實戰 C++

— 八個別具特色的實作經驗 —

(**The Art of C++**, by Herbert Schildt)

— 侯 捷 譯 —

出版序

《實戰 *C++* — 八個別具特色的實作經驗》與目前市面上眾多 C++ 書籍的最大不同，在於本書既非基礎觀念之教學書籍，亦非開發工具之使用手冊，而是以「一章一專案」的方式，從實際應用面引領讀者領略 C++。

本書是《*The Art of C++*》的中文譯本。原作者 Herbert Schildt 是一位學有專精、著作等身的 IT 技術作家，其作品普遍獲得良好評價。

本書承侯捷先生慨然相助，接下翻譯工作，在此特別致謝。侯捷先生對於譯作的嚴謹態度，向為兩岸三地讀者肯定。但盼好書加上優質譯者，能夠為讀者們帶來最好的學習經驗。

Herbert Schildt 另著有姊妹作《*The Art of Java*》，其中譯本《實戰 ***Java*** — 八個別具特色的實作經驗》亦由侯捷先生翻譯。

上奇科技出版事業處編輯部

侯捷譯序

市面上充滿了程式語言基本語法書籍、編程工具使用手冊、手把手蝸步緩進的「傻瓜系列」[1]。就像人生在不同層次有不同的體會，這些書籍在您不同的學習過程中也必然各有貢獻。但是當我們上升到某個檔次，驟然發現，高處不勝寒，無書可讀了！

倒不是說我們層次多高，高到無書可讀，而是書市中面向大眾的稍稍高檔次書籍 — 尤其在程式語言及更高階領域 — 很少。做為一個「對業內技術及程式員保持密切觀察」的讀者，我知道這種「稀缺」其實不能反映需求。做為一個技術書籍作者，我也知道，這種「稀缺」其實反映出陽春白雪的必然性。

學習一門語言，基本語法永遠不是問題，也不成其為關鍵，關鍵在於如何適當地運用。運用是最好的練習;實踐是檢驗力量的最佳辦法。這本《實戰 C++ — 八個別具特色的實作經驗》，就是很好的實踐藍本。您不但可以從書中獲得中大型程式的組織和寫作示範，還可以獲得八個不同領域（資源回收、多緒程 *Multithreading*、轉譯器 *Translator*、檔案下載、金融計算、人工智慧搜尋、自訂 STL 容器、直譯器 *Interpreter*）的領域知識。最終，您還獲得了八個可以做為個人開發起點的良好基礎。

本書的行進方式是，以程式碼為主軸，輔以大量說明。對於已經脫離語法學習階段，希望獲得實作經驗的讀者，這種行進方式頗為合適。程式碼很多，作者又不厭其煩地整體性條列、局部性條列，必會有些讀者嫌其囉嗦佔篇幅。但如果反省「買書當成買紙」的不當心態，從而不再以區區紙頁的角度去衡量一本書（的方方面面），而是從整體知識、便利閱讀、節省時間的角度來看，我相信詳盡的解說永遠受歡迎。

[1] 這裡並無不敬之意。與「傻瓜相機」用法同。

在中英術語的採納上，本書視情況時而使用中文，時而使用英文[2]，時而中英並陳。已做悉心安排，並多次檢閱試驗，必不致令讀者混亂困惑。本書目錄末尾列有中英對照表，每章首頁亦列有該章主要術語的中英對照。

本書翻譯過程中，北京大學計算機系辜新星同學協助我對稿件做了前期處理，使我得以快速完成工作，也使本書得以在這個時間點面世。新星的工作很好，品質很高，令我深喜得人。

本書另有姊妹作：《實戰 *Java* — 7 個別具特色的實作經驗》，亦由上奇出版。對於以 Java 為工具的程式員而言，又是一個好選擇。

侯捷 2005/04/20 於臺灣新竹
jjhou@jjhou.com
http://www.jjhou.com（繁體） http://jjhou.csdn.net（簡體）

[2] 使用英文術語的兩大用意：(1) 某些術語業界習慣以英文稱呼，(2) 某些術語譯為中文頗覺彆扭，例如 C++ "reference" 譯為 "參考" 或 "引用"，雖四平八穩但讀之不快，又不具術語突出性。當然，這是譯者的個人取向及其業界經驗和教學經驗使然，您也可能有不同看法。

目 錄

本書術語中英對照

計算機術語何其浩繁。下表只列本書中犖犖大者。各章第一頁另有該章術語摘要整理。

英文術語	中文術語	英文術語	中文術語
abstract	抽象的	access	存取、取用
adapter	配接器	address	位址
algorithm	演算法	allocator	配置器
argument	引數	array	陣列
assignment	賦值、指派	A.I.	人工智慧
batch	批次（整批作業）	cache	快取（區）
check box	核取方塊	class	類別
client	客端、客戶端、客戶	client-server	主從架構
collection	群集	combo box	複合方塊、複合框
command line	命令列	compiler	編譯器
component	組件（用於 GUI 則譯為控件）	concurrent	並行
connection	連接、連線	control	控件
constructor	建構式	escape code	轉義碼
evaluate	評估、求值、核算	event	事件
exception	異常	expression	算式、表達式
file	檔案	flag	旗標
function	函式	GUI	圖形介面
handle	識別碼	header file	表頭檔
import	匯入	instance	實體
instantiate	具現化	interface	介面（編程）、界面（GUI）
Internet	網際網路	interpreter	直譯器
invoke	喚起	library	程式庫
link	連結	list	串列、清單、系列
macro	巨集	memory	記憶體
menu	選單	object	物件
overload	重載	parameter	參數
parse	解析	polymorphism	多型
process	行程	programming	編程、程式設計
progress bar	進度條	source	源碼
statement	述句	stream	資料流、串流
tag	頁籤	thread	緒、緒程
token	語彙單元	type	型別
variable	變數	WWW	萬維網

關於作者

Herbert Schildt 是 C, C++, Java 和 C# 等語言的領導專家，亦擅長編寫 Windows 程式。他的編程書籍全球銷量超過 300 萬冊，並被翻譯成所有主要語言。他是多部 C++ 暢銷書的作者，包括《*C++: The Complete Reference*》,《*C++ From the Ground Up*》,《*C++: A Beginner's Guide*》, 以及《*STL Programming From the Ground Up*》。他的其他暢銷書包括：《*C: The Complete Reference*》,《*Java 2:The Complete Reference*》和《*C#: The Complete Reference*》。Schildt 獲得伊利諾州立大學計算機科學碩士學位。您可透過其諮詢辦公室電話（217）586-4683 與他聯繫。

關於譯者

侯捷 對於 C, C++, Java 和 C# 等語言皆有研究，亦擅長編寫 Windows 程式。他的各種著作和譯作在華人世界擁有 50 萬以上讀者。他的主要著作包括《深入淺出 *MFC*》、《多型與虛擬》、《*STL* 源碼剖析》、《無責任書評》卷 1,2,3、《*Word* 排版藝術》、《左手程式右手詩》,譯作包括：《*C++Primer*》、《*Effective C++*》、《*More Effective C++*》、《*The C++ Standard Library*》、《*Generic Programming and the STL*》、《*Refactoring*》、《*System Programming in Windows 95*》和《*Windows 95 System Programming Secrets*》等十數本。侯捷目前在元智大學、南京大學、同濟大學授課。您可透過電子郵箱 jjhou@jjhou.com 與他聯繫，也可以訪問他的個人網站 http://www.jjhou.com（中文繁體）和 http://jjhou.csdn.net（中文簡體）。

0

作者

By Herb Schildt

從早期的 FORTRAN 年代開始，持續發展的計算機語言走過了一條可被稱之為進化的道路，過去的成就影響著未來的發展。過程中，無用的性質和誤導人的成果被逐漸淘汰。這些年來，這個進化過程推動編程語言從比較純淨的形式變成完全純粹的形式，這才是編程語言純粹的本質所在。這樣的結果就是 C++，沒有任何其他語言在編程歷史上佔據比它更重要的地位。

C++ 的成功有很多因素。它的語法簡潔雅致，它的物件模型（object model）優良並且概念簡潔，它的程式庫設計精良並且環環相扣。然而這些特徵的設計並非是為了讓 C++ 贏得歷史地位，而是為了和 C++ 交到程式員手上的強大力量聯繫起來。沒有任何一種語言，無論先或晚於 C++，曾經給予程式員更多對計算機的直接控制權。擁有 C++，程式員成了機器的主宰 — 這是所有程式員都希望的。

沒有界線，沒有束縛，不受約束。這就是 C++。

本書有些什麼

本書和大多數 C++ 書籍相當不同。其他許多書籍教授語言基礎，本書卻是告訴您如何把 C++ 應用到一些更有趣、更有用，有時候甚至不可思議的的編程工作上。過程中將展現 C++ 語言的威力和優雅。唯有透過 C++ 運用上的藝術，才能將 C++ 設計上的藝術性展露無遺。

本書包含兩大類應用程式。第一類我稱之為「純淨碼」（pure code），它們的焦點放在 C++ 編程環境的擴展。例如第 2 章的垃圾回收器，第 3 章的緒程控制面板和第 8 章的訂製 STL 容器。第二種類型示範如何將 C++ 應用至各種編程任務，例如第 5 章開發了一個可續傳的 Internet 下載器，第 6 章展現如何開發金融計算程式，第 7 章則是將 C++ 用於人工智慧。

本書以一個獨特而有趣的程式結束：一個可直譯 C++ 小型子集的「Mini C++ 直譯器」。Mini C++ 幫助我們深刻理解 C++ 關鍵字和語法如何一起構成這個語言的文法。它讓您「深入語言內部」，為您顯示這個語言設計上的一些背後因素。儘管 Mini C++只是為了好玩，但它可以作為我們開發自己的語言的起點，也可以被改造為另一種語言的直譯器。

每一章都逐步闡明程式碼，您可以毫不改動地直接使用它。例如第 2 章的垃圾回收器就適用於許多編程任務。然而只有當您將這些程式作為自己的開發起點，真正的好處才顯露出來。例如第 5 章的 Internet 檔案下載器可以被強化為「在某個指定時間開始下載」，或者監控下載網址，等到檔案更新後才下載。總之，請把各種程式和次系統當作自己專案的墊腳石。

假設您有這些 C++知識

本書假設讀者擁有堅實的 C++基礎。您應該能夠建立、編譯和執行 C++程式。您應該知道如何使用指標、模板（templates）和異常（exceptions），理解 *copy* 建構式（copy ctor），並熟悉標準程式庫中最常用的部分。本書假設您已經擁有 C++ 初階課程中應該獲得的能力。

如果您需要更新或加強這些基礎知識，我推薦以下書籍：

- *C++ From the Ground Up*
- *C++: A Beginner's Guide*
- *C++: The Complete Reference*

它們都由 McGraw-Hill/Osborne 公司出版。

別忘了：程式碼就在網上

記住，所有實例的源碼和書中專案都在 www.osborne.com 上免費提供。

Herb Schildt 的其他作品

本書只是 Herb Schildt 編程系列書籍中的一冊。下面是您可能也感興趣的其他書籍。

如果想學習更多的 C++ 編程，我們推薦以下書籍：

- *C++: The Complete Reference*
- *C++: A Beginner's Guide*
- *Teach Yourself C++*

- *C++ From the Ground Up*
- *STL Programming From the Ground Up*

如果想學習 Java，您會發現下面這些書特別有用：

- *Java 2: The Complete Reference*
- *The Art of Java (Co-authored with James Holmes)*
- *Java 2: A Beginner's Guide*
- *Java 2: Programmer's Reference*

如果想學習 C#，我們建議以下書籍：

- *C#: A Beginner's Guide*
- *C#: The Complete Reference*

如果想學習更多 C 語言 — 現代編程技術的基礎，下面這些書頗有趣味：

- *C: The Complete Reference*
- *Teach Yourself C*

1

1. C++的威力

The Power of C++

本章中英術語摘要：

allocate⇔配置；class⇔類別；collection⇔群集；library⇔程式庫；memory⇔記憶體；memory leak⇔記憶體洩漏；object⇔物件；object model⇔物件模型；object oriented⇔物件導向；operand⇔運算元；operator⇔運算子；overload⇔重載；postfix⇔後置式（後序式）；programming⇔編程（程式設計）；prefix⇔前置式（前序式）；specifier⇔飾詞；STL⇔Standard Template Library（標準模板庫）；template⇔模板；type⇔型別；

C++能力超凡：能夠從最底層控制機器，能夠產生高度優化的程式碼，能夠與作業系統直接打交道。這種能力作用得既深且廣。有了 C++，您可以細部控制一個 object（物件）的產生、銷毀和繼承，可以存取指標，還可以支援低階 IO。您可以定義 class、重載 operator，藉此加入新特性。您可以打造自己的程式庫，或是加工優化的程式碼。如有必要，您甚至可以「打破常規」。C++不是膽小者的語言，它是為那種要求獲得世上最具威力的語言的人而準備的。

當然，C++ 絕非只是生猛而已。它是一種受引導、集中、受控制的力量。它的謹慎設計、豐富的程式庫和精巧的語法，導致了一個同時具備靈活與敏捷的編程環境。C++ 不僅在產生高效率的系統程式碼上非常出眾，也適合其他任何類型的編程任務。例如它的字串處理能力一流，它的數學和數值運算效率使它十分適合科學方面的編程，它產生「快速目的碼」（fast object code）的能力使它完全適合任何 CPU 密集型任務（CPU intensive task）。

本書目的就是要展現 C++ 的威力、適用範圍和靈活性。透過將 C++ 應用至各式各樣程式，本書達到了目的。有些程式演示了語言本身能力，我們稱之為「純淨碼」，因為這些碼彰顯出 C++ 語法的表現力和其設計上的優美。第 2 章的垃圾回收器和第 9 章的 C++ 直譯器就是此類。另一些程式示範 C++ 多麼容易被應用於一般編程任務。例如第 5 章的下載器就展現出 C++ 建立高效率網絡程式的能力，第 6 章則是將 C++ 應用於各類金融計算。所有這些都表現出 C++ 的多功能和多用途。

然而，在開始接觸各類應用之前，我們應該仔細思考 C++ 成為如此優秀語言的原因。為此，本章將花一些時間指出決定 C++ 強大能力的幾個特性。

1.1. 簡潔而豐富的語法

如果說 C++ 有一種決定性的本質特徵，那就是它那簡潔精鍊的語法。C++ 只定義了 63 個關鍵字。看似矛盾的是，C++ 的強大能力很大程度並非源自於向語言添加不必要的特性，而是它定義了一種豐富且緊湊的語法來支援控制述句、運算子、資料結構，以及任何現代語言必備的物件導向特性。除此之外別無更多。因此，C++ 的語法是簡潔、協調、規整的。

這種精簡哲理有兩個重要好處。第一，C++ 的關鍵字和語法適用於所有可使用 C++ 的環境。也就是說，無論執行環境如何，C++ 的核心特徵在所有程式中普遍適用。對執行環境比較敏感的特徵，例如多緒（multithreading），被留給最有能力高效處理的作業系統來處理。C++ 並不企圖成為一種大小通吃型解決方案，因為這樣做可能導致執行效率下降。

第二，這套精簡而且邏輯連貫的語法可以清楚表達出複雜的構件（constructs）。當程式成長

得過於龐大時，這的確是個重要優點。雖然一個糟糕的程式員會寫出糟糕的 C++ 程式碼，但一個好程式員卻可以編寫出清晰簡練得令人吃驚的程式碼。這種表達複雜邏輯的能力使得 C++ 語法幾乎成為編程界的通用語言。

1.2. 強大的程式庫

當然，現代編程環境除了 C++ 關鍵字和語法外，還要求更多特性。C++ 以非常優雅的方式提供了通往這些特性的途徑：藉由其標準程式庫。C++ 定義了一套在任何主流語言中都有理由被認為是最佳設計的程式庫。在繼承自 C 語言的函式庫中，包含了眾多非物件導向的函式，例如 **char*** 字串處理函式，字元操作函式，以及各地程式員經常使用的轉換函式。C++ 類別庫則提供了對 I/O、字串、STL 等的物件導向支援。

因為 C++ 倚賴程式庫而非關鍵字，所以新功能可以簡單地透過「擴展 C++ 程式庫」加入 C++ 語言中，無需發明新關鍵字。這使得 C++ 能夠在不改變語言核心的情況下適應編程環境的變化。因此，C++ 整合了兩個看似矛盾的特性：穩定性與靈活性。

即使在其函式庫和類別庫中，C++ 也採取了一種「簡潔就是美」的方式，避免大小通吃的陷阱。程式庫僅僅提供那些「為適用於廣泛編程環境而有理由存在」的特性。對於某些特定環境獨有的性質，C++ 的作法是供給存取作業系統的途徑，因此程式員能夠運用執行平台的所有特性。這種作法使您可以寫出能夠儘可能運用執行平台特性和能力的高效程式碼。

1.3. STL

標準類別庫中有一部分非常重要，您有必要特別注意，那就是標準模板程式庫（STL; Standard Template Library）。STL 的誕生非常關鍵，改變了程式員對程式庫的看法和使用。其影響至為深遠，乃至於影響了後來語言的設計。例如 Java 和 C# 的群集框架（Collections frameworks）就直接以它為模範。

從核心上講，STL 是一個複雜的 template classes（模板類別）和 template functions（模板函式）的集合，實現許多普及並被廣泛使用的資料結構 — STL 稱之為容器，例如 vectors、lists、queues 和 stacks。由於 STL 以 template classes 和 template functions 構造而得，因此其容器可應用於幾乎任何一種資料型別。換句話說 STL 提供了對許多編程問題的現成解決方案。

雖然 STL 對 C++ 程式員的實用重要性不至於被低估，不過其實還有更重要的原因使 STL 如此重要。是的，它是軟體組件革命的先頭部隊。盤古開天以來，程式員就不斷尋尋覓覓復用程式碼的辦法。因為開發和除錯是高代價程序，所以程式碼復用非常重要。在早期，程式碼

的復用是透過源碼的剪剪貼貼來實現（當然，今天還是有人使用此法）。後來程式員開發了可復用的程式庫，如 C++ 提供的那些。不久就出現了標準類別庫。

以類別庫為基礎，STL 進一步加深了這個觀念：它將程式庫模組化，使成為一種適用於各種資料的通用（泛型）組件。此外由於 STL 可擴充，所以您可以定義自己的容器，添加自己的演算法，甚至改造 STL 內建容器。這種擴充、改造和復用的能力正是軟體組件的本質精髓。

今天，軟體組件革命已趨完成，我們因而很容易遺忘當初是 STL 鋪設了大部分地基，包括功能模組化、介面標準化，以及透過繼承帶來的可擴充性。將來歷史學家會把 STL 作為語言設計的重要里程碑載入電算史冊！

1.4. 程式員掌握控制權

關於編程語言，存在兩種彼此對立的哲學。一方說，語言本身應該消除那些可能引發問題的特性，以免程式員出錯。雖然這種說法聽起來不錯，但它通常導致某些「功能強大但有潛在危險」的特性被限制、被減弱，甚至被完全略去。指標和顯式記憶體配置（explicit memory allocation）就是兩個例子。指標被認為是有風險的，因為它們常被新手錯用，有時超出安全界限。顯式記憶體配置（例如透過 **new** 和 **delete**）也被認為帶有風險，因為程式員可能不理智地配置大塊記憶體，或忘記釋放不再用到的記憶體從而導致記憶體洩漏（memory leak）。雖然這兩種特性伴隨著風險，但它們也給程式員帶來細緻入微的控制權和創造高效程式的能力。幸運的是 C++ 並不認同這種哲學。

第二種哲學，也就是 C++ 所堅持的，認為「程式員是主宰」。這意味程式員擁有控制權。避免您成為一位糟糕的程式員並不是語言的責任。相反地，語言的主要目標是為程式員提供一個敏捷、不唐突的工作環境。如果您是一位優秀程式員，您的工作會反應出來。如果您的編程能力很差，您的工作一樣會反應出來。本質而言，C++ 賦予您能力，然後走開。C++ 程式員永遠不需要「向語言開火」。

顯然，大多數程式員更喜歡 C++ 的哲學！

1.5. 細緻入微的控制

C++ 不僅給程式員以控制權，而且給的是細緻入微的控制權。試考慮遞增運算子：++，正如您所知道，C++ 同時定義了它的前置（prefix）版本和後置（postfix）版本，前者先累加運算元（operand）然後才獲得其值，後者先獲得運算元然後才累加其值。這使得您得以精準

控制內含 ++ 之運算式的執行。細緻控制權的另一個例子是 **register** 飾詞。以它來修飾某個變數宣告，便是要求編譯器優化對該變數的存取動作。這麼一來您就可以決定哪些變數獲得優化之最高優先權。C++ 給予程式員細緻的控制權，正是它有效取代 assembly（組合）語言成為系統編程熱門選項的一個原因。

1.6. 運算子重載

Operator Overloading

C++ 的一個最重要特性是運算子重載，因為 C++ 支援型別擴展（type extensibility）。所謂「型別擴展」使您能夠添加新的資料型別，並完全將它們整合至 C++ 編程環境。型別擴展建立在兩個特性之上，第一個是 class（類別），它使您得以定義一個新資料型別。第二個是運算子重載，它讓您定義各運算子之於某 class 型別的含義。藉由運算子重載和 class，您可以創造一種新型別，然後按照「與內建型別相同」的方式來操作新型別 — 透過運算子。

型別擴展非常強大，因為它使 C++ 成為一個開放而非封閉系統。舉個例子，假設您需要管理 3D 座標。您可以透過「定義一個名為 **ThreeD** 的新型別，然後定義在此型別物件上的各種運算子」從而完成工作。例如您可以使用 + 號讓兩個 **ThreeD** 座標相加，或定義 == 判斷兩套座標是否相等。您可以編寫程式碼，讓「對 **ThreeD** 的操作」就像「對任何內建型別的操作」一樣，如下所示：

```
L 1-1
ThreeD a(0, 0, 0), b(1, 2, 3), c(5, 6, 7);

a = b + c;
// ...
if(a == c) // ...
```

如果沒有運算子重載，操作 **ThreeD** 物件時可能必須使用 **addThreeD()** 或 **isEqualThreeD()** 等函式來處理，這是一個令人很不愉快的作法。

1.7. 乾淨而有效率的物件模型

A Clean, Streamlined Object Model

C++ 物件模型是簡潔的傑作！ISO *C++ Standard* 只用了不到一頁（精確說是六段）來描述物件模型。在這僅有的數段中，*C++ Standard* 解釋了物件及其生命期以及多型精髓。例如該標準規格這樣定義物件：「物件是一塊儲存區」。正是這種根本的簡潔性使得 C++ 物件模型如此卓越。

當然，語法和語意都必須支援物件，包括其建立、銷毀、繼承等等。*C++ Standard* 用很多篇幅描述這些。然而這是因為 C++ 給予您對物件的深層操作，並非出於其物件模型的怪僻或內在矛盾。此外，由於設計優雅，C++ 物件模型被 Java 和 C# 當作模範。

1.8. C++的餘蔭

Dennis Ritchie 於 1970 年代發明了 C 語言，開啟編程技術根本轉型的先河。雖然某些更早的語言，特別是 Pascal，獲得了重大成功，但 C 才成其為影響一代計算機語言的楷模。有了 C 語言，現代編程世紀才宣告開始。

C 語言發明後不久，一個新技術出現了：物件導向編程（OOP）。雖然今天我們把 OOP 看作理所當然，但在它被發明的那個年代，卻是向前邁出的一大步。物件導向思維很快抓住了程式員的想像力，因為它提供一種完成編程任務的有力新法。在當時，程式規模愈大，複雜度愈是昇高，因此需要一些處理複雜性的方法。OOP 就提供了一種解決方案，可將龐大複雜的程式組織成為按功能劃分的單元（物件）。這就使得複雜系統被簡化為更易管理的部分。問題是，C 語言並不支援物件（objects）。

以 C 語言為基礎，Bjarne Stroustrup 設計了 C++ 語言。Stroustrup 在 C 語言中添加了物件導向編程所需要的關鍵字和語法。藉由「向普及的 C 語言加入物件導向特性」，Stroustrup 引導成千上萬的程式員進入 OOP 領域。有了 C++ 之後，編程的現代紀元完全實現了。Stroustrup 大筆一揮，創造了世界上最強大的計算機語言，並描繪了未來語言的發展路線。

雖然 C++ 的餘蔭才剛剛展開，卻已導致兩個重要語言的誕生：Java 和 C#。除去一些細小差別，Java 和 C# 的語法、物件模型、整體風貌幾乎與 C++ 一樣。不僅如此，Java 和 C# 的程式庫也反映了 C++ 的設計，甚至於 Java 和 C# 的群集框架（Collections framework）更是直接從 STL 而來。C++ 開創性的設計在電算歷史中將永綻光彩。

「給予程式員強大能力」是 C++ 變得重要的原因。它深遠的影響使它目前依然是全世界程式員的超卓選擇。可以說，C++ 導出的終極力量就是它帶給我們的最珍貴餘蔭。

2

2. 簡易垃圾回收器

A Simple Garbage Collector in C++

本章中英術語摘要：

allocate⇔配置；array⇔陣列；assignment⇔賦值/指派；constructor⇔建構式；circular references⇔環狀引用；class⇔類別；destructor⇔解構式；garbage collection⇔垃圾回收；iterator⇔迭代器；mark and sweep⇔標記並清理；memory⇔記憶體；memory leak⇔記憶體洩漏；object⇔物件；reference counting⇔參用計數；utility⇔公用程式；

縱覽電算歷史，一直對於動態配置記憶體的最佳管理方式有著爭議。動態配置記憶體就是在執行期從 heap 取得的記憶體，heap 是可被程式使用的一塊空閒記憶體區，也常被稱為自由儲存區（free store）或動態記憶體（dynamic memory）。動態配置很重要，使程式得以在執行期獲得、使用、釋放後又重新使用記憶體。由於幾乎所有實用程式都以某種形式使用動態記憶體，因此其管理方式對程式的架構和效率有著深遠影響。

一般而言，動態記憶體有兩種處理方式。第一種是手動形式，即程式員必須明確釋放無用的記憶體以便它能夠再被運用。第二種則是倚賴常被稱為「垃圾回收」的自動形式，當記憶體不再被需要時將其回收。兩種方法各有千秋，隨著時間推移，人們對兩種方法的偏好也一直在轉變。

C++使用手動形式來管理動態記憶體。垃圾回收則是 Java 和 C# 使用的機制。既然 Java 和 C# 是更新的語言，那麼似乎計算機語言的當前設計趨勢傾向於垃圾回收。然而這並不意味 C++ 程式員被留在「歷史上錯誤的一邊」。由於 C++ 語言威力強大，所以有可能 — 甚至輕易地 — 為 C++ 建立一個垃圾回收器。因此 C++ 程式員可以左右逢源：根據需要選擇手動配置或自動回收。

本章將開發一個完整的 C++ 垃圾回收器。首先必須說明，這個垃圾回收器並非用來替代 C++ 內建的動態配置方式，而是用來更完善它。理解這一點很重要。因此，同一個程式可以同時使用手動系統和垃圾回收系統。

垃圾回收器不僅本身是一段有用（並且吸引人）的程式碼，而且它能清楚顯示 C++ 卓越的能力，因此被選作本書第一個例子。經由 template class（模板類別）、operator overloading（運算子重載）和 C++ 與生俱來的低階元素（例如記憶體位址）處理能力，我們有可能以透明的方式為 C++ 添加核心特性。對其他大多數語言而言，改變動態配置方式需要改變編譯器本身。然而由於 C++ 賦予程式員無與倫比的力量，這項任務可以在源碼層級完成。

垃圾回收器還示範如何「定義一個新型別並完全整合到 C++ 編程環境」。如此的「型別擴展性」是 C++ 的一個關鍵，也是常被忽視的一點。最後，由於垃圾回收器操作並管理了指標，所以它也證實了 C++「接近機器」的能力。是的，和其他「避免存取底層細節」的語言不同，C++ 總是讓程式員儘可能地接近硬件。

2.1. 比較兩種記憶體管理法

開發 C++ 垃圾回收器之前，先比較「垃圾回收」與 C++ 內建之「手動辦法」是有用處的。

通常在 C++ 中使用動態記憶體需要兩個步驟。第一，經由 **new** 從 heap 配置記憶體。第二，不再需要記憶體時透過 **delete** 將其釋放。因此每次動態配置都遵循以下順序：

```
p = new some_object;
// ...
delete p;
```

一般而言，每個 **new** 都需要對應的 **delete** 來平衡。如果沒有使用 **delete**，即使程式不再需要這塊記憶體，它也不會被釋放。

垃圾回收與手動方式相比有一個關鍵不同點：它自動釋放不再使用的記憶體。因此，有了垃圾回收，動態配置就成為單步操作。例如在 Java 和 C#中，透過 **new** 來配置記憶體，但從不明白釋放它。取而代之的是，垃圾回收器周期性地執行，尋找未被任何物件指向的記憶體區塊。當一塊記憶體沒有被某物件指向時，意味沒有任何程式元素使用這塊記憶體，於是就可以釋放它。因此在一個垃圾回收系統中，既不存在也不需要 **delete** 運算子。

乍見之下，垃圾回收與生俱來的簡易性使它看似成為管理動態記憶體的當然選擇。事實上您可能懷疑為什麼手動方法還被使用，尤其是在 C++ 這樣複雜的語言中。然而對動態配置而言，第一眼印象帶有欺騙性，因為其實兩種方法都伴隨著一些折衷。哪一種方法更合適，取決於具體應用程式。以下各小節將描述其中一些議題。

2.1.1. 手動管理的優缺點

手工式動態記憶體管理的最大好處在於高效率。由於沒有垃圾回收器，也就不會花時間跟蹤活動物件（active objects）或周期性地尋找未用記憶體（unused memory）。當程式員知道不再需要某個已配置物件時，就明白地將它釋放，沒有任何額外開銷。由於少了垃圾回收器帶來的額外開銷，手動法得以寫出更高效的程式。這就是 C++ 必須支援記憶體手動管理的原因之一：它能產生高效率程式。

手動法的另一個優點在於控制。雖然同時要求處理記憶體的配置和釋放，對程式員是一種負擔，好處卻是程式員能夠完全掌控這個程序的兩半部。您能夠精確知道記憶體何時被配置以及何時被釋放。此外當使用 **delete** 釋放物件時，其解構式立即執行，不像垃圾回收器事後才執行。因此，使用手動法，您可以精確控制一個已配置物件的銷毀時間點。

雖然手動法很高效，但它容易受到一種相當惱人的錯誤的困擾：記憶體洩漏（memory leak）。由於記憶體必須手動釋放，所以程式員有可能（甚至很容易）忘記這麼做。不釋放未用記憶體，意味即使該記憶體不再被使用，它也會保持「已配置狀態」。垃圾回收環境不會發生記

憶體洩漏，因為垃圾回收器保證未使用物件最終會被釋放。記憶體洩漏在 Windows 編程中是一個特別麻煩的問題，未使用的資源如果不被釋放，會逐漸降低整體效率。

C++ 手動管理辦法可能發生的其他問題還包括提前釋放仍被使用的記憶體，以及意外地將同一塊記憶體釋放兩次。這兩種錯誤都可能導致嚴重後果，不幸的是它們可能不會立即顯示症狀，這使得它們難以被發現。

2.1.2. 垃圾回收的優缺點

實現垃圾回收機制的作法有好幾種，各具不同的效率特徵。然而若與手動法相比，所有垃圾回收系統都有一些共同特性。垃圾回收的主要優點是簡單和安全。在一個垃圾回收環境中，您可以透過 **new** 明確進行記憶體配置，但不需要明白地釋放它。是的，未使用的記憶體會被自動回收。因此，忘記釋放或過早釋放都是不可能的。這就簡化了編程並避免同一類型的所有問題。此外也不可能意外地將動態配置記憶體釋放兩次。因此可以說，垃圾回收機制為記憶體管理提供了一個易於使用、錯誤防範而又可靠的解決方案。

不幸的是，垃圾回收的簡單性和安全性都是有代價的。第一個代價是它帶來的額外開銷。所有垃圾回收方案都消耗一定程度的 CPU 時間，因為記憶體回收不是一個免費程序。這些額外開銷在手動法中不會出現。

第二個代價是無法控制某個物件何時被銷毀。在手動法中，物件被銷毀（並且其解構式被喚起）的時間點很確定 — 就在對該物件執行 **delete** 指令時，但垃圾回收並不具備這樣的硬性約定。使用垃圾回收機制時，除非垃圾回收器運轉並回收某個物件，否則該物件不會被銷毀，這可能發生在未來任何時間。例如垃圾回收器可能要等到空閒記憶體數量低於一定水平時才運轉。某些情況下，無法精確知道一個物件何時被銷毀可能會導致麻煩，因為這意味您的程式不能精確知道一個動態配置物件的解構式何時被喚起。

對於作為背景任務（background task）而運轉的垃圾回收系統，由於它為程式引入了本質上的不確定性，因此這種「控制權的喪失」有可能潛在導致某一類程式更嚴重的問題。一個在背景執行的垃圾回收器為了各種實用目的，會在不確定時刻回收未用記憶體，例如通常只在 CPU 空閒時間才運轉，而由於這將因不同的執行程式、不同的電腦或不同的作業系統而不相同，所以垃圾回收器在程式中的精確運轉時間點實際上是不確定的。對很多程式這不是問題，但它可能對即時（real-time）程式產生嚴重破壞，因為在意料之外將 CPU 時間分配給垃圾回收器，有可能導致某個事件（event）丟失。

2.1.3. 可同時進行的種作法

正如前面所解釋，手動管理和垃圾回收都在「最大化某個特性」的同時犧牲了另一個特性。手動法的效率和控制權最高，卻犧牲了使用上的安全性和易用性。垃圾回收使簡單性和安全性最大化，卻為此付出了執行期效率低落和控制權喪失的代價。因此，記憶體的垃圾回收和手動管理本質上是相反的，其中之一強調的某特性，就是另一個所犧牲的特性。這便是兩種方法都不是「所有編程情況下最佳動態記憶體管理法」的原因。

雖然兩者截然相反，但它們並非相互排斥。它們可以共存。因此，C++ 程式員有可能執其兩端用其中，為手頭的任務選擇合適的辦法。您需要做的只是為 C++ 創造一個垃圾回收器，這就是本章後續部分的主題。

2.2. 以 C++ 建立一個垃圾回收器

因為 C++ 是一種豐富而強大的語言，所以實現垃圾回收器的作法有很多種。一種明顯但受限的作法是創造一個垃圾回收器 base class，被意欲使用垃圾回收機制的 classes 繼承。這使您能夠在 class-by-class 的基礎上實現。不幸的是這種作法過於狹隘，不能令人滿意。

較好的解法是：垃圾回收器被用於各型動態配置物件。為提供這樣的解法，垃圾回收器必須：

1. 與 C++ 提供的內建手動法並存。
2. 不破壞任何既有程式碼，也不能對既有程式碼產生任何影響。
3. 工作方式透明，以便讓不論使用或未使用垃圾回收機制的配置手法都相同。
4. 與 C++ 內建方法一樣，透過 **new** 來配置記憶體。
5. 適用於所有型別，包括內建型別如 **int** 和 **double**。
6. 易於使用

簡而言之，垃圾回收系統進行的動態記憶體配置，其所使用的機制和語法，必須非常類似 C++ 現有，並且不影響任何既有程式碼。

2.2.1. 了解問題所在

建立垃圾回收器時，我們所面對的關鍵挑戰是：如何得知某塊記憶體未被使用？要理解這個問題，請考慮以下程式碼：

```
int *p;
p = new int(99);
```

```
p = new int(100);
```

這裡的兩個 **int** 物件是動態配置而得。第一個含值 99，**p** 保存了指向該值的指標。然後配置一個含值 100 的整數，其位址也保存於 **p**，於是蓋掉第一個位址。此時 **int(99)** 的記憶體並未被 **p**（或其他任何物件）指向，因此可以被釋放。問題是，垃圾回收器如何得知 **p** 和其他物件都沒有指向 **int(99)**？

下面是這個問題的略微變化：

```
int *p, *q;
p = new int(99);
q = p; // 現在 q 指向與 p 所指相同的一塊記憶體
p = new int(100);
```

這種情況下 **q** 指向原本配置給 **p** 的記憶體。即使後來 **p** 指向另一塊記憶體，原先指的那塊記憶體也不能被釋放，因為它仍被 **q** 所用。問題是垃圾回收器如何得知這個事實？這些問題的確切答案由垃圾回收演算法決定。

2.3. 選擇一種垃圾回收演算法

為 C++ 實現一個垃圾回收器之前，有必要決定使用什麼樣的垃圾回收演算法。垃圾回收是個大題目，多年來都是嚴肅學術研究的焦點。由於它所呈現的引人問題有多種解決方案，所以人們已經設計出數種不同的垃圾回收演算法。對其中每一個都進行細緻分析，將遠遠超出本書範圍。然而可歸納出三種原型：參用計數（reference counting），標記並清理（mark and sweep），以及複製法（copying）。選擇之前，讓我們先回顧這三種演算法。

2.3.1. 參用計數（Reference Counting）

在參用計數法中，每一塊動態配置記憶體都有一個與之相關的參用計數。每次添加一個對該記憶體的參用，計數就加 1，每次刪除一個對該記憶體的參用，計數就減 1。以 C++ 術語來說，每當一個指標被設指向一塊配置記憶體，這塊記憶體的參用計數就加 1。當這個指標改指其他地方，剛才所說那塊記憶體的參用計數就減 1。當參用計數減至 0，記憶體就是處於未被使用的狀態，可以安心釋放它。

參用計數的最大好處是簡單 — 易於理解和實現。此外，它不對 heap 做任何限制，因為參用計數和物件實際位置無關。參用計數會在每次指標操作時增加額外開銷，不過收集階段的

開銷相當低。主要缺點是環狀引用（circular references）會阻止未被使用的記憶體獲得釋放。當兩個物件直接或間接指向彼此，就是所謂環狀引用。這種情況下兩個物件的參用計數器都不會減少至 0。目前已發明解決這個問題的一些方法，但它們都會增加複雜性和/或開銷。

2.3.2. 標記並清理（Mark and Sweep）

此法包含兩個階段。第一階段，heap 內的所有物件都被設為未標記狀態。接下來，所有能夠直接或間接被程式變數存取的物件都被標記為「使用中」。第二階段則是掃描所有配置記憶體（也就是進行記憶體的清理），然後釋放所有未被標記的元素。

「標記並清理」法有兩個主要優點。首先它能輕易解決環狀引用問題。第二，它在進行收集之前不會添加任何額外開銷。它的兩個主要缺點是，第一，操作時需要掃描整個 heap，會消耗相當大量的時間，因此整個垃圾回收機制的時間效率可能無法讓某些程式接受。第二，雖然「標記並清理」的概念很簡單，但卻需要相當技巧才能將它實現得高效率。

2.3.3. 複製/拷貝（Copying）

此法是將空閒記憶體組織為兩塊空間。第一個是 active（作用中的；保存著當前 heap），另一個是 idle（閒置的）。垃圾回收時，active 空間內正被使用的物件被辨識出來，並被複製到 idle 空間中。然後兩塊空間的角色互換，idle 空間變為 active，active 空間變為 idle。複製法的優勢在於能在複製過程中將 heap 緊湊壓實（compacting），缺點則是任何時候只允許一半自由記憶體（free memory）被使用。

2.3.4. 選擇哪一種演算法好？

認識三種垃圾回收經典方法的優缺點後，選擇一個而淘汰另一個似乎並不那麼容易。然而如果根據先前我們所列的約定條件，答案很明顯：參用計數。首先並且也是最重要的一點，參用計數很容易嵌入既有的 C++ 動態配置系統。其次，它的實現方式簡單易懂，不會影響既有程式碼。第三，它對 heap 的組織或結構沒有特殊要求，不影響 C++ 提供的標準配置系統。

參用計數的一個缺點是：它不好處理環狀引用（circular references）。對很多程式而言這不是問題，因為刻意的環狀引用並不常見，而且往往可以避免。即使有些東西被我們作成環狀，例如環狀佇列（circular queue），但不一定會形成環狀的指標參用。當然，還是會有需要環狀引用的情況，或者造成了環狀引用卻不自知 — 特別是使用協力廠商（第三方）程式庫時。因此垃圾回收器必須提供某種方法，能良好解決萬一存在的環狀引用。

為了處理環狀引用（circular references）問題，本章開發的垃圾回收器將在程式結束前一刻釋放所有遺留的配置記憶體。這便保證環狀引用所涉及的物件都被釋放，其解構式亦被呼叫。正常情況下程式結束時不該留有任何配得物件，這一點很重要。這種機制改以手動操作「因環狀引用而無法被釋放的物件」。您可能會想試驗其他解法，這的確是個有趣的挑戰。

2.3.5. 實作垃圾回收器

欲實現「參用計數」垃圾回收器，必須有辦法追蹤「指向某塊動態配置記憶體」的指標數。棘手的是 C++ 沒有內建機制可使一個物件知道另一個物件正指向它。幸運的是有一個解決辦法：您可以建立一個支援垃圾回收的新式指標型別。這就是本章垃圾回收器所使用的方法。

為支援垃圾回收，這個新式指標型別必須做三件事。第一，它必須為動態配置物件(s)維持一系列參用計數器。第二，它必須追蹤每一個指標操作，每當一個指標指向 A 物件時就將 A 物件對應的參用計數器加 1，每當指標改指向 B 物件時就將 A 物件的參用計數器減 1。第三，它必須回收參用計數為 0 的物件。除了支援垃圾回收，這個新式指標型別必須與一般指標的觀感一致。例如它必須支援所有指標操作，包括 `*` 和 `->`。

除了方便實現一個垃圾回收器，「垃圾回收式指標型別」還可以滿足「原始 C++ 動態配置系統不受影響」這個條件。需要垃圾回收時，就使用垃圾回收式指標。如果不需要垃圾回收，就使用一般 C++ 指標。因此，同一個程式可以使用兩種類型的指標。

2.3.6. 多緒與否？（To Multithread or Not?）

替 C++ 設計垃圾回收器時應考慮的另一個問題是，它該是單緒還是多緒？也就是說，垃圾回收器是否該被設計為一個獨立緒程在背景執行，並於 CPU 時間允許時進行垃圾回收？或者應該將垃圾回收器和其使用者執行於同一個緒程中，並在特定狀況發生時進行垃圾回收？唔，兩種方法各有千秋。

建立多緒垃圾回收器的主要優點是高效率，可以在 CPU 空檔周期進行垃圾回收。缺點是 C++ 並無內建任何多緒支援能力。這意味任何多緒作法都倚賴作業系統對多緒的支援。這使得程式碼不可移植。

單緒垃圾回收器的最大好處是可移植性。它可用於「不支援多緒」的環境下，或用於「多緒代價太高」的情況下。其主要缺點則是，進行垃圾回收時，程式的其他部分不得不暫停。

本章使用單緒作法，因為它能在所有 C++ 環境中執行，因此能被本書所有讀者使用。然而對於那些想要多緒解決方案的讀者，本書第三章給了一個實作品，該章涉及「在 Windows 環境中對一個 C++ 程式成功引入多緒」所需的技術。

2.3.7. 何時進行垃圾回收？

實現垃圾回收器前，最後一個要回答的問題是：何時進行垃圾回收？作為背景任務持續執行的多緒垃圾回收器，只在 CPU 空閒時才回收垃圾，所以這不成其為問題；但對於本章開發的單緒垃圾回收器，收集垃圾時會暫停程式其他部分，這就是個問題。

真實世界中，垃圾回收動作往往在充足理由下 — 例如記憶體不足 — 才進行。兩個因素使得這麼做有道理。第一，對某些垃圾回收演算法如「標記並清理」，如果不進行實際收集，就無法得知某塊記憶體未被使用。（也就是說有時候如果不實際進行回收就無法得知有垃圾需要被回收！）第二，垃圾回收是一個耗時的程序，不該在非必要時進行。

然而，對本章目的而言，直到記憶體不足才開始垃圾回收，並不合適，因為這會使得回收器的展示變得幾乎不可能。本章開發的垃圾回收器會頻繁回收垃圾，以便其行為容易被觀察。正如程式碼所示，每當一個指標離開其作用域（scope）就進行垃圾回收。當然，這種行為很容易被修改，以便迎合具體應用。

最後一點：使用參用計數垃圾回收器時，技術上有可能在一塊記憶體的參用計數降至 0 時將它及時回收，而不倚賴某個「垃圾回收獨立階段」。然而這種方法會在每次指標操作時都增加開銷。本章使用的方法是，每當指向一塊記憶體的指標被重新定向，就將該記憶體的參用計數器減 1，並讓回收程序在更方便時才進行實際回收。這可以減少指標操作帶來的執行期開銷，以便指標操作儘可能地快速。

2.3.8. auto_ptr 怎麼樣？

許多讀者知道，C++ 標準程式庫定義了一個 **auto_ptr** class。由於 **auto_ptr** 物件在超出其作用域時會自動將它所指的記憶體釋放，所以您可能認為它在開發垃圾回收器時有用，甚至有到奠基作用。然而並不是這麼回事。**auto_ptr** 是為一個被 ISO *C++ Standard* 稱為「嚴格擁有權」（strict ownership）的概念而設計的。在此概念中，一個 **auto_ptr** 物件擁有它所指的物件。這種擁有權能夠被傳遞到另一個 **auto_ptr** 物件，但在任何情況下有些 **auto_ptr** 物件會一直擁有其物件，直到它被銷毀 。由於 **auto_ptr** 的「嚴格所有權」特性，建立垃圾回收器時它並不是特別有用，本書的垃圾回收器也沒有使用它。

2.4. 一個簡式垃圾回收器

下面是一個完整的垃圾回收器。正如先前解釋，這個垃圾回收器是經由建立一個新式指標型別來完成，該型別以參用計數為基礎，提供對垃圾回收的內建支援。這個垃圾回收器是單緒形式，意味其可移植性良好，而且不倚賴執行環境（或對其有某種假設）。程式碼儲存在一個名為 **gc.h** 的檔案中。

瀏覽程式碼時請注意兩件事。第一，大多數成員函式都很短，並且為了追求效率都定義在各自的 class 內部。還記得嗎，一個在 class 內定義的函式會自動成為 inline 函式，可消除函式呼叫動作帶來的額外開銷。只有一些函式大到需要定義於 class 外部。

第二，請注意檔案前端的注釋。如果您想觀察垃圾回收器的活動，只需透過 **DISPLAY** 巨集打開顯示選項即可。

```
#001 //  一個單緒垃圾回收器
#002
#003 #include <iostream>
#004 #include <list>
#005 #include <typeinfo>
#006 #include <cstdlib>
#007
#008 using namespace std;
#009
#010 //  若要觀察垃圾回收器的活動，請定義 DISPLAY
#011 // #define DISPLAY
#012
#013 //  以下異常會在一個 Iter 超出其所在物件的範圍時被拋出。
#014 //
#015 class OutOfRangeExc {
#016   //  如果您的程式需要，請在此加入功能。
#017 };
#018
#019 //  一個 iterator-like class，用來巡訪 GCPtrs 所指陣列。Iter 指標
#020 //  不參加或影響垃圾回收器。因此，一個指向某物件的 Iter 物件
#021 //  並不會阻礙該物件被回收。
#022 //
#023 template <class T> class Iter {
#024   T *ptr;   // 當前指標（current pointer value）
#025   T *end;   // 指向終端的下一個位置
```

```
#026   T *begin; // 指向配置陣列的起點
#027   unsigned length; // 數列長度（length of sequence）
#028 public:
#029
#030   Iter() {
#031     ptr = end = begin = NULL;
#032     length = 0;
#033   }
#034
#035   Iter(T *p, T *first, T *last) {
#036     ptr =  p;
#037     end = last;
#038     begin = first;
#039     length = last - first;
#040   }
#041
#042   // 傳回這個 Iter 所指數列的長度。
#043   unsigned size() { return length; }
#044
#045   // 傳回 ptr 指向的值。不允許越界存取。
#046   T &operator*() {
#047     if( (ptr >= end) || (ptr < begin) )
#048       throw OutOfRangeExc();
#049     return *ptr;
#050   }
#051
#052   // 傳回 ptr 保存的位址。不允許越界存取。
#053   T *operator->() {
#054     if( (ptr >= end) || (ptr < begin) )
#055       throw OutOfRangeExc();
#056     return ptr;
#057   }
#058
#059   // 前置式（Prefix）++
#060   Iter operator++() {
#061     ptr++;
#062     return *this;
#063   }
#064
#065   // 前置式（Prefix）--
#066   Iter operator--() {
#067     ptr--;
#068     return *this;
#069   }
#070
```

```
#071  // 後置式（Postfix）++.
#072  Iter operator++(int notused) {
#073    T *tmp = ptr;
#074
#075    ptr++;
#076    return Iter<T>(tmp, begin, end);
#077  }
#078
#079  // 後置式（Postfix）--.
#080  Iter operator--(int notused) {
#081    T *tmp = ptr;
#082
#083    ptr--;
#084    return Iter<T>(tmp, begin, end);
#085  }
#086
#087  // 傳回一個 reference 指向特定索引上的物件。不允許越界存取。
#088  T &operator[](int i) {
#089    if( (i < 0) || (i >= (end-begin)) )
#090      throw OutOfRangeExc();
#091    return ptr[i];
#092  }
#093
#094  // 定義相對關係運算子（relational operators）
#095  bool operator==(Iter op2) {
#096    return ptr == op2.ptr;
#097  }
#098
#099  bool operator!=(Iter op2) {
#100    return ptr != op2.ptr;
#101  }
#102
#103  bool operator<(Iter op2) {
#104    return ptr < op2.ptr;
#105  }
#106
#107  bool operator<=(Iter op2) {
#108    return ptr <= op2.ptr;
#109  }
#110
#111  bool operator>(Iter op2) {
#112    return ptr > op2.ptr;
#113  }
#114
```

```
#115   bool operator>=(Iter op2) {
#116     return ptr >= op2.ptr;
#117   }
#118
#119   // 從 Iter 物件中減去某整數。
#120   Iter operator-(int n) {
#121     ptr -= n;
#122     return *this;
#123   }
#124
#125   // 為 Iter 物件加上某整數。
#126   Iter operator+(int n) {
#127     ptr += n;
#128     return *this;
#129   }
#130
#131   // 傳回兩個 Iter 物件間的元素個數
#132   int operator-(Iter<T> &itr2) {
#133     return ptr - itr2.ptr;
#134   }
#135
#136 };
#137
#138
#139 // 以下 class 定義保存於垃圾回收資訊串列（information list）中的元素
#140 //
#141 template <class T> class GCInfo {
#142 public:
#143   unsigned refcount; // 目前的 reference 個數
#144
#145   T *memPtr; // 指向配置而得的記憶體
#146
#147   /* 當 memPtr 指向一個配置陣列時，isArray 為 true，
#148      否則為 false。*/
#149   bool isArray; // 如果指向陣列則為 true
#150
#151   /* 如果 memPtr 指向一個配置陣列，
#152      arraySize 就用以表示其大小 */
#153   unsigned arraySize; // 陣列大小
#154
#155   // 在這裡，mPtr 指向被配置記憶體。如果那是個陣列，
#156   // size 就代表陣列大小。
```

```
#157   GCInfo(T *mPtr, unsigned size=0) {
#158     refcount = 1;
#159     memPtr = mPtr;
#160     if(size != 0)
#161       isArray = true;
#162     else
#163       isArray = false;
#164
#165     arraySize = size;
#166   }
#167 };
#168
#169 // 重載 == 運算子，使 GCInfo 物件可被比較。此為 STL list 所需要。
#170 template <class T> bool operator==(const GCInfo<T> &ob1,
#171                 const GCInfo<T> &ob2) {
#172   return (ob1.memPtr == ob2.memPtr);
#173 }
#174
#175
#176 // GCPtr 實現出一個指標型別，它以垃圾回收機制釋放未用記憶體。
#177 // GCPtr 物件只用來指向一塊以 new 動態配置而得的記憶體。
#178 // 如果用來指向一塊配置而得的陣列，必須指定陣列大小。
#179 //
#180 template <class T, int size=0> class GCPtr {
#181
#182   // gclist 用來維護回收串列（garbage collection list）
#183   static list<GCInfo<T> > gclist;
#184
#185   // addr 指向本 GCPtr 指標目前所指的配置記憶體。
#186   T *addr;
#187
#188   /* 如果這個 GCPtr 物件指向一塊配置陣列，isArray 是 true。
#189      否則它是 fasle。 */
#190   bool isArray; // 如果指向陣列則為 true
#191
#192   // 如果這個 GCPtr 物件指向一塊配置陣列，
#193   // arraySize 代表其大小。
#194   unsigned arraySize; // 陣列大小
#195
```

```
#196   static bool first; // 當第一個 GCPtr 被產生出來時為 true
#197
#198   // 傳回一個迭代器，指向 gclist 中的指標資訊。
#199   typename list<GCInfo<T> >::iterator findPtrInfo(T *ptr);
#200
#201 public:
#202
#203   // 為 GCPtr<T>定義一個迭代器型別（iterator type）
#204   typedef Iter<T> GCiterator;
#205
#206   // 建構初始化物件，或是非初始化物件。
#207   GCPtr(T *t=NULL) {
#208
#209     // 將 shutdown()註冊為退離函式（exit function）。
#210     if(first) atexit(shutdown);
#211     first = false;
#212
#213     list<GCInfo<T> >::iterator p;
#214
#215     p = findPtrInfo(t);
#216
#217     // 如果 t 已在 gclist 內，就累加其參用計數器。
#218     // 否則將它加入這個串列。
#219     if(p != gclist.end())
#220       p->refcount++; // 將參用計數器累加
#221     else {
#222       //  建立並儲存這一筆（entry）
#223       GCInfo<T> gcObj(t, size);
#224       gclist.push_front(gcObj);
#225     }
#226
#227     addr = t;
#228     arraySize = size;
#229     if(size > 0) isArray = true;
#230     else isArray = false;
#231     #ifdef DISPLAY
#232       cout << "Constructing GCPtr. ";
#233       if(isArray)
#234         cout << " Size is " << arraySize << endl;
#235       else
#236         cout << endl;
#237     #endif
#238   }
#239
```

```
#240   // copy建構式
#241   GCPtr(const GCPtr &ob) {
#242     list<GCInfo<T> >::iterator p;
#243
#244     p = findPtrInfo(ob.addr);
#245     p->refcount++; // 將參用計數器累加
#246
#247     addr = ob.addr;
#248     arraySize = ob.arraySize;
#249     if(arraySize > 0) isArray = true;
#250     else isArray = false;
#251     #ifdef DISPLAY
#252       cout << "Constructing copy.";
#253       if(isArray)
#254         cout << " Size is " << arraySize << endl;
#255       else
#256         cout << endl;
#257     #endif
#258   }
#259
#260   // GCPTr 解構式。
#261   ~GCPtr();
#262
#263   // 收集垃圾。如果至少釋放了一個物件，就傳回 true。
#264   static bool collect();
#265
#266   // 重載 "pointer 對 GCPtr" 的賦值函式（assignment）
#267   T *operator=(T *t);
#268
#269   // 重載 "GCPtr 對 GCPtr" 的賦值函式（assignment）
#270   GCPtr &operator=(GCPtr &rv);
#271
#272   // 傳回這個 GCPtr 物件所指的物件的 reference
#273   T &operator*() {
#274     return *addr;
#275   }
#276
#277   // 傳回被指向的位址
#278   T *operator->() { return addr; }
#279
#280   // 傳回一個 reference，指向 i 索引上的物件
#281   T &operator[](int i) {
#282     return addr[i];
#283   }
#284
```

```
#285   // 轉化函式（Conversion function），轉為 T*
#286   operator T *() { return addr; }
#287
#288
#289   // 傳回一個 Iter 物件，指向配置所得記憶體的起始位置
#290   Iter<T> begin() {
#291     int size;
#292
#293     if(isArray) size = arraySize;
#294     else size = 1;
#295
#296     return Iter<T>(addr, addr, addr + size);
#297   }
#298
#299   // 傳回一個 Iter 指向配置所得陣列之最後一個元素的下一位置
#300   Iter<T> end() {
#301     int size;
#302
#303     if(isArray) size = arraySize;
#304     else size = 1;
#305
#306     return Iter<T>(addr + size, addr, addr + size);
#307   }
#308
#309   // 傳回此種 GCPtr 型別的 gclist 大小
#310   static int gclistSize() { return gclist.size(); }
#311
#312   // 一個公用函式，用來顯示 gclist。
#313   static void showlist();
#314
#315   // 程式結束時清空 gclist
#316   static void shutdown();
#317 };
#318
#319 // 為靜態變數（static variables）保留空間
#320 template <class T, int size>
#321   list<GCInfo<T> > GCPtr<T, size>::gclist;
#322
#323 template <class T, int size>
#324   bool GCPtr<T, size>::first = true;
#325
```

```
#326 // GCPtr 解構式
#327 template <class T, int size>
#328 GCPtr<T, size>::~GCPtr() {
#329   list<GCInfo<T> >::iterator p;
#330
#331   p = findPtrInfo(addr);
#332   if(p->refcount) p->refcount--; // 減少 ref 計數
#333
#334   #ifdef DISPLAY
#335     cout << "GCPtr going out of scope.\n";
#336   #endif
#337
#338   //  當指標離開其作用域時，進行垃圾回收
#339   collect();
#340
#341   //  為了實際運用，您可能希望不那麼經常地收集未用記憶體，或許在
#342   //  gclist 達到特定大小，或特定數量的 GCPtr 物件超出作用域，或
#343   //  記憶體不足時，才進行收集。
#344 }
#345
#346 // 收集垃圾。如果至少有一個物件被釋放，就傳回 true。
#347 template <class T, int size>
#348 bool GCPtr<T, size>::collect() {
#349   bool memfreed = false;
#350
#351   #ifdef DISPLAY
#352     cout << "Before garbage collection for ";
#353     showlist();
#354   #endif
#355
#356   list<GCInfo<T> >::iterator p;
#357   do {
#358
#359     // 掃描 gclist，尋找未曾被引用的指標
#360     for(p = gclist.begin(); p != gclist.end(); p++) {
#361       // 如果正在使用，就跳過
#362       if(p->refcount > 0) continue;
#363
#364       memfreed = true;
#365
#366       // 從 gclist 中刪除未用項
#367       gclist.remove(*p);
#368
```

```
#369      // 除非 GCPtr 為空，否則釋放記憶體
#370      if(p->memPtr) {
#371        if(p->isArray) {
#372          #ifdef DISPLAY
#373            cout << "Deleting array of size "
#374                 << p->arraySize << endl;
#375          #endif
#376          delete[] p->memPtr; // 刪除陣列
#377        }
#378        else {
#379          #ifdef DISPLAY
#380            cout << "Deleting: "
#381                 << *(T *) p->memPtr << "\n";
#382          #endif
#383          delete p->memPtr; // 刪除單一元素
#384        }
#385      }
#386
#387      // 重新開始搜索
#388      break;
#389    }
#390
#391  } while(p != gclist.end());
#392
#393  #ifdef DISPLAY
#394    cout << "After garbage collection for ";
#395    showlist();
#396  #endif
#397
#398  return memfreed;
#399 }
#400
#401 // 重載「指向 GCPtr」的指標的賦值操作
#402 template <class T, int size>
#403 T * GCPtr<T, size>::operator=(T *t) {
#404  list<GCInfo<T> >::iterator p;
#405
#406  // 首先將當前所指記憶體的參用計數器減 1
#407  p = findPtrInfo(addr);
#408  p->refcount--;
#409
#410  // 然後，如果新位址已存在於系統，就累加其計數。
#411  // 否則為 gclist 新建一個條目（entry）
```

```
#412   p = findPtrInfo(t);
#413   if(p != gclist.end())
#414     p->refcount++;
#415   else {
#416     // 產生並保存該條目（entry）
#417     GCInfo<T> gcObj(t, size);
#418     gclist.push_front(gcObj);
#419   }
#420
#421   addr = t; // 保存位址
#422
#423   return t;
#424 }
#425
#426 // 重載「GCPtr 對 GCPtr」的賦值操作
#427 template <class T, int size>
#428 GCPtr<T, size> & GCPtr<T, size>::operator=(GCPtr &rv) {
#429   list<GCInfo<T> >::iterator p;
#430
#431   // 首先將當前所指記憶體的參用計數器減 1
#432   p = findPtrInfo(addr);
#433   p->refcount--;
#434
#435   // 然後累加新物件的參用計數器
#436   p = findPtrInfo(rv.addr);
#437   p->refcount++; // 累加參用計數器
#438
#439   addr = rv.addr;// 保存位址
#440
#441   return rv;
#442 }
#443
#444 // 以下是個公用函式，用以顯示 gclist。
#445 template <class T, int size>
#446 void GCPtr<T, size>::showlist() {
#447   list<GCInfo<T> >::iterator p;
#448
#449   cout << "gclist<" << typeid(T).name() << ", "
#450        << size << ">:\n";
#451   cout << "memPtr     refcount    value\n";
#452
#453   if(gclist.begin() == gclist.end()) {
#454     cout << "          -- Empty --\n\n";
#455     return;
#456   }
#457
```

```
#458   for(p = gclist.begin(); p != gclist.end(); p++) {
#459     cout <<  "[" << (void *)p->memPtr << "]"
#460          << "     " << p->refcount << "    ";
#461     if(p->memPtr) cout << "   " << *p->memPtr;
#462     else cout << "   ---";
#463     cout << endl;
#464   }
#465   cout << endl;
#466 }
#467
#468 // 在 gclist 中找到一個指標
#469 template <class T, int size>
#470 typename list<GCInfo<T> >::iterator
#471   GCPtr<T, size>::findPtrInfo(T *ptr) {
#472
#473   list<GCInfo<T> >::iterator p;
#474
#475   // 在 gclist 中找到 ptr
#476   for(p = gclist.begin(); p != gclist.end(); p++)
#477     if(p->memPtr == ptr)
#478       return p;
#479
#480   return p;
#481 }
#482
#483 // 程式結束前清除 gclist
#484 template <class T, int size>
#485 void GCPtr<T, size>::shutdown() {
#486
#487   if(gclistSize() == 0) return; // list 為空
#488
#489   list<GCInfo<T> >::iterator p;
#490
#491   for(p = gclist.begin(); p != gclist.end(); p++) {
#492     // 將所有參用計數器設為 0
#493     p->refcount = 0;
#494   }
#495
#496   #ifdef DISPLAY
#497     cout << "Before collecting for shutdown() for "
#498          << typeid(T).name() << "\n";
#499   #endif
#500
```

```
#501   collect();
#502
#503   #ifdef DISPLAY
#504     cout << "After collecting for shutdown() for "
#505          << typeid(T).name() << "\n";
#506   #endif
#507 }
```

2.4.1. 垃圾回收器 Classes 概述

垃圾回收器使用了四個 classes：**GCPtr**,**CGInfo**,**Iter** 和 **OutOfRangeExc**。在具體分析程式碼之前先理解每個 class 所扮演的角色，應該是有幫助的。

GCPtr

垃圾回收器的核心是 **GCPtr**，它實現了一個垃圾回收式指標。**GCPtr** 維護一個 list，將參用計數器和被 **GCPtr** 所用的配置記憶體連接起來。大體上它是這樣工作的：每當一個 **GCPtr** 物件指向一塊記憶體，該記憶體的參用計數就加 1。如果在此賦值操作之前 **GCPtr** 物件曾經指向另一塊記憶體，那麼該塊記憶體的參用計數就減 1。因此，向一塊記憶體增加指標，會增加該記憶體的參用計數，向它刪除一個指標則會減少該記憶體的參用計數。每當一個 **GCPtr** 物件超出作用域（scope），它當時所指記憶體的參用計數就減 1。一旦某個參用計數減少至 0，那塊記憶體就可以被釋放。

GCPtr 是一個 template class，重載了指標運算子 * 和 ->，以及陣列索引運算子 []。因此可以說，**GCPtr** 創造了一個新式指標型別，並將其整合至 C++ 編程環境。這就允許您使用和普通指標一樣的方式來使用 **GCPtr** 物件。然而 **GCPtr** 並不重載++，--或其他指標算術運算，原因稍後說明。因此除了賦值動作（assignment），您不能改變一個 **GCPtr** 物件所指的位址。這看似一個重大限制，但其實並非如此，因為 **Iter** class 提供了那些操作。

基於展示目的，垃圾回收器將在 **GCPtr** 物件超出作用域時起而行。彼時將檢查「回收 list」，釋放所有參用計數為 0 的記憶體 — 即使它並非與越界的那個 **GCPtr** 物件相關。如果要更早些回收記憶體，可以明白提出申請，要求立刻行動。

GCInfo

前面提過，**GCPtr** 維護一個將參用計數和配置記憶體連接起來的 list，其中每一筆元素都被封裝於 **GCInfo** 物件內。**GCInfo** 在其 **refcount** 資料欄儲存參用計數，在其 **memPtr** 資料欄儲存指向記憶體的指標。因此 **GCInfo** 物件便是將一個參用計數和一塊配置記憶體綁在一起。

GCInfo 還定義了另兩個資料欄：**isArray** 和 **arraySize**。如果 **memPtr** 指向一塊配置陣列，那麼 **isArray** 為 **true**，而陣列長度將被保存在 **arraySize** 資料欄內。

Iter Class

前面提過，**GCPtr** 物件允許您使用普通指標的 * 和 -> 運算子來存取其所指向的記憶體，但是不支援指標算術（++或--之類）。為了應付某些情況下必要的指標算術，您可以使用 **Iter** 物件。**Iter** 是一個功能類似 STL 迭代器的 template class，它定義了包括算術運算在內的所有指標操作。**Iter** 的主要用途是讓您巡訪動態配置陣列的各個元素，並提供邊界檢查。您可以呼叫 **GCPtr** 的 **begin()** 或 **end()** 獲得一個 **Iter** 物件，這和它們的 STL 兄弟的工作方式很類似。

雖然 **Iter** 和 STL 迭代器類似，但它們並不完全一樣，不能彼此替代。這一點很重要。

OutOfRangeExc

如果 **Iter** 物件嘗試存取配置範圍外的記憶體 ，就會有一個 **OutOfRangeExc** 異常被拋出。對本章目的而言，**OutOfRangeExc** 沒有任何成員。它只是一個可被拋出的型別。當然您可以自由地按照您的需要為它加入功能。

2.5. GCPtr 詳解

GCPtr 是我們這個垃圾回收器的核心。它實現了一個新式指標型別，為自 heap 配置而得的物件保持一個參用計數。它還提供垃圾回收功能，回收未被使用的記憶體。

GCPtr 是個 template class，宣告如下：

```
template <class T, int size=0> class GCPtr {
```

這裡要求您指定它將指向的資料型別，並將替代泛化型別 **T**。如果配置的是個陣列，您就必須在參數 **size** 中指定大小。否則 **size** 預設為 0，意味指向單一物件。這裡有兩個例子：

```
GCPtr<int> p;          // 宣告一個指標，指向單一整數。
GCPtr<int, 5> ap;      // 宣告一個指標，指向由 5 個整數形成的陣列。
```

其中，**p** 能夠指向 **int** 型別的單一物件，**ap** 能夠指向由 5 個 **int** 物件組成的陣列。

前面這兩個例子中，請注意，指定 **GCPtr** 物件名稱時並沒有使用運算子 *。也就是說，如果要為 **int** 型別建立一個 **GCPtr** 物件，不需要這麼寫：

```
GCPtr<int> *p; // 建立一個指標，指向 GCPtr<int>物件
```

這個宣告建立的是一個名為 **p** 的一般 C++ 指標，可以指向 **GCPtr<int>** 物件。它並不是建立一個「可指向 **int** 型別」的 **GCPtr** 物件。記住，**GCPtr** 本身定義了一個指標型別。

為 **GCPtr** 制定型別參數（type parameters）時請小心。應該指定 **GCPtr** 物件所能指向的物件的型別。因此如果這樣宣告：

```
GCPtr<int *> p; // 建立一個 GCPtr 物件，可指向*int。
```

產生出來的是一個指向 **int***的 **GCPtr** 物件，而不是一個指向 **int** 的 **GCPtr** 物件。

由於 **GCPtr** 的每個成員都很重要，以下各小節將詳細分析它們。

2.5.1. GCPtr 的資料成員（Data Members）

GCPtr 宣告以下資料成員：

```
#182   // gclist 用來維護回收串列（garbage collection list）
#183   static list<GCInfo<T> > gclist;
#184
#185   // addr 指向本 GCPtr 指標目前所指的配置記憶體。
#186   T *addr;
#187
#188   /* 如果這個 GCPtr 物件指向一塊配置陣列，isArray 是 true。
#189      否則它是 false。 */
#190   bool isArray; // 如果指向陣列則為 true
#191
#192   // 如果這個 GCPtr 物件指向一塊配置陣列，
#193   // arraySize 代表其大小。
#194   unsigned arraySize; // 陣列大小
#195
#196   static bool first; // 當第一個 GCPtr 被產生出來時為 true
```

gclist 資料欄內含一個 **GCInfo** object list。還記得嗎，**GCInfo** 將參用計數和配置記憶體連接起來。這個 list 被垃圾回收器用來判斷配置記憶體何時未被使用。注意 **gclist** 是 **GCPtr** 的一個 static 成員，意味對每個特定指標型別，只有一個 **gclist** 物件。例如所有 **GCPtr<int>** 型別的指標共享一個 list，所有 **GCPtr<double>**型別的指標共享另一個 list。**gclist** 是 STL list 的一個實體。充分運用 STL 能簡化 **GCPtr** 程式碼，因為不再需要自己造輪子。

GCPtr 在其 **addr** 資料欄內保存所指記憶體的位址。如果 **addr** 指向陣列，**isArray** 將為 **true**，而陣列長度將儲存於 **arraySize** 資料欄。

first 資料欄是個 static 變數，初始值是 **true**。它是一個旗標，被 **GCPtr** 建構式用來得知第一

個 **GCPtr** 物件何時被建立。一旦第一個 **GCPtr** 物件被建立後，**first** 即被設為 **false**。這個旗標被用來註冊一個終結函式，在程式結束時被喚起，用以關閉垃圾回收器。

2.5.2. findPtrInfo()

GCPtr 定義了一個 private **findPtrInfo()**。此函式在 **gclist** 中搜索一個特定位址，並傳回該筆元素的迭代器。如果沒有找到就傳回一個指向 **gclist** 末尾的迭代器。這個函式被 **GCPtr** 內部用來更新 **gclist** 所含元素（物件）的參用計數。實作如下：

```
#468 // 在 gclist 中找到一個指標
#469 template <class T, int size>
#470 typename list<GCInfo<T> >::iterator
#471   GCPtr<T, size>::findPtrInfo(T *ptr) {
#472
#473   list<GCInfo<T> >::iterator p;
#474
#475   // 在 gclist 中找到 ptr
#476   for(p = gclist.begin(); p != gclist.end(); p++)
#477     if(p->memPtr == ptr)
#478       return p;
#479
#480   return p;
#481 }
```

2.5.3. GCIterator typedef

GCPtr 的 public 區域一開頭是個型別定義：將 **Iter<T>**定義為 **GCIterator**。這個定義被捆綁到 **GCPtr** 的每一個實體上，因此每當某特定型別的 **GCPtr** 物件需要一個 **Iter** 物件時，就不再需要指定其型別參數。這就簡化了迭代器的宣告。例如，欲獲得一個迭代器指向「被某特定 **GCPtr** 物件所指」的記憶體，可以這樣宣告：

```
GCPtr<int>::GCiterator itr;
```

2.5.4. GCPtr 建構式（Constructor）

GCPtr 建構式如下：

```
#206   // 建構已初始化或尚未初始化的物件。
#207   GCPtr(T *t=NULL) {
#208
#209     // 將 shutdown()註冊為退離函式（exit function）。
#210     if(first) atexit(shutdown);
#211     first = false;
#212
```

```
#213    list<GCInfo<T> >::iterator p;
#214
#215    p = findPtrInfo(t);
#216
#217    // 如果 t 已在 gclist 內，就累加其參用計數器。
#218    // 否則將它加入這個串列。
#219    if(p != gclist.end())
#220      p->refcount++; // 累加 reference 計數器
#221    else {
#222      //  建立並儲存這一筆（entry）
#223      GCInfo<T> gcObj(t, size);
#224      gclist.push_front(gcObj);
#225    }
#226
#227    addr = t;
#228    arraySize = size;
#229    if(size > 0) isArray = true;
#230    else isArray = false;
#231    #ifdef DISPLAY
#232      cout << "Constructing GCPtr. ";
#233      if(isArray)
#234        cout << " Size is " << arraySize << endl;
#235      else
#236        cout << endl;
#237    #endif
#238  }
```

這個建構式允許產生初始化和未初始化兩種實體。如果宣告的是個初始化實體，那麼必須透過 **t** 傳遞 **GCPtr** 物件將指向的記憶體。否則 **t** 為空。以下詳細分析 **GCPtr** 建構式的動作。

首先，如果 **first** 為真，意味它是被產生的第一個 **GCPtr** 物件，那麼就呼叫 **atexit()** 將 **shutdown()** 註冊為終結函式。**atexit()** 是 C++ 標準程式庫的一部分，用來註冊一個將在程式結束時被喚起的函式。本例 **shutdown()** 釋放所有因環狀引用（circular reference）而無法被釋放的記憶體。

接下來，搜尋 **gclist**，尋找現有元素中吻合 **t** 者。如果找到一個，就將其參用計數加 1。如果沒找到，就建立一個新的 **GCInfo** 物件含有該位址，並將它加入 **gclist**。

然後，將 **addr** 設為 **t** 所指定的位址，並正確地設置 **isArray** 和 **arraySize**。記住，如果您配置的是個陣列，那麼在宣告一個 **GCPtr** 指向它時，必須明白指定陣列的大小。如果不這麼做，記憶體就不會被正確釋放；如果該陣列的元素是 class 物件，那麼那些元素的解構式也無法被正確喚起。

2.5.5. GCPtr 解構式（Destructor）

下面是 **GCPtr** 解構式：

```
#326 // GCPtr 解構式
#327 template <class T, int size>
#328 GCPtr<T, size>::~GCPtr() {
#329   list<GCInfo<T> >::iterator p;
#330
#331   p = findPtrInfo(addr);
#332   if(p->refcount) p->refcount--; // 減少 ref 計數
#333
#334   #ifdef DISPLAY
#335     cout << "GCPtr going out of scope.\n";
#336   #endif
#337
#338   //  當指標離開其作用域時，進行垃圾回收
#339   collect();
#340
#341   //  為了實際運用，您可能希望不那麼經常地收集未用記憶體，或許在
#342   //  gclist 達到特定大小，或特定數量的 GCPtr 物件超出作用域，或
#343   //  記憶體不足時，才進行收集。
#344 }
```

每當有 **GCPtr** 物件逾越其作用域，就進行垃圾回收。這件事由 **~GCPtr()** 處理。它首先搜尋 **gclist**，尋找與「將被銷毀之 **GCPtr** 物件所指位址」對應的元素。一旦找到就將其 **refcount** 減 1。然後呼叫 **collect()** 釋放所有未用記憶體（也就是那些參用計數為 0 的記憶體）。

正如 **~GCPtr()** 末尾註釋所言，在實際應用上，並不是每當 **GCPtr** 物件逾越作用域就進行垃圾回收。是的，如果不那麼頻繁效果可能更好。回收頻率較低，往往更為高效。然而前面解釋過，每當一個 **GCPtr** 物件被銷毀就回收垃圾，對於展示垃圾回收器是有幫助的，因為這樣能夠清楚顯示垃圾回收器的動作。

2.5.6. 以 collect() 收集垃圾

collect() 是垃圾回收動作的發生處。程式碼展示如下：

```
#346 //  收集垃圾。如果至少有一個物件被釋放，就傳回 true。
#347 template <class T, int size>
#348 bool GCPtr<T, size>::collect() {
```

```
#349  bool memfreed = false;
#350
#351  #ifdef DISPLAY
#352    cout << "Before garbage collection for ";
#353    showlist();
#354  #endif
#355
#356  list<GCInfo<T> >::iterator p;
#357  do {
#358
#359    // 掃描 gclist，尋找未曾被引用的指標
#360    for(p = gclist.begin(); p != gclist.end(); p++) {
#361      // 如果正在使用，就跳過
#362      if(p->refcount > 0) continue;
#363
#364      memfreed = true;
#365
#366      // 從 gclist 中刪除未用項目
#367      gclist.remove(*p);
#368
#369      // 除非 GCPtr 為空，否則釋放記憶體
#370      if(p->memPtr) {
#371        if(p->isArray) {
#372          #ifdef DISPLAY
#373            cout << "Deleting array of size "
#374                 << p->arraySize << endl;
#375          #endif
#376          delete[] p->memPtr; // 刪除陣列
#377        }
#378        else {
#379          #ifdef DISPLAY
#380            cout << "Deleting: "
#381                 << *(T *) p->memPtr << "\n";
#382          #endif
#383          delete p->memPtr; // 刪除單一元素
#384        }
#385      }
#386
#387      // 重新開始搜索
#388      break;
#389    }
#390
#391  } while(p != gclist.end());
#392
#393  #ifdef DISPLAY
```

```
#394     cout << "After garbage collection for ";
#395     showlist();
#396   #endif
#397
#398   return memfreed;
#399 }
```

collect() 的工作方式是，掃描 **gclist**，尋找 **refcount** 為零的元素項。一旦找到就呼叫 STL list 的 **remove()** 將它從 **gclist** 中移除。然後釋放與該元素項相關聯的記憶體。

回憶一下，C++ 中的單一物件應該透過 **delete** 釋放，物件陣列則應該經由 **delete[]** 釋放。因此，元素項的 **isArray** 決定了我們應該使用 **delete** 或 **delete[]** 來釋放記憶體。這是您必須為「指向陣列之 **GCPtr** 物件」指定陣列大小的原因之一：它導致 **isArray** 被設為 **true**，因此如果 **isArray** 設置不當，就無法正確釋放記憶體。

雖然垃圾回收目的是自動回收未被使用的記憶體，但如果有必要，也可以試試手動控制。任何人如果想申請垃圾回收，可以直接呼叫 **collect()**。注意，它是 **GCPtr** 的一個 static 函式，意味它可以不經由任何物件就被喚起。例如：

```
GCPtr<int>::collect(); // 收集所有未被使用的 int 指標
```

這導致 **gclist<int>** 被收集。由於不同型別的指標有不同的 **gclist** 物件，所以您需要為每個您想收集的 list 呼叫 **collect()**。老實說，如果您需要密切管理動態配置物件的釋放情況，使用 C++ 提供的手動配置系統或許更好。最好是在特殊情況下才直接呼叫 **collect()**，例如當可用記憶體變得出乎意料地少。

2.5.7. 重載賦值運算子 （Overloaded Assignment Operators）

GCPtr 重載了 **operator=()** 兩次：一次是為了將新位址賦予 **GCPtr** 指標，一次是為了將 **GCPtr** 指標賦予另一個 **GCPtr** 指標。以下顯示這兩個版本：

```
#401 // 重載「指向 GCPtr」的指標的賦值操作
#402 template <class T, int size>
#403 T * GCPtr<T, size>::operator=(T *t) {
#404   list<GCInfo<T> >::iterator p;
#405
#406   // 首先將當前所指記憶體的參用計數器減 1
#407   p = findPtrInfo(addr);
#408   p->refcount--;
#409
```

```
#410   // 然後，如果新位址已存在於系統，就累加其計數。
#411   // 否則為 gclist 新建一個條目（entry）
#412   p = findPtrInfo(t);
#413   if(p != gclist.end())
#414     p->refcount++;
#415   else {
#416     // 產生並保存該條目（entry）
#417     GCInfo<T> gcObj(t, size);
#418     gclist.push_front(gcObj);
#419   }
#420
#421   addr = t; // 保存位址
#422
#423   return t;
#424 }
#425
#426 // 重載「GCPtr 對 GCPtr」的賦值操作
#427 template <class T, int size>
#428 GCPtr<T, size> & GCPtr<T, size>::operator=(GCPtr &rv) {
#429   list<GCInfo<T> >::iterator p;
#430
#431   // 首先將當前所指記憶體的參用計數器減 1
#432   p = findPtrInfo(addr);
#433   p->refcount--;
#434
#435   // 然後累加新物件的參用計數器
#436   p = findPtrInfo(rv.addr);
#437   p->refcount++; // 累加 ref 計數
#438
#439   addr = rv.addr;// 保存位址
#440
#441   return rv;
#442 }
```

第一個重載函式處理的是「**GCPtr** 指標在左，位址在右」的賦值。它可以處理如下情況：

```
GCPtr<int> p;
// ...
p = new int(18);
```

在這裡，**new** 傳回的位址被賦予 **p**。當賦值動作發生，**operator=(T *t)**會被呼叫，新位址必須傳遞給 t。然後首先在 **gclist** 中搜尋目前所指記憶體的對應元素項，將其參用計數減 1。然後在 **gclist** 中尋找新位址。如果找到就將其參用計數加 1，否則就為新位址建立一個 **GCInfo** 物件，並將它加入 **gclist**。最後，將新位址保存在呼叫此函式之物件的 **addr** 資料欄中，並將該位址傳回。

賦值運算的第二個重載版本 **operator=(GCPtr &rv)**，用來處理以下類型的賦值：

```
GCPtr<int> p;
GCPtr<int> q;
// ...
p = new int(88);
q = p;
```

其中 **p** 和 **q** 都是 **GCPtr** 指標，而且 **p** 被賦予 **q**。此版本的賦值運算工作與上一個版本十分相似。首先在 **gclist** 中搜尋目前所指記憶體的對應元素項，將其參用計數減 1。然後在 **gclist** 中尋找 **rv.addr** 保存的新位址，並將其參用計數加 1。然後將呼叫此函式之物件的 **addr** 資料欄設為 **rv.addr** 內的位址。最後，將右側物件傳回，這就允許賦值操作成為鏈狀，像這樣：

```
p = q = w = z;
```

關於賦值運算子的工作方式還有另一個重點需要闡明。正如本章先前所提，技術上有可能在記憶體參用計數降至 0 時立刻將其回收，但這麼做會使每次指標操作時增加額外開銷。這就是重載的賦值運算子中，原先由左側運算元所指的記憶體的參用計數只是簡單減 1，沒有更進一步活動的原因。這就免除了實際釋放記憶體和維護 **gclist** 上的管理開銷 — 這些動作被推遲至垃圾回收器執行時才進行。這種作法有利於 **GCPtr** 的客戶碼執行得更快。它還使得在（可能）更有利於執行效率時才進行垃圾回收。

2.5.8. GCPtr 的 *copy* 建構式

由於需要跟蹤每個指向配置記憶體的指標，所以不能使用預設的 *copy* 建構式（該版本採用位元逐一拷貝，bitwise copy）。**GCPtr** 必須像這樣定義自己的 *copy* 建構式：

```
#240   // copy建構式
#241   GCPtr(const GCPtr &ob) {
#242     list<GCInfo<T> >::iterator p;
#243
#244     p = findPtrInfo(ob.addr);
```

```
#245     p->refcount++; // 累加 ref 計數
#246
#247     addr = ob.addr;
#248     arraySize = ob.arraySize;
#249     if(arraySize > 0) isArray = true;
#250     else isArray = false;
#251     #ifdef DISPLAY
#252       cout << "Constructing copy.";
#253       if(isArray)
#254         cout << " Size is " << arraySize << endl;
#255       else
#256         cout << endl;
#257     #endif
#258   }
```

還記得嗎，class 的 *copy* 建構式是在複製物件時被喚起，例如當物件被作為參數傳給某函式，或物件被函式傳回，或是當物件被用來初始化另一個物件時。**GCPtr** *copy* 建構式會將原物件所保存的資訊加以複製。它還將「原物件所指記憶體」的參用計數加 1。當這個拷貝超出作用域時，其參用計數會被減 1。

實際上，*copy* 建構式執行的額外工作常常沒有必要，因為絕大多數情況下重載後的賦值運算子都能夠正確維護「回收 list」。然而少數情況下 *copy* 建構式還是必需，例如某個函式配置一塊記憶體並傳回指向那塊記憶體的 GCPtr 物件。

2.5.9. 指標運算子和轉型函式（Pointer Operators and Conversion Function）

由於 **GCPtr** 是個指標型別，所以它必須重載指標運算子 * 和 ->，以及索引運算子 []。這些全由以下函式完成。重載這些運算子使我們有可能建立新式指標型別，而它們卻又是如此簡單，簡單得令人吃驚。

```
#272   // 傳回這個 GCPtr 物件所指的物件的 reference
#273   T &operator*() {
#274     return *addr;
#275   }
#276
#277   // 傳回被指向的位址
#278   T *operator->() { return addr; }
#279
#280   // 傳回一個 reference，指向 i 索引上的物件
#281   T &operator[](int i) {
#282     return addr[i];
#283   }
```

operator*() 傳回呼叫者（一個 **GCPtr** 物件）的 **addr** 資料欄所指物件的 reference，**operator[]** 則傳回被指定的元素的 reference；這個運算子只能對指向陣列的 **GCPtr** 物件使用。

正如前面提過，這裡並不支援指標算術。例如 **GCPtr** 沒有重載++或--運算子。原因是垃圾回收機制假設 **GCPtr** 物件指向被配置記憶體的起始點。如果 **GCPtr** 物件可被增量，那麼當該記憶體被當作垃圾而回收時，**delete** 取用的位址可能無效。

如果需要進行與指標算術相關的操作，有兩個選擇。第一，如果 **GCPtr** 物件指向陣列，您可以產生一個得以巡訪陣列的 **Iter** 物件。這種方法稍後再詳細介紹。第二，您可以呼叫 **GCPtr** 定義的 **T*** 轉型函式，將 **GCPtr** 物件轉成一個普通指標。該函式如下：

```
#285   // 轉化函式（Conversion function），轉為 T*
#286   operator T *() { return addr; }
```

這個函式傳回一個普通指標，指向 **addr** 記錄的位址。它可以被這樣使用：

```
GCPtr<double> gcp = new double(99.2);
double *p;

p = gcp;     // 現在，p 與 gcp 指向相同的記憶體
p++;         // 由於 p 是個普通指標，所以它可以被增量
```

此例中由於 **p** 是個普通指標，所以它可以像所有其他指標一樣地被使用。當然啦，這樣的操作能否產生有意義的結果，取決於您的應用程式。

轉型為 **T*** 的主要好處是，當您處理的程式碼要求的是普通指標時，您能夠以 **GCPtr** 替代 C++ 普通指標來用。例如，考慮這樣的程式碼：

```
GCPtr<char> str = new char[80];
strcpy(str, "this is a test");
cout << str << endl;
```

其中 **str** 是個指向 **char** 型別的 **GCPtr** 指標，在呼叫 **strcpy()** 時被使用。由於 **strcpy()** 希望其參數型別是 **char***，所以 **GCPtr** 內部自動呼叫轉型函式，轉換成 **T*** 型別，此時 **T** 是 **char** 型別。**str** 用於 **cout** 述句時也會自動呼叫同樣的轉型函式。轉型函式使 **GCPtr** 能夠無縫地整合到現有的 C++ 函式和類別中。

記住，這個轉型函式傳回的 **T*** 指標並不參與或影響垃圾回收。因此當某個 C++ 普通指標指向一塊配置記憶體時，仍有可能將該記憶體釋放。所以，最好明智地使用這個轉型函式，

而且不要太過頻繁地使用它。

2.5.10. begin() 和 end()

下面的 **begin()** 和 **end()** 與它們在 STL 中的兄弟很類似。

```
#289   // 傳回一個 Iter 物件，指向配置記憶體的起始位置
#290   Iter<T> begin() {
#291     int size;
#292
#293     if(isArray) size = arraySize;
#294     else size = 1;
#295
#296     return Iter<T>(addr, addr, addr + size);
#297   }
#298
#299   // 傳回一個 Iter 指向配置陣列之最後一個位置的下一位置
#300   Iter<T> end() {
#301     int size;
#302
#303     if(isArray) size = arraySize;
#304     else size = 1;
#305
#306     return Iter<T>(addr + size, addr, addr + size);
#307   }
```

begin() 傳回一個 **Iter** 物件，指向 **addr** 所指陣列的起點。**end()** 傳回一個 **Iter** 物件，指向該陣列末尾的下一位置。雖然沒有什麼能夠阻止一個「指向單一物件」的 **GCPtr** 去呼叫這些函式，不過請記住，它們的目的是為了支援對配置而得的陣列的操作。獲得一個「指向單一物件」的 **Iter** 並沒有害處，只不過不具意義罷了。

2.5.11. shutdown()

如果程式中出現 **GCPtr** 環狀引用，那麼當程式結束之際，仍會存在一些動態配置物件需要被釋放。這很重要，因為這些物件的解構式可能必須被喚起。**shutdown()** 處理這個任務。正如前面所說，此函式係在第一個 **GCPtr** 物件產生之際被 **atexit()** 註冊，這意味 **shutdown()** 將在程式結束時被喚起。

shutdown() 如下所示：

```
#483 // 程式結束前清除 gclist
#484 template <class T, int size>
#485 void GCPtr<T, size>::shutdown() {
```

```
#486
#487   if(gclistSize() == 0) return; // list 為空
#488
#489   list<GCInfo<T> >::iterator p;
#490
#491   for(p = gclist.begin(); p != gclist.end(); p++) {
#492     // 將所有參用計數器設為 0
#493     p->refcount = 0;
#494   }
#495
#496   #ifdef DISPLAY
#497     cout << "Before collecting for shutdown() for "
#498          << typeid(T).name() << "\n";
#499   #endif
#500
#501   collect();
#502
#503   #ifdef DISPLAY
#504     cout << "After collecting for shutdown() for "
#505          << typeid(T).name() << "\n";
#506   #endif
#507 }
```

首先，如果 list 為空（正常情況下應該如此），那麼 **shutdown()** 就僅僅返回。否則它會將仍存在於 **gclist** 內的元素項的參用計數設為 0，並呼叫 **collect()**。還記得嗎，**collect()** 會釋放參用計數為 0 的物件。因此，將參用計數設為 0，可保證所有物件都被釋放。

2.5.12. 公用函式（Utility Functions）

最後，**GCPtr** 定義了兩個公用函式。一個是 **gclistSize()**，用以傳回目前 **gclist** 保存的元素數目。另一個是 **showlist()**，用以顯示 **gclist** 內容。它們都不是實現「垃圾回收式指標型別」所必需，但當您想觀察垃圾回收器的操作時，它們很有用。

2.6. GCInfo

gclist 中的「回收 list」保存著 **GCInfo** 物件。**GCInfo** 如下所示：

```
#139 // 以下 class 定義保存於垃圾回收資訊串列（information list）中的元素
#140 //
#141 template <class T> class GCInfo {
#142 public:
#143   unsigned refcount; // 目前的 reference 個數
#144
#145   T *memPtr; // 指向配置記憶體
```

```
#146
#147   /* isArray 在 memPtr 指向一個已配置陣列時為 true，
#148      否則為 false。*/
#149   bool isArray; // 如果指向陣列則為 true
#150
#151   /* 如果 memPtr 指向一個已配置陣列，
#152      arraySize 就用以表示其大小 */
#153   unsigned arraySize; // 陣列大小
#154
#155   // 在這裡，mPtr 指向被配置記憶體。如果那是個陣列，size
#156   // 就代表陣列大小。
#157   GCInfo(T *mPtr, unsigned size=0) {
#158     refcount = 1;
#159     memPtr = mPtr;
#160     if(size != 0)
#161       isArray = true;
#162     else
#163       isArray = false;
#164
#165     arraySize = size;
#166   }
#167 };
```

正如前面所言，每個 **GCInfo** 物件在 **memPtr** 資料欄中保存著配置記憶體的指標，在 **refcount** 資料欄中保存著與之相聯的參用計數。如果 **memPtr** 所指的是個陣列，那麼該陣列長度必須在建立 **GCInfo** 物件時指明。這種情況下 **isArray** 資料欄會被設為 **true**，陣列長度將被保存於 **arraySize**。

GCInfo 物件被存放於一個 STL list 中。為能夠對 list 進行搜尋，有必要定義 **operator==()**：

```
#169 // 重載 == 運算子，使 GCInfo 物件可被比較。此為 STL list 所需要。
#170 template <class T> bool operator==(const GCInfo<T> &ob1,
#171                 const GCInfo<T> &ob2) {
#172   return (ob1.memPtr == ob2.memPtr);
#173 }
```

兩個 **GCInfo** 物件只有在其 **memPtr** 資料欄相等時才被視為相等。您可能需要重載其他運算子才能將 **GCInfo** 物件置入 STL list 中，這取決於您所使用的編譯器。

2.7. Iter

Iter class 實現了一個類似迭代器的東西，可用來巡訪一個配置陣列內的元素。**Iter** 在技術上並非必要，因為 **GCPtr** 物件可被轉換為其基礎型別（也就是其所指物件的型別）的普通指標，但是 **Iter** 提供兩個好處。第一，它使您以類似巡訪 STL 容器的方式巡訪一個陣列，因

此您很快就熟悉了 **Iter** 的用法。第二，**Iter** 不允許越界存取，因此和普通指標相比 **Iter** 物件比較安全。然而請理解，**Iter** 並不參與垃圾回收。因此如果一個 **Iter** 物件賴以為基礎的 **GCPtr** 物件超出作用域，無論它所指記憶體是否仍為這個 **Iter** 物件所需要，該記憶體都將被釋放。

Iter 是個 template class，它是這樣定義的：

```
template <class T> class Iter {
```

它所指向的資料的型別係由 T 來傳遞。

Iter 定義了這些實體變數：

```
#024   T *ptr;   // 當前指標（current pointer value）
#025   T *end;   // 指向終端的下一個位置
#026   T *begin; // 指向配置陣列的起點
#027   unsigned length; // 數列長度（length of sequence）
```

Iter 當前所指位址保存在 **ptr** 中，陣列起始位址保存在 **begin** 中，陣列末尾元素的下一位置（位址）保存在 **end** 中。動態所得陣列的長度保存在 **length** 中。

Iter 定義了下面兩個建構式。第一個是 *default* 建構式，第二個則依照給定之 **ptr** 初值以及陣列首尾元素指標，建構出一個 **Iter** 物件。

```
#030   Iter() {
#031     ptr = end = begin = NULL;
#032     length = 0;
#033   }
#034
#035   Iter(T *p, T *first, T *last) {
#036     ptr =  p;
#037     end = last;
#038     begin = first;
#039     length = last - first;
#040   }
```

對於本章所示的垃圾回收器，**ptr** 初值總是等於 **begin**。然而您可以任意建立 **Iter** 物件並讓其 **ptr** 有不同的初值。

為了使 **Iter** 的「類指標性質」起作用，它重載了指標運算子 `*` 和 `->`，以及索引運算子 `[]`，如下所示：

```
#045   // 傳回 ptr 指向的值。不允許越界存取。
#046   T &operator*() {
#047     if( (ptr >= end) || (ptr < begin) )
```

```
#048      throw OutOfRangeExc();
#049    return *ptr;
#050  }
#051
#052  // 傳回 ptr 保存的位址。不允許越界存取。
#053  T *operator->() {
#054    if( (ptr >= end) || (ptr < begin) )
#055      throw OutOfRangeExc();
#056    return ptr;
#057  }
...
#087  // 傳回一個 reference 指向特定索引上的物件。不允許越界存取。
#088  T &operator[](int i) {
#089    if( (i < 0) || (i >= (end-begin)) )
#090      throw OutOfRangeExc();
#091    return ptr[i];
#092  }
```

運算子 * 傳回陣列中當前所指元素的 reference。運算子 -> 傳回當前所指元素的位址。運算子[]則傳回位於給定位置上的元素。注意，這些運算都不允許越界存取。如果有人企圖那麼做，就會獲得一個 **OutOfRangeExc** 異常。

Iter 定義了指標的各種算術運算子，像是++, --等，它們將 **Iter** 物件增量或減量。這些運算子允許您巡訪整個配置而得的陣列。基於速度考量，這些算術運算本身並不進行範圍檢查，然而任何越界存取都將觸發一個異常。指標的算術運算和相對關係運算都很直接，很容易理解。

Iter 還定義了一個名為 **size** 的公用函式，它傳回 **Iter** 所指陣列的長度。

正如先前所提，在 **GCPtr** 中針對每個實體都將 **Iter<T>**定義為 **GCiterator** 型別，用以簡化迭代器宣告。這意味您可以使用型別名稱 **GCiterator** 獲得任何 **GCPtr** 的 **Iter** 物件。

2.8. 如何使用 GCPtr

使用 **GCPtr** 很容易。首先含入 **gc.h** 檔案，然後宣告一個 **GCPtr** 物件，指定它將指向的資料型別。舉個例子，為宣告一個「名稱為 **p** 並可指向 **int** 資料」的 **GCPtr** 物件，請這麼寫：

```
GCPtr<int> p; // p 可以指向 int 物件
```

接下來，以 **new** 動態配置記憶體，並將 **new** 傳回的指標賦予 **p**，如下所示：

```
p = new int; // 將一個 int 位址指派給 p
```

您可以這樣使用賦值操作，將一個數值指派給配置而得的記憶體：

```
*p = 88; // 給予那個 int 一個數值
```

當然，您可以將前面三條述句整合成這樣：

```
GCPtr<int> p = new int(88); // 宣告並初始化
```

也可以獲得 **p** 所指記憶體的內容值，如下所示：

```
int k = *p;
```

這些例子顯示，通常您可以像使用 C++ 普通指標一樣地使用 **GCPtr**。唯一的區別是當您不需要該指標時，不必將它刪除。因為針對該指標而配置的記憶體，如果不再被需要，會被自動釋放。

將上述程式碼整合起來，便形成一個完整程式如下：

```
#001 #include <iostream>
#002 #include <new>
#003 #include "gc.h"
#004
#005 using namespace std;
#006
#007 int main() {
#008   GCPtr<int> p;
#009
#010   try {
#011     p = new int;
#012   } catch(bad_alloc exc) {
#013     cout << "Allocation failure!\n";
#014     return 1;
#015   }
#016
#017   *p = 88;
#018
#019   cout << "Value at p is: " << *p << endl;
#020
#021   int k = *p;
#022
#023   cout << "k is " << k << endl;
#024
#025   return 0;
#026 }
```

打開 "display" 選項後，程式輸出如下。（記住，只要在 **gc.h** 中定義 **DISPLAY**，就可以觀察垃圾回收器的操作。）

```
Constructing GCPtr.
Value at p is: 88
k is 88
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F12C0]     0       88
[00000000]     0       ---

Deleting: 88
After garbage collection for gclist<int, 0>:
memPtr     refcount    value
          -- Empty --
```

程式結束時，**p** 將逾越作用域。這將導致其解構式被呼叫，並使得 **p** 所指記憶體的參用計數減 1。由於 **p** 是指向該記憶體的唯一指標，所以其參用計數變成 0。接下來 **p** 的解構式呼叫 **collect()**，掃描 **gclist**，尋找參用計數為 0 的元素項。而由於先前與 **p** 相聯的元素項的參用計數為 0，所以該元素項的記憶體就被釋放了。

請注意，垃圾回收之前，**gclist** 內還有一個空指標元素項。這個空指標是在 **p** 被建立時建立的。還記得嗎，如果 **GCPtr** 沒有得到初始位址，就會使用空位址（也就是 0）。雖然技術上並非必要在 **gclist** 內保存一個空指標（因為它從不被釋放），但這麼做可簡化 **GCPtr** 其他部分，因為這可保證每個 **GCPtr** 物件在 **gclist** 中都有對應的元素項。

2.8.1. 處理配置異常（Allocation Exceptions）

正如先前程式所展示，由於垃圾回收器並不改變「透過 **new** 配置記憶體」的方式，所以您可以按照一般方式來處理配置失敗，也就是說您可以捕捉 **bad_alloc** 異常。是的，當 **new** 失敗它會拋出一個 **bad_alloc** 異常。當然前面那個程式不會耗盡記憶體，從而 **try/catch** 區塊並非真正必要，但現實世界中的程式有可能耗盡 heap 空間。因此您還是應該檢查這種可能性。

一般而言，使用垃圾回收機制時，對於 **bad_alloc** 異常的最佳回應是回收所有未用記憶體，並重試上次失敗的配置動作。這個技術被本章稍後的測試程式採用。您可以在自己的程式中使用相同技術。

2.8.2. 一個比較有趣的例子

這裡有個更有趣的例子，它顯示一個 **GCPtr** 物件在程式結束前超越作用域的結果。

```
#001 // 顯示一個 GCPtr 物件在程式結束前超越其作用域
#002 #include <iostream>
#003 #include <new>
#004 #include "gc.h"
#005
#006 using namespace std;
#007
#008 int main() {
#009   GCPtr<int> p;
#010   GCPtr<int> q;
#011
#012   try {
#013     p = new int(10);
#014     q = new int(11);
#015
#016     cout << "Value at p is: " << *p << endl;
#017     cout << "Value at q is: " << *q << endl;
#018
#019     cout << "Before entering block.\n";
#020
#021     // 現在，產生一個區域物件
#022     { // 區塊起始
#023       GCPtr<int> r = new int(12);
#024       cout << "Value at r is: " << *r << endl;
#025     } // 區塊結束，造成 r 超出其作用域
#026
#027     cout << "After exiting block.\n";
#028
#029   } catch(bad_alloc exc) {
#030     cout << "Allocation failure!\n";
#031     return 1;
#032   }
#033
#034   cout << "Done\n";
#035
#036   return 0;
#037 }
```

打開 "display" 選項後，本程式產生以下輸出：

```
Constructing GCPtr.
Constructing GCPtr.
Value at p is: 10
Value at q is: 11
Before entering block.
```

```
Constructing GCPtr.
Value at r is: 12
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F31D8]     0       12
[002F12F0]     1       11
[002F12C0]     1       10
[00000000]     0       ---

Deleting: 12
After garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F12F0]     1       11
[002F12C0]     1       10

After exiting block.
Done
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F12F0]     0       11
[002F12C0]     1       10

Deleting: 11
After garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F12C0]     1       10

GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F12C0]     0       10

Deleting: 10
After garbage collection for gclist<int, 0>:
memPtr     refcount    value

          -- Empty --
```

讓我們仔細分析這個程式和其輸出。首先請注意，**p** 和 **q** 都在 **main()** 起始處被產生出來，但是 **r** 只在進入其附近區塊時才被產生。正如您知道的，在 C++ 中區域變數只有在進入其程式區塊後才會被建立起來。當 **r** 被產生時，它所指的記憶體被賦以初值 12。而後顯示該值，而後程式區塊結束。這導致 **r** 超出其作用域，意味其解構式將被喚起。這將使得 **r** 在 **gclist** 中的參用計數減為 0。最後，呼叫 **collect()** 回收垃圾。

由於打開了 "display" 選項，所以當 **collect()** 開始執行，會顯示 **gclist** 的內容。注意到它有四筆元素。第一筆是之前與 **r** 相聯者，注意其 **refcount** 資料欄為零，表示 **memPtr** 資料欄所指向的記憶體不再被任何程式元素使用。接下來兩筆仍然有效，分別與 **p** 和 **q** 相聯。由於它們仍在使用中，所以這次並不釋放它們所指的記憶體。最後一筆表示 **p** 和 **q** 產生時指向的空指標。由於它不再有用，所以會被 **collect()** 從 list 中移除。（當然，這個空指標被刪除時並不會釋放任何記憶體。）

因為沒有任何其他 **GCPtr** 物件和 **r** 指向同一塊記憶體，所以 **r** 的記憶體可以被釋放，正如 "Deleting: 12" 那一行所顯示。一旦完成這個動作，程式又從這個區塊之後繼續執行。最終 **p** 和 **q** 在程式結束時超出其作用域，於是記憶體被釋放。本例之中 **q** 的解構式先被喚起，意味它先被收集。最終 **p** 被銷毀，**gclist** 被清空。

2.8.3. 配置物件和拋棄物件（Allocating and Discarding Objects）

一旦記憶體參用計數降至 0（意味沒有 **GCPtr** 物件指向它），它就可以被當做垃圾收集起來。理解這一點至為重要。原先指向該物件的 **GCPtr** 物件倒不一定會超出作用域。因此，您可以使用單一 **GCPtr** 物件指向任意數量的配置物件，只需一再給予這個 **GCPtr** 物件新值即是。被丟棄的記憶體最終會被收集起來。例如：

```
#001 //  配置物件及丟棄物件
#002 #include <iostream>
#003 #include <new>
#004 #include "gc.h"
#005
#006 using namespace std;
#007
#008 int main() {
#009   try {
#010     //  配置並丟棄物件
#011     GCPtr<int> p = new int(1);
#012     p = new int(2);
#013     p = new int(3);
#014     p = new int(4);
#015
#016     //  為求展示，手動收集未使用的記憶體。
#017     GCPtr<int>::collect();
#018
#019     cout << "*p: " << *p << endl;
#020   } catch(bad_alloc exc) {
#021     cout << "Allocation failure!\n";
```

```
#022    return 1;
#023  }
#024
#025  return 0;
#026 }
```

打開 "display" 選項後，程式輸出如下：

```
Constructing GCPtr.
Before garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F1310]     1       4
[002F1300]     0       3
[002F12D0]     0       2
[002F12A0]     0       1

Deleting: 3
Deleting: 2
Deleting: 1
After garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F1310]     1       4

*p: 4
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr     refcount    value
[002F1310]     0       4

Deleting: 4
After garbage collection for gclist<int, 0>:
memPtr     refcount    value
         -- Empty --
```

在這個程式中，一個指向 **int** 的 **GCPtr** 物件 **p**，被賦予四塊動態配置而得的記憶體（的指標），這四塊記憶體分別有不同的初值。然後呼叫 **collect()** 強制進行垃圾回收。請注意 **gclist** 的內容：三筆元素被標記為無效，只有指向「最後配置之記憶體」的那一項還處於使用狀態。接下來，未用的元素項被刪除。最後程式結束，**p** 超出作用域，最後一項也被移除。

請注意，**p** 指向的前三塊動態記憶體的參用計數為 0。原因出在重載的賦值運算子的工作方式。還記得嗎，當一個 **GCPtr** 物件被賦予一個新位址，其原始值的參用計數會被減 1。因此，每當 **p** 被賦予一個新整數的位址，舊位址的參用計數就被減 1。

另請注意一點：由於 **p** 宣告時被初始化，所以 null 指標項不會產生，也不會被放入 **gclist**。

記住，null 指標項只有在 **GCPtr** 不帶初值進行宣告時才會產生。

2.8.4. 配置陣列（Allocating Arrays）

如果使用 **new** 配置一個陣列，那麼必須在宣告 **GCPtr** 物件指向該陣列時，指定陣列的大小，以便將情況通知給那個 **GCPtr** 物件。例如，下面配置一個陣列，具備 5 筆 **double** 元素：

```
GCPtr<double, 5> pda = new double[5];
```

有兩個原因使我們必須指出陣列大小。第一，它通知 **GCPtr** 建構式說，此物件將指向一個配置而得的陣列，這會導致 **isArray** 資料欄被設為 **true**，而當 **isArray** 為 **true** 時，**collect()** 便會以 **delete[]** 釋放記憶體（這個形式能夠釋放一個動態配置而得的陣列），而不使用只能釋放單一物件的 **delete**。因此在這個例子中，當 **pda** 超出作用域時，便以 **delete[]** 釋放 **pda** 的所有 5 個元素。如果配置的是 class 物件陣列，確保釋放正確個數的物件尤為重要。唯有使用 **delete[]**，才能夠確保每個物件的解構式都被喚起。

指定大小的第二個原因是，使用 **Iter** 物件巡訪配置而得的陣列時，可避免越界。記得嗎，每當需要一個 **Iter** 物件，陣列的大小（保存於 **arraySize**）都會被 **GCPtr** 傳給 **Iter** 建構式。

請保持這個觀念：沒有什麼能夠強制規定「配置而得的陣列」只能被「指向陣列之 **GCPtr** 物件」操作。這完全是您的責任。

一旦配置了一個陣列，有兩種方式可以巡訪其元素。第一，可以給指向該陣列的 **GCPtr** 物件編排索引。第二，可以使用迭代器。下面顯示兩種作法。

使用陣列索引（Using Array Indexing）

下面這個程式產生一個 **GCPtr** 物件，指向一個由 10 個 **int** 元素形成的陣列。它配置該陣列，將其初始化為 0~9，最後顯示這些值。它藉由對 **GCPtr** 物件編列索引來執行這些活動。

```
#001 //  示範如何對一個 GCPtr 物件編列索引（indexing a GCPtr）
#002 #include <iostream>
#003 #include <new>
#004 #include "gc.h"
#005
#006 using namespace std;
#007
#008 int main() {
#009
#010   try {
#011     // 產生一個 GCPtr，指向一個配置而得的陣列，其中有 10 個 ints.
```

```
#012    GCPtr<int, 10> ap = new int[10];
#013
#014    // 運用陣列索引，給予該陣列一些值
#015    for(int i=0; i < 10; i++)
#016      ap[i] = i;
#017
#018    // 現在，顯示陣列內容
#019    for(int i=0; i < 10; i++)
#020      cout << ap[i] << " ";
#021
#022    cout << endl;
#023
#024   } catch(bad_alloc exc) {
#025    cout << "Allocation failure!\n";
#026    return 1;
#027   }
#028
#029   return 0;
#030 }
```

打開 "display" 選項後，程式輸出如下：

```
0 1 2 3 4 5 6 7 8 9
```

由於 **GCPtr** 物件模仿了一個 C++ 普通指標，所以不進行任何邊界檢查，因此有可能超出動態配置陣列的上界或下界。為此，當您使用 **GCPtr** 物件來巡訪陣列時，請像「使用 C++ 普通指標巡訪陣列」那般小心。

使用迭代器（Using Iterators）

雖然「對著陣列使用索引」是巡訪一塊動態配置陣列的方便辦法，但這並非是您能支配的唯一辦法。對於很多應用程式，使用迭代器是更好的選擇，因為它可以避免邊界錯誤。回憶一下，對於 **GCPtr**，其迭代器是個 **Iter** 物件。**Iter** 支持完整指標操作，例如++。它還允許迭代器像個陣列似地被索引。

下面重寫前一個程式，其中用上了迭代器。記得嗎，欲獲得 **GCPtr** 物件的迭代器，最簡單的辦法是使用 **GCiterator**，那是自動綁定到泛化型別 **T** 身上的一個 **GCPtr** 內部定義。

```
#001 // 示範一個迭代器
#002 #include <iostream>
#003 #include <new>
#004 #include "gc.h"
#005
#006 using namespace std;
#007
```

```
#008 int main() {
#009
#010   try {
#011     // 建立一個 GCPtr 指向陣列，陣列中有 10 個 int 元素
#012     GCPtr<int, 10> ap = new int[10];
#013
#014
#015     //  宣告一個 int 迭代器
#016     GCPtr<int>::GCiterator itr;
#017
#018     //  將陣列起始指標賦予 itr
#019     itr = ap.begin();
#020
#021     //  使用陣列索引，為陣列賦予一組數值
#022     for(unsigned i=0; i < itr.size(); i++)
#023       itr[i] = i;
#024
#025     // 現在，使用迭代器巡訪陣列
#026     for(itr = ap.begin(); itr != ap.end(); itr++)
#027       cout << *itr << " ";
#028
#029     cout << endl;
#030
#031   } catch(bad_alloc exc) {
#032     cout << "Allocation failure!\n";
#033     return 1;
#034   } catch(OutOfRangeExc exc) {
#035     cout << "Out of range access!\n";
#036     return 1;
#037   }
#038
#039
#040   return 0;
#041 }
```

您可能會想自己試著增加 itr 的值，使它所指位置超出動態配置陣列的邊界。然後嘗試存取該位置上的值。正如您將看到的情況，會有一個 OutOfRangeExc 物件被拋出。一般而言，您可以隨意對一個迭代器增量或減量，不會導致異常。然而如果它並不是指向其做為底層的陣列內部，並嘗試獲得或設置該位置上的值，就會導致一個越界錯誤。

2.8.5. 對 Class Types 使用 GCPtr

對 class type 使用 **GCPtr**，其方式和對內建型別一樣。下面是個配置 **MyClass** 物件的小程式：

```
#001 // 對 class 型別使用 GCPtr
#002 #include <iostream>
```

```
#003 #include <new>
#004 #include "gc.h"
#005
#006 using namespace std;
#007
#008 class MyClass {
#009   int a, b;
#010 public:
#011   double val;
#012
#013   MyClass() { a = b = 0; }
#014
#015   MyClass(int x, int y) {
#016     a = x;
#017     b = y;
#018     val = 0.0;
#019   }
#020
#021   ~MyClass() {
#022     cout << "Destructing MyClass(" <<
#023         a << ", " << b << ")\n";
#024   }
#025
#026   int sum() {
#027     return a + b;
#028   }
#029
#030   friend ostream &operator<<(ostream &strm, MyClass &obj);
#031 };
#032
#033 // 一個經過重載的 inserter（<<），用來顯示 MyClass
#034 ostream &operator<<(ostream &strm, MyClass &obj) {
#035   strm << "(" << obj.a << " " << obj.b << ")";
#036   return strm;
#037 }
#038
#039 int main() {
#040   try {
#041     GCPtr<MyClass> ob = new MyClass(10, 20);
#042
#043     // 透過重載後的 inserter（<<）顯示內容值
#044     cout << *ob << endl;
#045
#046     // 改變 ob 所指物件
#047     ob = new MyClass(11, 21);
```

```
#048     cout << *ob << endl;
#049
#050     // 透過一個 GCPtr 物件呼叫其成員函式
#051     cout << "Sum is : " << ob->sum() << endl;
#052
#053     // 透過一個 GCPtr 物件，賦值給 class 成員
#054     ob->val = 98.6;
#055     cout << "ob->val: " << ob->val << endl;
#056
#057     cout << "ob is now " << *ob << endl;
#058   } catch(bad_alloc exc) {
#059     cout << "Allocation error!\n";
#060     return 1;
#061   }
#062
#063   return 0;
#064 }
```

請注意 **MyClass** 的成員如何經由運算子->而被存取。記住，**GCPtr** 定義的是個指標型別，因此對 **GCPtr** 的操作和對其他指標的操作方式一模一樣。

打開 "display" 選項後，程式輸入如下：

```
(10 20)
(11 21)
Sum is : 32
ob->val: 98.6
ob is now (11 21)
Destructing MyClass(11, 21)
Destructing MyClass(10, 20)
```

請特別注意最後兩行。它們是垃圾回收時 **~MyClass()** 的輸出。雖然只產生一個 **GCPtr** 指標，但卻配置了兩個 **MyClass** 物件。這兩個物件都被表現為「回收 list」中的元素項。當 **ob** 被銷毀時，**gclist** 會被掃描，試圖找出參用計數為 0 的元素項。本例找到了兩個這樣的元素項，於是釋放它們所指的記憶體。

2.8.6. 一個較大的示範程式

以下展示一個較大的例子，練習了 **GCPtr** 的所有特性：

```
#001 // 演示 GCPtr 類別
#002 #include <iostream>
#003 #include <new>
#004 #include "gc.h"
```

```
#005
#006 using namespace std;
#007
#008 //  以 class type 測試 GCPtr 的一個簡單類別
#009 class MyClass {
#010   int a, b;
#011 public:
#012   double val;
#013
#014   MyClass() { a = b = 0; }
#015
#016   MyClass(int x, int y) {
#017     a = x;
#018     b = y;
#019     val = 0.0;
#020   }
#021
#022   ~MyClass() {
#023     cout << "Destructing MyClass(" <<
#024          a << ", " << b << ")\n";
#025   }
#026
#027   int sum() {
#028     return a + b;
#029   }
#030
#031   friend ostream &operator<<(ostream &strm, MyClass &obj);
#032 };
#033
#034 //  為 MyClass 產生一個 inserter（<<）
#035 ostream &operator<<(ostream &strm, MyClass &obj) {
#036   strm << "(" << obj.a << " " << obj.b << ")";
#037   return strm;
#038 }
#039
#040 // 傳遞一個普通指標給一個函式。
#041 void passPtr(int *p) {
#042   cout << "Inside passPtr(): "
#043        << *p << endl;
#044 }
#045
#046 // 將一個 GCPtr 物件傳給某個函式
#047 void passGCPtr(GCPtr<int, 0> p) {
#048   cout << "Inside passGCPtr(): "
#049        << *p << endl;
#050 }
#051
```

```
#052 int main() {
#053
#054   try {
#055     // 宣告一個 GCPtr 物件，指向 int 型別
#056     GCPtr<int> ip;
#057
#058     // 配置一個 int 資料，並將其位址指派給 ip
#059     ip = new int(22);
#060
#061     // 顯示其值
#062     cout << "Value at *ip: " << *ip << "\n\n";
#063
#064     // 將 ip 傳給某函式
#065     passGCPtr(ip);
#066
#067     // ip2 被產生，然後超出作用域
#068     {
#069       GCPtr<int> ip2 = ip;
#070     }
#071
#072     int *p = ip; // 轉化為指標 int*
#073
#074     passPtr(p);  // 將 int* 傳給 passPtr()
#075
#076     *ip = 100;   // 為 ip 指派新值
#077
#078     // 現在，使用隱式轉換將其轉換成 int*
#079     passPtr(ip);
#080     cout << endl;
#081
#082     // 產生一個 GCPtr 物件指向 int 陣列
#083     GCPtr<int, 5> iap = new int[5];
#084
#085     // 初始化動態配置陣列
#086     for(int i=0; i < 5; i++)
#087       iap[i] = i;
#088
#089     // 顯示陣列內容
#090     cout << "Contents of iap via array indexing.\n";
#091     for(int i=0; i < 5; i++)
#092       cout << iap[i] << " ";
#093     cout << "\n\n";
#094
#095     // 產生一個 int GCiteraotr
#096     GCPtr<int>::GCiterator itr;
#097
```

```
#098    // 現在，使用迭代器來存取動態配置陣列
#099    cout << "Contents of iap via iterator.\n";
#100    for(itr = iap.begin(); itr != iap.end(); itr++)
#101      cout << *itr << " ";
#102    cout << "\n\n";
#103
#104    // 產生並丟棄許多物件
#105    for(int i=0; i < 10; i++)
#106      ip = new int(i+10);
#107
#108    // 現在， 對回收串列 GCPtr<int> list 進行手動式垃圾回收。
#109    // 記住，GCPtr<int,5> 指標並不會因此呼叫而被收集。
#110    cout << "Requesting collection on GCPtr<int> list.\n";
#111    GCPtr<int>::collect();
#112
#113    // 現在，對 class type 使用 GCPtr
#114    GCPtr<MyClass> ob = new MyClass(10, 20);
#115
#116    // 透過重載的 inserter（<<）顯示其值
#117    cout << "ob points to " << *ob << endl;
#118
#119    // 改變 ob 指向
#120    ob = new MyClass(11, 21);
#121    cout << "ob now points to " << *ob << endl;
#122
#123    // 透過 GCPtr 呼叫一個成員函式
#124    cout << "Sum is : " << ob->sum() << endl;
#125
#126    // 透過 GCPtr 為 class 成員賦值
#127    ob->val = 19.21;
#128    cout << "ob->val: " << ob->val << "\n\n";
#129
#130    cout << "Now work with pointers to class objects.\n";
#131
#132    // 宣告一個 GCPtr，指向含有 5 個 MyClass 元素的陣列
#133    GCPtr<MyClass, 5> v;
#134
#135    // 配置該陣列
#136    v = new MyClass[5];
#137
#138    // 得到一個 MyClass GCiterator
#139    GCPtr<MyClass>::GCiterator mcItr;
#140
```

```
#141    // 初始化 MyClass 陣列
#142    for(int i=0; i<5; i++) {
#143      v[i] = MyClass(i, 2*i);
#144    }
#145
#146    // 使用索引來顯示 MyClass 陣列的內容
#147    cout << "Cycle through array via array indexing.\n";
#148    for(int i=0; i<5; i++) {
#149      cout << v[i] << " ";
#150    }
#151    cout << "\n\n";
#152
#153    // 使用迭代器來顯示 MyClass 陣列的內容
#154    cout << "Cycle through array through an iterator.\n";
#155    for(mcItr = v.begin(); mcItr != v.end(); mcItr++) {
#156      cout << *mcItr << " ";
#157    }
#158    cout << "\n\n";
#159
#160    // 這是編寫先前的迴圈的另一種方法
#161    cout << "Cycle through array using a while loop.\n";
#162    mcItr = v.begin();
#163    while(mcItr != v.end()) {
#164      cout << *mcItr << " ";
#165      mcItr++;
#166    }
#167    cout << "\n\n";
#168
#169    cout << "mcItr points to an array that is "
#170         <<  mcItr.size() << " objects long.\n";
#171
#172    // 獲得兩個迭代器之間的元素數目
#173    GCPtr<MyClass>::GCiterator mcItr2 = v.end()-2;
#174    mcItr = v.begin();
#175    cout << "The difference between mcItr2 and mcItr is "
#176         << mcItr2 - mcItr;
#177    cout << "\n\n";
#178
#179    // 也可以透過迴圈來巡訪
#180    cout << "Dynamically compute length of array.\n";
#181    mcItr = v.begin();
#182    mcItr2 = v.end();
#183    for(int i=0; i < mcItr2 - mcItr; i++) {
#184      cout << v[i] << " ";
#185    }
#186    cout << "\n\n";
#187
#188
```

```
#189    // 現在，逆序顯示該陣列
#190    cout << "Cycle through array backwards.\n";
#191    for(mcItr = v.end()-1; mcItr >= v.begin(); mcItr--)
#192      cout << *mcItr << " ";
#193    cout << "\n\n";
#194
#195    // 當然，也可以使用普通指標來巡訪該陣列
#196    cout << "Cycle through array using 'normal' pointer\n";
#197    MyClass *ptr = v;
#198    for(int i=0; i < 5; i++)
#199      cout << *ptr++ << " ";
#200    cout << "\n\n";
#201
#202    // 可以透過一個 GCiterarot 物件來存取成員
#203    cout << "Access class members through an iterator.\n";
#204    for(mcItr = v.begin(); mcItr != v.end(); mcItr++) {
#205      cout << mcItr->sum() << " ";
#206    }
#207    cout << "\n\n";
#208
#209    // 可以像一般指標一樣，正常地配置和刪除一個指向 GCPtr 物件的指標
#210    cout << "Use a pointer to a GCPtr.\n";
#211    GCPtr<int> *pp = new GCPtr<int>();
#212    *pp = new int(100);
#213    cout << "Value at **pp is: " << **pp;
#214    cout << "\n\n";
#215
#216    // 由於 pp 不是一個「垃圾回收式」指標，所以必須手動刪除它
#217    delete pp;
#218  } catch(bad_alloc exc) {
#219    // 一個實際應用程式如果發生配置錯誤，
#220    // 可能會嘗試呼叫 collec() 來釋放記憶體
#221    cout << "Allocation error.\n";
#222  }
#223
#224  return 0;
#225 }
```

下面是關閉 "display" 選項後的輸出：

```
Value at *ip: 22

Inside passGCPtr(): 22
Inside passPtr(): 22
Inside passPtr(): 100

Contents of iap via array indexing.
0 1 2 3 4

Contents of iap via iterator.
0 1 2 3 4

Requesting collection on GCPtr<int> list.
ob points to (10 20)
ob now points to (11 21)
Sum is : 32
ob->val: 19.21

Now work with pointers to class objects.
Destructing MyClass(0, 0)
Destructing MyClass(1, 2)
Destructing MyClass(2, 4)
Destructing MyClass(3, 6)
Destructing MyClass(4, 8)
Cycle through array via array indexing.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array through an iterator.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array using a while loop.
(0 0) (1 2) (2 4) (3 6) (4 8)

mcItr points to an array that is 5 objects long.
The difference between mcItr2 and mcItr is 3

Dynamically compute length of array.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array backwards.
(4 8) (3 6) (2 4) (1 2) (0 0)

Cycle through array using 'normal' pointer
(0 0) (1 2) (2 4) (3 6) (4 8)
```

```
Access class members through an iterator.
0 3 6 9 12

Use a pointer to a GCPtr.
Value at **pp is: 100

Destructing MyClass(4, 8)
Destructing MyClass(3, 6)
Destructing MyClass(2, 4)
Destructing MyClass(1, 2)
Destructing MyClass(0, 0)
Destructing MyClass(11, 21)
Destructing MyClass(10, 20)
```

您可以嘗試打開 "display" 選項（也就是在 **gc.h** 檔案中定義 **DISPLAY**），然後編譯並執行這個程式，追蹤整個過程，將每個輸出和程式述句對應起來。這會讓您更好地體會垃圾回收器的工作。記住，只要一個 **GCPtr** 物件越過作用域，就會發生垃圾回收。這種情況會發生在程式中的某幾個地方，例如當一個「接收 **GCPtr** 物件副本」的函式結束時。這種情況下這份副本超出了作用域，於是發生垃圾回收。也請記住，每一種 **GCPtr** 型別都維護自己的一個 **gclist**。因此，從一個 list 中回收垃圾，並不會導致其他 list 也回收垃圾。

2.8.7. 負載測試（Load Testing）

以下程式對 **GCPtr** 進行負載測試，作法是重複配置和丟棄物件，直到自由記憶體被耗盡。一旦如此，**new** 將拋出一個 **bad_alloc** 異常。異常處理常式會明白呼叫 **collect()** 來回收未使用的記憶體，然後繼續這個程序。您可以在自己的程式中使用相同的技術。

```
#001 // 建立和丟棄成千上萬個物件，藉此對 GCPtr 進行負載測試（load test）。
#002 #include <iostream>
#003 #include <new>
#004 #include <limits>
#005 #include "gc.h"
#006
#007 using namespace std;
#008
#009 // 以下是對 GCPtr 進行負載測試時所用的一個簡單 class
#010 class LoadTest {
#011   int a, b;
#012 public:
#013   double n[100000]; // 僅僅為了占用記憶體
#014   double val;
#015
```

```
#016   LoadTest() { a = b = 0; }
#017
#018   LoadTest(int x, int y) {
#019     a = x;
#020     b = y;
#021     val = 0.0;
#022   }
#023
#024   friend ostream &operator<<(ostream &strm, LoadTest &obj);
#025 };
#026
#027 // 為 LoadTest 設計一個 inserter（<<）
#028 ostream &operator<<(ostream &strm, LoadTest &obj) {
#029   strm << "(" << obj.a << " " << obj.b << ")";
#030   return strm;
#031 }
#032
#033 int main() {
#034   GCPtr<LoadTest> mp;
#035   int i;
#036
#037   for(i = 1; i < 20000; i++) {
#038     try {
#039       mp = new LoadTest(i, i);
#040     } catch(bad_alloc xa) {
#041       // 一旦發生配置錯誤，呼叫 collect()回收垃圾
#042       cout << "Last object: " << *mp << endl;
#043       cout << "Length of gclist before calling collect(): "
#044            << mp.gclistSize() << endl;
#045       GCPtr<LoadTest>::collect();
#046       cout << "Length after calling collect(): "
#047            << mp.gclistSize() << endl;
#048     }
#049   }
#050
#051   return 0;
#052 }
```

下面顯示 "display" 選項關閉情況下的一部分程式輸出。當然，您的輸出可能不一樣，因為您的系統的自由記憶體數量以及您所使用的編譯器，可能和我的不同。

```
Last object: (518 518)
Length of gclist before calling collect(): 518
```

```
Length after calling collect(): 1
Last object: (1036 1036)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (1554 1554)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (2072 2072)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (2590 2590)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (3108 3108)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (3626 3626)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
```

2.8.8. 某些限制

使用 **GCPtr** 時，必須遵守某些限制：

1. 不能建立全域性（global）**GCPtr** 物件。還記得嗎，全域性物件只有在程式結束後才超出作用域。當一個全域性 **GCPtr** 物件超出作用域時，**GCPtr** 解構式呼叫 **collect()** 企圖釋放未被使用的記憶體。可問題是 **gclist** 可能已經被銷毀了（時間先後取決於您所使用的 C++ 編譯器）。這種情況下執行 **collect()**，會導致一個執行期錯誤。因此 **GCPtr** 應該只在產生區域性（local）物件時才被使用。
2. 使用動態配置陣列時，您必須在宣告一個 **GCPtr** 物件指向它時，同時指定陣列大小。但是本章並沒有設計任何機制強制您這麼做，所以請小心。
3. 不得透過顯式運用 **delete** 來釋放 **GCPtr** 物件所指的記憶體。如果您需要立刻釋放一個物件，請呼叫 **collect()**。
4. **GCPtr** 物件只允許指向一塊以 **new** 動態配置的記憶體。如果將其他任何記憶體（的指標）賦予 **GCPtr** 物件，那麼當那個 **GCPtr** 物件超出作用域時會導致錯誤，因為它將嘗試釋放一塊並非配置而得（並非來自 heap）的記憶體。
5. 基於本章前面所描述的理由，若能避免「環狀指標參用」（circular pointer references），再好不過。雖然所有配置而得的記憶體最終都會被釋放，但是內含「環狀引用」的物件在程式結束前都持續保持使用狀態（即使它們並不是），因此無法被釋放，也就永遠無法被其他程式元素所用。

2.9. 再接再厲

將 **GCPtr** 裁減適度以符合您的程式需要，是很簡單的事。正如前面所言，您可能會想嘗試的一個變化是：只在程式達到某些度量標準時才進行垃圾回收，例如 **gclist** 達到一個特定值，或是有特定數量的 **GCPtr** 物件超出了作用域。

對 **GCPtr** 的一項有趣改善是：重載 **new**，使它在配置失敗時自動回收垃圾。或是避免使用 **new** 來配置記憶體，改用 **GCPtr** 物件及其定義的 *factory method*（工廠方法；一種 design patterns）來替代 **new**。這樣做使您能夠仔細控制記憶體的動態配置，但卻也使得配置程序從根本上有別於 C++ 內建手法。

您可能會想嘗試其他辦法來解決「環狀引用」（circular references）問題。辦法之一是實現所謂的「弱引用」（weak reference）概念，它不會阻止垃圾回收。於是可以在任何需要「環狀引用」的時候使用「弱引用」。

您可以在第三章找到或許是對 **GCPtr** 最有趣的改變。那裡創造了一個多緒版本，其垃圾回收動作在 CPU 空閒時間自動運轉。