫它的 policy 子物件 `OpNewCreator<Widget>` 所提供的 `Create()`。選擇「生成策略」（**Creation policy**）是 `WidgetManager` 使用者的權利。藉由這樣的設計，可以讓 `WidgetManager` 使用者自行裝配他所需要的機能。

這便是 **policy-based class** 的設計主旨。

² 這個名稱稍許有點不夠精確，因為，如你所見，policy 的實作品也可以是 class templates。

³ 雖然 **host classes** 從技術上來說是 **host class templates**，但是就讓我們固守和注釋 2 相同的定義吧。不論 **host classes** 或 **host class templates**，都意味相同的概念。

1.5.1 運用 Template Template 參數實作 Policy Classes

如同先前例子所示，policy 的 template 引數往往是贅餘的。使用者每每需要傳入 template 引數給 OpNewCreator，這很笨拙。一般來說 host class 已經知道 policy class 所需參數，或是輕易便可推導出來。上述例子中 WidgetManager 總是操作 Widget 物件，這樣情況下還要求使用者「把 Widget 型別傳給 OpNewCreator」，就顯得多餘而且危險。

這時候程式庫可以使用「template template 參數」來描述 policies，如下所示：

```
// Library code
template <template <class Created> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
    ...
};
// 譯註:Created 是 CreationPolicy 的參數，CreationPolicy 則是 WidgetManager
// 的參數。Widget 已經寫入上述程式庫中，所以使用時不需要再傳一次參數給 policy。
```

儘管露了臉，上述 Created 並未對 WidgetManager 有任何貢獻。你不能在 WidgetManager 中使用 Created，它只是 CreationPolicy（而非 WidgetManager）的形式引數（formal argument），因此可以省略。

應用端現在只需在使用 WidgetManager 時提供 template 名稱即可：

```
// Application code
typedef WidgetManager<OpNewCreator> MyWidgetMgr;
```

搭配 policy class 使用「template template 參數」，並不單純只為了方便。有時候這種用法不可或缺，以便 host class 可藉由 templates 產生不同型別的物件。舉個例子，假設 WidgetManager 想要以相同的生成策略產生一個 Gadget 物件，程式碼長像如下：

```
// Library code
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
    ...
    void DoSomething()
    {
        Gadget* pW = CreationPolicy<Gadget>().Create();
        ...
    }
};
```

使用 policies 是否會為我們帶來一些優勢呢？乍看之下並不太多。首先，Creator policy 的實作本來就十分簡短。當然，WidgetManager 的作者應該會把「生成物件」的那份程式碼寫成 inline 函式，並避開「將 WidgetManager 建立為一個 template」時可能遭遇的問題。

然而，policy 的確能夠帶給 `WidgetManager` 非常大的彈性。第一，當你準備具現化 (instantiating) `WidgetManager` 時，你可以從外部變更 policies，就和改變 template 引數一樣簡單。第二，你可以根據程式的特殊需求，提供自己的 policies。你可以採用 `new` 或 `malloc` 或 `prototypes` 或一個專用於你自己系統上的罕見記憶體配置器。`WidgetManager` 就像一個小型的「程式碼生成引擎」(code generation engine)，你可以自行設定程式碼的產生方式。

爲了讓應用程式開發人員的日子更輕鬆，`WidgetManager` 的作者應該定義一些常用的 policies，並且以「template 預設引數」的型式提供最常用的 policy：

```
template <template <class> class CreationPolicy = OpNewCreator>
class WidgetManager ...
```

注意，policies 和虛擬函式有很大不同。雖然虛擬函式也提供類似效果：class 作者以基本的 (primitive) 虛擬函式來建立高端功能，並允許使用者覆寫這些基本虛擬函式的行爲。然而如前所示，policies 因爲有豐富的型別資訊及靜態連結等特性，所以是建立「設計元素」時的本質性東西。不正是「設計」指定了「執行前型別如何互相作用、你能夠做什麼、不能夠做什麼」的完整規則嗎？Policies 可以讓你在型別安全 (typesafe) 的前提下藉由組合各個簡單的需求來產出你的設計。此外，由於編譯期才將 `host class` 和其 policies 結合在一起，所以和手工打造的程式比較起來更加牢固並且更有效率。

當然，也由於 policies 的特質，它們不適用於動態連結和二進位介面，所以本質上 policies 和傳統介面並不互相競爭。

1.5.2 運用 Template 成員函式實作 Policy Classes

另外一種使用「template template 參數」的情況是把 template 成員函式用來連接所需的簡單類別。也就是說將 policy 實作爲一般 class (「一般」是相對於 class template 而言)，但有一個或數個 templated members。

例如，我們可以重新定義先前的 `Creator` policy 成爲一個 non-template class，其內提供一個名爲 `Create<T>` 的 template 函式。如此一來 policy class 看起來像下面這樣子：

```
struct OpNewCreator
{
    template <class T>
    static T* Create()
    {
        return new T;
    }
};
```

這種方式所定義並實作出來的 policy，對於舊式編譯器有較佳相容性。但從另一方面來說，這樣的 policy 難以討論、定義、實作和運用。

1.6 更豐富的 Policies

Creator policy 只規範一個 `Create()` 成員函式。然而 `PrototypeCreator` 卻多定義了兩個函式，分別為 `GetPrototype()` 和 `SetPrototype()`。讓我們來分析一下。

由於 `WidgetManager` 繼承了 policy class 而且 `GetPrototype()` 和 `SetPrototype()` 是 `PrototypeCreator` 的 public 成員，所以這兩個函式便被加至 `WidgetManager`，並且可以直接被使用者取用。

然而 `WidgetManager` 只要求 `Create()`；那是 `WidgetManager` 一切所需，也是用來保證它自己機能的唯一要求。不過使用者可以開發出更豐富的介面。

prototype-based Creator policy class 的使用者可以寫出下列程式碼：

```
typedef WidgetManager<PrototypeCreator>
    MyWidgetManager;
...
Widget* pPrototype = ...;
MyWidgetManager mgr;
mgr.SetPrototype(pPrototype);
... use mgr ...
```

如果此後使用者決定採用一個不支援 prototypes 的生成策略，那麼編譯器會指出問題：prototype 專屬介面已經被用上了。這正是我們希望獲得的堅固設計。

如此結果是很受歡迎的。使用者如果需要擴充 policies，可以在不影響 host class 原本功能的前提下，從更豐富的功能中得到好處。別忘了，決定「哪個 policy 被使用」的是使用者而非程式庫自身。和一般多重介面不同的是，policies 給予使用者一種能力，在型別安全（typesafe）的前提下擴增 host class 的功能。

1.7 Policy Classes 的解構式 (Destructors)

有一個關於建立 policy classes 的重要細節。大部分情況下 host class 會以「public 繼承」方式從某些 policies 衍生而來。因此，使用者可以將一個 host class 自動轉為一個 policy class（譯註：向上轉型），並於稍後 delete 該指標。除非 policy class 定義了一個虛擬解構式（virtual destructor），否則 delete 一個指向 policy class 的指標，會產生不可預期的結果⁴，如下所示：

```
typedef WidgetManager<PrototypeCreator>
    MyWidgetManager;
...
MyWidgetManager wm;
PrototypeCreator<Widget>* pCreator = &wm; // dubious, but legal
delete pCreator; // compiles fine, but has undefined behavior
```

然而如果為 policy 定義了一個虛擬解構式，會妨礙 policy 的靜態連結特性，也會影響執行效率。

⁴ 你可以在本書第 4 章「小型物件配置技術」（Small-Object Allocation）中找到一份精細討論。

許多 policies 並無任何資料成員，純粹只規範行為。第一個虛擬函式被加入後會為物件大小帶來額外開銷（譯註：因為引入一份 `vptr`），所以虛擬解構式應該儘可能避免。

一個解法是，當 host class 自 policy class 衍生時，採用 `protected` 繼承或 `private` 繼承。然而這樣會失去豐富的 policies 特性（1.6 節）。policies 應該採用一個輕便而有效率的解法 — 定義一個非虛擬的 `protected` 解構式：

```
struct OpNewCreator
{
    template <class T>
    static T* Create()
    {
        return new T;
    }
protected:
    ~OpNewCreator() {}
};
```

由於解構式屬於 `protected` 層級，所以只有衍生而得的 classes 才可以摧毀這個 policy 物件。這樣一來外界就不可能 `delete` 一個指向 policy class 的指標。而由於解構式並非虛擬函式，所以不會有大小或速度上的額外開銷。

1.8 透過不完全具現化而獲得的選擇性機能

事情還可以變得更好。C++ 的一些有趣特性造就了 policies 的威力。如果 class template 有一個成員函式未曾被用到，它不會被編譯器具體實現出來。編譯器不理會它，甚至也許不會為它進行語法檢驗⁵。

如此一來造成 host class 有機會指明並使用 policy class 的可選特性。舉個例子，讓我們為 `WidgetManager` 定義一個 `SwitchPrototype()`：

```
// Library code
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
    ...
    void SwitchPrototype(Widget* pNewPrototype)
    {
        CreationPolicy<Widget>& myPolicy = *this;
        delete myPolicy.GetPrototype();
        myPolicy.SetPrototype(pNewPrototype);
    }
};
```

下面是一些非常有趣的結果：

⁵ 據 *C++ Standard* 所載，「未被使用之 template functions」的語法分析層級，由編譯器自行決定。至於語意檢驗，編譯器根本不進行。

- 如果你採用一個「支援 prototype」的 Creator policy 來具現化 WidgetManager，你便可以使用 SwitchPrototype()。
- 如果你採用一個「不支援 prototype」的 Creator policy 來具現化 WidgetManager，並嘗試使用 SwitchPrototype()，會出現編譯錯誤。
- 如果你採用一個「不支援 prototype」的 Creator policy 來具現化 WidgetManager，並且從未試圖使用 SwitchPrototype()，程式是合法的。

這些都意味 WidgetManager 除了可從彈性且豐富的介面得到好處之外，也仍然可以搭配較簡單的介面而正常運作 — 只要你不試著去使用其中某些成員函式。

WidgetManager 的作者現在可以這樣定義 Creator policy：

Creator 設定一個型別為 T 的 class template，其中列有 Create()，此函式應該傳回一個指標指向新生之 T 物件。Creator 實作碼可以選擇性地定義兩個額外函式：T* GetPrototype() 和 SetPrototype(T*)，讓使用者可以在生成物件時擁有「取得」或「設定」某個 prototype 的機會。這種情況下，WidgetManager 於是可能出現一個 SwitchPrototype(T* pNewPrototype) 函式，可刪除目前的 prototype 並設定一個新的 prototype。

與各個 policy class 結合時，這些不完整具現化（incomplete instantiation）帶給你（使用者）有如程式庫設計者一般的非凡自由度。你可以實作出「傾斜的」（lean）host classes，它能夠使用額外的特性，並能夠姿態優雅地將層次降低至紀律性高的最小化 policies。

1.9 結合 Policy Classes

當你將 policies 組合起來，便是它們最有用的時候。一般而言，一個高度可組裝化的 class 會運用數個 policies 來達成其運作上的多種面向。一個程式庫使用者可以藉由組合不同的 policy classes 來選擇他所需的高階行為。

舉個例子，假設我們正打算設計一個泛型的 smart pointer（第 7 章有完整實作）。假設你分析出兩個得被建立為 policies 的設計：**threading model**（多緒模型）和 **check before dereference**（提領前先檢驗）。於是你實作出一個帶有兩個 policies 的 class template，名為 SmartPtr：

```
template <
    class T,
    template <class> class CheckingPolicy,
    template <class> class ThreadingModel
>
class SmartPtr;
```

SmartPtr 有三個 template 參數：一個代表 pointee type（被指物件的型別），另兩個是 policies。在 SmartPtr 之中你可以運用兩個 policies 組織出一份穩固的實作品。SmartPtr 成為「整合數個 policies」的協調層，而非一成不變的罐裝實作品。以這種方式來設計 SmartPtr，便是賦予使用者「以簡單的 typedef 對 SmartPtr 進行組態（configure）」的能力：

```
typedef SmartPtr<Widget, NoChecking, SingleThreaded>
    WidgetPtr;
```

在同一個應用程式中，你可以定義並使用數種不同的 smart pointer classes：

```
typedef SmartPtr<Widget, EnforceNotNull, SingleThreaded>
    SafeWidgetPtr;
```

兩個 policies 定義如下：

Checking：這個名為 `CheckingPolicy<T>` 的 class template 必須公開一個 `Check()` 成員函式，可接受一個型別為 `T*` 的左值。當 smart pointer 即將被提領（dereference）時會呼叫 `Check()`，傳入被指物件（pointee object）並作檢查。

ThreadingModel：這個名為 `ThreadingModel<T>` 的 class template 必須提供一個名為 `Lock` 的內部型別，該型別的建構式接受一個 `T&` 參數。對一個 `Lock` 物件來說，其生命中所有對此 `T` 物件的操作行為都是循序的、次第的（serialized）。

舉個例子，下面是兩個 policy classes `NoChecking` 和 `EnforceNotNull` 的實作內容：

```
template <class T> struct NoChecking
{
    static void Check(T*) {}
};
template <class T> struct EnforceNotNull
{
    class NullPointerException : public std::exception { ... };
    static void Check(T* ptr)
    {
        if (!ptr) throw NullPointerException();
    }
};
```

藉由抽換不同的 **Checking** policy class，你可以實作出各種不同行為。你甚至可以利用預設值來初始化被指物件（pointee object）— 只要傳入一個 `reference-to-pointer` 即可，像這樣：

```
template <class T> struct EnsureNotNull
{
    static void Check(T*& ptr)
    {
        if (!ptr) ptr = GetDefaultValue();
    }
};
```

`SmartPtr` 這樣使用 **Checking** policy：

```
template
<
    class T,
    template <class> class CheckingPolicy,
    template <class> class ThreadingModel
>
class SmartPtr
    : public CheckingPolicy<T>
    , public ThreadingModel<SmartPtr>
```



```

{
    ...
    T* operator->()
    {
        typename ThreadingModel<SmartPtr>::Lock guard(*this);
        CheckingPolicy<T>::Check(pointee_);
        return pointee_;
    }
private:
    T* pointee_;
};

```

注意上述同一個函式中對 `CheckingPolicy` 和 `ThreadingModel` 兩個 policy classes 的運用。根據不同的 `template` 引數，`SmartPtr::operator->` 會表現出兩種不同的正交（orthogonal）行為。這正是 policies 的組合威力所在。

一旦你設法把一個 class 分解成正交的 policies，便可利用少量程式碼涵蓋大多數行為。

1.10 以 Policy Classes 訂製結構

如同 1.4 節所說，templates 的限制之一是，你無法訂製（customize）class 的結構，只能訂製其行為。然而 policy-based design 支援結構方面的訂製。

假設你想支援「非指標形式」的 `SmartPtr`，例如某些平台上的某些指標也許會以 `handle` 型式呈現，這是一種用來傳給系統函式的整數值，藉以取得實際指標。為了解決這種情況，你可以透過一個所謂的 **Structure policy** 將指標的存取「間接化」。Structure policy 將指標的儲存概念抽象化，因此它應該提供一個 `PointerType` 型別（用以代表指標所指物件的型別）、一個 `ReferenceType` 型別（用以代表指標所指物件的 `reference` 型別），以及 `GetPointer()` 和 `SetPointer()` 兩函式。

不把 `pointer` 型別硬性規定為 `T*`，這種作法帶來重大好處。例如你可以將 `SmartPtr` 應用於非標準型別之上（有點像是 `segment` 架構上的 `near` 指標和 `far` 指標），或者你可以輕鬆實作出靈巧解法諸如 `before` 和 `after` 函式（Stroustrup 2000a）。這些可能性都非常有趣。

smart pointer 的預設儲存型式是一個帶有 Structure policy 介面的一般指標，像下面這樣：

```

template <class T>
class DefaultSmartPtrStorage // 譯註:Loki 無此class,但有一個DefaultSPStorage
{
public:
    typedef T* PointerType;
    typedef T& ReferenceType;
protected:
    PointerType GetPointer() { return ptr_; }
    void SetPointer(PointerType ptr) { ptr_ = ptr; }
}

```



```
private:
    PointerType pointee_;
};
```

實際指標的儲存型式已被完全隱蔽於 **Structure** 介面之內。現在，**SmartPtr** 可以運用一個 **Storage policy** 來取代對 **T*** 的聚合（aggregating）：

```
template
<
    class T,
    template <class> class CheckingPolicy,
    template <class> class ThreadingModel,
    template <class> class Storage = DefaultSmartPtrStorage
>
class SmartPtr;
```

當然，為了內嵌所需的結構，**SmartPtr** 必須繼承自 **Storage<T>** 或聚合（aggregate）一個 **Storage<T>** 物件（[譯註](#)：詳見第 7 章）。

1.11 Policies 的相容性

假設你要產生兩個 **SmartPtr**：**FastWidgetPtr** 是個不需檢驗的指標，**SafeWidgetPtr** 則必須在提領（dereference）之前先檢驗。這時有個有趣的問題：你能將一個 **FastWidgetPtr** 物件指派（賦值）給一個 **SafeWidgetPtr** 物件嗎？你應該有能力以其他方法指派它們嗎？如果你想實現出這樣的功能，該如何實作？

讓我們從推理跨出第一步。**SafeWidgetPtr** 比 **FastWidgetPtr** 有更多限制，因此我們很容易接受「把 **FastWidgetPtr** 轉為 **SafeWidgetPtr**」的想法。這是因為 C++ 原就支援隱式轉換（implicit conversion），不過也存在一些限制，例如 **non-const** 型別轉為 **const** 型別。

從另一方面說，「自由地將 **SafeWidgetPtr** 物件轉換為 **FastWidgetPtr** 物件」是危險的。因為應用程式大都使用 **SafeWidgetPtr**，只有小型且需要考慮速度的核心程式碼才會考慮使用 **FastWidgetPtr**。只在明確受控的情況下才允許將 **SafeWidgetPtr** 轉換為 **FastWidgetPtr**，此舉將有助於保持 **FastWidgetPtr** 的最小用量。

Policies 之間彼此轉換的各種方法中，最好又最具擴充性的實作法是以 **policy** 來控制 **SmartPtr** 物件的拷貝和初始化，如下所示（讓我們將先前程式簡化為只有一個 **policy**：**Checking**）。

```
template
<
    class T,
    template <class> class CheckingPolicy
>
class SmartPtr : public CheckingPolicy<T>
{
    ...
    template
```

```

    <
        class T1,
        template <class> class CP1,
    >
    SmartPtr(const SmartPtr<T1, CP1>& other)
        : pointee_(other.pointee_), CheckingPolicy<T>(other)
    { ... }
};

```

SmartPtr 實作出一個「接受任何一種 SmartPtr 物件」的 template *copy* 建構式。其中粗體那一行係根據其引數 SmartPtr<T1, CP1> 的內容，將 SmartPtr 的內容初始化。

下面介紹其運作方式（請接續上述建構式）。假設你有個 ExtendWidget class，衍生自 Widget。當你以一個 SmartPtr<ExtendedWidget, NoChecking> 初始化一個 SmartPtr<Widget, NoChecking> 時，編譯器會試著以一個 ExtendWidget* 初始化 Widget*（這會成功），然後以一個 SmartPtr<Widget, NoChecking> 初始化 NoChecking。這看起來很可疑，但是別忘了 SmartPtr 衍生自其 policy，所以編譯器可以輕易知道你想要以一個 NoChecking 初始化一個 NoChecking。整個初始化過程可以良好進行。

接下來就有意思了。假設你打算以一個 SmartPtr<ExtendedWidget, NoChecking> 初始化一個 SmartPtr<Widget, EnforceNotNull>。此時 ExtendedWidget* 被轉為 Widget*，一如前面所說。然後編譯器試圖將 SmartPtr<ExtendedWidget, NoChecking> 拿來匹配 EnforceNotNull 建構式。

如果 EnforceNotNull 實作出可接受 NoChecking 物件的建構式，那麼編譯器會找到那個建構式，完成轉換。如果 NoChecking 實作出可將自己轉換為 EnforceNotNull 的轉型運算子，那麼轉換也可以進行。除此之外，都會產生編譯錯誤。

如你所見，當你進行 policies 轉換時，兩邊都有彈性。左手邊你可以實作轉型建構式，右手邊你可以實作轉型運算子。

assignment 運算子也有一樣的難纏問題，幸運的是 Sutter 2000（譯註：*Exceptional C++* 條款 41）闡述了一種非常漂亮的技術，可以讓你根據 *copy* 建構式實作出 *assignment* 運算子。這是一個非常漂亮的手法，你應該讀讀那篇文章。Loki 的 SmartPtr 也運用了這項技術。

雖然 NoChecking 轉換為 EnforceNotNull 或反向轉換感覺都十分合理，但有些轉換卻是一點也不合理。想像將一個 reference-counted 指標轉換為一個支援其他「ownership（擁有權）策略」的指標，將是一場毀滅性的 *copy*（有點像 `std::auto_ptr`）。這樣的轉換造成語意上的錯誤。所謂 *reference counting* 是「所有指向同一物件的指標都為大家所知，並且可根據一個獨一無二的計數器加以追蹤」，一旦你嘗試將某個指標設為另一種 *ownership policy*，你便是破壞了 *reference counting* 賴以有效運作的不變性（恆長性）。

總之，ownership 的轉換不該是隱式轉換，應該特別小心處理。你最好明確呼叫某個函式來改變「reference-counted 指標」的 *ownership policy*。唯有源端指標的 *reference count* 數值為 1，這個函式才有可能轉換成功。

1.12 將一個 Class 分解為一堆 Policies

建立 policy-based class design 的最困難部分，便是如何將 class 正確分解為 policies。一個準則就是，將參與 class 行為的設計鑑別出來並命名之。任何事情只要能以一種以上的方法解決，都應該被分析出來，並從 class 中移出來成為 policy。別忘了，湮沒於 class 設計之中的 constraints（約束條件）就像湮沒於程式碼中的魔術常數一樣不好。

舉個例子，讓我們考慮 WidgetManager class。如果 WidgetManager 內部生成新的 Widget 物件，生成方式應該推遲至 policy 才確定。如果 WidgetManager 打算儲存一大群 Widget 物件，比較合理的設計是將群集設計為一個 storage policy，除非你對特定的儲存機制有強烈偏好。

極端情形下，host class 幾乎就是 policies 的集合，它將設計期的全部決定和約束條件都委派（delegates）給 policies，這樣的 host class 只不過是個涵蓋 policies 集合的外層而已，只能處理 policies 組合出來的行為。

過於泛化的 host class 會產生缺點，它會有過多的 template 參數。實用上，4~6 個以上的 template 參數會造成合作上的笨拙。不過如果 host class 打算提供複雜而有用的功能，該有的 template 參數還是要有。

型別定義（也就是 typedef）是運用 policy-based classes 時的一個重要工具。這不僅是為了方便，也可以確保有條理地運用和易維護性。例如下面這個型別定義：

```
typedef SmartPtr
<
    Widget,
    RefCounted,
    NoChecked
>
WidgetPtr;
```

在程式碼中使用冗長定義的 SmartPtr 而不使用上述簡潔的 WidgetPtr，實在乏味而令人厭煩。不過，「乏味」與程式的「可維護性」和「可讀性」相比，還是小問題。隨著設計的演進，WidgetPtr 的定義也許會跟著改變，例如也許會改用與「除錯期所用之 NoChecking」不同的一個 checking policy。所有程式都採用 WidgetPtr 而不採用「硬以 SmartPtr 寫出的實作品」是很重要的。其間差別就像「函式」和「功能相等的 inline 函式」一樣。雖然技術上 inline 函式做相同的事情，但我們無法在它背後建立抽象性。

當你將 class 分解為 policies 時，找到正交分解（orthogonal decomposition）很重要。正交分解會產生一些彼此完全獨立的 policies。你很容易發現一個非正交分解 — 如果各式各樣的 policies 需要知道彼此，那就是了。

舉個例子。試想 smart pointer 裡頭的一個 Array policy。它非常簡單，主訴求是「無論 smart pointer 有無指向 array，這個 policy 都會定義一個 T& ElementAt(T* ptr, unsigned int index)，和一個針對 const T 的類似版本。至於 non-array policy，由於並沒有定義 ElementAt()，所

如果有人試圖使用它，會出現編譯錯誤。以 1.6 節的話來說，`ElementAt()` 是一個可選用的、豐富介面下的行為。

下面是兩個實作出 `Array` policy 的 policy classes：

```
template <class T>
struct IsArray
{
    T& ElementAt(T* ptr, unsigned int index)
    {
        return ptr[index];
    }
    const T& ElementAt(T* ptr, unsigned int index) const
    {
        return ptr[index];
    }
};
template <class T> struct IsNotArray {};
```

問題是無論 `smart pointer` 是否指向 `array`，都會與另一個 policy：`destruction`（解構）產生不良互動。是的，你必須使用 `delete` 來摧毀指標所指物件，卻必須使用 `delete[]` 來摧毀指標所指的 `object array`。

兩個 policies 之間如果沒有互相影響，才稱為正交（*orthogonal*）。根據這個定義，`Array` 和 `Destroy` policies 不是正交。

如果你仍然需要將 `array` 的生成和摧毀設為獨立的 policy，你得建立一個讓它們溝通的辦法。你必須讓 `Array` policy 除了提供一個函式外，還提供一個 `bool` 常數，並將它傳入 `Destroy` policy。這會使 `Array` 和 `Destroy` 的設計變成更複雜，而且不由得多了些約束條件（*constraints*）。

非正交的 policies 是不完美的設計，應該儘量避免，因為這樣的設計會降低編譯期型別安全性（*type safety*），並造成 `host class` 和 `policy class` 的設計更加複雜。

如果你必須使用非正交的 policies，請儘可能藉著「把 `policy class` 當作引數傳給其他 `policy class` `template function`」來降低相依性。這樣一來你還是可以從 `template-based` 介面帶來的彈性中獲得利益。剩下的缺點就是，policy 必須曝露它的某些實作細節給其他 policy，這會降低封裝性。

1.13 摘要

「設計」就是一種「選擇」。大多數時候我們的困難並不在於找不到解決方案，而是有太多解決方案。你必須知道哪一組方案可以圓滿解決問題。大至架構層面，小至程式碼片段，都需要抉擇。此外，抉擇是可以組合的，這給設計帶來了可怕的多樣性。

為了在合理大小的程式碼中因應設計的多樣性，我們應該發展出一個以設計為導向（*design oriented*）的程式庫，並在其運用一些特別技術。這些被特意構想出來用以支援巨大彈性的程

式碼產生器，由小量基本裝置（`primitive device`）組合而成。程式庫本身供應有一定數量的基本裝置。此外程式庫也供應一些用以建立基本裝置的規格，因此客端（`client`）可以建造出自己想要的裝置。這基本上使得 `policy-based design` 成為開放式架構。這些基本裝置我們稱為 `policies`，其實作品則被稱為 `policy classes`。

`Policies` 機制由 `templates` 和多重繼承組成。一個 `class` 如果使用了 `policies`，我們稱其為 `host class`，那是一個擁有多個 `template` 參數（通常是「`template template` 參數」）的 `class template`，每一個參數代表一個 `policy`。`Host class` 的所有機能都來自 `policies`，運作起來就像是一個聚合了數個 `policies` 的容器。

環繞著 `policies` 而設計出來的 `classes`，支援「可擴充的行為」和「優雅的機能削減」。由於採用「`public` 繼承」之故，`policy` 得以透過 `host class` 提供追加機能。而 `host classes` 也能運用「`policy` 提供的選擇性機能」實作出更豐富的功能。如果某個選擇性機能不存在，`host class` 還是可以成功編譯，前提是該選擇性機能未被真正用上。

`Policies` 的最大威力來自於它們可以互相混合搭配。一個 `policy-based class` 可以組合「`policies` 實作出來的某些簡單行為」而提供非常多的行為。這極有效地使 `policies` 成為對付「設計期多樣性」的好武器。

透過 `policy classes`，你不但可以訂製行為，也可以訂製結構。這個重要的性質使得 `policy-based design` 超越了簡單的型別泛化（`type genericity`）——後者對於容器類別（`container classes`）效力卓著。

型別轉換對 `policy-based classes` 而言也是一種彈性的表現。如果你採用 `policy-by-policy` 拷貝方式，每個 `policy` 都能藉由提供適當的轉型建構式或轉型運算子（甚至兩者都提供）來控制它自己接受哪個 `policies`，或它自己可以轉換為哪個 `policy`。

欲將 `class` 分解為 `policies` 時，你應該遵守兩條重要準則。第一，把你的 `class` 內的「設計決定」局部化、命名、分離出來。這也許是一種取捨，也許需要以其他方式明智地完成。第二，找出正交的 `policies` ——也就是彼此之間無交互作用、可獨立更動的 `policies`。

2

技術

Techniques

本章呈現許多貫穿本書的 C++ 技術。爲了在各式各樣的情境 (context) 中都有用，它們傾向於泛化 (一般化，general) 和可復用 (reusable)，如此便可在其他情境中找出他們的應用。有些技術如 `partial template specialization` (模板偏特化) 是語言本身的特性，有些如「編譯期 assertions」則需藉由程式碼實作出來。

本章之中你將了解下列這些技術和工具：

- `Partial template specialization` (模板偏特化)
- `Local classes` (區域類別)
- 型別和數值之間的映射 (`Int2Type` 和 `Type2Type class templates`)
- 在編譯期察覺可轉換性 (convertibility) 和繼承性
- 型別資訊，以及一個容易上手的 `std::type_info` 外覆類別 (wrapper)
- `Select class template`。這是一個工具，可在編譯期間根據某個 `bool` 狀態選擇某個型別
- `Traits`，一堆 traits 技術集合，可施行於任何 C++ 型別身上

如果分開來看，每個技術和其所用之程式碼也許都不怎麼樣；它們都由 5~10 行淺顯易懂的程式碼組成。然而這些技術有一個重要特性：它們沒有極限。也就是說，你可以把它們組合成一個高階慣用手法 (idioms)。當它們合作，便形成一個強壯的服務基礎，可協助我們建立比較強壯的結構。

這些技術都帶有範例，所以討論起來並不枯燥。閱讀本書其餘部分時，也許你會回頭參考本章。

2.1 編譯期 (Compile-Time) Assertions

隨著泛型編程在 C++ 大行其道，更好的靜態檢驗 (static checking) 以及更好的可訂製型錯誤訊息 (customizable error messages) 的需求浮現了出來。

舉個例子，假設你發展出一個用來作安全轉型 (safe casting) 的函式。你想將某個型別轉爲其他型別，而爲了確保原始資訊被保留，較大型別不能轉型爲較小型別。

```
template <class To, class From>
```

```

To safe_reinterpret_cast(From from)
{
    assert(sizeof(From) <= sizeof(To));
    return reinterpret_cast<To>(from);
}

```

你可以像運用「C++ 內建之型別轉換操作」一樣地呼叫上述函式：

```

int i = ...;
char* p = safe_reinterpret_cast<char*>(i);

```

你必須明白指定 `To` 這個 `template` 引數；編譯器會根據 `i` 的型別推導出另一個 `template` 引數 `From`。藉由上述的「大小比較」`assertion` 動作，便可確定「標的型別」足以容納「源端型別」的所有 bits。如此一來上述程式碼便可達到正確的型別轉換⁶，或導致一個執行期 `assertion`。

很顯然，我們都希望錯誤能夠在編譯期便被偵測出來。一則因為轉型動作可能是你的程式中偶而執行的分支，當你將程式移植到另一個編譯器或平台時，你可能不會記住每一個潛在的不可移植部分（譯註：上例 `reinterpret_cast` 就是不可移植的），於是留下潛伏臭蟲，而它可能在用戶面前讓你出醜。

這裡有一道曙光。算式（`expression`）在編譯期評估所得結果是個定值（常數），這意味你可以利用編譯器（而非程式碼）來作檢查。這個想法是傳給編譯器一個語言構造（`language construct`），如果是非零算式便合法，零算式則非法。於是當你傳入一個算式而其值為零時，編譯器會發出一個編譯期錯誤。

最簡單的方式稱為 *compile-time assertions*（Van Horn 1997），在 C 和 C++ 語言中都可以良好運作。它依賴一個事實：大小為零的 `array` 是非法的。

```

#define STATIC_CHECK(expr) { char unnamed[(expr) ? 1 : 0]; }

```

現在如果你這樣寫：

```

template <class To, class From>
To safe_reinterpret_cast(From from)
{
    STATIC_CHECK(sizeof(From) <= sizeof(To));
    return reinterpret_cast<To>(from);
}
...
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);

```

而如果在你的系統中，指標大小大於字元，編譯器會抱怨你「正試著產生一個長度為零的 `array`」。

問題是，你收到的錯誤訊息無法表達正確資訊。「無法產生一個長度為零的 `array`」這句話無法暗示「`char` 型別用來持有一個指標實在太小了」。供應「可訂製、可移植的錯誤訊息」是很困難的一件事。錯誤訊息之間並沒有什麼必須遵循的規則，端視編譯器而定。例如，如果錯

⁶ 有了 `safe_reinterpret_cast`，在大多數機器上的確如此，但不能完全保證。

誤訊息指出一個未定義變數，該變數名稱不一定得出現在錯誤訊息裡頭。

較好的解法是依賴一個名稱帶有意義的 `template`（因為，很幸運地，編譯器會在錯誤訊息中指出 `template` 名稱）：

```
template<bool> struct CompileTimeError;
template<> struct CompileTimeError<true> {};

#define STATIC_CHECK(expr) \
    (CompileTimeError<(expr) != 0>())
```

`CompileTimeError` 需要一個非型別參數（一個 `bool` 常數），而且它只針對 `true` 有所定義。如果你試著具現化 `CompileTimeError<false>`，編譯器會發出 "Undefined specialization `CompileTimeError<false>`" 訊息。這個訊息比錯誤訊息好，因為它是我們故意製造的，不是編譯器或程式的臭蟲。

當然這其中還有很多改善空間。如何訂製錯誤訊息？我的想法是傳入一個額外引數給 `STATIC_CHECK`，並讓它在錯誤訊息中出現。唯一的缺點是這個訂製訊息必須是合法的 C++ 辨識符號（不能間雜空白、不能以數字開頭...）。這個想法引出了一個改良版 `CompileTimeError`，如下所示。此後 `CompileTimeError` 之名不再適用，改為 `CompileTimeChecker` 更具意義：

```
template<bool> struct CompileTimeChecker
{
    CompileTimeChecker(...);    // 譯註：這是 C/C++ 支援的非定量任意參數列
};
template<> struct CompileTimeChecker<false> { };
#define STATIC_CHECK(expr, msg) \
    {\
        class ERROR_##msg {}; \    // 譯註：## 是個罕被使用的 C++ 運算子
        (void)sizeof(CompileTimeChecker<(expr)>(ERROR_##msg())); \
    }
```

假設 `sizeof(char) < sizeof(void*)`（注意，*C++ Standard* 並不保證這一定為真）。讓我們看看當你寫出下面這段程式碼，會發生甚麼事：

```
template <class To, class From>
To safe_reinterpret_cast(From from)
{
    STATIC_CHECK(sizeof(From) <= sizeof(To),
        Destination_Type_Too_Narrow);
    return reinterpret_cast<To>(from);
}
...
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);
```

巨集被處理完畢後，上述的 `safe_reinterpret_cast` 會被展開成下列樣子：

```
template <class To, class From>
To safe_reinterpret_cast(From from)
{
    {
        class ERROR_Destination_Type_Too_Narrow {};
        (void)sizeof(
            CompileTimeChecker<(sizeof(From) <= sizeof(To))>(
                ERROR_Destination_Type_Too_Narrow());
    }
    return reinterpret_cast<To>(from);
}
```

這段程式定義了一個名為 `ERROR_Destination_Type_Too_Narrow` 的 local class，那是一個空類別。然後生成一個型別為 `CompileTimeChecker<(sizeof(From) <= sizeof(To))>` 的暫時物件，並以一個型別為 `ERROR_Destination_Type_Too_Narrow` 的暫時物件加以初始化。最終，`sizeof` 會量測出這個物件的大小。

這是個小技巧。`CompileTimeChecker<true>` 這個特化體有一個可接受任何參數的建構式；它是一個「參數列為簡略符號 (ellipsis)」的函式。這意味如果編譯期的算式評估結果為 `true`，這段程式碼就有效。如果大小比較結果為 `false`，就會有編譯期錯誤發生：因為編譯器找不到將 `ERROR_Destination_Type_Too_Narrow` 轉成 `CompileTimeChecker<false>` 的方法。最棒的是編譯器能夠輸出如下正確訊息："Error: Cannot convert ERROR_Destination_Type_Too_Narrow to CompileTimeChecker<false>"，這真是太棒了！

2.2 Partial Template Specialization (模板偏特化)

Partial template specialization 讓你在 template 的所有可能實體中特化出一組子集。讓我們先扼要解釋 template specialization。如果你有這樣一個 class template，名為 `Widget`：

```
template <class Window, class Controller>
class Widget
{
    ... generic implementation ...
};
```

你可以像下面這樣明白加以特化：

```
template <>
class Widget<ModalDialog, MyController>
{
    ... specialized implementation ...
};
```

其中 `ModalDialog` 和 `MyController` 是你另外定義的 classes。

有了這個 `Widget` 特化定義之後，如果你定義 `Widget<ModalDialog, MyController>` 物件，編譯器就使用上述定義，如果你定義其他泛型物件，編譯器就使用原本的泛型定義。

然而有時候你也許想要針對任意 `Window` 並搭配一個特定的 `MyController` 來特化 `Widget`。這時就需要 **Partial Template Specialization** 機制：

```
// Partial specialization of Widget
template <class Window>           // 譯註：Window 仍是泛化
class Widget<Window, MyController> // 譯註：MyController 是特化
{
    ... partially specialized implementation ...
};
```

通常在一個 `class template` 偏特化定義中，你只會特化某些 `template` 參數而留下其他泛化參數。當你在程式中具體實現上述 `class template`，編譯器會試著找出最匹配的定義。這個尋找過程十分複雜精細，允許你以富創意的方式來進行偏特化。例如，假設你有一個 `Button class template`，它有一個 `template` 參數；那麼，你不但可以拿任意 `Window` 搭配特定 `MyController` 來特化 `Widget`，還可以拿任意 `Button` 搭配特定 `MyController` 來偏特化 `Widget`：

```
template <class ButtonArg>
class Widget<Button<ButtonArg>, MyController>
{
    ... further specialized implementation ...
};
```

如你所見，偏特化的能力十分令人驚訝。當你具現化一個 `template` 時，編譯器會把目前存在的偏特化和全特化 `templates` 作比較，並找出其中最合適者。這樣的機制給了我們很大彈性。不幸的是偏特化機制不能用在函式身上（不論成員函式或非成員函式），這樣多少會降低一些你所能作出來的彈性和粒度（*granularity*）。

- 雖然你可以全特化 `class template` 中的成員函式，但你不能偏特化它們。
- 你不能偏特化 `namespace-level (non-member)` 函式。最接近「`namespace-level template functions`」偏特化機制的是函式重載 — 就實際運用而言，那意味你對「函式參數」（而非回返值型別或內部所用型別）有很精緻的特化能力。例如：

```
template <class T, class U> T Fun(U obj);           // primary template
// template <class U> void Fun(void, U>(U obj); // illegal partial
// specialization
template <class T> T Fun (Window obj);             // legal (overloading)
```

如果沒有偏特化，編譯器設計者的日子肯定會好過一些，但卻對程式開發者造成不好的影響。稍後介紹的一些工具（如 `Int2Type` 和 `Type2Type`）都顯現偏特化的極限。本書頻繁運用偏特化，`typelist` 的所有設施（第 3 章）幾乎都建立在這個機制上。

2.3 區域類別 (Local Classes)

這是一個有趣而少人知道的 C++ 特性。你可以在函式中定義 `class`，像下面這樣：

```
void Fun()
{
    class Local
    {
        ... member variables ...
        ... member function definitions ...
    };
    ... code using Local ...
}
```

不過還是有些限制，`local class` 不能定義 `static` 成員變數，也不能存取 `non-static` 區域變數。`local classes` 令人感興趣的是，可以在 `template` 函式中被使用。定義於 `template` 函式內的 `local classes` 可以運用函式的 `template` 參數。以下所列程式碼中有一個 `MakeAdapter` `template function`，可以將某個介面轉接為另一個介面。`MakeAdaptery` 在其 `local class` 的協助下實作出一個介面。這個 `local class` 內有泛化型別的成員。

```
class Interface
{
public:
    virtual void Fun() = 0;
    ...
};

template <class T, class P>
Interface* MakeAdapter(const T& obj, const P& arg)
{
    class Local : public Interface
    {
    public:
        Local(const T& obj, const P& arg)
            : obj_(obj), arg_(arg) {}
        virtual void Fun()
        {
            obj_.Call(arg_);
        }
    private:
        T obj_;
        P arg_;
    };
    return new Local(obj, arg);
}
```

事實證明，任何運用 local classes 的手法，都可以改用「函式外的 template classes」來完成。換言之並非一定得 local classes 不可。不過 local classes 可以簡化實作並提高符號的地域性。

Local classes 倒是有個獨特性質：它們是「最後一版」（也即 Java 口中的 final）。外界不能繼承一個隱藏於函式內的 class。如果沒有 local classes，爲了實現 Java final，你必須在編譯單元（譯註：也就是個別檔案）中加上一個無具名的命名空間（namespace）。

我將在第 11 章運用 local classes 產生所謂的「彈簧墊」函式（trampoline functions）。

2.4 常整數映射為型別 (Mapping Integral Constants to Types)

下面是最初由 Alexandrescu (2000b) 提出的一個簡單 template，對許多泛型編程手法很有幫助：

```
template <int v>
struct Int2Type
{
    enum { value = v };
};
```

Int2Type 會根據引數所得的不同數值來產生不同型別。這是因爲「不同的 template 具現體」本身便是「不同的型別」。因此 Int2Type<0>不同於 Int2Type<1>，以此類推。用來產生型別的那個數值是一個列舉元（enumerator）。

當你想把常數視同型別，便可採用上述的 Int2Type。這麼一來便可根據編譯期計算出來的結果選用不同的函式。實際上你可以運用一個常數達到靜態分派（static dispatching）功能。

一般而言，符合下列兩個條件便可使用 Int2Type：

- 有必要根據某個編譯期常數呼叫一個或數個不同的函式。
- 有必要在編譯期實施「分派」（dispatch）。

如果打算在執行期進行分派（dispatch），可使用 if-else 或 switch 述句。大部分時候其執行期成本都微不足道。然而你還是無法常常那麼做，因爲 if-else 述句要求每一個分支都得編譯成功，即使該條件測試在編譯期才知道。困惑了嗎？讀下去！

假想你設計出一個泛形容器 NiftyContainer，它將元素型別參數化：

```
template <class T> class NiftyContainer
{
    ...
};
```

現在假設 NiftyContainer 內含指標，指向型別爲 T 的物件。爲了複製 NiftyContainer 裡頭的某個物件，你想呼叫其 copy 建構式（針對 non-polymorphic 型別）或虛擬函式 clone()（針對 polymorphic 型別）。你以一個 boolean template 參數取得使用者所提供的資訊：

```

template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    void DoSomething()
    {
        T* pSomeObj = ...;
        if (isPolymorphic)
        {
            T* pNewObj = pSomeObj->Clone();
            ... polymorphic algorithm ... (多型演算法)
        }
        else
        {
            T* pNewObj = new T(*pSomeObj);    // 譯註：copy 建構式
            ... non-polymorphic algorithm ... (非多型演算法)
        }
    }
};

```

問題是，編譯器不會讓你僥倖成功。如果多型演算法使用 `pObj->Clone()`，那麼面對任何一個未曾定義成員函式 `Clone()` 之型別，`NiftyContainer::DoSomething()` 都無法編譯成功。雖然編譯期間很容易知道哪一條分支會被執行起來，但這和編譯器無關，因為即使最佳化工具可以評估出哪一條分支不會被執行，編譯器還是會勤勞地編譯每個分支。如果你試著呼叫 `NiftyContainer<int, false>` 的 `DoSomething()`，編譯器會停在 `pObj->Clone()` 處並說「嗨，不行唷」。

上述的 non-polymorphic 部分也有可能編譯失敗。如果 `T` 是個 polymorphic 型別，而上述的 non-polymorphic 程式分支想作 `new T(*pObj)` 動作，這樣也有可能編譯失敗。舉個實例，如果 `T` 藉著「把 `copy` 建構式置於 `private` 區域以產生隱藏效果」，就像一個有良好設計的 polymorphic class 那樣，那麼便有可能發生上述的失敗情況。

如果編譯器不去理會那個不可能被執行的程式碼就好了，然而目前情況下是不可能的。甚麼才是令人滿意的解決方案呢？

事實證明有很多解法，而 `Int2Type` 提供了一個特別明確的方案。它可以把 `isPolymorphic` 這個型別的 `true` 和 `false` 轉換成兩個可資區別的不同型別。然後程式中便可以運用 `Int2Type<isPolymorphic>` 進行函式重載。瞧，可不是嗎！

```

template <typename T, bool isPolymorphic>
class NiftyContainer
{
private:
    void DoSomething(T* pObj, Int2Type<true>)
    {
        T* pNewObj = pObj->Clone();
        ... polymorphic algorithm ...
    }
}

```

```

void DoSomething(T* pObj, Int2Type<false>)
{
    T* pNewObj = new T(*pObj);
    ... nonpolymorphic algorithm ...
}
public:
    void DoSomething(T* pObj)
    {
        DoSomething(pObj, Int2Type<isPolymorphic>());
    }
};

```

Int2Type 是一個用來「將數值轉換為型別」的方便手法。有了它，你便可以將該型別的一個暫時物件傳給一個重載函式 (overloaded function)，後者實現必要的演算法。（譯註：這種手法在 STL 中亦有大量實現，唯形式略有不同；詳見 STL 源碼，或《STL 源碼剖析》by 侯捷）

這個小技巧之所以有效，最主要的原因是，編譯器並不會去編譯一個未被用到的 template 函式，只會對它做文法檢查。至於此技巧之所以有用，則是因為在 template 程式碼中大部分情形下你需要在編譯期作流程分派 (dispatch) 動作。

你會在 Loki 的數個地方看到 Int2Type 的運用，尤其是本書第 11 章：Multimethods。在那兒，template class 是一個雙分派 (double-dispatch) 引擎，運用 bool template 參數決定是否要支援對稱性分派 (symmetric dispatch)。

2.5 型別對型別的映射 (Type-to-Type Mapping)

就如 2.2 節所說，並不存在 template 函式的偏特化。然而偶爾我們需要模擬出類似機制。試想下面的程式：

```

template <class T, class U>
T* Create(const U& arg)
{
    return new T(arg);
}

```

Create() 會將其參數傳給建構式，用以產生一個新物件。

現在假設你的程式有個規則：Widget 物件是你碰觸不到的老舊程式碼，它需要兩個引數才能建構出物件來，第二引數固定為 -1。Widget 衍生類別則沒有這個問題。

現在你該如何特化 Create()，使它能夠獨特地處理 Widget？一個明顯方案是另寫出一個 CreateWidget() 來專門處理。但這麼一來你就沒有一個統一的介面用來生成 Widgets 和其衍生物件。這會使得 Create() 在任何泛型程式中不再有用。

由於你無法偏特化一個函式，因此無法寫出下面這樣的程式碼：

```

// Illegal code — don't try this at home
template <class U>
Widget* Create<Widget, U>(const U& arg)

```

```

{
    return new Widget(arg, -1);
}

```

由於函式缺乏偏特化機制，因此（再一次地）你只有一樣工具可用：多載化（重載）機制。我們可以傳入一個型別為 `T` 的暫時物件，並以此進行重載：

```

template <class T, class U>
T* Create(const U& arg, T /* dummy */)
{
    return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Widget /* dummy */)
{
    return new Widget(arg, -1);
}

```

這種解法會輕意建構未被使用的複雜物件，造成額外開銷。我們需要一個輕量級機制來傳遞「型別 `T` 的資訊」到 `Create()` 中。這正是 `Type2Type` 扮演的角色，它是一個型別代表物，一個可以讓你傳給多載化函式的輕量級 ID。`Type2Type` 定義如下：

```

template <typename T>
struct Type2Type
{
    typedef T OriginalType;
};

```

它沒有任何數值，但其不同型別卻足以區分各個 `Type2Type` 實體，這正是我們所要的。現在你可以這麼寫：

```

// An implementation of Create relying on overloading
// and Type2Type
template <class T, class U>
T* Create(const U& arg, Type2Type<T>)
{
    return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Type2Type<Widget>)
{
    return new Widget(arg, -1);
}
// Use Create()
String* pStr = Create("Hello", Type2Type<String>());
Widget* pW = Create(100, Type2Type<Widget>());

```


`Create()` 的第二參數只是用來選擇適當的重載函式，現在你可以令各種 `Type2Type` 實體對應於你的程式中的各種型別，並根據不同的 `Type2Type` 實體來特化 `Create()`。

2.6 型別選擇 (Type Selection)

有時候，泛型程式需要根據一個 `boolean` 變數來選擇某個型別或另一型別。

2.4 節討論的 `NiftyContainer` 例子中，你也許會以一個 `std::vector` 作為後端儲存結構。很顯然，面對 `polymorphic` (多型) 型別，你不能儲存其物件實體，必須儲存其指標。但如果面對的是 `non-polymorphic` (非多型) 型別，你可以儲存其實體，因為這樣比較有效率。

在你的 `class template` 中：

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
};
```

你需要存放一個 `vector<T*>` (如果 `isPolymorphic` 為 `true`) 或 `vector<T>` (如果 `isPolymorphic` 為 `false`)。根本而言，你需要根據 `isPolymorphic` 來決定將 `ValueType` 定義為 `T*` 或 `T`。你可以使用 **traits class template** (Alexandrescu 2000a) 如下：

```
template <typename T, bool isPolymorphic>
struct NiftyContainerValueTraits
{
    typedef T* ValueType;
};
template <typename T>
struct NiftyContainerValueTraits<T, false>
{
    typedef T ValueType;
};
template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    typedef NiftyContainerValueTraits<T, isPolymorphic> Traits;
    typedef typename Traits::ValueType ValueType;
};
```

這樣的做法其實笨拙難用，此外它也無法擴充：針對不同的型別的選擇，你都必須定義出專屬的 **traits class template**。

Loki 提供的 `Select class template` 可使型別的選擇立時可用。它採用偏特化機制 (partial template specialization)：

```

template <bool flag, typename T, typename U>
struct Select
{
    typedef T Result;
};
template <typename T, typename U>
struct Select<false, T, U>
{
    typedef U Result;
};

```

其運作方式是：如果 `flag` 為 `true`，編譯器會使用第一份泛型定義式，因此 `Result` 會被定義成 `T`。如果 `flag` 為 `false`，那麼偏特化機制會進場運作，於是 `Result` 被定義成 `U`。

現在你可以更方便地定義 `NiftyContainer::ValueType` 了：

```

template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    typedef Select<isPolymorphic, T*, T>::Result    ValueType;
    ...
};

```

2.7 編譯期間偵測可轉換性 (Convertibility) 和繼承性 (Inheritance)

實作 `template functions` 和 `template classes` 時我常常發現一個問題：面對兩個陌生的型別 `T` 和 `U`，如何知道 `U` 是否繼承自 `T` 呢？在編譯期間發掘這樣的關係，實在是實作泛型程式庫的一個優化關鍵。在泛型函式中，如果你確知某個 `class` 實作有某個介面，你便可以採用某個最佳演算法。在編譯期發現這樣的關係，意味不必使用 `dynamic_cast` — 它會耗損執行期效率。

發掘繼承關係，靠的是一個用來偵測可轉換性 (convertibility) 的更一般化機制。這裡我們面臨更一般化的問題：如何測知任意型別 `T` 是否可以自動轉換為型別 `U`？

有個方案可以解決問題，並且只需仰賴 `sizeof`。`sizeof` 有著驚人的威力：你可以把 `sizeof` 用在任何算式 (expression) 身上，不論後者有多複雜。`sizeof` 會直接傳回大小，不需拖到執行期才評估。這意味 `sizeof` 可以感知重載 (overloading)、模板具現 (template instantiation)、轉換規則 (conversion rules)，或任何可發生於 C++ 算式身上的機制。事實上 `sizeof` 背後隱藏了一個「用以推導算式型別」的完整設施。最終 `sizeof` 會丟棄算式並傳回其大小⁷。

「偵測轉換能力」的想法是：合併運用 `sizeof` 和重載函式。我們提供兩個重載函式：其中一

⁷目前正有一份提案，準備為 C++ 語言加入 `typeof` 運算子，它會傳回一個算式的型別。有了這個 `typeof` 運算子，泛型程式將會更好寫，也更易被了解。Gnu C++ 已實作出 `typeof` 作為語言擴充功能。很顯然 `typeof` 和 `sizeof` 擁有共同的後端實作，因為 `sizeof` 也需要判斷型別。

個接受 `U` (`U` 型別代表目前討論中的轉換標的)，另一個接受「任何其他型別」。我們以型別 `T` 的暫時物件來喚起這些重載函式，而「`T` 是否可轉換為 `U`」正是我們想知道的。如果接受 `U` 的那個函式被喚起，我們就知道 `T` 可轉換為 `U`；否則 `T` 便無法轉換為 `U`。為了知道哪一個函式被喚起，我們對這兩個重載函式安排大小不同的回返型別，並以 `sizeof` 來區分其大小。型別本身無關緊要，重要的是其大小必須不同。

讓我們先建立兩個不同大小的型別。很顯然 `char` 和 `long double` 的大小不同，不過 C++ 標準規格書並未保證此事，所以我想到一個極其簡單的作法：

```
typedef char Small;
class Big { char dummy[2]; };
```

根據定義，`sizeof(Small)` 是 1。而 `Big` 的大小肯定比 1 還大，這正是我們所需要的保證。

接下來需要兩個重載函式，其一如先前所說，接受一個 `U` 物件並傳回一個 `Small` 物件：

```
Small Test(U);
```

但，接下來，我該如何寫出一個可接受任何其他種物件的函式呢？`template` 並非解決之道，因為 `template` 總是要求最佳匹配條件，因而遮蔽了轉換動作。我需要一個「比自動轉換稍差」的匹配，也就是說我需要一個「唯有在自動轉換缺席情況下」才會雀屏中選的轉換。我很快看了一下施行於函式呼叫的各種轉換規則，然後發現所謂的「簡略符號比對」準則，那是最壞的情況了，位於整個列表的最底端。於是我寫出這樣一個函式：

```
Big Test(...);
```

(呼叫一個帶有簡略符號的函式並傳入一個 C++ 物件，無人知道結果會如何。不過沒關係，我們並不真正呼叫這個函式，它甚至沒被實作出來。還記得嗎，`sizeof` 並不對其引數求值)

現在我們傳一個 `T` 物件給 `Test()`，並將 `sizeof` 施行於其傳回值身上：

```
const bool convExists = sizeof(Test(T())) == sizeof(Small);
```

就是這樣！`Test()` 會取得一個 *default* 建構物件 `T()`，然後 `sizeof` 會取得此一算式結果的大小，可能是 `sizeof(Small)` 或 `sizeof(Big)`，取決於編譯器是否找到轉換方式。

這裡還有一個小問題。萬一 `T` 讓自己的 *default* 建構式為 `private`，那麼 `T()` 會編譯失敗，我們所有的舞台支架也將倒塌。慶幸的是有一個簡單解法：以一個「稻草人函式」(strawman function) 傳回一個 `T` 物件。還記得嗎，我們處於 `sizeof` 的神奇世界中，並不會真有任何算式被求值 (evaluated)。本例之中編譯器高興，我們也高興。

```
T MakeT(); // not implemented
const bool convExists = sizeof(Test(MakeT())) == sizeof(Small);
```

(順便提一下，像 `MakeT()` 和 `Test()` 這樣的函式，你能在它們身上做多少事情？它們不只沒做任何事情，甚至根本不真正存在。這不是很好玩嗎？)

現在讓它運作，把所有東西以 `class template` 包裝起來，隱藏「型別推導」的所有細節，只曝露結果：

```
template <class T, class U>
class Conversion
{
    typedef char Small;
    class Big { char dummy[2]; };
    static Small Test(U);
    static Big Test(...);
    static T MakeT();
public:
    enum { exists =
        sizeof(Test(MakeT())) == sizeof(Small) };
};
```

下面程式碼用來測試上述的 `Conversion` `class template`：

```
int main()
{
    using namespace std;
    cout << Conversion<double, int>::exists << ' '
        << Conversion<char, char*>::exists << ' '
        << Conversion<size_t, vector<int> >::exists << ' ';
}
```

這個小程序式會印出 "100"。注意，雖然 `std::vector` 實作出一個接受 `size_t` 引數的建構式，但上述轉換測試卻傳回 0，因為該建構式是 `explicit`（譯註：explicit 建構式無法擔任轉換函式）

我們在 `Conversion` 中更實作出兩個常數：

- `exists2Way`，表示 `T` 和 `U` 之間是否可以雙向轉換。例如 `int` 和 `double` 可以雙向轉換。使用者自定型別也可以實作出這樣的轉換。
- `sameType`；如果 `T` 和 `U` 是相同型別，這個值便為 `true`。

```
template <class T, class U>
class Conversion
{
    ... as above ...
    enum { exists2Way = exists &&
        Conversion<U, T>::exists };
    enum { sameType = false };
};
```

我們也可以透過 `Conversion` 的偏特化（`partial specialization`）來實作出 `sameType`：

```
template <class T>
class Conversion<T, T>
{
```

```
public:
    enum { exists = 1, exists2Way = 1, sameType = 1 };
};
```

最後，讓我們回頭看看，有了 `Conversion` 的幫助，要決定兩個 `classes` 之間是否存在繼承關係，變得很容易：

```
#define SUPERSUBCLASS(T, U) \
    (Conversion<const U*, const T*>::exists && \
     !Conversion<const T*, const void*>::sameType)
```

如果 `U` 是 `public` 繼承自 `T`，或 `T` 和 `U` 是同一型別，那麼 `SUPERSUBCLASS(T,U)` 會傳回 `true`。當 `SUPERSUBCLASS(T, U)` 對 `const U*` 和 `const T*` 作「可轉換性」評估時，只有三種情況下 `const U*` 可以隱式轉換為 `const T*`：

1. `T` 和 `U` 是同一種型別。
2. `T` 是 `U` 的一個 `unambiguous`（不模稜兩可的、非歧義的）`public base`。
3. `T` 是 `void`。

第三種情況可以在前述第二次測試中解決掉。如果把第一種情況（`T` 和 `U` 是同一型別）視為 `is-a` 的退化，實作時會很有用，因為實用場合中你常常可以將某個 `class` 視為它自己的 `superclass`。如果你需要更嚴謹的測試，可以這麼寫：

```
#define SUPERSUBCLASS_STRICT(T, U) \
    (SUPERSUBCLASS(T, U) && \
     !Conversion<const T, const U>::sameType)
```

為何這些程式碼都加上 `const` 飾詞？原因是我們不希望因 `const` 而導致轉型失敗。如果 `template` 程式碼實施 `const` 兩次（對一個已經是 `const` 的型別而言），第二個 `const` 會被忽略。簡單地說，藉由在 `SUPERSUBCLASS` 中使用 `const`，我們得以更安全一些。

為甚麼選用 `SUPERSUBCLASS` 而不是更可愛的名稱如 `BASE_OF` 或 `INHERITS` 之類？這是基於一個非常實際的理由。一開始 `Loki` 對它的命名是 `INHERITS`，但是當 `INHERITS(T,U)` 運作時，出現了一個問題：它說的是 `T` 繼承 `U` 或相反呢？改名為 `SUPERSUBCLASS(T,U)` 之後，誰先誰後就變得很清楚。

2.8 type_info 的一個外覆類別 (Wrapper)

標準 C++ 提供了一個 `std::type_info` class，使你能夠於執行期間查詢物件型別。通常 `type_info` 必須和 `typeid` 運算子並用，後者會傳回一個 `reference`，指向一個 `type_info` 物件：

```
void Fun(Base* pObj)
{
    // Compare the two type_info objects corresponding
    // to the type of *pObj and Derived
```

```

        if (typeid(*pObj) == typeid(Derived))
        {
            ... aha, pObj actually points to a Derived object ...
        }
        ...
    }

```

除了支援比較運算 `operator==` 和 `operator!=`，`type_info` 還提供兩個函式如下：

- `name()`，傳回一個 `const char*` 代表型別名稱。但是並無標準方法可以將 `class` 名稱對應於字串，所以你不應該期望 `typeid(Widget)` 會傳回 `"Widget"`。一個可資遵循（但不至於獲獎）的作法是讓 `type_info::name()` 對所有型別都傳回空字串。
- `before()`，帶來 `type_info` 物件的次序關係。程式員可運用 `type_info::before()` 對 `type_info` 物件建立索引。

不幸的是這些好用的 `type_info` 功能被包裝得難以發揮。`type_info` 關閉了 `copy` 建構式和 `assignment` 運算子，使我們不可能儲存 `type_info` 物件。但你可以儲存一個指向 `type_info` 物件的指標。`typeid` 傳回的物件採用 `static` 儲存方式，所以你不必擔心其壽命問題，只需擔心指標間的辨識就行了。

C++ Standard 並不保證每次呼叫（例如）`typeid(int)`，會傳回「指向同一個 `type_info` 物件」的 `reference`。如此一來你就無法比較「指向 `type_info` 物件」的指標了。因此你應該做的是儲存 `type_info` 物件，並且以 `type_info::operator==` 施行於提領後的指標上。

如果你想對 `type_info` 物件排序，再一次你必須儲存「指向 `type_info` 物件」的指標，此時你必須使用成員函式 `before()`。因此如果你想要使用 STL 的帶序容器（ordered containers），你必須寫出一個小小的仿函式（functor）並處理指標。

以上這些已經夠笨拙到足以讓我們委派一個外覆類別（wrapper class）來包裝 `type_info` 了，它應該儲存「指向 `type_info` 物件」的指標，並提供下列功能：

- `type_info` 的所有成員函式。
- *value* 語意（譯註：相對於 *reference* 語意），也就是 `public copy` 建構式和 `public assignment` 運算子。
- 定義出 `operator<` 和 `operator==`，使比較動作更完整。

Loki 已經實作出如此好用的外覆類別，名為 `TypeInfo`，概要如下：

```

class TypeInfo
{
public:
    // Constructors/destructors
    TypeInfo();    // needed for containers
    TypeInfo(const std::type_info&);

```

```

TypeInfo(const TypeInfo&);
TypeInfo& operator=(const TypeInfo&);
// Compatibility functions
bool before(const TypeInfo&) const;
const char* name() const;
private:
    const std::type_info* pInfo_;
};
// Comparison operators
bool operator==(const TypeInfo&, const TypeInfo&);
bool operator!=(const TypeInfo&, const TypeInfo&);
bool operator<(const TypeInfo&, const TypeInfo&);
bool operator<=(const TypeInfo&, const TypeInfo&);
bool operator>(const TypeInfo&, const TypeInfo&);
bool operator>=(const TypeInfo&, const TypeInfo&);

```

由於其中的「轉換建構式」接受一個 `std::type_info` 參數，所以你可以拿一個 `TypeInfo` 物件和一個 `std::type_info` 物件來比較，像這樣：

```

void Fun(Base* pObj)
{
    TypeInfo info = typeid(Derived);
    ...
    if (typeid(*pObj) == info)
    {
        ... pObj actually points to a Derived object ...
    }
    ...
}

```

許多情況下，`TypeInfo` 物件的複製和比較能力是很重要的。第 8 章的 *cloning factory* 和第 11 章的 *double-dispatch* 引擎都把 `TypeInfo` 運用得很好。

2.9 NullType 和 EmptyType

Loki 定義了兩個非常簡單的型別：`NullType` 和 `EmptyType`。你可以拿它們當作型別計算時的某種邊界標記。

`NullType` 是一個只有宣告而無定義的 class：

```

class NullType;    // no definition

```

你不能生成一個 `NullType` 物件，它只被用來表示「我不是個令人感興趣的型別」。2.10 節把 `NullType` 用在有語法需求卻無語意概念的地方（例如「`int` 指的是什麼型別」）。第 3 章的 `typelist` 以 `NullType` 標記 `typelist` 的末端，並用以傳回「找不到型別」這一訊息。

第二個輔助型別是 `EmptyType`。和你想的一樣，`EmptyType` 定義如下：

```

struct EmptyType {};

```

這是一個可被繼承的合法型別，而且你可以傳遞 `EmptyType` 物件。你可以把這個輕量級型別視為 `template` 的預設（可不理會的）參數型別。第 3 章的 `typelist` 就是這樣用它。

2.10 Type Traits

Traits 是一種「可於編譯期根據型別作判斷」的泛型技術，很像你在執行期根據數值進行判斷一樣（Alexandrescu 2000a）。眾所皆知，加上一個間接層便可解決很多工程問題，`trait` 讓你得以在「型別確立當時」以外的其他地點做出與型別相關的判斷。這會讓最終程式碼變得比較乾淨，更具可讀性，而且更好維護。

通常，當你的泛型程式需要時，你會寫出自己的 `trait templates` 和 `trait classes`。然而某些 `traits` 可應用於任何型別，它們可以幫助泛型程式員根據型別特性修改出適當的泛型碼。

舉個例子。假設你想實作 `copying` 演算法：

```
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result)
{
    for (; first != last; ++first, ++result)
        *result = *first;
}
```

理論上你並不需要實作這樣的演算法，因為這和 `std::copy()` 的功能重複了。但你也許需要針對某些型別，對這個演算法進行特化。

假設你要在一台多處理器（`multiprocessor`）機器上發展程式，它已有一個非常快的內建函式 `BitBlast()`，而你希望儘可能發揮該函式的好處。

```
// Prototype of BitBlast in "SIMD_Primitives.h"
void BitBlast(const void* src, void* dest, size_t bytes);
```

當然，`BitBlast()` 只對基本型別和簡樸的舊式結構（[譯註](#)：意指類似 C `struct` 這樣的東西）運作。你不能把 `BitBlast()` 用於擁有「`nontrivial copy` 建構式」的型別上。現在，你希望儘可能利用 `BitBlast()` 來實作前述的 `Copy` 演算法，並希望退卻至較為一般、謹慎的作法，希望適用於各種精巧型別。如此一來拷貝動作面對基本型別時便會「自動」快速執行。

你需要做兩個測試（[譯註](#)：以下兩變數是前述 `Copy()` 的參數）：

- `InIt` 和 `OutIt` 是一般指標嗎？（相對於較華麗、需高度技巧的迭代器（[iterator](#)）而言）。
- `InIt` 和 `OutIt` 所指的型別可以進行 `bitwise copy`（位元逐一拷貝）嗎？

如果你能夠在編譯期找出這兩個問題的答案，而且答案都是 `yes`，那麼便可以採用 `BitBlast()`，否則就只能以一般迴圈來完成。

Type traits 有助於解決這樣的問題。本章的 `type traits` 歸功於 `Boost`（一個 C++ 程式庫）實作出來的許多 `type traits`。

2.10.1 實作出 Pointer Traits

Loki 定義了一個 class template `TypeTraits`，收納很多泛型 traits。`TypeTraits` 內部使用 `template specialization`（模板特化）並將結果顯露出來。

大部分 type traits 的實作都需倚賴 `template` 的全特化或偏特化（2.2 節）。例如下面這段程式碼用來判斷型別 `T` 是否為指標：

```
template <typename T>
class TypeTraits
{
private:
    template <class U> struct PointerTraits
    {
        enum { result = false };
        typedef NullType PointeeType;
    };
    template <class U> struct PointerTraits<U*>
    {
        enum { result = true };
        typedef U PointeeType;
    };
public:
    enum { isPointer = PointerTraits<T>::result };
    typedef PointerTraits<T>::PointeeType PointeeType;
    ...
};
```

其中第一個 class template 導入 `PointerTraits` 的定義，其意義是：「`T` 不是指標，且所謂被指標型別（`PointeeType`）不能拿來運用」。回憶 2.9 節所說，是的，`NullType` 只是一種用於「不能被使用」情況下的佔位型別（placeholder type）。

第二個 `PointerTraits`（粗體那一行）是上一個 `PointerTraits` 的偏特化，是一個「針對任意指標型別」的特化體。對任意指標而言，這個特化體比其泛型版本更適合作為候選人。因此如果面對的是指標，就進入這個特化體運作，因此 `result` 是 `true`，此外 `PointeeType` 會被適當定義。

現在請觀察 `std::vector::iterator` 的實作內容，它是個簡樸指標還是個精巧型別？

```
int main()
{
    const bool
        iterIsPtr = TypeTraits<vector<int>::iterator>::isPointer;
    cout << "vector<int>::iterator is " <<
        iterIsPtr ? "fast" : "smart" << '\n';
}
```

同樣道理，`TypeTraits` 也實作出一個 `isReference` 常數和一個 `ReferencedType` 型別。對於一個 reference type `T` 而言，`ReferencedType` 代表「`T` 所參考（指向）」的型別；如果 `T` 是個直率的型別（譯註：亦即 non-reference），那麼 `ReferencedType` 就是 `T` 自己。

如欲偵測 `pointers to members`（參見第 5 章相關說明），情況有點不同。我們需要特化如下：

```
template <typename T>
class TypeTraits
{
private:
    template <class U> struct PToMTraits
    {
        enum { result = false };
    };
    template <class U, class V>
    struct PToMTraits<U V::*>
    {
        enum { result = true };
    };
public:
    enum { isMemberPointer = PToMTraits<T>::result };
    ...
};
```

2.10.2 偵測基本型別（fundamental types）（譯註：或稱 primitive types）

`TypeTraits<T>` 實作出一個編譯期常數 `isStdFundamental`，用來表示 `T` 是否為基本型別。標準基本型別包括 `void` 和所有數值型別（分為浮點數和整數兩大集團）。`TypeTraits` 定義了一些常數用來表示某個型別屬於哪一分類。

讓我未雨綢繆一下，先說明第 3 章的 `typelists` 的神奇魔法。它可以偵測某個型別是否隸屬某一組型別。此刻你唯一應該知道的是：

```
TL::IndexOf<T, TYPELIST_nn (以逗號隔開的 types list)>::value
```

其中的 `nn` 代表 `type list` 中有多少個型別。上式會傳回 `T` 在 `type list` 中的位置（從零起算）；如果 `T` 不在其中就傳回 `-1`。舉個例子，如果 `TL::IndexOf<T, TYPELIST_4(signed char, short int, int, long int)>::value` 的值大於或等於零，表示 `T` 是個帶正負號的整數。

`TypeTraits` 針對基本型別，有如下定義：

```
template <typename T>
class TypeTraits
{
    ... as above ...
public:
    typedef TYPELIST_4(
        unsigned char, unsigned short int,
        unsigned int, unsigned long int)
        UnsignedInts;
    typedef TYPELIST_4(signed char, short int, int, long int)
        SignedInts;
```

```

typedef TYPELIST_3(bool, char, wchar_t) OtherInts;
typedef TYPELIST_3(float, double, long double) Floats;
enum { isStdUnsignedInt =
    TL::IndexOf<T, UnsignedInts>::value >= 0 };
enum { isStdSignedInt = TL::IndexOf<T, SignedInts>::value >= 0 };
enum { isStdIntegral = isStdUnsignedInt || isStdSignedInt ||
    TL::IndexOf<T, OtherInts>::value >= 0 };
enum { isStdFloat = TL::IndexOf<T, Floats>::value >= 0 };
enum { isStdArith = isStdIntegral || isStdFloat };
enum { isStdFundamental = isStdArith || isStdFloat ||
    Conversion<T, void>::sameType };
...
};

```

使用 `typelists` 和 `TL::IndexOf`，你會獲得一種能力，可以很快推論出型別資訊，不需要多次撰寫 `template` 特化體。如果你忍不住現在就想研究 `typelists` 和 `TL::Find` 的細節，可以先跳去看第 3 章，但是別忘了回到這裡喔。

「基本型別偵測動作」的真正實作品比這裡所說的更為完整而嚴謹，並允許廠商自行提供擴充型別（例如 `int64` 或 `long long`）。

2.10.3 最佳化的參數型別

在泛型程式碼中，你常常需要回答下列問題：任意給定一個型別 `T`，什麼是「將 `T` 物件傳入函式當作參數」的最有效作法？一般而言最有效的方法是在傳入一個精巧型別時採用 **by reference** 傳遞方式，面對純量型別時採用 **by value** 傳遞方式。純量型別由前述之數值型別、列舉型別（`enums`）、指標、指向成員之指標組成。對精巧型別而言，你應該避免額外暫時物件帶來的額外開銷（那會帶來建構式和解構式的額外呼叫動作）；對純量型別而言，你應該避免 **reference** 帶來的間接性所造成的額外開銷。

有一個細節必須謹慎處理，那就是 C++ 不允許 **references to references**。因此如果 `T` 已經是個 **reference**，千萬別對它再加一層 **reference**。

我對函式呼叫的最佳參數型別作了一些分析之後，產生以下演算法。讓我們把即將尋求的參數型別稱為 `ParameterType`：

如果 `T` 是某型別的 **reference**，則 `ParameterType` 就是 `T`，因為 C++ 不允許 **references to references**。
否則：
 如果 `T` 是個純量型別（`int`、`float` 等等），`ParameterType` 便是 `T`。因為基本型別的最佳傳遞方式是 **by value**。
 否則 `ParameterType` 將是 `T&`，因為一般「非基本型別」的最佳傳遞方式是 **by reference**。

這個演算法的一個重要成就是它可以免除 **reference-to-reference** 的錯誤——這個錯誤可能出現在你結合標準程式庫的 `bind2nd` 和 `mem_fun` 時發生（[譯註](#)：二者都是配接器，`adapters`）。

如果想以我們目前手上的技術，加上先前定義的 `referencedType` 和 `isPrimitive`，實作出 `TypeTraits::ParameterType` 是很容易的：

```

template <typename T>
class TypeTraits
{
    ... as above ...
public:
    typedef Select<isStdArith || isPointer || isMemberPointer,
                  T, ReferencedType&>::Result
                ParameterType;
};

```

不幸的是這個方案如果遇上「以 `by value` 方式傳遞列舉型別 (enums)」會失敗，因為目前已知的任何方法都無法偵測出某個型別是否為 `enum`。

本書第5章的 `Functor class template` 使用了上述的 `TypeTraits::ParameterType`。

2.10.4 卸除飾詞 (Stripping Qualifiers)

已知某個型別 `T`，你可以輕易藉由 `const T` 取得其常數版兄弟。然而，相反的動作（卸除某個型別的 `const`）就有點難度了。同樣道理，有時候你會想要卸除某個型別的 `volatile` 飾詞。

舉個例子。考慮建立一個名為 `SmartPtr` 的 `smart pointer class`（第7章會詳盡討論這玩意兒）。雖然你允許使用者產生一個 `smart pointer to const object`，例如 `SmartPtr<const Widget>`，但在內部你還是需要改變指標，令它指向 `Widget`。這種情況下你需要在 `SmartPtr` 內部維護一個由 `const Widget` 轉來的 `Widget` 物件。

實作一個「`const` 卸除器」很容易，我們再次運用 `partial template specialization`（模板偏特化）：

```

template <typename T>
class TypeTraits
{
    ... as above ...
private:
    template <class U> struct UnConst
    {
        typedef U Result;
    };
    template <class U> struct UnConst<const U>
    {
        typedef U Result;
    };
public:
    typedef UnConst<T>::Result NonConstType;
};

```

2.10.5 運用 TypeTraits

`TypeTraits` 可以協助你做出很多有趣事情。其一是你現在可以結合本章提供的一些技術來使用 `BitBlast()` (2.10 節) 並實作出 `Copy()`。你可以利用 `TypeTraits` 判斷兩個迭代器 (iterators) 的型別資訊，並以 `Int2Type` 決定呼叫（分派至）`BitBlast()` 或典型的 `copy` 函式。

```

enum CopyAlgoSelector { Conservative, Fast };

// Conservative routine-works for any type
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIt first, InIt last, OutIt result, Int2Type<Conservative>)
{
    for (; first != last; ++first, ++result)
        *result = *first;
}

// Fast routine-works only for pointers to raw data
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIt first, InIt last, OutIt result, Int2Type<Fast>)
{
    const size_t n = last--first;
    BitBlast(first, result, n * sizeof(*first));
    return result + n;
}

template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result)
{
    typedef TypeTraits<InIt>::PointeeType SrcPointee;
    typedef TypeTraits<OutIt>::PointeeType DestPointee;
    enum { copyAlgo =
        TypeTraits<InIt>::isPointer &&
        TypeTraits<OutIt>::isPointer &&
        TypeTraits<SrcPointee>::isStdFundamental &&
        TypeTraits<DestPointee>::isStdFundamental &&
        sizeof(SrcPointee) == sizeof(DestPointee) ? Fast : Conservative };

    return CopyImpl(first, last, result, Int2Type<copyAlgo>);
}

```

雖然 `Copy()` 本身沒有做很多事情，不過最有趣的其實就在這裡。enum `copyAlgo` 會自動選擇某個演算法，邏輯如下：如果兩個迭代器（iterators）都是指標，而且都指向基本型別，而且所指型別的大小一樣，那麼就使用 `BitBlast()`。最後一個條件是個有趣變形，如果你這麼做：

```

int* p1 = ...;
int* p2 = ...;
unsigned int* p3 = ...;
Copy(p1, p2, p3);

```

那麼 `Copy()` 會呼叫快速版本，雖然「源端型別」和「標的端型別」並不相同。

這個 `Copy()` 的缺點是它無法加速所有可加速的東西。假設你有一個 C struct，其內除了基本型別的資料外，沒有其他任何東西，此即所謂 *plain old data*，或稱為 **POD** 結構。C++ Standard 允許對 POD 結構採取 bitwise copy（位元逐一拷貝）動作，但 `Copy()` 卻無法偵測出操作對象是否為 POD，因此它會呼叫慢速版本。這裡你不但需要 `TypeTraits`，還需要「古典的」traits 技法，如下：

```

template <typename T> struct SupportsBitwiseCopy
{
    enum { result = TypeTraits<T>::isStdFundamental };
};
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result, Int2Type<true>)
{
    typedef TypeTraits<InIt>::PointeeType SrcPointee;
    typedef TypeTraits<OutIt>::PointeeType DestPointee;
    enum { useBitBlast =
        TypeTraits<InIt>::isPointer &&
        TypeTraits<OutIt>::isPointer &&
        SupportsBitwiseCopy<SrcPointee>::result &&
        SupportsBitwiseCopy<DestPointee>::result &&
        sizeof(SrcPointee) == sizeof(DestPointee) };
    return CopyImpl(first, last, Int2Type<useBitBlast>);
}

```

現在，如欲解除「POD 型別無法運用 BitBlast()」的束縛，只需令 SupportsBitwiseCopy 針對你的 POD 型別進行特化，並在裡頭放進一個 true 即可：

```

template<> struct SupportsBitwiseCopy<MyType>
{
    enum { result = true };
};

```

2.10.6 包裝

表格 2.1 整理出一組由 Loki 定義並實作的完整 traits。

2.11 摘要

以下是本書各組件用到的技術。大部分技術都和 template 有關。

- 編譯期 assertions (2.1 節)，幫助程式庫為泛型碼產生有意義的錯誤訊息。
- 模板偏特化 (Partial template specialization, 2.2 節)，讓你可以特化 template — 並非針對特定的、固定集合的參數，而是針對「吻合某個式樣的一群參數」。
- 區域類別 (Local classes, 2.3 節)，讓你做些有趣的事，特別是對 template 函式。
- 常整數映射為型別 (Mapping integral constants to types, 2.4 節)，允許在編譯期以數值（特別是 boolean）作為分派 (dispatch) 的取決因素。
- 型別對型別的映射 (Type-to-type mapping, 2.5 節)，讓你利用函式重載取代 C++ 缺乏的一個特性：函式模板偏特化 (function template partial specialization)。
- 型別選擇 (Type selection, 2.6 節)，讓你得以根據 boolean 條件來選擇型別。
- 編譯期間偵測可轉換性 (convertibility) 和繼承性 (2.7 節)，讓你得以判斷任意兩型別是否可互相轉換，或是否為相同型別，或是否有繼承關係。

表格 2.1 `TypeTraits<T>` 的各個成員

名稱	種類	說明
<code>isPointer</code>	Boolean 常數	如果 <code>T</code> 是指標，此值為 <code>true</code> 。
<code>PointerType</code>	Type	如果 <code>T</code> 是個指標型別，此式求得 <code>T</code> 所指型別。如果 <code>T</code> 不是指標型別，核定結果為 <code>NullType</code> 。
<code>isReference</code>	Type	如果 <code>T</code> 是個 <code>reference</code> 型別，核定結果為 <code>true</code> 。
<code>ReferencedType</code>	Type	如果 <code>T</code> 是個 <code>reference</code> 型別，此式求得 <code>T</code> 所指型別。否則求得 <code>T</code> 自身型別。
<code>ParameterType</code>	Type	此式求得「最適合做為一個 <i>nonmutable</i> 函式（譯註：不會更改操作對象內容）的參數」的型別。可以是 <code>T</code> 或 <code>const T&</code> 。
<code>isConst</code>	Boolean 常數	如果 <code>T</code> 是個常數型別（經 <code>const</code> 修飾），則為 <code>true</code> 。
<code>NonConstType</code>	Type	將型別 <code>T</code> 的 <code>const</code> 飾詞拿掉（如果有的話）。
<code>isVolatile</code>	Boolean 常數	如果 <code>T</code> 是個經 <code>volatile</code> 修飾的型別，則為 <code>true</code> 。
<code>NonVolatileType</code>	Type	將型別 <code>T</code> 的 <code>volatile</code> 飾詞拿掉（如果有的話）。
<code>NonQualifiedType</code>	Type	將型別 <code>T</code> 的 <code>const</code> 和 <code>volatile</code> 飾詞拿掉（如有的話）。
<code>isStdUnsignedInt</code>	Boolean 常數	如果 <code>T</code> 是四個不帶正負號的整數型別之一（ <code>unsigned char</code> , <code>unsigned short int</code> , <code>unsigned int</code> , <code>unsigned long int</code> ），此值為 <code>true</code> 。
<code>isStdSignedInt</code>	Boolean 常數	如果 <code>T</code> 是四個帶正負號的整數型別之一（ <code>char</code> , <code>short int</code> , <code>int</code> , <code>long int</code> ），此值為 <code>true</code> 。
<code>isStdIntegral</code>	Boolean 常數	如果 <code>T</code> 是個標準整數型別，此值為 <code>true</code> 。
<code>isStdFloat</code>	Boolean 常數	如果 <code>T</code> 是個標準浮點數型別（ <code>float</code> , <code>double</code> , <code>long double</code> ），此值為 <code>true</code> 。
<code>isStdArith</code>	Boolean 常數	如果 <code>T</code> 是個標準算術型別（整數或浮點數），此值為 <code>true</code> 。
<code>isStdFundamental</code>	Boolean 常數	如果 <code>T</code> 是個基本型別（算術型別或 <code>void</code> ），此值為 <code>true</code> 。

- `TypeInfo` (2.8 節) 實作出一個包裝了 `std::type_info` 的 `template class`，其內包含 *value* 語意和次序比較 (*ordering comparisons*) 等特性。
- `NullType` 和 `EmptyType` (2.9 節)，其功能像是在 *template metaprogramming* 中的佔位型別 (*placeholder types*)。
- `TypeTrait` (2.10 節) 提供許多一般用途的 *traits*，讓你可以根據不同的型別裁製你的程式碼。

3

Typelists

Typelists 是一個用來操作一大群型別的 C++ 工具。就像 lists 對數值提供各種基本操作一樣，typelists 對型別也提供相同操作。

有些設計範式 (design patterns) 具體指定並操作一群型別，其中也許有繼承關係 (但也許沒有)。顯著的例子是 *Abstract Factory* 和 *Visitor* (Gamma et al. 1995)。如果以傳統編程技術來操作一大群型別，將是全然的重複性工作。如此重複會導致隱微的程式碼膨脹。多數人不會想到其實它可以比現在更好。Typelists 帶給你一種能力，可以將經常性的巨集工作自動化。Typelists 將來自外星球的強大威力帶到 C++ 中，讓它得以支援新而有趣的一些手法 (idioms)。

本章介紹一個專為 C++ 設計的 typelist 完整設施，以及許多運用範例。閱讀本章之後，你將：

- 對 typelist 的概念有所了解
- 了解 typelists 如何被產生及運作
- 能夠更高效地操作 typelists
- 了解 typelists 的主要用途，以及他們可支援的編程手法 (programming idioms)

第 9, 10, 11 章都以 typelists 作為前提技術。

3.1 Typelists 的必要性

有時候你必須針對某些型別重複撰寫相同的程式碼，而且 templates 無法幫上忙。假設你需要實作一個 *Abstract Factory* (Gamma et al. 1995)。Abstract Factory 規定你必須針對設計期間已知的一群型別中的每一個型別定義一個虛擬函式，像這樣：

```
class WidgetFactory
{
public:
    virtual Window* CreateWindow() = 0;
    virtual Button* CreateButton() = 0;
    virtual ScrollBar* CreateScrollBar() = 0;
};
```

如果你想將 *Abstract Factory* 的概念泛化，並將它納入程式庫中，你必須讓使用者得以產生針對任意型別（而不只是 `Window`、`Button` 和 `ScrollBar`）的工廠（*factories*）。`templates` 無法支援此一特性。

雖然一開始 *Abstract Factory* 似乎沒有提供太多抽象和泛化的機會，但還是有些事情值得研究：

1. 如果你不試圖泛化基礎概念，就不太有機會泛化這些概念的具象實體。這是很重要的原則。如果你沒能將本質（*essence*）泛化，你仍然得和本質所衍生的具象實體糾纏奮戰。在 *Abstract Factory* 中，雖然抽象的 `base class` 十分簡單，但你會在實作各式各樣 *factories* 時遇到很多煩人又重複的程式碼。
2. 你無法輕易操作 `WidgetFactory` 成員函式（見稍早程式碼）。本質上我們不可能以泛型方式處理一群虛擬函式標記式（*virtual function signatures*）。例如，請考慮下面這個：

```
template <class T>
T* MakeRedWidget(WidgetFactory& factory)
{
    T* pW = factory.CreateT(); // huh???
    pW->SetColor(RED);
    return pW;
}
```

你得根據 `T` 是個 `Window`、`Button` 或 `ScrollBar`，各別呼叫 `CreateWindow()`、`CreateButton()` 或 `CreateScrollBar()`。但 `C++` 不允許你做這樣的文字替換。

3. 最後一點（但並非不重要），優秀程式庫可擺脫「命名習慣的無盡爭議（到底是 `createWindow` 好呢，還是 `create_window` 或 `CreateWindow?`）」。它們引入一個更好、更標準化的方法來做這些事。大致而言 *Abstract Factory* 的確有這些良好副作用。

讓我們把以上三點放到需求清單中。為滿足第一點，我們最好能夠將一串參數傳給 *AbstractFactory* `template`，產出一個 `WidgetFactory`，像這樣：

```
typedef AbstractFactory<Window, Button, ScrollBar> WidgetFactory;
```

為滿足第二點，我們需要對各種 `CreateXxx()` 函式有一個類似 `template` 的呼叫形式，例如 `Create<Window>()`，`Create<Button>()` 等等，然後就可以在泛型程式中這樣呼叫它們：

```
template <class T>
T* MakeRedWidget(WidgetFactory& factory)
{
    T* pW = factory.Create<T>(); // aha!
    pW->SetColor(RED);
    return pW;
}
```

然而，我們無法滿足這樣的需求。首先，先前的 `WidgetFactory` 型別定義是不可能的，因為 `templates` 無法擁有不定量參數。其次 `template` 語法 `Create<Xxx>` 不合法，因為虛擬函式不可以是 `templates`。

透過這些分析，我想你看到了，我們有多好的抽象化和復用化的機會，開發這些機會時我們又將遭遇多糟的語言限制。

`Typelists` 將使 *Abstract Factories* 泛化成真，並帶來更多其他利益。

3.2 定義 Typelists

由於種種因素，C++ 是一種會讓其使用者有時說出『這是我寫過最精巧的五行程式碼』的程式語言。這或許是由於其語意的豐富性，或源於其總是令人興奮（並驚訝）的特性。與這種傳統一致的是，`typelists` 的本質亦十分簡樸：

```
namespace TL
{
    template <class T, class U>
    struct Typelist
    {
        typedef T Head;
        typedef U Tail;
    };
}
```

首先，所有 `typelists` 相關東西都定義於 `TL` 命名空間內，而 `TL` 位於 `Loki` 命名空間中，因此整個都算是 `Loki` 程式碼。為了簡化範例，本章略而未寫 `TL` 命名空間。當你使用 `Typelist.h` 時可別疏忽了這一點（如果你忘記，編譯器會提醒你）。

`Typelist` 持有兩個型別，我們可透過其內部型別 `Head` 和 `Tail` 加以存取。就這樣了！我們不需要持有三個或更多個型別的 `typelist`，因為我們已經有了。例如下面就是有著三個 C++ `char` 型別變體的 `typelist`（注意尾端兩個 `>` 符號之間需要空格，這很煩人，但是必要）：

```
typedef Typelist<char, Typelist<signed char, unsigned char> >
CharList;
```

`Typelists` 內部沒有任何數值（`value`）：它們的實體是空的，不含任何狀態（`state`），也未定義任何函式。執行期間 `typelists` 不帶任何數值。它們存在的理由只是為了攜帶型別資訊。因此對 `typelist` 的任何處理都一定發生在編譯期而非執行期。`Typelists` 並未打算被具現化 — 雖然這樣做也沒什麼損害。因此，本書無論何時提到 “a `typelist`”，實際指的是一個 `typelist` 型別，不是一個 `typelist` 物件 — 那不是我們所關注的。`typelist` 的型別才是有用的（3.13.2 節展示如何以 `typelist` 建立一組物件）。

這裡使用的性質是：`template` 參數可以是任何型別，包含該 `template` 的其他具現體。這是一個眾所週知的 `template` 特性，尤其常被用來實作 `array`，例如 `vector< vector<double> >`。由

於 `Typelist` 接受兩個參數，所以我們總是可以藉由「將其中一個參數置換為另一個 `Typelist`」來達到無限延伸的目的。

然而有個小問題。雖然我們可以表達出持有兩個型別或更多型別的 `typelists`，但我們無法表達出持有零個或一個型別的 `typelist`。我們需要一個 *null list type*，而第 2 章的 `NullType` 正適合這樣的用途。

現在我來制定一個習慣：每個 `typelist` 都必須以 `NullType` 結尾。`NullType` 可被視為一個結束記號，類似傳統 C 字串的 `\0` 功能。現在我們可以定義一個只持有單一元素的 `typelist` 如下：

```
// See Chapter 2 for the definition of NullType
typedef Typelist<int, NullType> OneTypeOnly;
```

內含三個 `char` 變體的 `typelist` 則是像下面這樣：

```
typedef Typelist<char, Typelist<signed char,
    Typelist<unsigned char, NullType> > > AllCharTypes;
```

由此我們得以擁有一個無限的 `Typelist` template，藉由組合「基本單元」而持有任意量型別。

現在讓我們看看如何操作 `typelists`（再一次強調，這裡指的是 `Typelist` 型別而非 `Typelist` 物件）。準備好開始有趣的旅程吧。從這兒開始我們將鑽研 C++ 地下技術，一個由奇異和新規則組成的世界，一個「編譯期編程世界」（*compile-time programming*）。

3.3 將 `Typelist` 的生成線性化（linearizing）

詳細討論之前，我必須先說，`typelists` 實在太過偏向 LISP 風格了，以至於不好使用。LISP 風格對 LISP 程式員當然好，對 C++ 程式員卻不怎麼對味（兩個 `>` 之間的空格沒什麼好說的，不過你還是得注意它）。例如以下是一個整數型別的 `typelist`：

```
typedef Typelist<signed char,
    Typelist<short int,
        Typelist<int,
            Typelist<long int, NullType> > > >
    SignedIntegrals;
```

`Typelists` 或許是個酷點子，不過很顯然它需要更好的包裝。

為了將 `typelist` 的生成線性化，`typelist` 程式庫（Loki 中的 `Typelist.h`）定義出很多巨集，用來將遞迴型式轉換成比較簡單的列舉型式，取代冗長的重複動作。這不是問題，重複工作只需在程式庫中做一次即可。Loki 把 `typelists` 的長度擴充到 50（這是 Loki 內部定義的一個數值）。這些巨集看起來像這樣：

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2)>
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3)>
#define TYPELIST_4(T1, T2, T3, T4) Typelist<T1, TYPELIST_3(T2, T3, T4)>
...
#define TYPELIST_50(...) ...
```

每個巨集使用前一個巨集，這讓程式庫使用者可以輕易擴充上限 — 如果有此需求的話。

現在我們可以把先前的 `SignedIntegrals` 定義式以更優雅的方式表現出來：

```
typedef TYPELIST_4(signed char, short int, int, long int)
    SignedIntegrals;
```

將 `typelist` 的生成線性化，只是包裝手法的一個開端。`Typelist` 的操作仍然非常笨拙。例如存取 `SignedIntegrals` 最後一個元素竟需用到 `SignedIntegrals::Tail::Tail::Tail`。這對我們「泛化操作 `typelists`」而言還是不夠清晰易懂。現在我們應該思考傳統的 `lists` 提供哪些基本操作，並以此定義出 `typelist` 的一些基本操作。

3.4 計算長度

這是一個簡單的操作。假設有個 `TList typelist`，它有一個編譯期常數代表其長度。這個常數理當是個編譯期常數，因為 `typelists` 是一種靜態構件，我們預期所有與 `typelists` 相關的計算都將在編譯期完成。

大部份 `typelist` 操作函式的基本概念是「開拓「遞迴式(recursive)templates」，這是指某種 templates 以其本身具現體當作其定義的一部分。當這麼做時，它們會傳遞一個不同的 `template` 引數列。這種形式所產生的遞迴將止於一個「被作為邊界情況運用」的特化體 (explicit specialization)。

下面這段用來計算 `typelist` 長度的程式碼十分簡明：

```
template <class TList> struct Length;
template <> struct Length<NullType>
{
    enum { value = 0 };
};
template <class T, class U>
struct Length<Typelist<T, U> >
{
    enum { value = 1 + Length<U>::value };
};
```

以 C++ 的說法是：「`null typelist` 的長度為 0，其他 `typelist` 的長度是「`tail` 的長度加 1」。

實作 `Length` 時需要運用 `template` 偏特化 (partial specialization，參見第 2 章) 來區分 `null type` 和 `typelist`。上述第一版本是 `Length` 全特化，只匹配 `NullType`。第二版本是 `Length` 偏特化，可匹配任何 `Typelist<T, U>`，包括「複合型 (compound) `typelists`」 — 亦即 `U` 本身又是個 `Typelist<V, W>`。

第二份特化完成了遞迴式運算。它將 `value` 定義為數值 1（用來將 `head T` 計算進去）加上 `tail` 的長度。當 `tail` 變成 `NullType` 時，吻合第一份特化定義，於是停止遞迴並巧妙地傳回結果。這便是長度計算過程。如果你想定義一個 `C array`，用來存放指標（指向所有帶正負號整數之 `std::type_info` 物件），有了 `Length` 你便可以這麼寫：

```
std::type_info* intsRtti[Length<SignedIntegrals>::value];
```

於是，透過編譯期的運算，你為 `intsRtti` 配置了 4 個元素⁸。

3.5 間奏曲

你可以在 Veldhuizen(1995) 找到一些早期的 `template meta-programs` 實例。Czarnecki 和 Eisenecker (2000) 很深入地討論過這個問題，並提供一個完整的編譯期「C++ 述句組」模擬。

`Length` 的概念和實作很類似計算機科學中的經典遞迴範例：一個用來計算單向 `linked list` 長度的演算法（雖然兩者之間其實還有兩點主要不同：`Length` 所用的演算法在編譯期執行，而且它只操作型別，不操作實物）。

這很自然引導出一個問題：我們不能為 `Length` 發展一個迭代（`iteration`）版本用以取代遞迴（`recursion`）版本嗎？畢竟迭代對 C++ 來說比遞迴更自然。而且為了獲得答案，我們可能進而發展出其他 `TypeList` 設施。

答案是否定的，原因很有趣。

我們的「C++ 編譯期編程」工具是：`template`、編譯期整數計算、`typedefs`。現在讓我們看看這些工具以什麼方式來幫助我們。

- **Templates** — 更明確地說是指 `template specialization`（模板特化）— 提供編譯期間的 `if` 敘述。一如先前見過的 `Length`，特化版本能夠在 `typelists` 和其他型別之間形成差異。
- **Integer calculations**（整數計算）提供真實的數值計算能力，用以從型別轉為數值。然而其中有些古怪：所有編譯期數值都是不可變的（`immutable`），一旦你為它定義了一個整數常數，例如一個列舉值（`enumerated value`），就不能再改變它（也就是說不能重新賦予他值）。
- **typedefs** 可被視為用來引進「具名的型別常數」（`named type constants`）。它們也是定義之後就被凍結 — 你不能將 `typedef` 定義的符號重新定義為另一個型別。

編譯期計算有兩點特點，使它根本上不相容於迭代（`iteration`）。所謂迭代是持有一個迭代器（`iterator`）並改變它，直到某些條件吻合。由於編譯期間我們並沒有「可資變化的任何東西」（`mutable entities`），所以無法實現「迭代」。因此，雖然 C++ 是一種極重要的語言，但任何編譯期運算所倚賴的技術明顯讓人聯想到過去的純粹函式型（`functional`）語言 — 那些語言不會改變數量內容。現在，準備接受高劑量遞迴吧。

⁸ 你也可以不靠重複的程式碼完成 `array` 初始化。這留給讀者作為練習。

3.6 索引式存取 (Indexed Access)

透過索引來存取 `typelist` 元素，這種能力令人嚮往。這將使 `typelist` 的存取線性化，使我們能更輕鬆地操作 `typelist`。當然啦，就像我們在 `static` 世界中操作的所有物體一樣，索引必須是個編譯期數值 (compile-time value)。

一個帶有索引操作的 `template` 宣告式，看起來像這樣：

```
template <class TList, unsigned int index> struct TypeAt;
```

現在讓我們來定義演算法。記住，不能使用任何可變數值 (mutable value, modifiable value)。

TypeAt

輸入：typelist `TList`，索引值 `i`

輸出：內部某型別 `Result`

如果 `TList` 不為 `null`，且 `i` 為 0，那麼 `Result` 就是 `TList` 的頭部。

否則

如果 `TList` 不為 `null` 且 `i` 不為 0，那麼 `Result` 就是「將 `TypeAt` 施行於 `TList` 尾端及 `i-1`」的結果。

否則 逾界 (out-of-bound) 存取，造成編譯錯誤。

下面就是 `TypeAt` 演算法的化身：

```
template <class Head, class Tail>
struct TypeAt<Typelist<Head, Tail>, 0>
{
    typedef Head Result;
};
template <class Head, class Tail, unsigned int i>
struct TypeAt<Typelist<Head, Tail>, i>
{
    typedef typename TypeAt<Tail, i-1>::Result Result;
};
```

如果你試著逾界存取，編譯器會抱怨找不到 `TypeAt<NullType, x>` 特化版本，其中 `x` 是你所指定的逾界索引。這個訊息還可以更好一些，但這樣也還不錯了。

Loki 的 `Typelist.h` 還定義了一個 `TypeAt` 變型，名為 `TypeAtNonStrict`，實作出類似 `TypeAt` 的功能，不同之處是它對逾界存取更加寬容，以 `NullType` 為回傳值，取代編譯錯誤訊息。第 5 章介紹的「泛化 callback 實作碼」使用上了 `TypeAtNonStrict`。

對 `typelist` 進行索引存取，花費的時間與 `typelist` 大小有關。對 `value list` 來說這種方法不夠高效（所以 `std::list` 並未定義 `operator[]`），然而對 `typelist` 來說，時間花在編譯期，就某種意義來說這是「免費的」⁹。

⁹ 事實上，對大型專案而言這種說法不十分正確。至少理論上我們知道，大量 `typelist` 的操作可能導致編譯時間拉長。無論如何，程式如果內含大型 `typelists`，那麼若非造成執行期對速度十分飢渴（於是你可能只得接受較慢的編譯），就是太過耦合 (coupled)，於是你可能需要重新檢閱你的設計。

3.7 搜尋 Typelists

如何在 typelist 中找到某個型別呢？讓我們試著實作出 `IndexOf` 演算法，用以計算 typelist 內某型別所在位置。如果找不到就傳回一個非法值 `-1`。這個演算法是古典的線性搜尋，以遞迴方式完成。

IndexOf

輸入：typelist `TList`，type `T`

輸出：內部編譯期常數 `value`

如果 `TList` 是 `NullType`，令 `value` 為 `-1`。

否則

如果 `TList` 的頭端是 `T`，令 `value` 為 `0`。

否則

將 `IndexOf` 施行於「`TList` 尾端和 `T`」，並將結果置於一個暫時變數 `temp`。

如果 `temp` 為 `-1`，令 `value` 為 `-1`。

否則 令 `value` 為 `1+temp`。

`IndexOf` 是一個相對簡單的演算法。需特別注意的是如何將「沒找到」（`value` 為 `-1`）往外傳為計算結果。我們需要三個特化版本，每個版本對應演算法中的一個分支點。最後一個分支（根據 `temp` 計算 `value`）是個數值計算，我以條件運算子 `?:` 完成。下面是實作碼：

```
template <class TList, class T> struct IndexOf;
template <class T>
struct IndexOf<NullType, T>
{
    enum { value = -1 };
};
template <class Tail, class T>
struct IndexOf<Typelist<T, Tail>, T>
{
    enum { value = 0 };
};
template <class Head, class Tail, class T>
struct IndexOf<Typelist<Head, Tail>, T>
{
private:
    enum { temp = IndexOf<Tail, T>::value };
public:
    enum { value = temp == -1 ? -1 : 1 + temp };
};
```


3.8 附加元素至 Typelists

我們還需要一個可將「某個 type 或整個 typelist」加入 typelist 的工具。先前已討論過，修改 typelist 是不可能的，但我們將以 *by value* 方式傳回一個我們所期望的新 typelist。

Append

輸入：typelist TList，type or typelist T

輸出：內部某型別 Result

如果 TList 是 NullType 而且 T 是 NullType，那麼令 Result 為 NullType。

否則

 如果 TList 是 NullType，且 T 是個 type（而非 typelist），那麼 Result 將是「只含唯一元素 T」的一個 typelist。

 否則

 如果 TList 是 NullType，且 T 是一個 typelist，那麼 Result 便是 T 本身。

 否則 如果 TList 是 non-null，那麼 Result 將是個 typelist，以 TList::Head 為其頭端，並以「T 附加至 TList::Tail」的結果為其尾端。

這個演算法對應下列程式碼：

```
template <class TList, class T> struct Append;
template <> struct Append<NullType, NullType>
{
    typedef NullType Result;
};
template <class T> struct Append<NullType, T>
{
    typedef TYPELIST_1(T) Result;
};
template <class Head, class Tail>
struct Append<NullType, Typelist<Head, Tail> >
{
    typedef Typelist<Head, Tail> Result;
};
template <class Head, class Tail, class T>
struct Append<Typelist<Head, Tail>, T>
{
    typedef Typelist<Head,
        typename Append<Tail, T>::Result>
        Result;
};
```

再次請注意最後一個 Append 偏特化版本，它遞迴具現 Append，每次遞迴都將 tail 和「待附加型別」傳遞進去。

現在，面對單一 type 和 typelist 兩者，我們有了一致的 Append 操作型式。下面這個述句：

```
typedef Append<SignedIntegrals,
    TYPELIST_3(float, double, long double)>::Result
    SignedTypes;
```

便是定義一個 typelist，涵蓋 C++ 所有帶正負號（signed）數值型別。

3.9 移除 Typelist 中的某個元素

現在考慮「附加」的相對操作：從一個 typelist 中「移除」某個型別。我們有兩個選擇：只移除第一個出現個體，或是移除所有出現個體。

首先讓我們考慮只移除第一個出現者。

Erase

輸入：typelist TList, type T

輸出：內部某型別 Result

如果 TList 是 NullType，那麼 Result 就是 NullType。

否則

如果 T 等同於 TList::Head，那麼 Result 就是 TList::Tail。

否則 Result 將是一個 typelist，它以 TList::Head 為頭端，並以「將 Erase 施行於 TList::Tail 和 T」所得結果為尾端。

以下是上述演算法對應的 C++ 程式碼：

```
template <class TList, class T> struct Erase;
template <class T> // Specialization 1
struct Erase<NullType, T>
{
    typedef NullType Result;
};
template <class T, class Tail> // Specialization 2
struct Erase<Typelist<T, Tail>, T>
{
    typedef Tail Result;
};
template <class Head, class Tail, class T> // Specialization 3
struct Erase<Typelist<Head, Tail>, T>
{
    typedef Typelist<Head,
        typename Erase<Tail, T>::Result>
        Result;
};
```

如同 TypeAt 一樣，這裡也沒有 template 預設版本，這意味你不能以任意型別來具現化 Erase。

例如 Erase<double, int> 會導致編譯錯誤，因為它沒有匹配者。Erase 的第一參數必須是個 typelist。

繼續沿用先前的 `SignedTypes` 定義，現在我們可以寫出這樣的碼：

```
// SomeSignedTypes contains the equivalent of
// TYPELIST_6(signed char, short int, int, long int,
// double, long double)
typedef Erase<SignedTypes, float>::Result SomeSignedTypes;
```

另一項移除操作是 `EraseAll`，它會移除 `typelist` 中某個型別的所有出現個體。其實作手法類似 `Erase`，唯一不同的是，發現移除對象後演算法並不停止下來，而是繼續搜尋下一個符合條件的元素並刪除之，直到 `list` 尾端。

```
template <class TList, class T> struct EraseAll;
template <class T>
struct EraseAll<NullType, T>
{
    typedef NullType Result;
};
template <class T, class Tail>
struct EraseAll<Typelist<T, Tail>, T>
{
    // Go all the way down the list removing the type
    typedef typename EraseAll<Tail, T>::Result Result;
};
template <class Head, class Tail, class T>
struct EraseAll<Typelist<Head, Tail>, T>
{
    // Go all the way down the list removing the type
    typedef Typelist<Head,
        typename EraseAll<Tail, T>::Result>
        Result;
};
```

3.10 移除重複元素 (Erasing Duplicates)

`typelist` 的另一項重要操作就是移除重複元素。第 11 章的 `static double-dispatch engine` 廣泛運用了此項機能。

這兒的需求是：轉換 `typelist`，讓每種型別只出現一次。例如面對下面這個 `typelist`：

```
TYPELIST_6(Widget, Button, Widget, TextField, ScrollBar, Button)
```

我們希望獲得這樣的 `typelist`：

```
TYPELIST_4(Widget, Button, TextField, ScrollBar)
```

過程有點複雜，但也許你猜到了，我們可以透過 `Erase` 獲得一些幫助。

NoDuplicates

輸入：`typelist TList`

輸出：內部某型別 `Result`

如果 `TList` 是 `NullType`，那麼就令 `Result` 為 `NullType`。

否則

將 `NoDuplicates` 施行於 `TList::Tail` 身上，獲得一個暫時的 `typelist L1`。

將 `Erase` 施行於 `L1` 和 `TList::Head`。獲得結果 `L2`。

`Result` 是個 `typelist`，其頭端為 `TList::Head`，尾端為 `L2`。

現在把上述演算法轉換為程式碼：

```
template <class TList> struct NoDuplicates;
template <> struct NoDuplicates<NullType>
{
    typedef NullType Result;
};
template <class Head, class Tail>
struct NoDuplicates< Typelist<Head, Tail> >
{
private:
    typedef typename NoDuplicates<Tail>::Result L1;
    typedef typename Erase<L1, Head>::Result L2;
public:
    typedef Typelist<Head, L2> Result;
};
```

為什麼當我們以為 `EraseAll` 較為恰當的時候，`Erase` 就夠用了呢 — 我們不是希望移除所有重複型別嗎？答案是，`Erase` 被實施於 `NoDuplicates` 遞迴操作之後。這意味我們將從 `list` 之中移除的是一個「不再有任何重複個體」的型別，所以最多只會有一個型別個體（instance of the type）被移除。這一段遞迴編程相當有趣。

3.11 取代 Typelist 中的某個元素

有時候我們需要元素的「代換」而非「移除」。一如你將於 3.12 節所見，將某個型別取代為另一個型別是慣用技法（idioms）中很重要的部分。

我們需要在一個 `typelist TList` 中以型別 `U` 取代型別 `T`。

Replace

輸入：typelist `TList`, type `T`（被取代者），以及 type `U`（取代者）

輸出：內部某型別 `Result`

如果 `TList` 是 `NullType`，令 `Result` 為 `NullType`。

否則

如果 `typelist TList` 的頭端是 `T`，那麼 `Result` 將是一個 `typelist`，以 `U` 為其頭端並以 `TList::Tail` 為其尾端。

否則 `Result` 是一個 `typelist`，以 `TList::Head` 為其頭端，並以「`Replace` 施行於 `TList`, `T`, `U`」的結果為其尾端。

一旦理解上述遞迴演算法，很自然寫出以下這樣的程式碼：

```

template <class TList, class T, class U> struct Replace;
template <class T, class U>
struct Replace<NullType, T, U>
{
    typedef NullType Result;
};
template <class T, class Tail, class U>
struct Replace<Typelist<T, Tail>, T, U>
{
    typedef Typelist<U, Tail> Result;
};
template <class Head, class Tail, class T, class U>
struct Replace<Typelist<Head, Tail>, T, U>
{
    typedef Typelist<Head,
        typename Replace<Tail, T, U>::Result>
        Result;
};

```

如果改變上述第二個特化版本，將演算法遞迴施行於 Tail 身上，輕易便可獲得 ReplaceAll 演算法（譯註：請看 Loki "typelist.h" 便知道實際作法）。

3.12 為 Typelists 局部更換次序 (Partially Ordering)

假設我們打算根據繼承關係進行排序，例如希望衍生型別出現在基礎型別之前。舉個例子，我們手上有一個圖 3-1 這樣的 class 繼承體系：

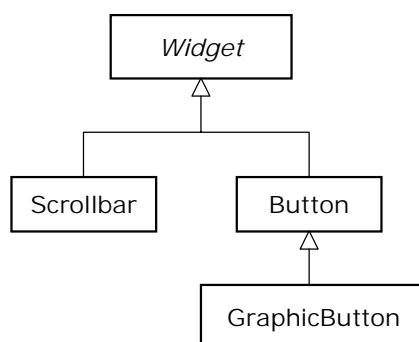


圖 3-1 一個簡單的 class 繼承體系

如果我們有這樣一個 `typelist`：

```
TYPELIST_4(Widget, ScrollBar, Button, GraphicButton)
```

現在的挑戰是如何將它轉換為：

```
TYPELIST_4(ScrollBar, GraphicButton, Button, Widget)
```

也就是把「最末端衍生型別」（most derived types）帶向前，並讓其他兄弟型別的順序不變。

這似乎只是個智力練習，但其實有其重要應用。重新排列一個 `typelist` 的元素次序，使最末端衍生型別放在最前面，可以確保由下而上走訪（遍歷）這個 `class` 繼承體系。第 11 章的 `double-dispatch engine` 便是運用這個重要技術來解析型別資訊。

當我們準備對一群型別排序時，需要一個排序函式。先前已經有了一個在編譯期用來偵測繼承關係的工具，第 2 章曾有詳細討論。回想一下，我們有一個方便的巨集 `SUPERSUBCLASS(T, U)`，如果 `U` 衍生自 `T`，它會傳回 `true`。太好了，我們只需將「繼承偵測機制」合併到 `typelist` 即可。

這裡不能使用充分而成熟的排序演算法，因為我們沒有完整的順序關係；是的，對 `classes` 而言並沒有與 `operator<` 相應的東西，於是本例兩個兄弟型別無法被 `SUPERSUBCLASS(T, U)` 排序。我將使用一個自製演算法，將衍生型別帶到前面，並令其他 `classes` 的相對位置保持不變。

DerivedToFront

輸入：`typelist TList`

輸出：內部某型別 `Result`

如果 `TList` 是 `NullType`，令 `Result` 為 `NullType`。

否則

從 `TList::Head` 到 `TList::Tail`，找出最末端衍生型別，儲存於暫時變數 `TheMostDerived` 中

以 `TList::Head` 取代 `TList::Tail` 中的 `TheMostDerived`，獲得 `L`。

建立一個 `typelist`，以 `TheMostDerived` 為其頭端，以 `L` 為其尾端。

將這個演算法施行於 `typelist` 身上，衍生型別便會被移至 `typelist` 頭端，基礎型別被推移至尾端。

這裡還缺少一樣東西：在某個 `typelist` 中搜尋「某型別之最深層衍生型別」的演算法。由於 `SUPERSUBCLASS` 會傳回一個編譯期 `Boolean` 值，我們發現第 2 章一個小型的 `Select class template` 可以派上用場。還記得嗎，`Select` 可根據編譯期 `Boolean` 值在兩個型別中選擇一個。

MostDerived 演算法接受一個 typelist 和一個 Base 型別，傳回 typelist 中 Base 的最深層衍生型別（如果找不到任何衍生型別，就傳回 Base 自己）。看來像這樣：

MostDerived

輸入：typelist TList, type T

輸出：內部某型別 Result

如果 TList 是 NullType，令 Result 為 T。

否則

將 MostDerived 施行於 TList::Tail 和 T 身上，獲得一個 Candidate。

如果 TList::Head 衍生自 Candidate，令 Result 為 TList::Head。

否則 令 Result 為 Candidate。

MostDerived 實作如下：

```
template <class TList, class T> struct MostDerived;

template <class T>
struct MostDerived<NullType, T>
{
    typedef T Result;
};

template <class Head, class Tail, class T>
struct MostDerived<Typelist<Head, Tail>, T>
{
private:
    typedef typename MostDerived<Tail, T>::Result Candidate;
public:
    typedef typename Select<
        SUPERSUBCLASS(Candidate, Head),
        Head, Candidate>::Result Result;
};
```

前述的 DerivedToFront 演算法便是以 MostDerived 為基礎，下面是其實作：

```
template <class T> struct DerivedToFront;

template <>
struct DerivedToFront<NullType>
{
    typedef NullType Result;
};

template <class Head, class Tail>
struct DerivedToFront< Typelist<Head, Tail> >
{
private:
    typedef typename MostDerived<Tail, Head>::Result
```

```

    TheMostDerived;
    typedef typename Replace<Tail,
        TheMostDerived, Head>::Result L;
public:
    typedef Typelist<TheMostDerived, L> Result;
};

```

這個複雜的 `typelist` 操作用處很大。「DerivedToFront 轉換」可以高效地將「型別處理工序」自動化。沒有了它，我們就只能仰賴大量訓練和注意力的集中才能完成任務。

3.13 運用 Typelists 自動產生 Classes

如果此刻你認為 `typelist` 很有魅力，很有趣，或是很醜陋，你其實還沒真正看到什麼東西。我將在這一小節中定義數個基本構想，以 `typelists` 做為程式碼生成機制。這麼一來我們就不必手寫太多程式碼，而是驅動編譯器幫我們自動產生程式碼。這些概念採用了 C++ 最具威力的特性之一，一個不存在於任何其他語言的特性 — **template template parameters**。

截至目前，`typelist` 的操作都沒有產生真正的程式碼；運作過程只產生 `typelists`、`types` 或編譯期常數（例如 `Length`）。讓我們儘可能產生一些真正的程式碼，也就是真正能夠在編譯結果中留下足跡的東西。

`Typelist` 物件本身沒什麼用；它們缺乏執行期狀態（state）和機能（functionality）。從 `typelist` 中產生 `classes` 是很重要的編程需求。應用程式編寫者有時需要以一種「`typelist` 所指示的方向」來編寫 `classes` — 填以虛擬函式、資料宣告或函式實作。我將試著運用 `typelist` 將這樣的過程自動化。

由於 C++ 缺乏「編譯期迭代或遞迴巨集」（compile-time iteration or recursive macros），想為 `typelist` 內含的每個型別加入一些程式碼是很困難的。你可以運用「`template` 偏特化」手法，組合前述演算法，但是在客端實作這樣的方案，不但笨拙而且複雜。這時候 `Loki` 可以派上用場。

3.13.1 產生「散亂的繼承體系（scattered hierarchies）」

`Loki` 提供一個極具威力的工具，可輕易將 `typelist` 裡頭的每一個型別套用於一個由用戶提供的基本 `template` 身上。這麼一來，將 `typelist` 內的型別分發（distributing）至客端程式碼的一些不便過程就被完全封裝於 `Loki` 程式庫中，使用者只需自行定義一個單一參數的 `template` 即可。

這樣一個 `class template` 名為 `GenScatterHierarchy`。雖然其定義十分簡單，很快你會看見 `GenScatterHierarchy` 引擎蓋下的驚人馬力。下面是其定義式¹⁰：

```

template <class TList, template <class> class Unit>
class GenScatterHierarchy;

```

¹⁰ 此處的情況是，在潛在應用尚未浮現之前，先將設計構想呈現出來。


```

// GenScatterHierarchy specialization: Typelist to Unit
template <class T1, class T2, template <class> class Unit>
class GenScatterHierarchy<TYPELIST_2(T1, T2), Unit>
    : public GenScatterHierarchy<T1, Unit>
    , public GenScatterHierarchy<T2, Unit>
{
};
// Pass an atomic type (nontypelist) to Unit
template <class AtomicType, template <class> class Unit>
class GenScatterHierarchy : public Unit<AtomicType>
{
};
// Do nothing for NullType
template <template <class> class Unit>
class GenScatterHierarchy<NullType, Unit>
{
};

```

template template parameters（見第 1 章）如你預期般地進行大量工作。你傳入一個名為 `Unit` 的 `template class` 當作 `GenScatterHierarchy` 的第二參數。`GenScatterHierarchy` 內部對該 `Unit` 的使用就像面對任何「帶有單一 `template` 參數」的一般 `template class` 一樣。它所能顯現的威力來自於你（`GenScatterHierarchy` 使用者）：給它一個你自己寫的 `template` 吧。

`GenScatterHierarchy` 做了甚麼？如果其第一引數是個 **atomic type**（意指單一型別，相對於 `typelist`），`GenScatterHierarchy` 便把該型別傳給 `Unit`，然後繼承 `Unit<T>`。如果 `GenScatterHierarchy` 的第一引數是個 `typelist TList`，就遞迴產生 `GenScatterHierarchy<TList::Head, Unit>` 和 `GenScatterHierarchy<TList::Tail, Unit>` 並繼承此二者。`GenScatterHierarchy<NullType, Unit>` 則是個空類別。

最終，一個 `GenScatterHierarchy` 具現體會繼承 `Unit`「對 `typelist` 中每一個型別」的具現體。例如下面這段程式碼：

```

template <class T>
struct Holder
{
    T value_;
};
typedef GenScatterHierarchy<
    TYPELIST_3(int, string, Widget),
    Holder>
WidgetInfo;

```

由 `WidgetInfo` 產生的繼承體系如圖 3-2。我們稱此種 `class` 繼承體系是亂散的（*scattered*），因為 `typelist` 內的型別亂散分佈於不同的 `root class` 之下。`GenScatterHierarchy` 的要旨是：它藉由重複具現化一個你所提供的 `class template`（視之為模型），為你產生一個 `class` 繼承體系，然後將所有這樣產生出來的 `classes` 聚焦至一個 `leaf class`（葉類別、末端類別），亦即本例的 `WidgetInfo`。

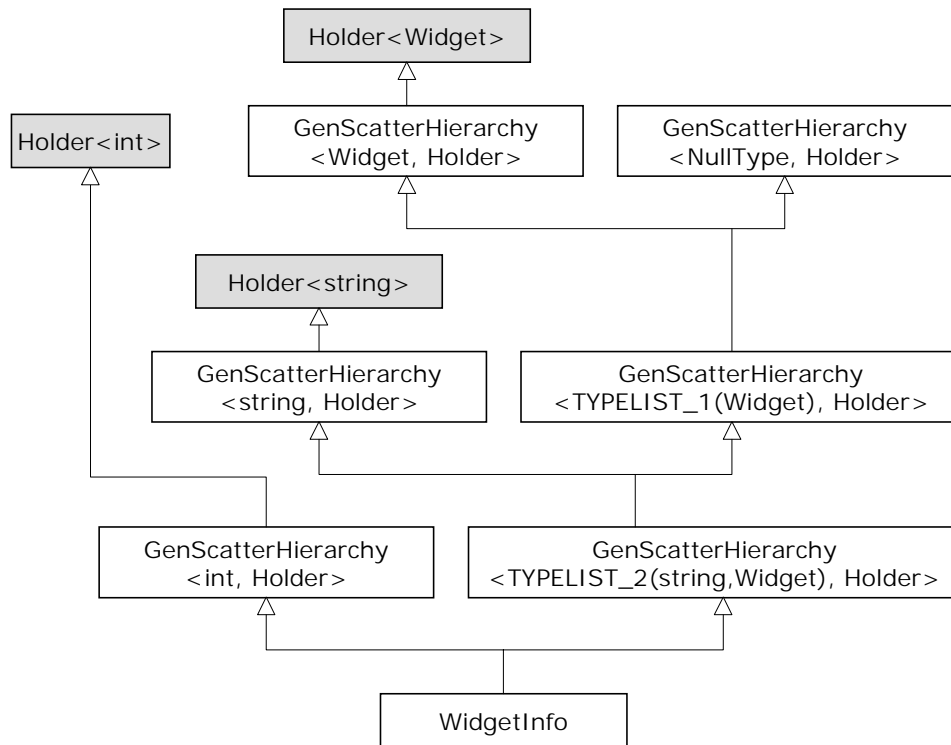


圖 3-2 WidgetInfo 的繼承結構

由於繼承了 `Holder<int>`、`Holder<string>` 和 `Holder<widget>`，`WidgetInfo` 因而針對它們（亦即 `typelist` 中的每個型別）各自擁有一個成員變數 `value_`。圖 3-3 顯示 `WidgetInfo` 物件的記憶體配置可能型式。這樣的配置型式首先假設空類別 `GenScatterHierarchy<NullType, Holder>` 會被優化掉，不會在這個複合物件中佔據實體位置。

你還可以對 `WidgetInfo` 物件作些有趣的事情。例如以這種方式存取其中的 `string` 物件：

```
WidgetInfo obj;
string name = (static_cast<Holder<string>&>(obj)).value_;
```

其中的顯式轉型（`explicit cast`）是必要的，以消除成員變數名稱 `value_` 的可能含糊意義，否則編譯器不知道你打算取用哪一個 `value_`。

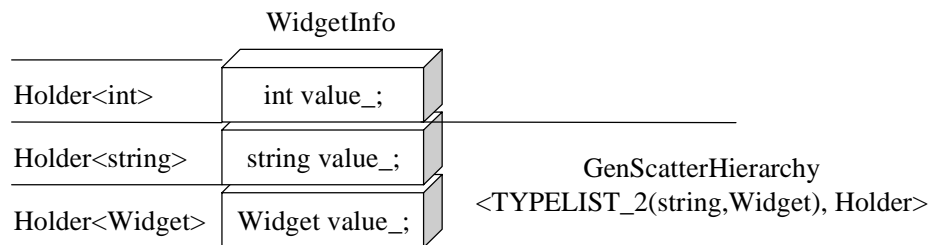


圖 3-3：WidgetInfo 的記憶體配置圖

如此的轉型十分難看，讓我們試著藉由提供某些方便的存取函式，把 GenScatterHierarchy 變得更容易使用。例如提供一個成員函式，根據成員的型別來取用成員變數。這很容易：

```
// Access a base class by type name
template <class T, class TList, template <class> class Unit>
Unit<T>& Field(GenScatterHierarchy<TList, Unit>& obj)
{
    return obj;
}
```

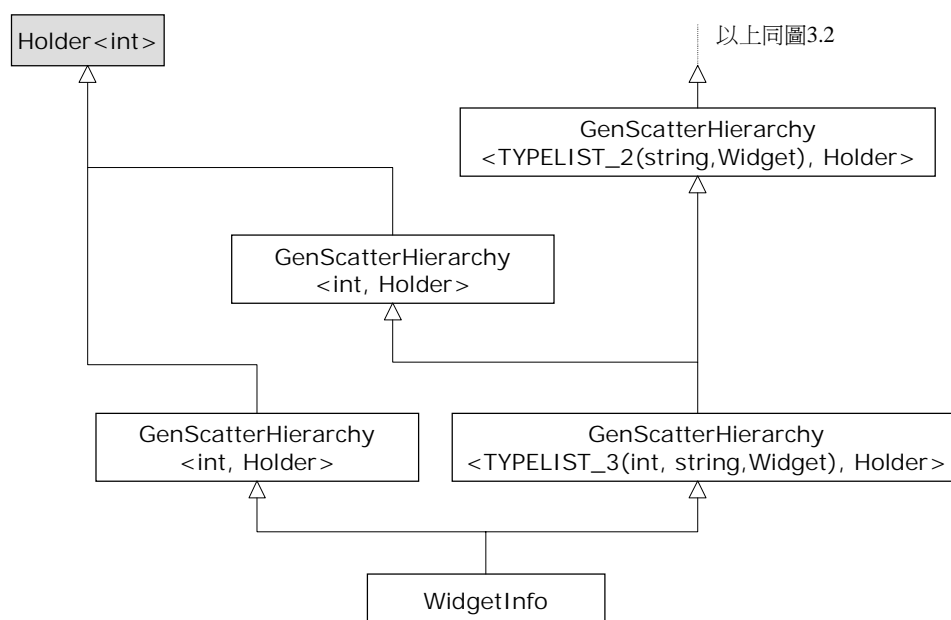
Field 倚賴 **derived-to-base** 隱式轉換。如果你呼叫 Field<Widget>(obj)（其中 obj 是一種 WidgetInfo），編譯器會知道 Holder<Widget> 是 WidgetInfo 的 base class 並只傳回複合物件中該成份的 reference。

為何 Field 是個 namespace-level 函式而不是一個成員函式呢？因為在如此高階的泛型程式設計中，我們必須小心處理名稱。舉個例子，想像一下，如果 Unit 自己定義了一個名為 Field 的符號，而 GenScatterHierarchy 很糟糕地自己也定義了一個名為 Field 的成員函式，後者將會遮蔽前者。這會成為惹惱使用者的一個主要原因。

Field 另有一個問題也是惹惱使用者的主因：當你的 typelist 內含重複型別時，你不能使用 Field。看一下這個稍作修改的 WidgetInfo：

```
typedef GenScatterHierarchy<
    TYPELIST_4(int, int, string, Widget),
    Holder>
WidgetInfo;
```

現在 WidgetInfo 有了兩個「型別為 int」的 value_ 成員。如果你試著對某個 WidgetInfo 物件呼叫 Field<int>，編譯器會抱怨出現模稜兩可（歧義）情況。這個問題無法輕鬆解決，因為 WidgetInfo 最終確實透過了不同的路徑繼承 Holder<int> 兩次，如圖 3-4。

圖 3-4 `WidgetInfo` 繼承 `Holder<int>` 兩次

因此，我們需要一個「以索引選擇 `GenScatterHierarchy` 實體欄位」的方法，而非透過型別名稱。如果你可以藉由在 `typelist` 中的位置指出兩個 `int` 欄位的話（例如這麼寫：`Field<0>(obj)` 和 `Field<1>(obj)`），就可以擺脫模稜兩可（歧義）的困境。

讓我們試著實作一個以索引為基礎的欄位存取函式。我們必須在編譯期分派（`dispatch`）索引值零（用以存取 `typelist head`）和非零（用以存取 `typelist tail`）。有了第 2 章定義的 `Int2Type` `template` 的協助，要做分派動作（`dispatching`）是很容易的。`Int2Type` 可以直接將不同常數轉換為不同型別。

```

template <class TList, template <class> class Unit>
Unit<TList::Head>& FieldHelper(
    GenScatterHierarchy<TList, Unit>& obj,
    Int2Type<0>)
{
    GenScatterHierarchy<TList::Head, Unit>& leftBase = obj;
    return leftBase;
}
template <int i, class TList, template <class> class Unit>
Unit<TypeAt<TList, index>::Result>&
FieldHelper(
    GenScatterHierarchy<TList, Unit>& obj,
    Int2Type<i>)
{
    GenScatterHierarchy<TList::Tail, Unit>& rightBase = obj;
    return FieldHelper(rightBase, Int2Type<i-1>());
}
template <int i, class TList, template <class> class Unit>
Unit<TypeAt<TList, index>::Result>&
Field(GenScatterHierarchy<TList, Unit>& obj)
{
    return FieldHelper(obj, Int2Type<i>());
}

```

寫出這樣的實作碼需要相當時間，幸運的是它很容易說明。名為 `FieldHelper` 的兩個多載函式做了實際工作。第一版本接受一個型別為 `Int2Type<0>` 的參數，第二版本接受的型別是 `Int2Type<any integer>`。因此，第一版本回傳的物件相當於 `Unit<T1>&`，第二版本傳回的是 `typelist` 之中被索引標示出來的型別。`Field` 和 `FieldHelper` 都用上了 3.6 節定義的 `TypeAt` 演算法。`FieldHelper` 第二版本遞迴呼叫自己的一個特化體，傳入 `GenScatterHierarchy` 的右側 base class 和 `Int2Type<index-1>`。這麼做是因為，對任何非零值 N 而言，`typelist` 內第 N 個欄位其實就是 `tail` 的第 $N-1$ 個欄位（ $N=0$ 的情況則由第一個多載版本負責處理）。

為了獲得更有效率的介面，我們還需兩個額外的 `Field` 函式：兩個 `const Field` 函式。它們和 `non-const` 版本類似，但可接受和回傳 `references to const types`。

`Field` 使得 `GenScatterHierarchy` 非常容易被使用。現在我們可以這樣寫：

```

WidgetInfo obj;
...
int x = Field<0>(obj).value_;    // first int
int y = Field<1>(obj).value_;    // second int

```

`GenScatterHierarchy` 很適合從 `typelist` 中產生繁複的 classes（只需與一個簡單的 `template` 合作）。你可以運用 `GenScatterHierarchy` 對 `typelist` 中的每一個 `types` 產生一些虛擬函式。第 9 章的 *Abstract Factory* 便運用 `GenScatterHierarchy` 從 `typelist` 產生出抽象生成函式（abstract creation functions），該章也展示如何利用 `GenScatterHierarchy` 產生 classes 繼承體系。

3.13.2 產生 Tuples

有時候你也許只是希望產生一個帶有無名欄位的小型結構（某些語言，例如 ML，將這種東西稱爲一個 **tuple**）。C++ tuple 設施由 Jakko Järvi（1999a）首先提出，而後經過 Järvi 和 Powell（1999b）改良。

甚麼是 tuples 呢？看看下面的例子：

```
template <class T>
struct Holder
{
    T value_;
};
typedef GenScatterHierarchy<
    TYPELIST_3(int, int, int),
    Holder>
Point3D;
```

Point3D 的運用方式有點笨拙，因爲你必須在每一個欄位存取函式之後寫出 `.value_`（[譯註](#)：就像 p69 最下那樣）。你需要的其實只是採用和 GenScatterHierarchy 相同的方法產生一個結構，但是讓 Field 存取函式直接傳回 `value_` reference。也就是說 Field<n> 不該傳回 Holder<int>&，應該傳回 int&。

Loki 定義了一個 Tuple template class，其實作手法類似 GenScatterHierarchy，但提供「欄位直接存取」機能。Tuple 用起來像這樣：

```
typedef Tuple<TYPELIST_3(int, int, int)>
    Point3D;
Point3D pt;
Field<0>(pt) = 0;
Field<1>(pt) = 100;
Field<2>(pt) = 300;
```

Tuples 很適合用來產生一個無任何成員函式的無名結構。例如你可以在函式中利用 tuples 傳回多個數值，像這樣：

```
Tuple<TYPELIST_3(int, int, int)>
GetWindowPlacement(Window&);
```

上述函式 GetWindowPlacement 可讓使用者只以一個函式呼叫就取得一個 window 座標和它在 windows stack 中的位置。程式庫作者不需要爲三個整數型別的 tuples 提供各自不同的結構。

你可以在 Tuple.h 中看到 Loki 提供的其他 tuple 相關函式。

3.13.3 產生線性繼承體系

考量下面這個簡單的 template，它定義了一個事件處理介面（event handler interface），其中只定義了一個成員函式 OnEvent：

```
template <class T>
class EventHandler
{
public:
    virtual void OnEvent(const T&, int eventId) = 0;
    virtual void ~EventHandler() {}
};
```

爲了策略正確性，EventHandler 還定義了一個虛擬解構式，這雖然和我所要討論的主題無關，卻是絕對必要的（原因見第 4 章）。

我們可以運用 GenScatterHierarchy 傳發（*distribute*）給 typelist 裡頭的任何型別一個 EventHandler：

```
typedef GenScatterHierarchy
<
    TYPELIST_3(Window, Button, ScrollBar),
    EventHandler
>
WidgetEventHandler;
```

GenScatterHierarchy 的缺點是它使用了多重繼承。如果你很在乎物件大小的優化，那麼 GenScatterHierarchy 也許就不那麼好了，因為 WidgetEventHandler 內有三個指向虛擬函式表（vtables）的指標¹¹，一個指標針對一個 EventHandler 函式實體（[譯註](#)：此類基礎知識可參考 *Inside The C++ Object Model*，《深度探索 C++ 物件模型》）。如果 sizeof(EventHandler) 是 4 bytes，sizeof(WidgetEventHandler) 可能高達 12 bytes，而且會隨著你加入 typelist 的型別個數而增加大小。如果希望獲得最佳空間使用率，應該把所有虛擬函式宣告到 WidgetEventHandler 裡頭，但這會破壞程式碼產生機會。

一個好的組態（configuration）是將 WidgetEventHandler 分解成「每個虛擬函式配一個 class」，如圖 3-5，是謂「線性繼承體系」。藉由單一繼承（而非多重繼承），WidgetEventHandler 只有一個 vtable 指標，達到最佳空間效能。

什麼機制才能自動化完成這樣一個線性繼承體系呢？類似 GenScatterHierarchy 的遞迴性 template 將可帶來幫助。然而其中有個不同點：使用者提供的 class template 如今必須接受兩個 template 參數，其一是 typelist 內的當前型別（這一點和 GenScatterHierarchy 相同），另一是具具體的 base class。後者之所以需要，因為如圖 3-5 所示，客端程式碼如今需要參與繼承體系，而不只是做為根類別（像 GenScatterHierarchy 那樣）。

現在讓我們寫出遞迴的 GenLinearHierarchy template。它很類似 GenScatterHierarchy，不同的是對於「繼承關係」和「使用者提供之 template unit」的處理。

¹¹ 並無任何規定說 C++ 編譯器一定需要一個虛擬函式表（virtual tables），不過大部分編譯器的確是需要的。關於虛擬函式表可參考 Lippman(1994)（[譯註](#)：按版權頁顯示，該書乃 1996 年出版）

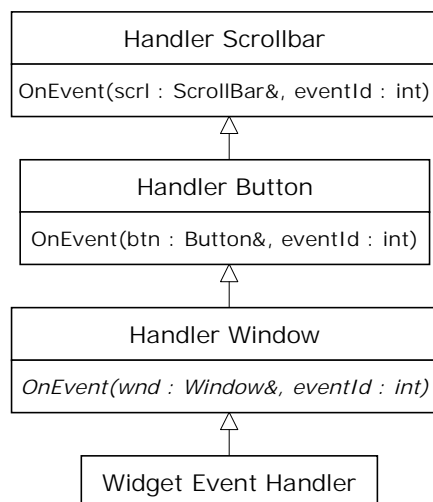


圖 3-5：一個有結構大小最佳化的 WidgetEventHandler

```

template
<
    class TList,
    template <class AtomicType, class Base> class Unit,
    class Root = EmptyType // For EmptyType, consult Chapter 2
>
class GenLinearHierarchy;
template
<
    class T1,
    class T2,
    template <class, class> class Unit,
    class Root
>
class GenLinearHierarchy<Typelist<T1, T2>, Unit, Root>
    : public Unit< T1, GenLinearHierarchy<T2, Unit, Root> >
{ };
template
<
    class T,
    template <class, class> class Unit,
    class Root
>
class GenLinearHierarchy<TYPELIST_1(T), Unit, Root>
    : public Unit<T, Root>
{ };
  
```

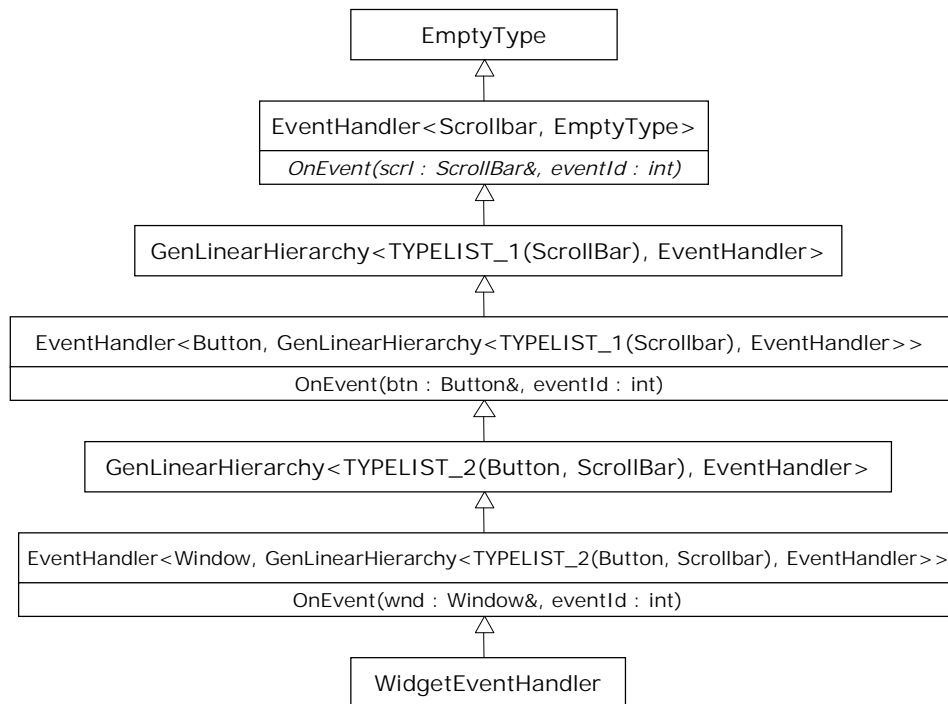



圖 3-6 GenLinearHierarchy 產生的 class 繼承體系

這一段程式碼比 GenScatterHierarchy 稍微複雜些，但產出之繼承體系結構比較簡單。現在讓我們驗證「一張圖勝過千言萬語」的箴言，看看圖 3-6，其中顯示由下列程式碼產生出來的繼承體系。

```

template <class T, class Base>
class EventHandler : public Base
{
public:
    virtual void OnEvent(T& obj, int eventId);
};
  
```

```
typedef GenLinearHierarchy
<
    TYPELIST_3(Window, Button, ScrollBar),
    EventHandler
>
MyEventHandler;
```

爲了結合 `EventHandler`，`GenLinearHierarchy` 定義了一個線性的、繩索般的、單一繼承式的 `classes` 繼承體系，其中任一節點都有個純虛擬函式，形式如 `EventHandler` 所描述。結果，`MyEventHandler` 一如所求地定義了三個虛擬函式。`GenLinearHierarchy` 爲其 **template template parameter** 加上一個新條件：`Unit`（本例中爲 `EventHandler`）必須接受第二個 `template` 引數，然後成爲 `GenLinearHierarchy` 的基礎類別。爲了彌補，`GenLinearHierarchy` 十分勤勉地產生 `classes` 繼承體系。

要讓 `Field` 多載函式提供某些機能（類似給 `GenScatterHierarchy` 運用的那些）給 `GenLinearHierarchy` 使用，是很容易的。`Loki` 總共定義了 8 個 `Field` 多載函式，分別針對：`GenScatterHierarchy` 和 `GenLinearHierarchy`、`const` 和 `non-const` 物件、「以索引爲據」和「以型別爲據」。`GenScatterHierarchy` 和 `GenLinearHierarchy` 通常焦不離孟，大部分情況下你需要以 `GenScatterHierarchy` 產生一個介面，並以 `GenLinearHierarchy` 實作之。第 9 章和第 10 章實際展示了這兩個「class 繼承體系產生器」的使用方式。

3.14 摘要

`Typelists` 是一個重要的泛型編程技術。它可以提供程式庫作者一些新性能：表達及操作任意數量的型別，並從這些型別中產出資料結構和程式碼…等等。

編譯期間 `typelist` 提供了如同一般 `list` 的絕大部分基本功能：附加、刪除、搜尋、取用、取代、刪除重複，乃至「依據繼承關係局部排序」。操作 `typelist` 的程式碼，限制爲純函式風格（`pure functional style`），因爲缺乏編譯期變數可用——不論型別或編譯期常數，一旦被定義，就再不能被改變了。因爲這個緣故，大部分 `typelist` 操作都倚賴遞迴式（`recursive`）`templates`，並根據 `template` 偏特化（`partial specialization`）來完成範式匹配（`pattern matching`）。

當你必須針對一大群型別撰寫相同的程式碼時，`typelist` 很有用——不論是用於宣告或用於實作。它們讓你得以抽象化和泛化那些被其他所有泛型編程技術遺忘的東西。正因如此，`typelists` 帶來了新特性、新手法、新的程式庫實作技術，一如你將於第 9 章和第 10 章所見。

`Loki` 提供兩個強而有力的基本工具 `GenScatterHierarchy` 和 `GenLinearHierarchy`，讓你根據 `typelist` 自動產生繼承體系。它們會產生兩種 `classes` 結構：散亂的（*scatter*，圖 3-2）和線性的（*linear*，圖 3-6）。線性繼承體系比較高效，散亂繼承體系則具備一個有用性質：使用者自定之 `template` 所具現出來的所有實體，都是最後產出之 `class` 的父類別，如圖 3-2。

3.15 Typelist 要點概覽

- 表頭檔：Typelist.h。
- 所有 typelist 工具都歸屬於 `Loki::TL` 這一命名空間（namespace）。
- 定義了 `class template Typelist<Head, Tail>`。
- Typelist 的生成：定義了 `TYPELIST_1` 至 `TYPELIST_50` 共 50 個巨集，它們接受的參數個數如其名稱所示。
- 巨集上限（50）可以擴充如下：


```
#define TYPELIST_51(T1, repeat here up to T51) \
    Typelist<T1, TYPELIST_50(T2, repeat here up to T51)>
```
- 根據習慣，typelists 應該合度 — 它們總是有個單一型別（non-typelist）的頭端（head，第一元素），至於尾端（tail）可以是個 typelist 或是個 `NullType`。
- 表頭檔內定義了一組用來操作 typelists 的基本工具。習慣上，所有基本工具都傳回 `Result`，那是一個巢狀的（內部的）`public` 型別。如果操作完畢後傳回一個數值，該數值通常命名為 `value`。
- 上述所謂基本工具，完整列於表 3-1。
- `class template GenScatterHierarchy` 概要如下：


```
template <class TList, template <class> class Unit>
class GenScatterHierarchy;
```
- `GenScatterHierarchy` 會產生一個繼承體系，由「根據 typelist `TList` 內的每一個型別，將 `Unit` 具現化」後的所有成果構成。`GenScatterHierarchy` 具現體直接或間接繼承自每一個 `Unit<T>`，其中 `T` 是 typelist 內的任一型別。
- 圖 3-2 描繪出 `GenScatterHierarchy` 所產生的繼承體系結構。
- `class template GenLinearHierarchy` 概要如下：


```
template <class TList, template <class, class> class Unit>
class GenLinearHierarchy;
```
- 圖 3-6 描繪出 `GenLinearHierarchy` 所產生的繼承體系結構。
- `GenLinearHierarchy` 會將 typelist `TList` 內的每一個型別傳入 `Unit` 當做其第一參數，以此具現化 `Unit`。很重要的一點：`Unit` 必須以 `public` 方式繼承其第二 `template` 參數。
- 多載函式 `Field` 允許使用者「根據型別名稱」和「根據索引」存取繼承體系中的任一節點。`Loki` 共提供八個 `Field` 多載函式，包含 `const` 和 `non-const` 版本、「根據型別」和「根據索引」版本，以及「針對 `GenScatterHierarchy`」和「針對 `GenLinearHierarchy`」版本。
- `Field<Type>(obj)` 傳回一個 `reference` 指向某 `Unit` 具現體，後者相應於特定型別 `Type`。
- `Field<index>(obj)` 傳回一個 `reference` 指向某 `Unit` 具現體，後者相應於「以常數 `index` 標記出來的那個型別」。

表 3-1 作用於 Typelists 身上的各種編譯期演算法

基本工具 名稱	說明
Length<TLi st>	計算 TLi st 的長度
TypeAt<TLi st, i dx>	傳回 TLi st 某位置（以零為基準）上的型別。如果 index 大於或等於 TLi st 長度，會發生編譯期錯誤。
TypeAtNonStri ct<TLi st, i dx>	傳回 TLi st 某位置（以零為基準）上的型別。如果 index 大於或等於 TLi st 長度，會傳回 Nul l Type。
I ndexOf<TLi st, T>	傳回第一次出現型別 T 的位置（以零為基準）。如果沒找到就傳回 -1。
Append<TLi st, T>	將一個 type 或 typelist 附加到 TLi st 中。
Erase<TLi st, T>	移除 TLi st 內的第一個 T（如果有的話）。
EraseAl l <TLi st, T>	移除 TLi st 內的所有 T（如果有的話）。
NoDupl i cates<TLi st>	移除 TLi st 內所有重複的型別。
Repl ace<TLi st, T, U>	以 U 取代 TLi st 內的第一個 T。
Repl aceAl l <TLi st, T, U>	用 U 取代 TLi st 內的所有 T。
MostDeri ved<TLi st, T>	傳回 TLi st 內最深層衍生型別（most derived type）。如果沒有這樣的型別，就傳回 T 本身。
Deri vedToFront<TLi st>	把最深層衍生型別（most derived type）移到最前面。

4

小型物件配置技術

Small Object Allocation

譯註：本章術語包括 `block`（區塊）、`chunk`（大塊記憶體；不譯）、`allocate`（配置）、`deallocate`（歸還）、`release`（釋放）、`free`（釋放）。請注意，`deallocate` 和 `release` 在本章的意義並不相同。

本章講述小型物件快速配置器的設計和實現。如果你使用這種配置器來動態配置物件，其所花費的額外開銷比起在 `stack` 內動態配置物件的額外開銷顯得微不足道。

在不同的場合下，`Loki` 會用到非常小的物件 — 甚至小至數個 `bytes`。第 5 章（`Generalized Functors`，泛化仿函式）和第 7 章（`Smart Pointers`，精靈指標）廣泛運用了小型物件。基於各種原因，多型（`polymorphic`）行為乃是物件導向編程中最重要的性質，因此這些小型物件不能夠儲存在 `stack` 內，只能位於 `free store`（自由空間）中。

`C++` 提供 `new` 和 `delete` 運算子作為 `free store` 的主要使用方法。問題是這些運算子都是通用型的，它們的「小型物件配置性能」很糟糕。糟到什麼程度呢？配置小型物件時，某些標準的 `free store` 配置器和本章配置器相比，執行速度會慢一個數量級，記憶體則耗費兩倍之多。

「過早最佳化是一切罪惡的根源」，`Knuth` 如是說。但另一方面，按照 `Len Lattanzi` 的說法：「過時的悲觀毫無益處」。對於核心物件如 `functors`（仿函式）、`smart pointers`（精靈指標）或 `strings`（字串）來說，如果它們在執行期產生一個數量級的惡化，對整個專案的成功與否可能帶來極大影響。廉價（無額外開銷）而快速地動態配置小型物件，好處十分巨大，能讓你在運用高級技術的同時，無需擔心效能上出現重大損失。因此，探究小型物件在 `free store`（自由空間）中的配置策略，對程式員具有極大誘惑力。

很多 `C++` 書籍，例如 `Sutter`（2000）和 `Meyers`（1998a），都談到了專用型記憶體配置器的用處。然而 `Meyers` 將細節「當做一道超難習題留給讀者」，`Sutter` 則將你打發到「你最喜愛的高階或通用 `C++` 編程教本」中。你手上這本書並不指望成為你的最愛，不過本章的確打破砂鍋，詳盡實現一個奉行 `C++` 標準的配置器。

閱讀本章之後，你將對記憶體配置器優化所涉及的某些微妙而有趣的問題有所理解，並知道如何使用 `Loki` 中責任重大的「小型物件配置器」，以及勞苦功高的 `smart pointers` 和泛化 `functors`。

4.1 預設的 Free Store 配置器

由於某些「神祕原因」，系統預設的 free store 配置器速度極慢，惡名昭彰。其中一個可能的原因是，它通常只是 C heap 配置器 (malloc/realloc/free) 的淺層包裝。C heap 配置器並未特別針對小塊記憶體的配置進行優化。C 程式通常十分有條理地、保守地使用動態記憶體，決不會採用任何「導致小塊記憶體被大量配置」的手法或技巧。C 程式通常配置中大型物件（數百個或數千個 bytes）。因此 malloc/free 有必要針對小型物件的配置進行優化。

除了速度慢，C++ 預設配置器的通用性也造成小型物件空間配置的低效。預設配置器管理一個記憶體池 (memory pool)，而這種管理通常需要耗用一些額外記憶體。一般而言，對於透過 new 配得的每一塊記憶體，其用於簿記管理的部分達到數個 (4~32 個) bytes。如果配置「大小為 1024 bytes」的區塊，每一區塊的額外開銷微不足道 (0.4%~3%)，但如果配置的物件大小為 8bytes，每個物件的額外開銷就變成了 50%~400%，令人慌目驚心，尤其是如果你需要大量配置這類小型物件的話。

在 C++ 中，動態配置很重要。執行期多型性 (runtime polymorphism) 和動態配置的聯繫最為緊密。「Pimpl 手法」(sutter 2000) 就要求「以 free store 配置取代 stack 配置」為前提。

因此，在邁向高效 C++ 程式開發的道路上，預設配置器的低劣性能成爲一種障礙。老練的 C++ 程式員會盡量避免使用「採行 free store 配置行爲」的語言構件，因為根據經驗他們知道它的成本高昂。預設配置器不僅是個具體問題，還可能成爲一個心理障礙。

4.2 記憶體配置器的工作方式

研究「記憶體在程式中的運用情況」是一項非常有趣的事，這在 Knuth 劃時代著作 (Knuth 1998) 內已經得到證明。Knuth 建立了很多基礎的記憶體配置策略，其後又有更多發明。

記憶體配置器如何運作？它管理一個由 raw bytes (譯註：意指尚未被配置的記憶體) 所組成的記憶體池，能夠從池中配置任意大小的記憶體區塊。簿記結構可以是像這樣的簡單控制塊：

```
struct MemControlBlock {  
    std::size_t    size_;  
    bool          available_;  
};
```

MemControlBlock 物件所管理的區塊緊接其後，大小為 size_ bytes，然後又是另一個控制塊 MemControlBlock，依此類推。

程式開始執行時，記憶體起始處只有一個 MemControlBlock，並將所有記憶體視爲一大塊來管理。這就是所謂 root 控制塊，永不離開最初位置。圖 4.1 顯示程式剛啓動時 1M 記憶體的佈局。

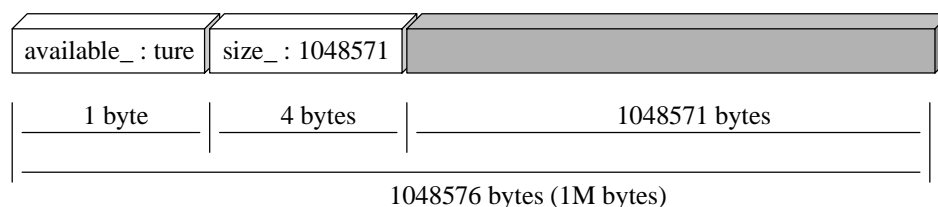


圖 4.1 程式啟動期 (program startup) 的記憶體映射圖 (memory map)

每次配置申請都會引發對記憶體的一次線性搜尋，以求找到一個合適區塊（等於或大於申請值）來滿足需求。滿足需求的策略之多之怪，令人驚訝。你可以使用最先匹配法則（first fit）、最佳匹配法則（best fit）、最差匹配法則（worst fit），甚至隨機匹配法則（random fit）。有趣的是最差匹配比最佳匹配好——呃，怎麼會有如此的矛盾呢？

每次歸還 (deallocate) 區塊，同樣需要一次線性搜索，找出待還區塊的前一區塊並調整其大小。

正如你所看到的，此一策略在時間上並非特別高效。但它在空間上的額外開銷相當小——對於每一區塊，只需額外付出一個 `size_t` 和一個 `bool`。大多數實際場合下你甚至可以將 `size_` 中的一個 bit 挪作 `available_` 之用，從而將 `MemControlBlock` 包網 (pack) 至極限：

```
// platform- and compiler- dependent code
struct MemControlBlock {
    std::size_t    size_ : 31;
    bool          available_ : 1;
};
```

如果將指向前一個和指向下一個 `MemControlBlock` 的指標儲存在每個 `MemControlBlock` 中，就可以得到常數時間的記憶體歸還動作。這是因為，根據待還區塊，我們可以直接取得臨近的區塊並調整之。這種控制塊的結構必然是：

```
struct MemControlBlock {
    bool available_ ;
    MemControlBlock* prev_;
    MemControlBlock* next_;
};
```

圖 4.2 顯示這樣一種支持 doubly linked list (雙向串列) 區塊的記憶體池佈局。你可以看到其中不再需要 `size_`，因為我們很容易以 `this->next - this` 計算出區塊大小。然而每一塊配置而得的記憶體，必須承擔兩個指標和一個 `bool` 的額外開銷。當然你也可以使用因平台而異的某種技巧，將上述的 `bool` 和指標包網 (pack) 在一起。

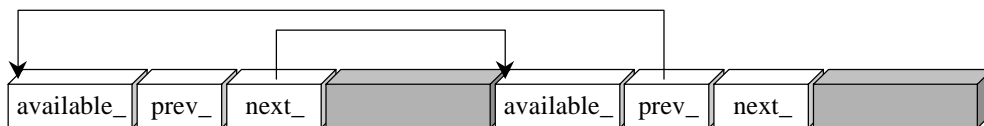


圖 4.2 本圖所示的配置器可擁有常數時間的區塊歸還動作

儘管如此，配置動作還是得消耗線性時間。要減輕這個問題，有很多巧妙技術可用，每一種技術各有利弊。有趣的是，沒有任何一種記憶體配置策略堪稱完美；每一種策略在某種情況下的執行效率都可能比其他方式差。

我們並不需要將「通用型配置器」最佳化。我們的焦點放在可最佳處理小型物件的「專用型配置器」身上。

4.3 小型物件配置器 (Small-Object Allocator)

本章介紹的小型物件配置器分為四層結構，如圖 4.3 所示。下層提供功能供上層使用。最底層是 Chunk 型別。每一個 Chunk 物件包含並管理一大塊記憶體（所謂 chunk），此大塊記憶體本身包含整數個固定大小的區塊（block）。Chunk 內含邏輯資訊，使用者可根據它來配置和歸還區塊。當 chunk 之中不再剩餘 blocks 時，配置失敗並傳回零。

第二層是 FixedAllocator class，其物件以 Chunk 為構件。FixedAllocator 主要用來滿足那些「累計總量超過 Chunk 容量」的記憶體配置請求。FixedAllocator 會透過一個 array（譯註：其實是個 vector）將 Chunks 組合起來以達成目的。如果出現一筆新申請，但現有的 Chunks 都被佔用了，此時 FixedAllocator 會產生（配置）一塊新 Chunk，並將它添入 array（譯註：其實是 vector）內，再由新 Chunk 滿足需求。

第三層 SmallObjectAllocator 提供的是通用性的配置/歸還函式。此物擁有數個 FixedAllocator 物件，每一個負責配置某特定大小的物件。根據「申請的 bytes 個數」不同，SmallObjectAllocator 物件會將記憶體配置申請分發給轄下某個 FixedAllocator。但如果請求量過大，會轉發給預設的（系統提供的）::operator new。

最後一層是 SmallObject，它包裝 FixedAllocator 以便向 C++ classes 提供封裝良好的配置服務。SmallObject 重載 operator new 和 operator delete，將任務轉給 SmallObjectAllocator 物件去完成。採用這種方法，你可以讓你的物件享受專用配置器的好處，而這一切只需你的物件從 SmallObject 衍生出來就可以辦到。

你也可以直接使用 SmallObjectAllocator 和 FixedAllocator（Chunk 則過於原始，而且不夠安全，因而被定義在 FixedAllocator 的 private 區段內），但大多數情況下客端程式碼只需衍生自 SmallObject 就可以運用高效率配置行為。這是一個十分簡單易用的介面。

Small Object	* 提供物件層次（object level）的服務 * 通透性 — classes 只需繼承自 Small Object 即可享受服務
Small Object Allocator	* 能夠配置多種不同大小的小型物件 * 可根據參數進行組態（configurable）
Fixed Allocator	* 配置特定大小的物件
Chunk	* 根據某個特定大小來配置物件 * 物件配置數量的上限是固定的

圖 4-3 小型物件配置器（small object allocator）的四層構造

4.4 Chunks（大塊記憶體）

每一個 Chunk 物件包含並管理一大塊記憶體（chunk），其中包含固定數量的區塊（blocks）。你可以在建構期間設定區塊的大小和數量。

Chunk 包含邏輯資訊，讓你得以從該大塊記憶體中配置和歸還區塊。一旦 chunk 之中沒有剩餘區塊，配置函式便傳回零。

Chunk 定義如下：

```
// Nothing is private - Chunk is a Plain Old Data (POD) structure
// structure defined inside FixedAllocator
// and manipulated only by it
struct Chunk
{
    void Init(std::size_t blockSize, unsigned char blocks);
    void Release();
    void* Allocate(std::size_t blockSize);
    void Deallocate(void* p, std::size_t blockSize);
    unsigned char* pData_;
    unsigned char
        firstAvailableBlock_,
        blocksAvailable_;
};
```

除了以一個指標指向被管理之記憶體本身，Chunk 還保存以下整數值：

- firstAvailableBlock_，chunk 內的第一個可用區塊的索引號
- blocksAvailable_，chunk 內的可用區塊總數

Chunk 的介面非常簡單。Init() 用於初始化，Release() 用來釋放已配得的記憶體。Allocate() 用來配置一個區塊，Deallocate() 用來歸還某個區塊。你必須傳給 Allocate() 和 Deallocate() 一個區塊大小值，因為 Chunk 不保存它（譯註：如果是 system new 就會以所謂 "cookie" 保存它）。區塊的大小在較高層級中已知，如果此處多設一個 blockSize_ 成員，會給 Chunk 造成空間和時間上的浪費 — 不要忘記我們現在處於最底層，每件事情每樣東西都至關重要。基於效率上的考量，Chunk 並未定義建構式、解構式和 assignment（賦值）運算子。在這一層定義自己的 copy 語意會損及上一層的效率 — 上一層將 Chunks 儲存於一個 vector 內。

Chunk 結構反映出設計中的一個重要折衷。由於 blocksAvailable_ 和 firstAvailableBlock_ 都是 unsigned char 型別，因此一個 Chunk 在一部 8-bit char 機器上無法擁有 255 個以上的區塊。很快你就會看到，這個決定還不賴，可幫你省去許多頭疼的事。

現在看看最有趣的部份。區塊(s) 若非正被使用，就是尚未被使用。未被使用的區塊當然可以拿來儲存任何東西。是的，我們要善用這一點，拿「未被使用的區塊」的第一個 bytes 來放置「下一個未被使用的區塊」的索引號。由於 firstAvailableBlock_ 已經持有第一個可用區塊的索引號，因此我們便有了一個由「可用區塊」組成的完整單向串列（singly linked list），無須佔用額外記憶體。

初始化時，Chunk 物件看起來像圖 4.4 那樣。初始化函式如下：

```
void Chunk::Init(std::size_t blockSize, unsigned char blocks) {
    pData_ = new unsigned char[blockSize * blocks];
    firstAvailableBlock_ = 0;
    blocksAvailable_ = blocks;
    unsigned char i = 0;
    unsigned char* p = pData_;
    for (; i != blocks; p += blockSize) {
        *p = ++i;
    }
}
```

融合於這一資料結構內的單向串列（singly linked list）真是個好東西。它提供了一種快速、高效的方法來尋找這一大塊記憶體中的可用區塊，卻不佔用額外空間。在 Chunk 內配置和歸還區塊時，之所以只消耗常數時間，得特別感謝此一內嵌單向串列。

現在你應該可以明白為什麼我要將區塊數量限制在 unsigned char 的大小了。假設我們使用一個較大型別，例如 unsigned short（它在很多機器上是 2 bytes），我們將遭遇兩個問題，一個是大問題，一個是小問題。

- 我們無法配置小於 sizeof(unsigned short) 的區塊，這真令人尷尬，因為我們現在正打算建立一個小型物件配置器。這是我所謂的小問題。
- 我們會遇到齊位（alignment）問題。假設你為一個 5 bytes 區塊建立一個專屬配置器。這種情況下如果想將「指向如此一個 5 bytes 區塊」的指標轉換為 unsigned int，會造成不確定（未定義）的行為。這是我所謂的大問題。

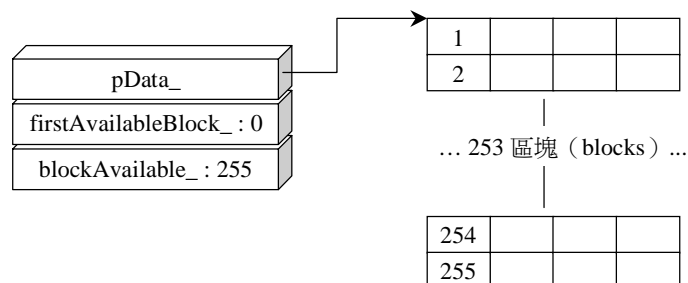


圖 4.4 一個內含 255 個區塊的 chunk，每個區塊大小都是 4 bytes。

解決辦法很簡單：以 `unsigned char` 作為這個「特異索引號」的資料型別。根據定義，`char` 的大小為 1，無齊位問題，因為即使指向 raw memory 的指標也是指向 `unsigned char`。

這個設計會限制 chunk 所含區塊（blocks）的最大數量。chunk 所擁有的區塊數將無法多於 `UCHAR_MAX`（其值在大多數系統中為 255）。這一點可以接受，即使區塊真的非常小，例如 1~4 bytes。對於較大區塊，這個限制沒什麼差別，畢竟我們不想配置太大的 chunks。

配置函式 `Allocate()` 的動作是取出 `firstAvailableBlock_` 所代表的區塊，然後調整 `firstAvailableBlock_` 使指向下一個可用區塊。這是典型的 list 操作。

```
void* Chunk::Allocate(std::size_t blockSize)
{
    if (!blocksAvailable_) return 0; // 譯註：一個比較動作
    unsigned char* pResult =          // 譯註：一個賦值動作
        pData_ + (firstAvailableBlock_ * blockSize); // 譯註：一個索引存取
    // Update firstAvailableBlock_ to point to the next block
    firstAvailableBlock_ = *pResult; // 譯註：一個賦值動作和一個提領動作
    --blocksAvailable_;             // 譯註：一個遞減動作
    return pResult;
}
```

`Chunk::Allocate()` 的成本是：一個比較動作、一個索引存取、一個提領（dereference）動作、兩個賦值動作、一個遞減動作；成本很小。最重要的是不需搜尋動作。目前為止一切良好。圖 4.5 顯示第一次區塊配置完成後，Chunk 物件的佈局。

歸還函式 `Deallocate()` 的行為完全相反：將區塊傳回給自由串列（free list），然後累加 `blocksAvailable_`。不要忘記，由於 `Chunk` 對區塊大小一無所知，所以你必须將區塊大小當作參數傳給 `Deallocate()`。

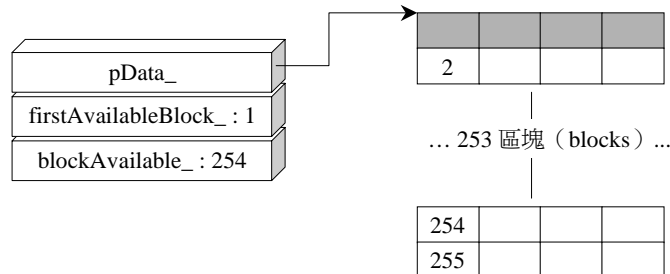


圖 4.5 chunk 經過一次配置後的情況。被配置的區塊以灰色呈現。

```
void Chunk::Deallocate(void* p, std::size_t blockSize)
{
    assert(p >= pData_);
    unsigned char* toRelease = static_cast<unsigned char*>(p);
    // Alignment check
    assert((toRelease - pData_) % blockSize == 0);
    *toRelease = firstAvailableBlock_;
    firstAvailableBlock_ = static_cast<unsigned char>(
        (toRelease - pData_) / blockSize);
    // Truncation check
    assert(firstAvailableBlock_ ==
        (toRelease - pData_) / blockSize);
    ++blocksAvailable_;
}
```

上述的歸還函式很精煉，但有很多 assertions（而且尚未能夠考量所有出錯狀態）。Chunk 秉承了 C 和 C++ 在記憶體配置上的重要傳統：準備面對最壞情況——也就是你將錯誤指標傳給 `Chunk::Deallocate()`。

4.5 大小一致（Fixed-Size）的配置器

小型物件配置器的第二層由 `FixedAllocator` 構成。此物知曉如何配置和歸還「特定大小的區塊」，其大小不受限於 chunk，只受限於系統可用記憶體量。

為達到這一點，`FixedAllocator` 將 `Chunk` 物件集合存放到一個 `vector` 中。任何時候出現配置請求，`FixedAllocator` 便找出一個適當的 `Chunk` 企圖滿足之。如果所有 `Chunk` 都用光了，`FixedAllocator` 會添一個新 `Chunk`。下面是 `FixedAllocator` 定義式中的相關部份：

```
class FixedAllocator
{
    ...
private:
    std::size_t          blockSize_;
    unsigned char        numBlocks_;
    typedef std::vector<Chunk> Chunks;
    Chunks chunks_;
```

```

    Chunk* allocChunk_;
    Chunk* deallocChunk_;
};

```

爲提高搜尋速度，面對每一次配置，FixedAllocator 並非巡訪整個 `chunks_` 尋找空間。它保存一個指標 `allocChunk_` 指向「最近一次配置所使用的 `chunk`」。任何時候只要出現配置請求，FixedAllocator::Allocate() 首先檢查 `allocChunk_`，看看是否有可用空間。如果 `allocChunk_` 尚有可用空間，配置請求將藉由 `allocChunk_` 瞬間獲得滿足。否則就會引發一次線性搜尋（甚至可能會有個新 `Chunk` 添加到 `chunks_ vector` 中）。無論哪一種情況，`allocChunk_` 都會更新，指向剛找到的或新添加的 `chunk`。採用這種方法可以增加「提高下次配置速度」的可能性。以下程式碼實作出上述演算法：

```

void* FixedAllocator::Allocate()
{
    if (allocChunk_ == 0 || allocChunk_->blocksAvailable_ == 0) {
        // No available memory in this chunk
        // Try to find one
        Chunks::iterator i = chunks_.begin();
        for (;;) {
            if (i == chunks_.end()) {
                // All filled up - add a new chunk
                chunks_.push_back(Chunk());
                Chunk& newChunk = chunks_.back();
                newChunk.Init(blockSize_, numBlocks_);
                allocChunk_ = &newChunk;
                deallocChunk_ = &chunks_.front();
                break;
            }
            if (i->blocksAvailable_ > 0) {
                // Found a chunk
                allocChunk_ = *i;
                break;
            }
        }
    }
    assert(allocChunk_ != 0);
    assert(allocChunk_->blocksAvailable_ > 0);
    return allocChunk_->Allocate(blockSize_);
}

```

運用這個策略，FixedAllocator 可以在常數時間內滿足大多數配置請求，只偶爾會因爲搜尋或添加新區塊而變慢。某些特殊的記憶體配置狀況會使上述策略效率不高，但現實世界中那種情況並不頻繁。別忘了，每一種配置器都有弱點，就像阿契里斯 (Achilles) 的腳踵一樣（[譯註](#)：希臘神話中的阿契里斯，除了腳踵，渾身刀槍不入）。

記憶體歸還 (deallocation) 比較棘手，因為歸還的時候缺少一則資訊 — 我們擁有的只是一個指標指向待還區塊，我們不知道那個指標屬於哪個 Chunk。是的，我們可以巡訪 `chunks_`，檢查指標是否落在 `pData_` 和 `pData_ + blockSize_ * numBlocks_` 之間。如果是，就將指標傳給那個 Chunk 的 `Deallocate()`。問題是這麼做很耗時間。雖然配置速度很快（常數時間），歸還卻需耗用線性時間，這不好。我們需要另一種方法來提高歸還速度。

我們可以為已歸還的區塊增設一個快取 (cache，高速緩衝) 儲存區。當客端程式透過 `FixedAllocator::Deallocate(p)` 歸還一個區塊時，`FixedAllocator` 並不將 `p` 傳回給對應的 Chunk，而是將 `p` 添加到一塊內部儲存區，那是一塊用以保存可用區塊的快取裝置內。一旦出現新的配置請求，`FixedAllocator` 首先在快取區中搜尋，如果快取區不為空，就立即從中取出一個可用指標。這是速度很快的操作。只有當快取區用罄，`FixedAllocator` 才走標準途徑，將配置請求轉給一個 Chunk。這是一個很有價值的策略，但對於小型物件經常使用的某些配置和歸還狀況而言，它表現得並不理想。

配置小型物件時有四種主要傾向：

- 批量配置。一次配置很多小型物件。當你初始化一群指標，而它們都指向小型物件，就屬於這種情況。
- 以相同次序歸還。很多小型物件的歸還次序和配置次序相同。大多數 STL 容器被摧毀時就會發生這種情況¹²。
- 以相反次序歸還。很多小型物件的歸還次序和配置次序相反。當你在 C++ 程式中呼叫「操作小型物件」的函式時，這種情況很自然會發生。函式引數和暫時性 stack 變量都屬此類。
- 蝶式 (Butterfly) 配置和歸還。物件的生成和銷毀不遵循一定順序。當你的程式正在運行並偶爾需要小型物件時，這種情況會發生。

快取 (caching) 策略非常適合「蝶式配置和歸還」情況，因為如果配置和歸還隨機發生，這種策略能使它們持續保持快速。但對於批量配置和歸還，快取裝置無甚幫助，更糟的是它甚至會降低歸還速度，因為快取裝置本身的清理也需要時間¹³。

一個較好的策略是採用與配置相同的概念。成員變數 `FixedAllocator::deallocChunk_` 指向歸還動作所用的最後那個 Chunk 物件。任何時候只要發生歸還動作，首先便檢查 `deallocChunk_`。然後，如果那是個錯誤的 chunk，`Deallocate()` 會執行一次線性搜索，滿足需求並更新 `deallocChunk_`。

¹² 標準 C++ 並未定義標準容器內的物件解構次序，因此每一位實作人員都需要自己抉擇。容器往往經由 *forward* 迭代器被摧毀。然而某些實作者會選擇比較「自然」的次序：他們會以相反次序來摧毀物件。基本理由是 C++ 物件以其生成次序的相反次序被摧毀。

¹³ 我無法舉一個適當的快取方案來討論它在「同序歸還」和「逆序歸還」是否一樣好。永遠是魚與熊掌不可得兼。由於兩種傾向都很可能發生，所以快取 (caching) 其實不是個好選擇。

針對先前所列的配置情況，有兩個重要調整可以提高 `Deallocate()` 的速度。首先 `Deallocate()` 從 `deallocChunk_` 附近開始搜尋合適的 `Chunk`。也就是說 `chunks_` 的搜尋是從 `deallocChunk_` 開始，以兩個迭代器（`iterators`）分別向上和向下進行。對於以正序或逆序進行的批量歸還動作，這可以大大提高速度。批量配置期間 `Allocate()` 按序添加 `chunks`，歸還期間要不立即發現 `deallocChunk_` 就是目標，要不就在下一步找到正確的 `chunk`。

第二個調整是避免邊界條件的發生。假設 `allocChunk_` 和 `deallocChunk_` 都指向 `vector` 的最後一個 `Chunk`，而該 `chunk` 已無剩餘空間。那麼，假設執行以下程式碼：

```
for (...)
{
    // Some smart pointers use the small-object
    // allocator internally (see Chapter 7)
    SmartPtr p;
    ... use p ...
}
```

迴圈的每一次迭代都會生成（而後銷毀）一個 `SmartPtr` 物件。生成時由於沒有更多記憶體，`FixedAllocator::Allocate()` 會產生一個新 `Chunk`，並將它添加到 `chunks_ vector` 中。銷毀時 `FixedAllocator::Deallocate()` 會檢測到一個空區塊並將它歸還。每一次迭代，上述昂貴過程便重複一次。

這樣的低效讓人無法接受。因此歸還過程中，只有當「存在兩個空 `chunk` 時」，其中一個 `chunk` 才會被釋放。如果只有一個空 `chunk`，它會被高效地置換至 `chunk_ vector` 尾部。這樣我們便能避免昂貴的 `vector<Chunk>::erase()` 動作，因為我們永遠只需刪除最後一個元素。

當然，某些情形會使上述簡單的設想失效。如果你在迴圈內配置一個 `vector`，每個元素都是 `SmartPtr`，具有合適大小，那麼你會回到老問題上。但是這種情形較少出現。而且就像本章簡介所提，任何配置器都可能在某個特定情況下表現得比其他配置器差。

這樣的歸還策略也適合蝶式配置（`butterfly allocation`）。縱使不按一定次序配置資料，程式也具有某種地域性（`locality`）傾向。也就是說它們一次只存取少量資料。指標 `allocChunk_` 和 `deallocChunk_` 可以良好處理這種情況，因為它們就像「最近一次配置和歸還」的快取裝置。

結論是，我們現在有了一個 `FixedAllocator class`，能夠滿足特定大小的區塊配置請求，速度和記憶體運用效率都令人滿意，並且針對小型物件配置的典型情況進行了最佳化。

4.6 SmallObjAllocator Class

本章配置器分層架構中的第三層是 `SmallObjAllocator`，這是個 `class`，能夠配置任意大小的物件。`SmallObjAllocator` 藉由「聚集數個 `FixedAllocator` 物件」來達到這一服務。當 `SmallObjAllocator` 收到一個配置請求時，也許將該配置請求轉給最佳匹配的 `FixedAllocator`，要不就轉給預設的 `::operator new`。

下面是 `SmallObjAllocator` 的概貌，文字說明列於程式碼之後。

```
class SmallObjAllocator
{
public:
    SmallObjAllocator(
        std::size_t chunkSize,
        std::size_t maxObjectSize);
    void* Allocate(std::size_t numBytes);
    void Deallocate(void* p, std::size_t size);
    ...
private:
    std::vector<FixedAllocator> pool_;
    ...
};
```

上述建構式接受兩個參數，用以對 `SmallObjAllocator` 進行組態設定（configure）。第一參數 `ChunkSize` 代表 `chunk` 的預設大小（`Chunk` 物件的長度皆以 `bytes` 計算），第二參數 `maxObjectSize` 是所謂「小型物件」的最大認可值 — 所有小型物件皆需小於此值。如果申請的區塊大小超過 `maxObjectSize`，`SmallObjAllocator` 會將請求轉給 `::operator new`。

奇怪的是 `Deallocate()` 有一個參數用來表示「待歸還大小」。這麼做是為了讓配置更快，否則這個函式就不得不搜尋 `pool_` 中的所有 `FixedAllocator`，以期找到第一參數（一個指標）所屬的那個 `FixedAllocator`。這成本太高了，所以 `SmallObjAllocator` 要求你傳入待歸還區塊的大小。下一節你會看到，這一任務由編譯器優雅地處理掉了。

`FixedAllocator` 區塊的大小和 `pool_` 之間有什麼映射關係？換句話說如果給出一個大小，哪個 `FixedAllocator` 會負責這類區塊的配置和歸還任務？

一個簡單而且高效的作法是，讓 `pool_[i]` 處理大小為 `i` 的物件。首先初始化 `pool_`，使其大小為 `maxObjectSize`，然後初始化每一個相應的 `FixedAllocator`。一旦接到 `numBytes` 配置請求，`SmallObjAllocator` 便將該請求轉給 `pool_[numBytes]`（引發一個常數時間的操作），抑或轉給 `::operator new`。

然而，這個方案並不如看上去那麼巧妙。「有效果」並不總是意味「有效率」。問題在於你可能只需少量配置器，用以配置特定大小的物件（具體情況取決於你的應用程式）。例如也許你只需產生 4 `bytes` 和 64 `bytes` 兩種物件，再沒其他的了。這種情況下你但是還是得為 `pool_` 配置 64 個或更多個元素，雖然你只使用其中兩個。

齊位（alignment）和填補（padding）會進一步造成 `pool_` 的空間浪費。很多編譯器會為所有「客戶自定型別」進行填補，使其大小為某數（2, 4, 或更大）的倍數。如果編譯器將所有結構都填補為 4 的倍數，你就會只用到 `pool_` 的 25%，其餘都被浪費了。

因此，比較好的作法是：為節約記憶體而犧牲一點點搜尋速度¹⁴。只有「某種大小的配置需求」至少發生一次，我們才儲存對應的 `FixedAllocator`。採用這種方法，`pool_` 就可以容納各種物件大小而不需要成長太多。如果要提高搜尋速度，可以將 `pool_` 按區塊大小排序。

為改善搜尋速度，我們可以採用 `FixedAllocator` 所採用的相同策略。讓 `SmallObjAllocator` 保存兩個指標，分別指向最近一次配置或最近一次歸還所用的 `FixedAllocator`。以下完整列出 `SmallObjAllocator` 的成員變量：

```
class SmallObjAllocator {
    ...
private:
    std::vector<FixedAllocator> pool_;
    FixedAllocator* pLastAlloc_;
    FixedAllocator* pLastDealloc_;
};
```

一旦發生配置請求，首先檢查 `pLastAlloc_`。如果大小不對，`SmallObjAllocator::Allocate()` 會在 `pool_` 身上執行二分搜尋（binary search）。歸還請求亦以類似方式處理，唯一區別是 `SmallObjAllocator::Allocate()` 可以在 `pool_` 中插入一個新的 `FixedAllocator` 物件。

就像先前對 `FixedAllocator` 的討論一樣，這個簡單的快取（caching）方案對於批量配置和歸還很有效，其操作效率為常數時間。

4.7 帽子下的戲法

本章小型物件配置器的第四層結構為 `SmallObject`。這是個 `base class`，將 `SmallObjAllocator` 提供的功能做更方便運用的包裝。

`SmallObject` 重載了系統提供的 `operator new` 和 `operator delete`。這麼一來只要你生成一個 `SmallObject` 衍生物件，重載後的行為會加入整體行動之中，於是將配置請求發送給前述的「定量配置器」。 `SmallObject` 的定義非常簡短，只不過情節可能有點複雜：

```
class SmallObject {
public:
    static void* operator new(std::size_t size);
    static void operator delete(void* p, std::size_t size);
    virtual ~SmallObject() {}
};
```

看起來非常簡短，但有些小地方要注意。很多 C++ 書籍（例如 Sutter 2000）告訴我們：如果想在 `class` 中重載系統預設的 `operator delete`，就必須以一個「指向 `void` 的指標」作為 `operator delete` 的唯一參數。

¹⁴ 現代系統中，你可以在使用較少記憶體的同時，寄望速度的增加。這是因為主記憶體（main memory）和快取記憶體（cache memory）之間有很大的差異：前者量大而慢，後者量小而快。

這是 C++ 的一個漏洞，我們對此漏洞很感興趣（請回憶一下，我們設計 `SmallObjectAllocator` 時是將「待歸還區塊之大小」設計為一個參數）。在標準 C++ 中，你其實可以藉由兩種方式重載系統預設的 `operator delete`，一則是這樣：

```
void operator delete(void* p);
```

再則是這樣：

```
void operator delete(void* p, std::size_t size);
```

Sutter (2000) p.144 對此曾有詳盡的討論。

如果採用第一形式，意味你不在意「待釋區塊之大小」。但事實上我們非常需要知道這個區塊大小，才能將它傳給 `SmallObjectAllocator`。所以 `SmallObject` 採用上述第二形式來重載 `operator delete`。

編譯器如何自動提供物件大小呢？看起來似乎無可避免得增加每個物件的記憶體額外開銷，而額外開銷卻正是本章希望避免的。

不，根本沒有任何額外開銷！請看以下程式碼：

```
class Base {
    int a_[100];
public:
    virtual ~Base() {}
};
class Derived : public Base {
    int b_[200];
public:
    virtual ~Derived() {}
};
...
Base* p = new Derived;
delete p;
```

`Base` 和 `Derived` 大小不同。為避免因儲存「`p` 所指物件的大小」而帶來額外開銷，編譯器玩了個花招：它即時產生一些程式碼用以計算物件大小。有四種技術可以做到這一點，一一列出於下。（偶而轉換角色戴上編譯器設計者的帽子，不失為一件趣事，轉眼之間你可以玩些不可思議的小把戲，而一般程式員卻無法做到☺）

1. 將一個 `Boolean` 旗標傳給解構式，表示「銷毀物件之後要不要喚起 `operator delete`」。Base 擁有一個 `virtual` 解構式，因此本例中的 `delete p` 會作用於正確物件（亦即 `Derived`）上。彼時物件大小可以靜態獲得（亦即 `sizeof(Derived)`），編譯器只需將該值傳給 `operator delete` 即可。
2. 讓解構式傳回物件大小。你可以要求每一個解構式在銷毀物件之後傳回 `sizeof(Class)`（別忘了你現在是編譯器設計者）。這一方案之所以有效是因為 `Base` 擁有一個 `virtual` 解構式。喚起解構式後，C++ runtime（執行期系統）會喚起 `operator delete` 並將解構式回傳值傳

給它。

3. 實作出一個隱藏的 `virtual` 成員函式，例如 `_Size()`，用以獲得物件大小。而後 C++ runtime（執行期系統）呼叫該函式並保存結果、銷毀物件、喚起 `operator delete`。這個作法看起來似乎缺乏效率，優點是 `_Size()` 可另做它用。
4. 在每個 class 的虛擬函式表（virtual function table, vtable）某處直接保存大小。這個作法既靈活又高效，但技術上比較困難。

（現在請摘下編譯器設計者的帽子）如你所見，爲了向你的 `operator delete` 傳遞正確大小值，編譯器費了很大工夫。那麼，每次歸還物件時，何必忽略此值又執行一次昂貴的搜尋呢？這一切都配合得井井有條：`SmallObjectAllocator` 需要知道「待歸還區塊的大小」、編譯器提供其值、`SmallObject` 將該值轉給 `FixedAllocator`。

以上大多數方案都假設你爲 Base 定義了一個 `virtual` 解構式。這再次說明將多型性 (polimorphic) classes 的解構式定義爲 `virtual` 是多麼重要。如果你沒那麼做，萬一你 `delete` 一個 base class 指標，而該指標實際卻指向一個 derived class 物件，就會造成不確定（未定義）行爲。就本章配置器而言，這會使程式在除錯模式下因爲 `assertion` 而中斷執行，在非除錯模式下則造成崩潰。喔，任何人都會同意這種行爲的確屬於「不確定」範疇☹。

爲了讓你不至於總是必須記住這一切，也爲了避免將時間浪費於「不這麼做因而導致的沒日沒夜的除錯」，`SmallObject` 定義了一個 `virtual` 解構式。從 `SmallObject` 衍生出來的任何 classes 都會繼承這個 `virtual` 解構式。這同時也將我們帶到「`SmallObject` 的實作」話題上。

對整個程式而言，我們只需要唯一一個 `SmallObjectAllocator`，它必須被正確地建構並被正確地摧毀。這對它本身而言是個棘手難題。幸運的是透過 `SingletonHolder` template，Loki 完全解決了這個問題。`SingletonHolder` template 將於第 6 章討論（讓讀者跳閱後繼章節實在是很遺憾，但如果浪費這一「復用 `SingletonHolder`」的好機會恐怕我會更遺憾）。眼下請暫時先將 `SingletonHolder` 視爲一個設備（device），借助這個設備我們可以高階管理「某個 class 的唯一實體」。假設 class 名爲 X，我們可以透過 `SingletonHolder<X>` 將它具現化 (instantiate)，而後可呼叫 `SingletonHolder<X>::Instance()` 取得該唯一實體。關於 *Singleton* 設計範式，Gamma 四人之著作（1995）中有詳盡闡述。

有了 `SingletonHolder`，`SmallObject` 的實作就極爲簡單了：

```
typedef SingletonHolder<SmallObjectAllocator> MyAlloc;
void* SmallObject::operator new(std::size_t size) {
    return MyAlloc::Instance().Allocate(size);
}
void SmallObject::operator delete(void* p, std::size_t size) {
    MyAlloc::Instance().Deallocate(p, size);
}
```

4.8 簡單，複雜，終究還是簡單

Small Object 的實作十分簡單。但如果將多緒 (multithreading) 納入考量，就不可能那麼簡單。是的，上述唯一一個 Small Object Allocator 實體 (物件) 被所有 Small Object 實體 (物件) 共享。如果那些實體屬於不同的執行緒，我們實際上就是在多緒之間共享這個 Small Object Allocator。正如本書附錄所言，這種情況下我們必須採取特殊對策。我們似乎得回到小型物件配置器的各層實作細節中，找出關鍵操作，適當增加鎖定 (locking) 機制。

毋庸置疑，多緒確實會給事情帶來一定的複雜度，但還不至於太過複雜，因為 Loki 已經定義了高階的物件同步機制 (synchronization)。邁向復用 (reuse) 的最好途徑就是真正用它，因此我含入 Loki Threads.h，並對 Small Object 作如下修改 (粗體即修改之處)：

```
template <template <class T> class ThreadingModel>
class SmallObject : public ThreadingModel<SmallObject>
{
    ... 如同以往 ...
};
```

operator new 和 operator delete 的定義也需稍作改動：

```
template <template <class T> class ThreadingModel>
void* SmallObject<tm>::operator new(std::size_t size)
{
    Lock lock;
    return MyAlloc::Instance().Allocate(size);
}

template <template <class T> class ThreadingModel>
void SmallObject<tm>::operator delete(void* p, std::size_t size)
{
    Lock lock;
    MyAlloc::Instance().Deallocate(p, size);
}
```

就這樣！不需對底層做任何修改 — 有了高階鎖定 (locking) 機制，它們的功能完全獲得保護。

此處對 Loki 所提供的「單件 (Singleton) 管理」和「多緒特性」的運用，證實了「復用」的強大威力。「全域變數的生命期」和「多緒」兩個題目各有複雜度，如果單純從基本原理出發，試圖在 Small Object 中處理這些問題，將會苦不堪言。是的，心平氣和地想像一下，如果你正如火如荼地實作 Fixed Allocator 的複雜快取 (caching) 功能，卻遇上「多個執行緒將同一物件初始化 (這當然是錯誤的)」的情況…，喔天啊。

4.9 使用細節

本節討論如何在應用程式中使用 `SmallObj.h`。

爲了使用 `SmallObject`，你必須爲 `SmallObjAllocator` 建構式提供適當參數：`chunk` 大小和小型物件的最大尺寸。何謂小型物件？物件多小才算是小型物件呢？

爲了回答這個問題，讓我們回頭看看生成小型物件的目的：我們希望降低「系統預設配置器」附帶的空間和時間上的額外開銷。

「系統預設配置器」帶來的空間額外開銷在不同情況下差異很大，畢竟它也有可能採取類似本章所討論的改善策略。然而對大多數通用型配置器而言，每個物件的開銷在典型桌面系統上大約是 4 ~ 32 bytes。如果額外開銷爲 16 bytes，對一個 64 bytes 物件的浪費率便是 25%，唔，很高。因此 64 bytes 物件應該被視爲小型物件。

如果你讓 `SmallObjAllocator` 處理太大物件，你就會配置出比需求量多得多的記憶體 — 別忘了，即使你釋放所有小型物件，`FixedAllocator` 還是會保留一個 `chunk`。

Loki 允許你選擇，也爲你準備了適當的預設值。`SmallObj.h` 中有三個預處理符號（preprocessor symbols），如表 4.1 所示。面對專案中的所有原始碼，你應該使用同一個預處理符號進行編譯（或乾脆不定義它們而使用預設值）。如果不這麼做，也沒什麼大不了，只不過會產出更多不同大小的 `FixedAllocators`。

預設值是針對「具有合理記憶體容量」的桌面系統而設。如果你將 `MAX_SMALL_OBJECT_SIZE` 或 `DEFAULT_CHUNK_SIZE` 兩者中的任一個定義（`#define`）爲零，`SmallObj.h` 會經由條件編譯產出「只採用系統預設之 `::operator new` 和 `::operator delete`」的程式碼，不帶任何額外開銷。物件介面沒變，但其函式爲 `inline` 函式，而記憶體請求將被轉至系統預設的 `free store`（自由空間）配置器身上。

`class template SmallObject` 原本只有一個參數。爲支持不同的 `chunk` 大小和物件大小，如今另帶兩個 `template` 參數，分別預設爲 `DEFAULT_CHUNK_SIZE` 和 `MAX_SMALL_OBJECT_SIZE`：

```
template
<
    template <class T>
        class ThreadingModel = DEFAULT_THREADING,
        std::size_t chunkSize = DEFAULT_CHUNK_SIZE,
        std::size_t maxSmallObjectSize = MAX_SMALL_OBJECT_SIZE
    >
    class SmallObject;
```

因此，如果使用 `SmallObject<>` 形式，你將獲得一個 `class`，可在預設的多緒模式下運作，並具有記憶體管理方面的預設選擇。

表 4.1：SmallObj.h 的預處理符號（preprocessor symbols）

符號	意義	預設值
DEFAULT_CHUNK_SIZE	chunk 預設大小（單位：byte）	4096
MAX_SMALL_OBJECT_SIZE	SmallObject Allocator 所處理的最大型「小型物件」	64
DEFAULT_THREADING	應用程式所使用之預設執行緒模式。 多緒程式應該將此符號定義為 ClassLevelLockable	繼承自 Thread.h

4.10 摘要

某些 C++ 技術亟需運用「來自於 free store（自由空間）的小型物件」，這是因為 C++ 的執行期多型性（runtime polymorphism）離不開「動態配置」和「pointer/reference 語意」。但系統預設的配置器（以全域的 `::operator new` 和 `::operator delete` 呈現）通常只針對大型物件（而非小型物件）的配置進行優化，造成「預設配置器不適合用來配置小型物件」的現象，因為那會導致速度變慢，而且每個小型物件的記憶體額外開銷也難以忽視。

解決方案是使用專用型小型物件配置器，這是一種經過優化的配置器，專門用來處理小區塊（數十個或數百個 bytes）配置。小型物件配置器使用 chunks（大塊空間），並運用獨特方式將 chunks 組織起來以減少空間和時間上的損失。C++ 執行期系統有助於達成此一目標，因為它可以提供「待還區塊」的大小 — 只要運用較少人知的一種 `operator delete` 重載形式，就可以獲得該大小值。

Loki 的小型物件配置器達到「最大可能速度」了嗎？沒有！Loki 配置器只是在標準 C++ 的限定範圍內工作。正如你在本章所見，面對諸如齊位（alignment）這樣的問題，我們必須保守處理，而保守意味「未達最佳狀態」。但 Loki 配置器的確已經相當快速、簡單和穩定，而且具有可移植的優點。

4.11 小型物件配置器（Small-Object Allocator）要點概覽

- Loki 所實現的配置器有四層結構。第一層由 private type Chunk 組成，它將相等大小的記憶體組織為一個個大區塊（chunks）。第二層為 FixedAllocator，使用一個「具可變長度的 vector」來管理 chunks，以滿足配置需求，使配置總量可至「整個系統的可用記憶體」。第三層 SmallObject Allocator 運用多個 FixedAllocator 物件，得以配置任意大小的物件，其中對小型物件的配置係透過 FixedAllocator 完成，對大型物件的配置則轉由 `::operator new` 完成。最後的第四層由 SmallObject 擔綱，這個 class template 對 SmallObject Allocator 採行進一步包裝。

- `SmallObject` class template 大致定義如下：

```
template
<
    template <class T>
        class ThreadingModel = DEFAULT_THREADING,
        std::size_t chunkSize = DEFAULT_CHUNK_SIZE,
        std::size_t maxSmallObjectSize = MAX_SMALL_OBJECT_SIZE
    >
class SmallObject
{
public:
    static void* operator new(std::size_t size);
    static void operator delete(void* p, std::size_t size);
    virtual ~SmallObject() {}
};
```

- 只要繼承自 `SmallObject`，你的 class 就可以盡享本章小型物件配置器帶來的好處。你可以藉由預設參數 (`SmallObject<>`) 來具現化 (instantiated) `SmallObject` class template，也可以調整其執行緒模式或記憶體配置參數。
- 如欲在多個執行緒中透過 `new` 生成物件，你必須為上述的 `ThreadingModel` 參數選定某個多緒模式。本書附錄對 `ThreadingModel` 有更多介紹。
- `DEFAULT_CHUNK_SIZE` 預設為 4096。
- `MAX_SMALL_OBJECT_SIZE` 預設為 64。
- 你可以運用 `#define` 定義 `DEFAULT_CHUNK_SIZE` 或 `MAX_SMALL_OBJECT_SIZE` (或兩者)，從而造成其預設值被忽略。這些巨集 (macros) 被展開後必須是型別為 `std::size_t` 的常數，或是可轉換為 `std::size_t` 的常數。
- 如果將 `DEFAULT_CHUNK_SIZE` 或 `MAX_SMALL_OBJECT_SIZE` 定義為零，`SmallObject.h` 會透過條件編譯方式產生程式碼，使配置任務直接轉給系統預設的 `free store` (自由空間) 配置器，介面維持不變。萬一你想比較「專用記憶體配置器」使用前後對程式帶來的影響，這就很有用。

參考書目

Bibliography

- Alexandrescu, Andrei. 2000a. Traits: The else-if-then of types. *C++ Report*, April.
— 2000b. On mappings between types and values. *C/C++ Users Journal*, October.
- Austern, Matt. 2000. The standard librarian. *C++ Report*, April.
- Ball, Steve, and John Miller Crawford. 1998. Channels for inter-applet communication. *Doctor Dobb's Journal*, September. Available at <http://www.ddj.com/articles/1998/9809/9809a/9809a.htm>.
- Boost. The Boost C++ Library. <http://www.boost.org>.
- Coplien, James O. 1992. *Advanced C++ Programming Styles and Idioms*. Reading, MA: Addison-Wesley.
— 1995. The column without a name: A curiously recurring template pattern. *C++ Report*, February.
- Czarnecki, Krzysztof, and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Reading, MA: Addison-Wesley.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Järvi, Jaakko. 1999a. *Tuples and Multiple Return Values in C++*. TUCS Technical Report No. 249, March.
— 1999b. The Lambda Library. <http://lambda.cs.utu.fi>
- Knuth, Donald E. 1998. *The Art of Computer Programming*. Vol. 1. Reading, MA: Addison-Wesley.
- Koenig, Andrew, and Barbara Moo. 1996. *Ruminations on C++*. Reading, MA: Addison-Wesley.
- Lippman, Stanley B. 1994. *Inside the C++ Object Model*. Reading, MA: Addison-Wesley.
- Martin, Robert. 1996. Acyclic Visitor. Available at <http://objectmentor.com/publications/acv.pdf>.

- Meyers, Scott. 1996a. *More Effective C++*. Reading, MA: Addison-Wesley.
- 1996b. Refinements to smart pointers. *C++ Report*, November-December.
 - 1998a. *Effective C++*, 2nd ed. Reading, MA: Addison-Wesley.
 - 1998b. Counting objects in C++. *C/C++ Users Journal*, April.
 - 1999. auto_ptr update. Available at http://www.awl.com/cseng/titles/0-201-63371-X/auto_ptr.html.
請注意：Colvin/Gibbons 技法並未出現於任何論文之中。Meyers 對 auto_ptr 所做的註釋是迄今對 Greg Colvin 和 Bill Gibbons 所找出的解法的最精確描述。這個技法乃是利用 auto_ptr 來解決函式回返問題。
- Schmidt, D. 1996. Reality check. *C++ Report*, March. Available at <http://www.cs.wustl.edu/~schmidt/editorial-3.html>.
- 2000. The ADAPTIVE Communication Environment (ACE). Available at <http://www.cs.wustl.edu/~schmidt/ACE.html>.
- Stevens, Al. 1998. Undo/Redo redux. *Doctor Dobb's Journal*, November.
- Stroustrup, Bjarne. 1997. *The C++ Programming Language*, 3rd ed. Reading, MA: Addison-Wesley.
- 2000. Wrapping calls to member functions. *C++ Report*, June.
- Sutter, Herb. 2000. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Reading, MA: Addison-Wesley.
- Van Horn, Kevin S. 1997. Compile-time assertions in C++. *C/C++ Users Journal*, October. Available at <http://www.xmission.com/~ksvhsoft/ctassert/ctassert.html>.
- Veldhuizen, Todd. 1995. Template metaprograms. *C++ Report*, May. Available at <http://extreme.indiana.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html>.
- Vlissides, John. 1996. To kill a singleton. *C++ Report*, June. Available at <http://www.stat.cmu.edu/~lamj/sigs/c++-report/cppr9606.c.vlissides.html>.
- 1998. *Pattern Hatching*. Reading, MA: Addison-Wesley.
 - 1999. Visitor in frameworks. *C++ Report*, November-December.

索引

Index

A

- Abstract Factory design pattern, 69, 40-51
 - architectural role of, 219-222
 - basic description of, 219-234
 - implementing, 226-233
 - interface, generic, 223-226
 - quick facts, 233-234
- AbstractEnemyFactory, 221-222, 224-227
- AbstractFactory, 219-234
- AbstractProduct, 209-214, 216-217
- abstract products, 209-214, 216-217, 222, 231
- Accept, 240-251, 254
- AcceptImpl, 252, 256, 259
- ACE, 309
- Acquire, 161-162, 305, 306
- Action, 100
- Adapter, 284
- Add, 278-282, 284, 300
- address-of, 170-171
- after, 16
- AFUnit, 223-226
- Alexandrescu, Andrei, 29
- algorithms
 - copy, 40
 - compile-time, 76
 - linear search, 56
 - operating on typelists, 76
- Allocate, 83, 85
- Allocated, 82
- allocation, small-object. *See also* allocators
 - basic description of, 77-96
 - default free store allocators and, 78
 - fixed-size allocators and, 84-87
 - hat trick and, 89-91
 - memory chunks and, 81-84
 - quick facts, 94-95
- allocators. *See also* allocation, small-object
 - default free store, 78
 - fixed-size, 84-87
 - memory chunks and, 81-84
 - workings of, 78-79
- allocChunk_, 85, 87
- AllowConversion, 192-193
- ALU (arithmetic logic unit), 303
- AnotherDerived, 197
- APIs (application program interfaces), 161, 306
- Append, 57-58, 76
- Application, 100
- arguments, 114-115, 285-290
- Array policy, 19-20
- arrays, 19-20, 183-184
- ArrayStorage, 189-190
- assert, 193
- AssertCheck, 193
- AssertCheckStrict, 193
- assertions, compile-time, 23-26
- associative collections, 203
- AssocVector, 210, 277-278
- asynchronous execution, 302
- atexit, 134-139, 142-143, 149
- ATEXIT_FIXED, 139
- AtExitFn, 144-145
- AtomicAdd, 305
- AtomicAssign, 305
- AtomicDecrement, 186
- AtomicIncrement, 186
- auto_ptr, 108, 159, 170
- available_, 79

B

backEnd_, 281, 294
 BackendType, 294
 BadMonster, 219-222
 BadSoldier, 219-222, 230
 BadSuperMonster, 219-221
 BankAccount, 306
 Bar, 139
 Base, 63, 90, 91, 176
 BASE_IF, 37
 BaseLhs, 272, 299
 BaseProductList, 227
 BaseRhs, 272, 284, 299
 BaseSmartPtr, 46
 BaseVisitor, 249, 251, 254
 BaseVisitor, 245, 249-250, 252, 254, 256, 261
 BaseVisitorImpl, 260, 262
 BasicDispatcher, 277-284, 292-294, 297-299
 BasicFastDispatcher, 291-294, 297, 298-299
 before, 16
 BinderFirst, 121
 BindFirst, 121, 127-128
 binding, 119-121, 127-128
 BitList, 40-41, 44-46
 blockAvailability_, 81-83
 blockSize_, 82
 bookkeeping data level, 185-187
 bool, 105, 174-175, 177, 191
 _buffer, 135
 bulk allocation, 86
 butterfly allocation, 86
 Button, 50, 61-62, 219-221

C

callable entities, 103-104
 callBackMap_, 280
 callbacks, 103-104, 205, 280-281
 callBacks_, 205, 281
 CallbackType, 277, 280, 299
 CastingPolicy policy, 288, 299
 CatchAll policy, 260, 262
 Chain, 122, 128
 chaining requests, 122
 char, 35, 38, 114-115
 checking issues, 181-182
 checkngimpl, 193
 Checking policy, 15-16, 188, 193-194
 chunks, of memory, 81-87
 chunkSize, 88

Circle, 203
 class(es). *See also* inheritance; policy classes
 base, 228
 client, 153-154
 decomposing, 19-20
 derived, 228
 final, 29
 generating, with typelists, 64-65
 -level locking operations, 187
 local, 28-29
 object factories and, 200-201
 visitable, 248-255
 visitor, 248-255
 Class, 200
 ClassLevelLockable, 153, 307-309
 Clone, 30, 107, 123, 164, 211-213, 230, 232, 234, 284
 CloneFactory, 214-218
 clone object factories, 211-215
 CLOS, 263
 columns_, 292
 COM (Component Object Model), 133, 192
 command(s). *See also* Command design pattern
 active, 102
 forwarding, 102
 Command, 100
 Command design pattern, 99-104
 basic description of, 100-102
 in the real world, 102-103
 comparison operators, 178-181
 compile-time
 assertions, 23-26
 detecting convertibility and inheritance at, 34-37
 CompileTimeChecker, 25-26
 CompileTimeError, 25
 COMRefCounted, 192
 ConcreteCommand, 100, 102
 ConcreteFact, 232, 234
 ConcreteFactory, 226-228, 230-231, 233-234
 ConcreteLifetimeTracker, 144-145
 concrete products, 222, 227
 const, 44, 114-115, 182-183
 constant(s)
 mapping integral, to types, 29-31
 -time multimethods, 290-293
 Conventional Dialog, 219-221
 conversion
 argument, 114-115, 285-290
 binding as, 119-121

- implicit, to raw pointer types, 171-173
- return type, 114-114
- user-defined, 172
- Conversion policy, 36-37, 188, 192-193
- convertibility, detecting, at compile time, 35-37
- copy
 - construction, eliding of, 123
 - deep, 123, 162-164, 192
 - destructive, 168-170
 - on write (COW), 165
- Copy, 40, 45
- copy_backward, 144
- copyAlgo, 45
- CORBA (Common Object Request Broker Architecture), 115, 133
- counting, reference, 165-167
- covariant return types, 212-213
- Create, 9, 11-12, 14, 31-32, 198, 224, 234
- CreateButton, 50
- CreateDocument, 199
- CreateObject, 208-209
- CreateScrollBar, 50
- CreateShape, 205-206
- CreateShapeCallback, 205
- CreateStatic, 153
- CreateT, 223
- CreateUsingMalloc, 153
- CreateUsingNew, 153
- CreateWindow, 50
- Creation policy, 151, 153
- Creator policy, 7-9, 11-14, 149, 154, 156
- cyclic
 - dependencies, 243-248
 - references, 168
- CyclicVisitor, 255-257, 261
- Czarnecki, Krzysztof, 54
- D**
- dead reference problem, 135-142
- Deallocate, 82-87
- deallocChunk_, 86-87
- deep copy, 123, 163-164, 192
- DeepCopy, 192
- DEFAULT_CHUNK_SIZE, 93-94, 95
- default free store allocator, 78
- DefaultLifetime, 153
- DEFAULT_THREADING, 94
- #define preprocessor directive, 93, 139
- DEFINE_CYCLIC_VISITABLE, 257
- DEFINE_VISITABLE, 252, 254, 256
- delete, 12, 89-91, 94, 108, 132, 143, 159, 172-178
- delete[], 184, 189-190
- DeleteChar, 125
- dependency, circular, 141
- DependencyManager, 141
- Deposit, 306
- dereference, checking before, 182
- Derived, 90, 197-198
- DerivedToFront algorithm, 63-65, 76, 272
- design patterns
 - Abstract Factory pattern, 69, 49-51, 219-234
 - Command pattern, 99-104
 - Double-Checked Locking pattern, 146-147, 149
 - Prototype design pattern, 228-233
 - Strategy design pattern, 8
- Destroy policy, 20
- destroyed_, 136, 138
- _DestroySingleton, 135
- DestructiveCopy, 192
- destructors, 12-13
- detection, dead-reference, 135-137
- Dialo, 219-221
- Dijkstra, Edgar, 305-306
- DisallowConversion, 192-193
- DispatcherBackend policy, 294
- DispatchRhs, 269-270
- Display, 135-142, 169
- DisplayStatistics, 230
- do-it-all interface, failure of, 4-5
- DocElement, 236-248, 249, 251, 256, 259
- DocElementVisitor, 239-248
- DocElementVisitor.h, 243-244
- DoClone, 213
- DoCreate, 223-224, 227, 232
- DocStats, 236-237, 240-242, 246-247, 248
- Document, 198
- DocumentManager, 198-199
- DottedLine, 212
- double, 306
- Double-Checked Locking pattern, 146-147, 149
- DoubleDispatch, 267, 268
- double dispatcher, logarithmic, 263, 276-285, 297-300
- double switch-on-type, 264-268
- Dr. Dobb's Journal, 125
- Drawing, 201-203
- DrawingDevices, 290

DrawingType, 203
 Dylan, 263
 dynamic_cast, 238, 243, 246, 251, 255, 267,
 285-290, 299
 cost of, 255-256
 DynamicCaster, 288, 289

E

EasyLevel EnemyFactory, 227, 229, 231
Effective C++ (Meyers), 132
 efficient resource use, 302
 Eisenecker, Ulrich, 54
 ElementAt, 20
 Elixir, 203, 271, 273
 else, 238
 EmptyType, 39-40, 48, 110
 encapsulation, 99
 EnforceNotNull, 15, 18
 enum, 45
 equality, 173-178
 erase, 278
 Erase, 58-59, 76
 EraseAll, 76, 59
 error(s)
 messages, compile-time assertions and, 24
 reporting, 181-182
 EventHandler, 71
 events, 71, 309
 exceptions, 209
 Execute, 100, 102
 Executor, 269-272
 exists2Way, 36
 ExtendedWidget, 18, 164

F

factorie(s)
 basic description of, 197-218
 classes and, 200-201
 generalization and, 207-210
 implementing, 201-206
 need for, 198-200
 quick facts, 216-217
 templates, 216-218
 type identifiers and, 206-207
 using, with generic components, 215
 Factory, 207-208, 215-217
 FactoryErrorImpl, 209
 FactoryError policy, 208-209, 217-218, 234
 FactoryErrorPolicy, 217-218, 234
 FastWidgetPtr, 17

Field, 67-70, 74-75
 Fire, 275, 270, 271
 firstAvailableBlock, 81-83
 FixedAllocator, 80-81, 84-85
 FnDispatcher, 280-282, 285, 288, 293-294,
 299-300
 FnDoubleDispatcher, 280
 Foo, 103, 139
 forwarding functions, cost of, 122-124
 free, 143, 189-190
 fun, 113, 115
 functionality, optional, through incomplete
 instantiation, 13-14
 functions
 forwarding, cost of, 122-124
 static, singletons and, 130
 functor(s)
 argument type conversions and, 114-115
 basic description of, 99-128
 binding and, 119-121
 chaining requests and, 122
 command design pattern and, 100-102
 double dispatch to, 282-285
 generalized, 99-128
 handling, 110-112
 heap allocation and, 124-125
 implementing Undo and Redo with, 125-126
 multimethods and, 282-285
 quick facts, 126-128
 real-world issues and, 122-125
 return type conversions and, 114-115
 Functor1, 106
 Functor2, 106
 FunctorDispatcher, 283-285, 288, 293,
 299-300
 FunctorHandler, 110-112, 117-119, 126
 Functor template, 99-108, 114, 117, 120,
 125-126, 215
 FunctorImpl, 107-112, 123-124, 128, 283, 284
 FunctorType, 284
 FunkyDialog, 219-221

G

GameApp, 230
 Gamma, Ralph, 248
 generalization, 207-210. *See also* generalized
 functors
 generalized functors. *See also* functors
 argument type conversions and, 114-115

- basic description of, 99-128
- binding and, 119-121
- chaining requests and, 122
- command design pattern and, 100-102
- handling, 110-112
- heap allocation and, 124-125
- implementing Undo and Redo with, 125-126
- quick facts, 126-128
- real-world issues and, 122-125
- return type conversions and, 114-115
- GenLi nearHierarchy, 71-75, 227, 230, 231
- GenScatterHierarchy, 64-75, 223-226, 227-228
- geronimosWork, 117
- GetClassIndex, 292-293
- GetClassIndexStatic, 292
- GetImpl, 162, 173, 183, 190
- GetImplRef, 162, 190
- GetLongevity, 154
- GetPrototype, 12, 14
- Go, 269, 270, 272, 279
- GoF (Gang of Four) book, 100, 122, 125, 199, 235, 248-249, 255-262
- granular interfaces, 224
- GraphicalButton, 61-62
- GUI (graphical user interface), 102

H

- handles, 161
- Harrison, Tim, 146
- Haskell, 263
- hat trick, 89-91
- HatchingDispatcher, 272
- HatchingExecutor, 271
- HatchRectanglePoly, 279
- header files
 - DocElementVisitor.h, 243-244
 - Multimethods.h, 294
 - Typelist.h, 51, 55, 75
 - SmallAlloc.h, 93-95
- heaps, 124-125, 189-190
- HeapStorage, 189-190
- hierarchies
 - linear, 70-74
 - scattered, 64-70
- HTMLDocument, 198

I

- IdentifierType, 209, 213, 215
- IdToProductMap, 214

- #ifdef preprocessor directive, 138-139
- if-else statements, 29, 267, 268, 270
- if statements, 238, 267, 305
- IMPLEMENT_INDEXABLE_CLASS, 293
- implicit conversion, to raw pointer types, 171-173
- IncrementFontSize, 240, 241
- indexed access, 55
- IndexOf, 56, 76, 275
- inequality, 173-178
- inheritance
 - detecting, at compile time, 34-37
 - logarithmic dispatcher and, 279
 - multiple, 5-6
- INHERTS, 37
- Int, 40, 82
- initialization
 - checking, 181-182
 - dynamic, 132
 - lazy, 182
 - object factories and, 197
 - static, 132
- insert, 205
- InsertChar, 120, 122, 125-126
- Instance, 131-132, 135-137, 146, 151
- instantiation, 13-14, 120, 272, 274
- int, 159
- Int2Type, 29-31, 68-69
- interface(s)
 - Abstract Factory design pattern, 223-226
 - application program (APIs), 161, 306
 - do-it-all, failure of, 4-5
 - granular, 224
 - graphical user (GUI), 102
 - separation, 101
- intrusive reference counting, 167. *See also*
 - reference counting
- IntType, 186, 304
- InvocationTraits, 275
- isConst, 47
- isPointer, 47
- isReference, 41, 47
- isStdArit, 47
- isStdFloat, 47
- isStdFundamental, 42-43, 47
- isStdIntegral, 47
- isStdSignedInt, 47
- isStdUnsignedInt, 47
- isVolatile, 47

K

KDL problem, 135-142, 155
 Keyboard, 135-142, 155
 Kill PhoenixSingleton, 138
 Knuth, Donald E., 77, 78

L

Lattanzi, Len, 77
 Length, 76
 Less, 180-181
 LhsTypes, 272
 Lifetime policy, 149-153
 LifetimeTracker, 142-143
 LIFO (last in, first out), 134
 Line, 203, 204, 212-213
 linking, reference, 167-168
 List, 52
 ListOfTypes, 295
 Lock, 146, 184, 306
 LockedStorage, 189-190
 locking

- class-level, 307
- object-level, 306-308
- pattern, double-checked, 146-147, 149
- semantics, 306-308

 LockingProxy, 184-185
 Log, 135-142
 logarithmic double dispatcher, 263, 276-285, 297-300
 logic_error, 152
 Loki, 70, 77, 91-92, 210, 303, 309

- multimethods and, 263, 268, 277-278, 288, 295-300
- mutexes and, 306
- smart pointers and, 163

 long double data type, 35
 longevity, 139-145, 149, 151
 Lower_bound, 277

M

MacroCommand, 122
 macros, 122, 251-252, 254, 256-257, 261
 "maelstrom effect," 170
 MAKE_VISIBILITY, 261
 MakeAdapter, 28-29
 MakeCopy, 164
 MakeT, 223
 malloc, 143
 map, 204-205, 210, 277, 278

mapping

- integral constants to types, 31-32
- type-to-type, 31-33

Martin, Robert, 245

maxObjectSize, 88

MAX_SMALL_OBJECT_SIZE, 93-94, 95

MemControlBlock, 78-79

MemFunHandler, 117-119, 126

memory

- allocators, workings of, 78-80
- chunks of, 81-87
- heaps, 124-125, 189-190
- RMW (read-modify-write) operation, 303-304

Meyers, Scott, 77, 133-134, 276, 280

Meyers singleton, 133-134

ML, 263

Modal Dialog, 27

Monster, 219-222, 224, 229

More Effective C++ (Meyers), 276, 280

MostDerived, 63, 76

multimethods

- arguments and, 285-290
- basic description of, 263-300
- constant-time, 290-293
- double switch-on-type and, 265-268
- logarithmic double dispatcher and, 276-285
- need for, 264-265
- quick facts, 297-300
- symmetry and, 273-274

Multimethods.h, 294

Multithreaded, 6

MultithreadedRefCounting, 186-187

multithreading, 145-148, 302-306

- at the bookkeeping data level, 185-187
- critique of, 302-303
- library, 301-309
- mutexes and, 305-306
- at the pointee object level, 184-185
- reference counting and, 186
- reference tracking and, 186-187
- smart pointers and, 184-187

mutex_, 146

mutexes, 146-147, 305-306

MyController, 27

MyOnlyPrinter, 130

MyVisitor, 257

N

name, 206

name cyclic dependency, 243
 new[], 183
 next_, 187
 NiftyContainer, 28-30, 33-34
 NoChecking, 15, 18-19
 NoCopy, 192
 NoDestroy, 153
 NoDuplicates, 60-61, 76
 NonConstType, 47
 nontemplated operators, 176-177
 NonVolatileType, 47
 NoQualifiedType, 47
 NullType, 39-41, 48, 52, 54-56, 62-63, 270

O

object factorie(s)
 basic description of, 197-218
 classes and, 200-201
 generalization and, 207-210
 implementing, 201-206
 need for, 198-200
 quick facts, 216-217
 templates, 216-218
 type identifiers and, 206-207
 using, with generic components, 215
 ObjectLevelLockable, 307-309
 OnDeadReference, 136-139, 150
 OnError, 272
 OnEvent, 70-71
 OnUnknownVisitor, 260, 262
 operators
 operator *, 104, 116-117
 operator !=, 174, 176, 178
 operator(), 103-113, 116, 122-127, 283, 285
 operator->, 157, 160-162, 165, 182-185
 operator->*, 104, 116-117
 operator*, 157, 161-162, 178-179, 182
 operator<, 144
 operator<=, 178-179
 operator=, 162
 operator==, 176-177, 178
 operator>, 178-179
 operator>=, 178-179
 operator[], 55, 183, 278
 operator T*, 172
 OpNewCreator, 10
 OpNewFactoryUnit, 226, 227, 231, 234
 OrderedTypeInfo, 217, 276-277
 orthogonal policies, 20

OutIt, 40
 ownership-handling strategies, 163-170
 Ownership policy, 183-184, 186, 188, 190-192

P

Paragraph, 237, 241, 247, 249, 254, 256
 ParagraphVisitor, 246, 247, 248, 251
 parameter(s)
 template, 10-11, 64, 105
 types, optimized, 43-44
 ParameterType, 43-44, 47, 123
 ParentFunctor, 111, 120-121
 Parm1, 111
 Parm2, 111
 ParmN, 109
 Parrot, 117-119
 pattern(s)
 Abstract Factory pattern, 69, 49-51, 219-234
 Command pattern, 99-104
 Double-Checked Locking pattern, 146-147, 149
 Prototype design pattern, 228-233
 Strategy design pattern, 8
Pattern Hatching (Vlissides), 133
 pData_, 86
 pDocElem, 246
 pDuplicateShape, 212
 pDynObject, 140
 pFactory_, 222
 Phoenix Singleton, 137-142, 149, 153
 pimpl idiom, 78
 plinstance_, 131-132, 136, 138, 146-148, 151
 placement new, 138
 pLastAlloc_, 89
 POD (plain old data) structure, 45-46
 Point3D, 70
 pointer_, 160, 161, 178, 183
 PointerType, 41-42, 47, 160-161
 pointee object level, 184-185
 pointer(s)
 address-of operator and, 170-171
 arrays and, 183-184
 basic description of, 157-195
 checking issues and, 181-182
 copy on write (COW) and, 165
 deep copy and, 163-164
 destructive copy and, 168-170
 equality and, 173-178
 error reporting and, 181-182

failure of the do-it-all interface and, 5
 handling, to member functions, 115-119
 implicit conversion and, 171-173
 inequality and, 173-178
 multithreading and, 184-187
 ordering comparison operators and, 178-181
 ownership-handling strategies, 163-170
 quick facts, 194-195
 raw, 171-173
 reference counting and, 165-167, 186
 reference linking and, 166-168
 reference tracking and, 186-187
 traits of, implementing, 41-42
 types, implicit conversion, 171-173
 PointerToObj, 118
 PointerTraits, 41-42
 PointerType, 160, 190-191
 policies. *See also* policy classes
 basic description of, 3, 7-11
 BasicDispatcher and, 293-294
 BasicFastDispatcher and, 293-294
 compatible, 17-18
 decomposing classes in, 19-20
 enriched, 12
 multimethods and, 293-294
 noncompatible, 17-18
 orthogonal, 20
 singletons and, 149-150, 152-153
 stock, 152-153
 policies (listed by name). *See also* policies
 Array policy, 19-20
 CastingPolicy policy, 288, 299
 CatchAllPolicy, 260, 262
 Checking policy, 15-16, 188, 193-194
 Conversion policy, 36-37, 188, 192-193
 Creation policy, 151, 153
 Creator policy, 8-9, 11-14, 149, 155, 156
 Destroy policy, 20
 DispatcherBackend policy, 294
 FactoryError policy, 208-209, 217-218, 234
 Lifetime policy, 149-153
 Ownership policy, 183-184, 186, 188, 190-192
 Storage policy, 17, 185, 188, 189-190
 Structure policy, 16-17
 ThreadingModel policy, 16, 149, 151-153, 186, 303-309
 policy classes. *See also* classes
 basic description of, 3-22
 combining, 14-16

customizing structure with, 16-17
 destructors of, 12-13
 implementing, 10-11
 Poly, 271, 279
 Polygon, 203, 212
 polymorphism, 78, 163-164
 Abstract Factory implementation and, 228-229
 multimethods and, 264
 object factories and, 197-200, 211-212
 prev_, 187
 Printer, 161-162
 printf, 105
 printingPort_, 130
 priority_queue, 142-145
 ProductCreator, 210-211, 216-217
 Prototype design pattern, 228-233
 PrototypeFactoryUnit, 231-232, 234
 prototypes, 228-233
 pTrackerArray, 143

Q

qualifiers, stripping, 44

R

RasterBitmap, 237, 241, 247, 249, 256
 RasterBitmapVisitor, 247, 251
 realLoc, 143
 Receiver, 100
 Rectangle, 267, 279, 289
 RectanglePoly, 279
 Redo, 125-126
 RefCounted, 6, 192
 RefCountedMT, 192
 reference(s)
 counting, 165-167, 186
 linking, 167-168, 186-187
 ReferencedType, 41-42, 47
 RefLinked, 192
 RegisterShape, 206
 RejectNull, 194
 RejectNullStatic, 193-194
 RejectNullStrict, 194
 Release, 161-162, 192, 305, 306
 Replace, 76
 ReplaceAll, 61, 76
 Reset, 162
 resource leaks, 133
 Result, 55
 ResultType, 111, 269, 284

- return type(s)
 - conversion, 114-115
 - covariant, 228
 - generalized functors and, 114-115
- RISC processors, 148
- RMW (read-modify-write) operation, 303-304
- RoundedRectangle, 272, 286-287, 289
- RoundedShape, 286-287, 289
- runtime_error, 137, 209-210, 300
- runtime type information (RTTI), 215, 245, 255, 276
- S**
- safe_reinterpret_cast, 26
- SafeWidgetPtr, 17
- sameType, 36
- Save, 201-203
- scanf, 105
- ScheduleDeconstruction, 149-150
- Schmidt, Douglas, 146-147
- Scroll, 122
- ScrollBar, 50, 61-62
- Secretary, 5
- Section, 254
- Select, 33-34, 62
- semantics
 - failure of the do-it-all interface and, 4-5
 - functors and, 99
 - locking, 306-308
- semaphores, 309
- SetLongevity, 140-145, 149
- SetPrototype, 12, 14, 232
- Shape, 201-202, 265, 268, 279, 286-289
- ShapeCast, 289
- ShapeFactory, 204-205, 215
- Shapes, 290
- ShillyMonster, 219-222, 226
- ShillySoldier, 219-222, 230
- ShillySuperMonster, 219-222
- SingletonThreaded, 153, 308
- singleton(s). *See also* SingletonHolder
 - basic C++ idioms supporting, 131-132
 - dead reference problem and, 135-142
 - destroying, 133-135
 - double-checked locking pattern and, 146-147
 - failure of the do-it-all interface and, 4-5
 - implementing, 129-154
 - longevity and, 139-145
 - Meyers singleton, 133-134
 - multithreading and, 145-148
 - Phoenix, 137-142, 149, 153
 - static functions and, 130
 - uniqueness of, enforcing, 132-133
- SingletonHolder, 3, 91, 129-130, 148-153, 215.
 - See also* singletons
 - assembling, 150-152
 - decomposing, 149-150
 - quick facts, 155-156
 - working with, 153-156
- SingletonWithLongevity, 153, 154
- sizeof, 25, 34-35, 82, 90-91
- size_t, 36, 79
- skinnable programs, 103
- SmallAllocator, 93-95
- small-object allocation
 - basic description of, 77-96
 - default free store allocators and, 78
 - fixed-size allocators and, 84-87
 - hat trick and, 89-91
 - memory chunks and, 81-84
 - quick facts, 94-95
- SmallObjectLocator, 80-81, 87-89, 92-95
- SmallObject, 80-81, 89, 91-95, 123-124
- smart pointer(s)
 - address-of operator and, 170-171
 - arrays and, 183-184
 - basic description of, 157-195
 - checking issues and, 181-182
 - copy on write (COW) and, 165
 - deep copy and, 163-164
 - destructive copy and, 168-170
 - equality and, 173-178
 - error reporting and, 181-182
 - failure of the do-it-all interface and, 5
 - implicit conversion and, 171-173
 - inequality and, 173-178
 - multithreading and, 184-187
 - ordering comparison operators and, 178-181
 - ownership-handling strategies, 163-170
 - quick facts, 194-195
 - raw, 171-173
 - reference counting and, 165-167, 186
 - reference linking and, 167-168
 - reference tracking and, 186-187
 - types, implicit conversion, 171-173
- SmartPtr, 3, 6-7, 14-19, 44, 87, 157-195
- Soldier, 219-222, 224, 229-230
- SomeLhs, 278, 284

SomeRhs, 278, 284
 SomeThreadingModel, 309
 SomeVisitor, 250, 254
 splmpl_, 112
 statements
 if, 238, 267, 305
 if-else, 29, 267, 268, 270
 static, 280
 static_cast, 114, 285-290, 299
 STATIC_CHECK, 25
 StaticDispatcher, 268-276, 297-298
 static manipulation, 6
 Statistics, 249
 stats, 246-247
 STL, 115, 171
 StorageImpl, 189, 190
 Storage policy, 17, 185, 188, 189-190
 Strategy design pattern, 8
 String, 302-303, 307
 Stroustrup, Bjarne, 202
 struct, 45
 structure(s)
 customizing, with policy classes, 16-17
 POD (plain old data), 45-46
 specialization of, 7
 Structure policy, 16-17
 SuperMonster, 219-222, 229
 SUPERSUBCLASS, 37, 62
 Surprises.cpp, 224
 Sutter, Herb, 77
 switch, 203
 SwitchPrototype, 13-14
 symmetry, 273-274
 synchronization objects, 303

T

template(s)
 advantages of, 6-7
 implementing policy classes with, 11
 skeleton, Functor class, 104-108
 specialization, partial, 53-54
 template parameters, 10-11, 64, 105
 templated operators, 176-177
 TemporarySecretary, 5
 Tester, 178
 Tester*, 178
 TestFunction, 113-115
 TextDocument, 198

ThreadingModel policy, 16, 149, 151-153, 186, 303-309
 time
 separation, 101
 slicing, 201
 TList, 53-65, 75, 120, 127, 223, 231-234, 261
 Tools menu, 102
 trampoline functions (thunks), 280-283
 Trampoline, 281
 translation units, 132
 Triangle, 289
 tuples, generating, 70
 type(s)
 atomic operations on, 303-304
 conversions, generalized functors and, 114-115
 detection of fundamental, 42-43
 identifiers, 201, 206-207
 integral, 303-305
 modifiers, 308-309
 multiple inheritance and, 6
 multithreading and, 303-305
 parameters, optimized, 43-44
 replacing an element in, 60-61
 safety, loss of static, 4
 selection, 33-34
 -to-type mapping, 31-33
 traits, 38-46
 Type2Type, 32-33, 223
 TypeAt, 55, 76
 TypeAtNonStrict, 55, 76, 109
 typedef, 14, 19, 51, 54, 215, 295
 typeid, 38, 267
 type_info, 37-39, 54, 206-207, 213-215, 276-277, 295
 TypeInfo, 38-39, 48
 TYPELIST, 295
 TypeList.h, 51, 55, 75
 typelists, 223, 268, 272
 appending to, 57-58
 basic description of, 49-76
 calculating length and, 53-54
 class generation with, 64-75
 compile-time algorithms operating on, 76
 creating, 52-53
 defining, 51-52
 detecting fundamental types and, 42-43
 need for, 49-51
 partially ordering, 61-64
 quick facts, 75

searching, 56-57
TypesLhs, 269-270, 274
TypesRhs, 269, 270, 274
TypeTraits, 41-48, 123, 183

U

UCHAR_MAX, 83
Undo, 125-126
unique bouncing virtual functions, 238
Unlock, 184
UpdateStats, 237-238
upper_bound, 144
use_Size, 91

V

ValueType, 33
vector, 183
VectorGraphic, 244
VectorizedDrawing, 259
Veldhuizen, Todd, 54
virtual
 constructor dilemma, 229
 functions, 238
Visit, 249-250, 254, 256
VisitTable, 249, 251, 252-254
Visitor design pattern
 acyclic, 243-248

 basic description of, 235-262
 catch-all function and, 242-243, 258-259
 cyclic, 243-248, 254-257
 generic implementation of, 248-255
 hooking variations, 258-260
 nonstrict visitation and, 261
 quick facts, 261-262
VisitParagraph, 240, 242
VisitRasterBitmap, 242
VisitVectorGraphic, 244
Vlissides, John, 133
void*, 172
volatile, 151, 308-309, 308-309
VolatileType, 151, 309

W

Widget, 26-32, 38, 44, 61-62, 159, 164, 184, 309
WidgetEventHandler, 71-74
WidgetFactory, 49-51
WidgetInfo, 65-67
WidgetManager, 9-14, 19-20
Window, 50
Withdrawal, 306

X

X Windows, 103