簡介

學習 C++ 為一探索旅程，尤其在於此語言容納多種程式設計規範，包括物件導向程式設計、一般程式設計，以及傳統的程序式程式設計。本書的第五版以 ISO C++標準，簡稱 C++99 及 C++03，或 C++99/03，來描述語言。（2003 版本絕大多數為針對 1999 標準所作的技術修正，並無加入任何新功能。）從那時候開始，C++ 持續發展。在撰寫本書的時候，國際 C++ 標準委員會剛通過標準的新版本。此標準在開發期間被簡稱為 C++0x，如今被稱作 C++11。大部分同時期的編譯器皆很完善地支援 C++99/03，本書絕大多數的範例皆遵從該標準。但新標準的多種功能已出現在某些實作內容裡，此版本的 C++ Primer Plus 會探討這些新功能。

C++ Primer Plus 整合基本 C 語言和現在 C++ 特性的討論，使得這本書是完整獨立的。它呈現 C++ 的基本原理並用簡短、切重要點且易於複製和實驗的程式說明之。您會學習如何輸入和輸出，如何使程式執行反覆的工作並做選擇，處理資料的許多方法，以及如何使用函數。您會學到 C++ 在 C 上新增的許多特性，包含下列各點：

- 類別和物件
- 繼承
- 同名異式，虛擬函數，以及 RTTI（執行期型態識別）
- 函數多載
- reference 變數
- 泛型，或是與型態無關的程式設計，例如樣版（template）和 Standard Template Library（STL）所提供
- 處理錯誤狀況的異常機制
- 管理函數、類別和變數名稱的名稱空間

碁峰 www.gotop.com.tw

入門的方法

C++ Primer Plus 對於呈現這些內容的工作有數項優點。將近 20 年前，入門書傳統上都從 C Primer Plus 開始，並包含下列成功的哲學：

- 入門書應該是易於使用、親切的指南。
- 入門書不應假設您已經熟悉所有相關的程式設計概念。
- 入門書強調動手學習簡短、容易輸入的範例，使您一次瞭解一至兩個概念。
- 入門書用圖形闡述觀念。
- 入門書提供問題和練習使您可以檢視自己的瞭解程度，因此本書適於自我學習或是課堂用書。

跟隨這些原則，本書幫助您瞭解這豐富的語言，以及如何使用它。例如：

- 對於何時使用特殊的特性，例如使用公用繼承模擬所謂的 is-a 關係，它提供概念性的指引。
- 它說明常見的 C++ 程式設計習慣用法和技術。
- 它提供各種方塊文章，包括小技巧、注意事項和真實世界註記。

我們（作者和編輯者）盡其所能使內容是切重要點、簡單，而且有趣。我們的目標是在看完本書時，您可以撰寫完整、有效的程式並且樂在其中。

本書的範例程式

本書提供相當多精彩的範例程式，並且大部分都具有完整的程式碼。與前幾版一樣，本書採用泛型的 C++，所以不會受限於特定種類的電腦、作業系統，或是編譯程式。所有的程式都以 Windows 7、Macintosh OS X 與 Linux 等作業系統測試過。只有一些程式會受編譯程式不一致的影響。自本書上一版發行之後，編譯程式對於 C++ 標準的相容性已有許多的改善。

本書之範例程式碼下載：http://books.gotop.com.tw/download/ACL033800

本書的架構

這本書分成 18 章及 10 份附錄（本書之附錄部分以 PDF 格式供讀者下載，請至碁峰網站 http://books.gotop.com.tw/v_ACL033800 下載），總結如下：

碁峰 www.gotop.com.tw

- **第 1 章進入 C++ 的世界**：這一章敘述 Bjarne Stroustrup 如何在 C 語言中，加入物件導向程式設計，而產生 C++ 程式設計語言。您會學到程序式語言（Procedural language），如 C，和物件導向語言（如 C++）之間的差異。也會讀到有關聯合 ANSI/ISO 的工作發展出 C++ 標準。這章討論如何產生 C++ 程式的方法，概述數個目前 C++ 編譯程式的方法。最後，它描述本書使用的慣例。
- **第 2 章開始 C++**：此章指引您完成簡單 C++ 程式的過程。學到 `main()` 函數的任務，以及有關 C++ 程式所使用的敘述種類。利用已定義的 `cout` 和 `cin` 物件，來處理程式的輸出和輸入，而且也會學到如何產生和使用變數。最後，您會看到函數的介紹，這是 C++ 程式設計的模組。
- **第 3 章資料處理**：C++ 提供內建的型態儲存兩種資料：整數（無小數部分的數字）和浮點數字（有小數部分的數字）。為了符合各種程式設計師的需求，C++ 對於每一類提供數種型態。這章討論這些型態，包括產生變數和撰寫各種型態的常數。您也會學習 C++ 如何處理隱含式（implicit）和明確（explicit）的型態轉換。
- **第 4 章複合型態**：C++ 使您可以從基本的內建型態建構出更精巧的型態。最高階的形式是類別，在第 9 到第 13 章有做深入的探討。本章也討論其他的形式，包括陣列，這儲存單一型態的數個值；結構，可儲存數個不相同型態的值；以及指標，這標示記憶體的位置。您也會學到如何產生和儲存文字字串，以及使用 C- 格式字元陣列和 C++ 的 `string` 類別處理文字輸入和輸出。最後，會學到 C++ 處理記憶體配置的一些方法，包括明確管理記憶體的 `new` 和 `delete` 運算子。
- **第 5 章迴圈和關係運算式**：程式常常必須執行反覆的動作，C++ 提供 3 種迴圈結構處理這種問題：`for` 迴圈、`while` 迴圈和 `do while` 迴圈。迴圈必須知道何時該結束，而 C++ 的關係運算子使您可以產生測試導引這種迴圈。在本章您也會學到如何產生迴圈，來逐一讀入並處理輸入字元。最後，學習如何產生二維陣列，以及如何用巢狀迴圈處理之。
- **第 6 章分支敘述與邏輯運算子**：若程式可以根據環境而調整其行為，則程式可以表現得很聰明。在這章，您會學到如何用 `if`、`if else` 和 `switch` 敘述，以及條件運算子來控制程式的流程。也會學到如何用邏輯運算子來幫助表達做決策的檢查。而且，會看到 `cctype` 函式庫的函數，可以計算字元的關係，如檢查字元是否為數字或是不可印出的字元。最後，還可以瞭解到檔案輸入與輸出的基本觀念。

碁峰 www.gotop.com.tw

- **第 7 章函數：C++ 的程式設計模組**：函數是 C++ 程式設計的基本建構區塊。這章重點在於 C++ 函數和 C 函數共有的特性。尤其是複習函數定義的一般格式，以及看到函數原型如何增加程式的可靠性。還有，探討如何撰寫函數處理陣列、字元字串，以及結構。接著，進入遞迴領域，這是會呼叫自己的函數，並瞭解它如何用來實作劃分解題的方法。最後是函數指標，使您可以使用函數引數，來告訴函數使用第二個函數。
- **第 8 章函數探索**：這章研究 C++ 函數的新增特性。您會學到內嵌函數，這可以加速程式的執行，但是會增加程式的大小。也會學到 reference 變數，這提供傳遞資訊至函數的另一種方法。函數呼叫時若省略引數，則預設引數可以自動提供函數引數之值。函數多載使您可以產生同名，但有不同簽名的函數。所有特性都經常用在類別的設計中。而且，您將會學到函數樣版，使您可以指定相關函數的設計。
- **第 9 章記憶體模式和名稱空間**：這章討論多個檔案的程式。它檢視配置記憶體的作法，管理記憶體的不同成員函數和範疇、連結性，以及名稱空間，這決定程式的哪些部分可以知道變數。
- **第 10 章物件及類別**：類別是使用者自訂的型態，而物件是類別的實體，如變數。這章介紹物件導向程式設計，以及類別設計。類別宣告是描述儲存在類別物件中的資訊，以及類別物件可以執行的操作。物件有一部分是可見於外界世界（公用部分），而有些是隱藏的（私有部分）。當物件產生和結束時，特定的類別成員函數（建構函數和解構函數）就會產生作用。在本章您會學到這些以及類別的其他細節，並且瞭解類別可以用來實作抽象資料型態（ADT），如堆疊。
- **第 11 章運用類別**：在本章您會更深入瞭解類別。首先，將學到運算子多載，這使您可以定義運算子如 +，來處理類別物件。還會學到夥伴函數，這可以存取外界無法存取的類別資料。同時瞭解如何用特定的建構函數，和多載的運算子成員函數，來管理類別型態的轉換。
- **第 12 章類別和動態記憶體配置**：使類別成員指向動態配置的記憶體經常是有用的。若您在類別建構函數中使用 new 配置動態記憶體，就要負責提供適當的解構函數，定義明確的複製建構函數，以及定義明確的指定運算子。這章告訴您，若沒有提供明確的定義，將如何產生隱含式的成員函數並討論其行為。您也可以利用物件指標，以及研究佇列模擬問題，來擴展您的類別經驗。

碁峰 www.gotop.com.tw

- **第 13 章類別繼承**：物件導向程式設計最有威力的特性之一是繼承，衍生類別可以繼承基礎類別的特性，使您可以重複利用基礎類別的程式碼。本章討論公用繼承，用來建構 is-a 的關係，意思是衍生物件是基礎物件的特例。例如，物理學家是科學家的一個特例。有些繼承關係是同名異式，意思是您可以撰寫程式碼使用混合的相關類別，則相同名稱的成員函數，其行為會視物件型態而定。實作這種行為必須使用新的成員函數，稱為虛擬函數。有時抽象基礎類別是繼承關係的最好方法。這章討論這些事，指出何時公用繼承是合適的，何時不是。
- **第 14 章 C++ 的程式碼再利用**：公用繼承只是重複利用程式碼的一種方法。這章檢視其他數種方法。包含表示一個類別其成員是另一類別的物件。它可以用來建構 has-a 的關係，此時一個類別具有另一個類別的元件。例如，汽車有引擎。也可以用私有和保護繼承來建構此種關係。這章告訴您如何建構並指出這些方法的不同處。還會學到類別樣版，它是用一些未指定的泛型定義類別，然後以特定型態使用樣版產生特殊類別。例如，堆疊樣版可以產生整數堆疊，或是字串堆疊。最後，談到多重繼承，這是可以從一個以上的類別衍生類別。
- **第 15 章夥伴、異常和其他**：這章延伸夥伴的討論，包含了夥伴類別和夥伴成員函數。然後它呈現數種 C++ 的新發展，從異常開始，這提供處理不尋常之程式事故的機制，如不正確的函數引數值，或是耗盡記憶體。之後您會學到 RTTI（執行期型態），這是標示物件型態的機制。最後，您會學到較安全的方法執行無限制的型態轉換。
- **第 16 章 string 類別和標準樣版函式庫**：這章討論一些最近加入的有用類別函式庫。`string` 類別是除了傳統的 C- 格式字串之外，另一個便利且好用的方法。`auto_ptr` 類別有助於管理動態配置的記憶體。STL（Standard Template Library）提供數種泛型的收納器，包括 array、queue、list、set 和 map 的樣版表示法。它也提供泛型演算法的有效率函式庫，可以處理 STL 收納器，也可以處理一般的陣列。`valarray` 樣版類別對數字型陣列提供了支援。

碁峰 www.gotop.com.tw

- **第 17 章輸入、輸出和檔案**：這章複習 C++ I/O 並討論如何格式化輸出。將會學到如何使用類別成員函數，來判斷輸入或輸出串流的狀態，以及檢查如輸入是否有型態不匹配，或是達到檔案終點。C++ 利用繼承衍生類別管理檔案的輸入和輸出。也會學到如何開啟檔案做輸入和輸出，如何附加資料至檔案，如何使用二進位檔案，以及如何隨機存取檔案。最後，您會學到如何應用標準的 I/O 成員函數從字串讀入資料及寫入字串。
- **第 18 章檢視新的 C++ 標準**：這章開始複習在前面章節所提及的 C++11 的特性。包括新的型態，均勻的初始化語法，自動型態推論（deduction），新的聰明指標及範圍列舉。接下來，討論有關新的 rvalue reference 型態，以及它如何實作於一稱為**移動語意**（move semantics）的新特性。同時也論及新的類別特性，拉姆達（lambda）表示式，以及可變的（variadic）類版。最後，本章也談及前面章節未列出的新特性。
- **附錄 A 基數**：此附錄討論八進位、十六進位和二進位數字。
- **附錄 BC++ 保留字**：此附錄列出 C++ 的關鍵字。
- **附錄 CASCII 字元集**：此附錄列出 ASCII 字元集及十進位、八進位、十六進位，以及二進位表示式。
- **附錄 D 運算子優先權**：此附錄以遞減優先權的順序列出 C++ 運算子。
- **附錄 E 其他運算子**：此附錄總結本文主體未涵蓋的 C++ 運算子，如位元運算子。
- **附錄 Fstring 樣版類別**：此附錄總結 `string` 類別成員函數和函數。
- **附錄 GSTL 成員函數和函數**：此附錄總結 STL 收納器成員函數和一般的 STL 演算法函數。
- **附錄 H 參考書單與網際網路的資源**：此附錄列出一些可以使您更深入瞭解 C++ 的書籍與相關資源。
- **附錄 I 轉換成 ANSI/ISO 標準的 C++**：此附錄提供從 C 和較舊之 C++ 實作轉換至 ANSI/ISO C++ 的指南。
- **附錄 J 問題回顧解答**：此附錄包含每章結束時提出之問題回顧的答案。

碁峰 www.gotop.com.tw

授課老師注意事項

這一版本的 C++ Primer Plus 目標之一，是能夠提供一本可以當作自我教育的書籍或是教科書。下面是使用 C++ Primer Plus 第六版作為教科書所提供的一些特性：

- 這本書描述泛型的 C++，所以它可以獨立於特定的實作平台。
- 這內容記錄 ISO/ANSI C++ 標準委員會的運作足跡並引入樣版，STL、string 類別、異常、RTTI，以及名稱空間的討論。
- 不需要有 C 的背景知識，所以在不瞭解 C 的情況下也可以使用它（但是有些程式設計的背景知識是必要的）。
- 主題的安排是為了使前面的各章，可以快速地涵蓋複習 C 的必要知識。
- 各章有問題回顧和程式設計練習。附錄 J 提供了問題回顧的解答。
- 這本書介紹幾個適於電腦科學課程的主題，包括抽象資料型態（ADT）、堆疊、佇列、簡單串列、模擬、泛型程式設計，以及使用遞迴完成劃分解題法。
- 大部分的章節都可以在一個星期或更短的時間內完成。
- 這本書討論**何時**使用某些的特性，以及**如何**使用它們。例如，它將公用繼承和 is-a 的關係連結在一起，組合和私有繼承與 has-a 的關係連結在一起，而且它討論何時使用虛擬函數，以及何時不要使用。

本書的使用慣例

在本書裡，使用了一些方式來表達文字上不同的意義：

- 有關程式碼、註解、敘述、變數、檔案名稱與程式輸出，會以如下的電腦程式字體表示：

```
#include <iostream>
int main()
{
    using namespace std;
    cout << "What's up, Doc!\n";
    return 0;
}
```

碁峰 www.gotop.com.tw

- 對於程式輸入則會以粗體的電腦程式字體表示：

```
Please enter your name:
Plato
```

本書另外還會使用數種方式來表達特別的重點：

方塊文章

方塊文章將提供更深入的討論與相關背景知識來幫助您瞭解主題內容。

對於特別的程式設計情況，提供簡短且有幫助的指引。

用以提醒潛在陷阱出現的情況。

對於不屬於任何範疇的註解，以這種不限定範疇之區塊說明加以表示。

開發本書範例程式所使用的系統

本書有關 C++11 的範例程式，是使用 Microsoft Visual C++ 2010 及 GNU g++ 4.5.0 的 Cygwin。這些系統皆在 64 位元的 Windows 7 下執行的。其餘的範例程式，除了上述的系統外，也可在 OS X 10.6.8 作業系統的 iMac，使用 g++ 4.2.1，或是在 Ubuntu Linux 作業系統，使用 g++ 4.4.1。大部分 C++11 之前的範例程式可在 Windows XP 作業系統下，使用 Microsoft Visual C++ 2003 及 Metrowerks CodeWarrior Development Studio 9，並利用 Borland C++ 5.5 命令列編譯程式，以及 GNU gpp 3.3.3 加以檢查。此外，也可以在 SuSE 9.0 Linux 作業系統下使用 Comeau 4.3.3 和 GNU g++ 3.3.1，以及在 OS 10.3 的 Macintosh G4 電腦上使用 Metrowerks Development Studio 9。

C++ 提供了相當多資源給程式設計師；請好好學習並希望您能夠樂在其中！

碁峰 www.gotop.com.tw

Chapter 09

記憶體模式和名稱空間

本章將學習以下的主題：

- 獨立編譯
- 儲存期間、範疇和連結性
- 定位放置 new
- 名稱空間

將資料儲存在記憶體中，C++ 提供許多選擇。您可以選擇資料要在記憶體中停留多久（儲存期間），或是選擇程式的哪些部分可以存取資料（範疇和連結性）。我們可以使用 new 動態地配置記憶體空間，而定位放置 new（placement new）更可以在其中提供使用上的變化。C++ 的名稱空間功能提供存取的控制權。較大的程式一般由數個原始碼檔案組成，這些會共享某些資料。這種程式需要獨立編譯程式檔案，所以本章從此開始。

獨立編譯

C++ 與 C 一樣，允許您甚至鼓勵您將程式的函數元件置於獨立的檔案中。在第 1 章中，您可以獨立編譯檔案，然後將它們連結成最終的執行檔（一般而言，C++ 編譯程式會編譯程式而且也管理連結程式）。若您只修改一個檔案，則可以只重新編譯此檔案，然後與其他先前已編譯好的檔案連結即可。這方法使得大型程式較容易管理。而且，大部分的 C++ 環境均提供額外的功能幫助程式管理。例如，UNIX 和 Linux 系統提供 make 程式；它記錄程式與哪些檔案相關，以及這些檔案的最後修改時間。若執行 make 而且它檢查自最後一次編譯後您已經修改過一個或多個原

碁峰 www.gotop.com.tw

始檔案，則 make 會執行正確的步驟重建程式。Embarcadero C++ Builder、Microsoft Visual C++、Apple Xcode 和 Freescale CodeWarrior 的 IDE（整合發展環境）以 Project 選單提供類似的功能。

我們來看簡單的範例。不看編譯細節，因為這與編譯程式的實作有關。著重於更一般的觀念，如設計。

例如，假設您決定將範例程式 7.12 的函數置於不同的檔案。範例程式 7.12 是將直角座標轉換為極座標，並顯示結果。這不只是將原始檔案從 main() 之後分成二個檔案。問題是 main() 和其他兩個函數都用到相同的結構宣告，所以需要將宣告放在每個檔案中。若逐一打字輸入，則容易產生錯誤，即使正確的複製結構，若您之後要做修改，您還要記得修改這兩個檔案。簡言之，將程式分佈在多個檔案中會產生新的問題。

有誰會希望製造更多的問題？C 和 C++ 的發展者當然不希望，所以他們提供 #include 功能來處理這種情況。不是將結構宣告置於每個檔案，而是置於標頭檔，然後在每個原始碼檔案中匯入此標頭檔。因此若您修改結構宣告，只須修改標頭檔。此外也可以將函數原型置於標頭檔，如此，原始程式就可以分成三部分：

- 包含結構宣告和使用這些結構之函數原型的標頭檔。
- 包含與結構相關之函數程式碼的原始碼檔案。
- 包含呼叫這些結構相關之函數程式碼的原始碼檔案。

對於組織程式這是有用的方法。例如，若您撰寫另一個程式使用這些相同的函數，只要匯入標頭檔，再將此函數檔案加入專案或是 make 清單中。而且這組織也表現了 OOP（物件導向程式設計）的方法。一個標頭檔包含使用者自訂的型態；第二個檔案包含處理使用者自訂型態的函數程式碼。二者合成一組軟體套件，可適用許多程式。

不要將函數定義或是變數宣告置於標頭檔。這也許只是簡單的指定，但是通常會導致問題。例如，若您將函數定義置於標頭檔，然後在同一程式的兩個其他檔案中匯入此標頭檔，則您最後在此程式中會有同一函數的兩份定義，這是錯誤的，除非這函數是 inline。下面是標頭檔中常見的內容：

- 函數原型
- 用 #define 或是 const 定義的符號常數
- 結構宣告
- 類別宣告

碁峰 www.gotop.com.tw

- 樣版宣告
- 內嵌函數（inline function）

將結構宣告置於標頭檔是沒問題的，因為它們不會產生變數，當您在原始碼檔案中宣告結構變數時，它們會告訴編譯程式如何產生。同樣的，樣版宣告不是要編譯的程式碼，它們是指令告訴編譯程式如何產生函數定義符合程式中的函數呼叫。宣告為 const 的資料和 inline 函數有特殊的連結性，所以可以放在標頭檔而不會產生問題。

範例程式 9.1、9.2 和 9.3 展示範例程式 7.12 分成不同檔案後的結果。注意，在匯入標頭檔時，我們使用 "coordin.h" 而非 <coordin.h>。若檔案名稱以角括號包住，C++ 編譯程式會檢查含有標準標頭檔的檔案系統。但是若檔案包在雙引號中，則編譯程式先檢查目前的工作目錄或是原始檔的目錄（或是類似的選擇，視編譯程式而定）。若在此處找不到標頭檔，就會檢查標準位置。所以在匯入您自己的標頭檔時，使用雙引號，不是角括號。

圖 9.1 描述在 Unix 系統上編譯此程式的步驟。注意您只要輸入 CC 的編譯指令，其他步驟會自動完成。g++指令列編譯程式和 Borland C++ 指令列編譯程式（bcc32.exe）都是這種作法。Apple Xcode、Embarcadero C++ Builder 和 Microsoft Visual C++ 基本上都使用相同的步驟，但是如第 1 章所述，起始的步驟略有不同，使用可以產生專案及連結相關原始檔的選單。在專案中，您只要加入原始碼檔案而不是標頭檔。這是因為 #include 指令會管理標頭檔，而且不要使用 #include 匯入原始碼檔案，因為這會導致多重宣告。

注意事項

在整合性發展環境中，不要將標頭檔加入專案清單中，而且在原始碼檔案中不要使用 #include 匯入其他原始碼檔案。

範例程式 9.1 coordin.h

```
// coordin.h -- structure templates and function prototypes
// structure templates
#ifndef COORDIN_H_
#define COORDIN_H_

struct polar
{
    double distance;    // distance from origin
    double angle;        // direction from origin
};
```

碁峰 www.gotop.com.tw

```
struct rect
{
    double x;       // horizontal distance from origin
    double y;       // vertical distance from origin
};

// prototypes
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);

#endif
```

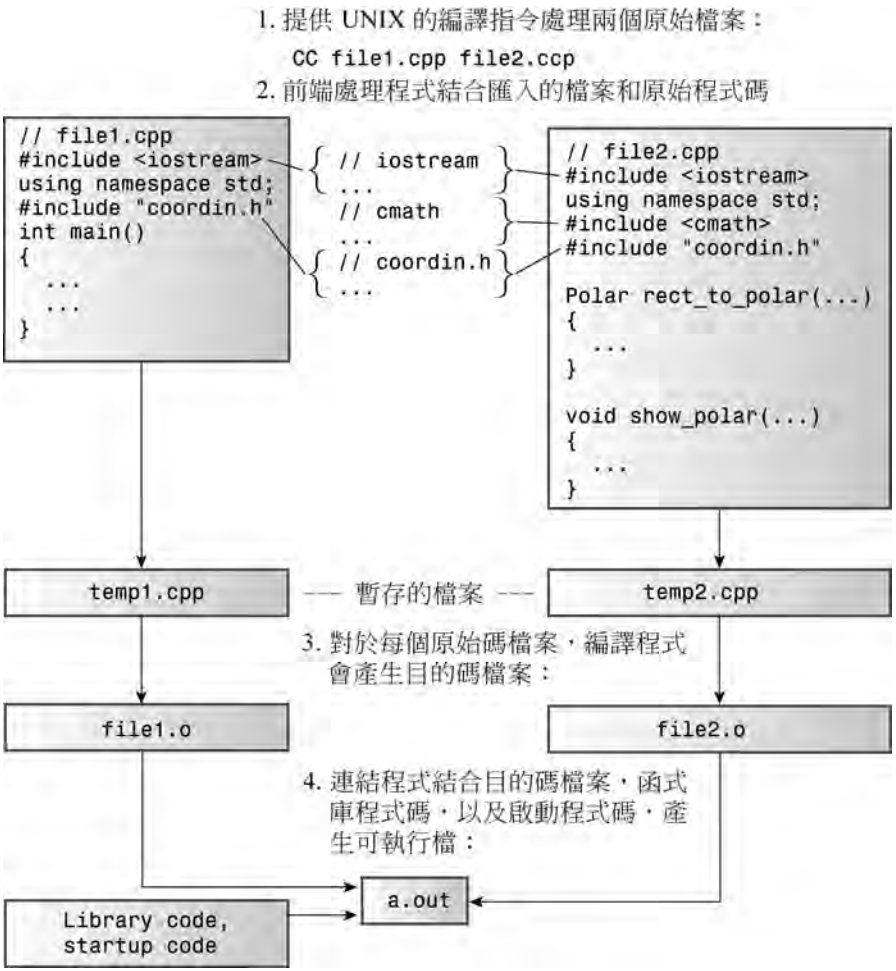

圖 9.1 在 Unix 系統上，編譯有多個檔案的 C++ 程式

碁峰 www.gotop.com.tw

標頭檔管理

在一個檔案中，您只能匯入標頭檔一次。這似乎是很容易記住的事情，但是您可能匯入一個標頭檔數次而不自知。例如，您也許用了一個匯入另一個標頭檔的標頭檔。有一個標準的 C/C++ 技術可以避免多重匯入標頭檔。這是根據前端處理指令 #ifndef（取自 if not defined）。一段程式碼如下：

```
#ifndef COORDIN_H_
...
#endif
```

意思是若名稱 COORDIN_H_ 之前未被前處理指令 #define 定義過，就處理 #ifndef 和 #endif 之間的敘述。

通常您用 #define 敘述產生符號常數，如下面的作法：

```
#define MAXIMUM 4096
```

但是 #define 只用在名稱上就可以使此名稱為定義過，如下面的敘述：

```
#define COORDIN_H_
```

範例程式 9.1 所用的技就是將檔案內容包在 #ifndef 中：

```
#ifndef COORDIN_H_
#define COORDIN_H_
// place include file contents here
#endif
```

編譯程式第一次遇到這個檔案時，名稱 COORDIN_H_ 應該未定義（這名稱是根據 include 的檔案名稱，再加上一些底線字元，使此名稱不可能在其他地方產生）。所以編譯程式會檢查 #ifndef 和 #endif 之間的內容，這就是我們希望的。在檢查內容的過程中，編譯程式讀到定義 COORDIN_H_ 的敘述。若在同一檔案中，之後它又遇到第二次匯入 coordin.h，編譯程式會注意到 COORDIN_H_ 已經定義過，並略過至 #endif 後的敘述。注意這方法不會避免編譯程式匯入檔案兩次，除了第一次匯入外，這只是使編譯程式忽略所有的內容。大部分的標準 C 和 C++ 標頭檔都用此方法。

範例程式 9.2 file1.cpp

```
// file1.cpp -- example of a three-file program
#include <iostream>
#include "coordin.h" // structure templates, function prototypes
using namespace std;
int main()
```

碁峰 www.gotop.com.tw

```
{
    rect rplace;
    polar pplace;

    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y)  // slick use of cin
    {
        pplace = rect_to_polar(rplace);
        show_polar(pplace);
        cout << "Next two numbers (q to quit): ";
    }
    cout << "Bye!\n";
    return 0;
}
```

範例程式 9.3 **file2.cpp**

```
// file2.cpp -- contains functions called in file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h" // structure templates, function prototypes

// convert rectangular to polar coordinates
polar rect_to_polar(rect xypos)
{
    using namespace std;
    polar answer;

    answer.distance =
        sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = std::atan2(xypos.y, xypos.x);
    return answer;      // returns a polar structure
}

// show polar coordinates, converting angle to degrees
void show_polar (polar dapos)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
```

碁峰 www.gotop.com.tw

使用新的標頭檔案並且編譯及連結這兩個原始檔案，將會產生一個新的可執行程式。以下是範例程式的執行結果：

```
Enter the x and y values: 120 80
distance = 144.222, angle = 33.6901 degrees
Next two numbers (q to quit): 120 50
distance = 130, angle = 22.6199 degrees
Next two numbers (q to quit): q
```

我們已經討論過獨立編譯檔案，C++ 標準語言使用**翻譯單元**（translation unit）名稱取代**檔案**（file），以保留較大的一般性；檔案不是組織電腦資訊的唯一可能方式。為了簡單起見，本書將使用檔案這個名稱，但也可以使用翻譯單元。

多重函式庫連結

C++ Standard 允許每個編譯程式設計者可以自由地實作他認為合適的名稱裝飾或是編碼（參考第 8 章有關名稱裝飾的真實世界註記），所以您應該知道以不同編譯程式產生的二進位模組（目的碼檔案）很有可能不能正確地連結。也就是說，對於同一個函數，兩個編譯程式會產生不同的裝飾名稱。

名稱的差異會使連結程式不能將一個編譯程式產生的函數呼叫，與第二個編譯程式產生的函數定義相互匹配。當要嘗試連結編譯過的模組時，要確定每個目的檔案或是函式庫是用同一個編譯程式產生。若您有原始碼，通常只要用您的編譯程式重新編譯就可以解決連結錯誤。

儲存期間、範疇和連結性

您已經見過有多個檔案的程式，所以可以進一步討論第 4 章有關儲存類別的記憶體架構，這會影響資訊如何跨檔案共用。因為第 4 章已經看過一陣子了，所以我們來複習它討論有關記憶體的內容。對於儲存資料，C++ 使用三種不同的方法，每種方法會使資料儲存在記憶體的時間不相同。

- **自動儲存期間**（automatic storage duration）— 宣告在函數定義中的變數（包括函數參數）—具有自動儲存期間。當程式執行進入定義這些變數的函數或區塊時會產生它們，而且當執行離開函數或區塊時就會釋放它們使用的記憶體。C++ 有兩種自動儲存期間的變數。
- **靜態儲存期間**（static storage duration）— 定義在函數定義之外或其他用關鍵字 `static` 定義的變數具有靜態儲存期間。在程式執行的整個期間，它們都會存在。C++ 有三種靜態儲存期間的變數。
- **執行緒儲存期**（thread storage duration）（C++11）— 近來多核心處理器已很常見。這些 CPUs 可同時處理數個執行工作。也就讓程式將運算分割成數個

碁峰 www.gotop.com.tw

單獨可同時進行的執行緒（threads），使用 `thread_local` 關鍵字宣告的變數，只要其包含的執行緒還在，便可持續保有儲存區。本書並沒有涵蓋平行程式設計（concurrent programming）的內容。

- **動態儲存期間**（dynamic storage duration）— 用 `new` 運算子配置的記憶體會一直存在，直到以 `delete` 運算子或是程式結束，視何者先發生而定，就會釋放之。這記憶體具有動態儲存期間，有時稱為**自由空間或自由儲存區**（free store）或稱**堆積**（Heap）。

接下來，我們將詳細說明不同型態的變數，其範疇中（scope），或是可見（visible）（可為程式使用）為何，以及有關連結性（linkage）的細節，這會決定檔案之間資訊的共享。

範疇和連結性

範疇（scope）是描述在檔案（編譯單元）中名稱的可見範圍。例如，定義在函數中的變數可以用在函數中，但是另一個函數就不能使用，但是若變數定義在檔案中，則在此變數定義之後的所有函數均可使用之。**連結性**（linkage）描述一個名稱在不同單元之間如何共用。具有**外部連結性**（external linkage）的名稱可以跨檔案共用，而具有**內部連結性**（internal linkage）名稱只能在單一檔案內函數之間共用。自動變數的名稱不具連結性，因為它們不能共用。

C++ 變數會有一種範疇。具有**區域範疇**（local scope）（也稱為**區塊範疇**）的變數只有定義它的區塊知道它的存在。**區塊**（block）是一串以大括號包住的敘述。例如，函數主體是是一個區塊，但是可以使其他區塊巢狀於函數主體內。具有**全域範疇**（global scope）（也稱為**檔案範疇**）的變數在其定義之後的所有檔案內容都會知道它。自動變數有區域範疇，靜態變數可能為這兩種範疇的任何一種，視其定義方式而定。用在函數原型範疇中的名稱，只有包住引數串列的括號內知道它的存在（這就是為何它們不是那麼重要，或是否存在的原因）。宣告在類別中的成員具有**類別範疇**（class scope）（請參閱第 10 章：「物件與類別」）。宣告在名稱空間中的變數具有**名稱空間範疇**（namespace scope）（現在這語言已經加入名稱空間，全域範疇變成名稱空間範疇的特例）。

C++ 函數可以是類別範疇或是名稱空間範疇，包括全域範疇，但是不能是區域範疇（因為函數不可以定義在區塊中，若函數是區域範疇，只有它自己知道自己，因此，其他函數就無法呼叫它，這種函數就不能稱為函數了）。

各種的 C++ 儲存選項都有儲存期間、範疇和連結性等性質。我們以這些性質來看 C++ 的儲存類別。我們先從未加入名稱空間之前的情況開始，再看名稱空間如何修改這局面。

碁峰 www.gotop.com.tw

自動儲存期間

函數參數和宣告在函數中的變數預設都為自動儲存期間。它們屬區域範疇，而且無連結性。也就是說，若您在 main() 中宣告變數 texas，在另外一個函數 oil() 中宣告同樣名稱的變數，則會產生兩個獨立的變數，只有定義它們的函數知道它們。您對 oil() 的 texas 所做的任何事情不會影響 main() 的 texas，反之亦然。而且每個變數的配置是在程式執行進入包含此定義的最內層區塊才會產生，當其函數結束，每個變數就會消失（注意當執行進入區塊時才會配置變數，但是範疇從宣告處開始就已經存在）。

若您在區塊中定義變數，變數的存在期間和範疇都限制在區塊中。例如，若您在 main() 的開始處定義一個變數 teledeli，而在 main() 中開始新的區塊並在此區塊中定義新變數 websight。之後，在區塊的內部和外部都可以看到 teledeli，而 websight 只存在區塊內部，進入範疇是從定義之處開始，直到程式執行通過區塊尾端。

```
int main()
{
    int teledeli = 5;
    {                           // websight allocated
        cout << "Hello\n";
        int websight = -2;      // websight scope begins
        cout << websight << ' ' << teledeli << endl;
    }                           // websight expires
    cout << teledeli << endl;
    ...
}
```

但是如果將內部區塊的變數名稱從 websight 改成 teledeli，這樣就有兩個相同名稱的變數，一個在區塊之外，一個在區塊之內，此時會如何？當程式執行區塊內的敘述時，程式會將 teledeli 名稱解釋為區域的區塊變數。我們稱此新定義會**遮蔽**（hide）先前的定義。新定義是在**範疇中**（in scope），而舊定義**暫時離開範疇**（out of scope）。當程式離開區塊，則原始的定義又會回到範疇中（請參閱圖 9.2）。

碁峰 www.gotop.com.tw

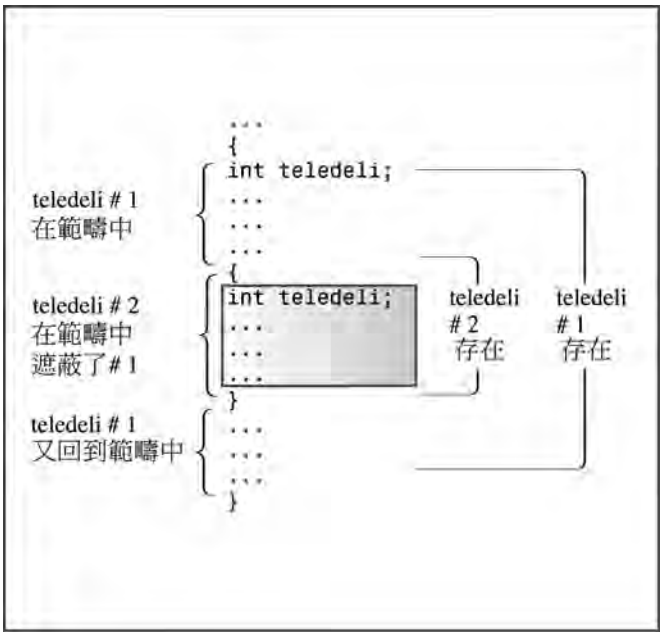

圖 9.2 區塊和範疇

範例程式 9.4 說明自動變數如何被區域化於包含它們的函數或區塊。

範例程式 9.4 autoscp.cpp

```
// autoscp.cpp -- illustrating scope of automatic variables
#include <iostream>
void oil(int x);
int main()
{
    using namespace std;

    int texas = 31;
    int year = 2011;
    cout << "In main(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In main(), year = " << year << ", &year = ";
    cout << &year << endl;
    oil(texas);
    cout << "In main(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In main(), year = " << year << ", &year = ";
    cout << &year << endl;
    return 0;
}
```

碁峰 www.gotop.com.tw

```
void oil(int x)
{
    using namespace std;
    int texas = 5;

    cout << "In oil(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In oil(), x = " << x << ", &x = ";
    cout << &x << endl;
    {                                // start a block
        int texas = 113;
        cout << "In block, texas = " << texas;
        cout << ", &texas = " << &texas << endl;
            cout << "In block, x = " << x << ", &x = ";
        cout << &x << endl;
    }                                // end ablock
    cout << "Post-block texas = " << texas;
    cout << ", &texas = " << &texas << endl;
    cout << ", &texas = " << &texas << endl;
}
```

範例程式 9.4 的執行結果如下：

```
In main(), texas = 31, &texas = 0012FED4
In main(), year = 2011, &year = 0012FEC8
In oil(), texas = 5, &texas = 0012FDE4
In oil(), x = 31, &x = 0012FDF4
In block, texas = 113, &texas = 0012FDD8
In block, x = 31, &x = 0012FDF4
Post-block texas = 5, &texas = 0012FDE4
In main(), texas = 31, &texas = 0012FED4
In main(), year = 2011, &year = 0012FEC8
```

注意，範例程式 9.4 中的三個 texas 變數的每一個都有自己不同的位址，在某一時點程式只使用在範疇中的特定變數，所以將 oil() 內層區塊中的 texas 設成 113，並不會影響其他同名的變數（同樣地，實際記憶體位址值，以及記憶體位址格式會因不同的系統而有不同表示方式）。

我們對上述動作做個總結。當 main() 開始時，程式配置 texas 和 year 的空間。而且這些變數進入範疇中。當程式呼叫 oil()，這些變數仍在記憶體中，但是不在範疇內。此時會配置兩個新變數，x 和 texas 兩個變數，並進入範疇中。當程式執行到 oil() 的內部區塊時，新的 texas 離開範疇，而被較新的定義取代。但是變數 x 仍在範疇中，因為區塊沒有定義新的 x。當執行離開此區塊，就會釋放最新的 texas，而第 2 個 texas 再回到範疇中。當 oil() 函數結束，x 和 texas 消失，最初的 texas 和 year 重新回到範疇中。

碁峰 www.gotop.com.tw

C++11 對 auto 的修改

在 C++11 裡，關鍵字 auto 用來處理自動型態推導，如同第 3、7、8 章所說明。但在 C 語言，以及較早期的 C++版本裡，卻有著完全不同的意思。其用來顯式識別帶有自動儲存的變數：

```
int froob(int n)
{
    auto float ford;  // ford has automatic storage
    ...
}
```

在 C++11 裡，這種用法已不復存在。準備標準內容的成員，並不願意提出新的關鍵字，因為這樣做的話，可能會讓將關鍵字用於其他用途上的程式碼失效。在這種情形下，大家覺得 auto 的舊式使用方法其實不常用到，因此最好直接對關鍵字做重新定義，而不須提出新的關鍵字。

自動變數的初始化

您可以用任何運算式初始化自動變數，當遇到此宣告時，這些運算式的值都要已知。以下的變數 x、y、big 及 z 會做初始化的動作：

```
int w;          // value of w is indeterminate
int x = 5;      // initialized with a constant expression
int big = INT_MAX – 1; // initialized with a constant expression
int y = 2 * x;  // use previously determined value of x
cin >> w;
int z = 3 * w;  // use new value of w
```

自動變數與堆疊

接下來我們進一步瞭解 C++ 編譯程式如何處理自動變數。因為自動變數的數目會隨著函數的開始和結束而增減，所以程式在執行時需要管理自動變數。通常的方法是撥出一塊記憶體，將它當做堆疊來管理這些來來去去的變數。它稱為堆疊是因為新資料會象徵性的堆在舊資料之上（也就是說，在相鄰的位置，而不是相同的位置），當程式結束後變數也從堆疊上移除。堆疊的預設大小依編譯程式而定，但是編譯程式通常會提供改變大小的功能。程式用兩個指標記錄堆疊，一個指標指向堆疊基部，這是配置給堆疊的記憶體的開始處，另一指標指向堆疊頂端，這是下一個可用的記憶體位置。當函數被呼叫時，其自動變數會加入堆疊中，更新指向堆疊頂端的指標指向這些變數之後的下一個可用的未用空間。當函數執行完畢，頂端指標之值會恢復到函數未呼叫前的值。

碁峰 www.gotop.com.tw

堆疊的設計是 LIFO（後進先出），意思是最後放入堆疊的變數最先移出。這種設計簡化了引數的傳遞。函數呼叫將引數值放入堆疊頂端，更新頂端指標。被呼叫的函數用其形式參數的描述決定每個引數的位址。例如，圖 9.3 顯示呼叫 fib() 函數時，傳遞 2-位元組的 int 和 4-位元組的 long。這些值會在堆疊上。當 fib() 開始執行時，會將兩個值關聯至名稱 real 和 tell。當 fib() 結束，頂端指標會重新設回之前的位置。新值不會被清除，但是再也沒有名稱，而這些空間會在下一次程序中用來儲存置於堆疊上的值（圖 9.3 事實上有些簡化，因為函數呼叫時還會傳遞其他的資訊，如回傳位址）。

暫存器變數

C 最初提出暫存器（register）關鍵字，是用來告知編譯器使用 CPU 暫存器來儲存自動變數：

```
register int count_fast;  // request for a register variable
```

目的是為了更快速地對變數做存取。

在 C++11 之前，C++用相同的方式使用關鍵字，除了硬體與編譯器以較複雜的方式開發，背後的含意為變數頻繁被使用時，編譯器或許可提供特殊的方式來處理。到了 C++11，即使這個含意為舊有概念，留下的 register 關鍵字用來顯式辨別自動變數。既然 register 只能用於自動變數，使用這個關鍵字的理由之一，便是用來強調使用的變數為自動變數，而這些變數可能與外部變數名稱相同。這跟關鍵字 auto 原本的使用目的一樣。然而，保留 register 關鍵字更重要的原因，在於避免使用該關鍵字的程式碼失效。

靜態期間變數

與 C 一樣，C++ 提供靜態儲存期間的變數，這些變數有 3 種連結性：外部連結性（檔案之間可存取）、內部連結性（同一檔案函數之間的存取）和沒有連結性（在一函數或函數內區段的存取）。這三種都會與程式共存亡，它們比自動變數有較長的生命。因為靜態變數的數目不會隨著程式執行而變動，所以不需要一個像堆疊的特殊裝置來管理它們。相反的，編譯程式會配置固定大小的記憶體區塊儲存這些靜態變數，只要程式在執行，這些變數就會存在。此外，若您未明確地初始化靜態變數，則編譯程式會將它們設成 0。靜態陣列和結構預設上會使每個元素或是每個成員的每個位元都設為 0。

碁峰 www.gotop.com.tw

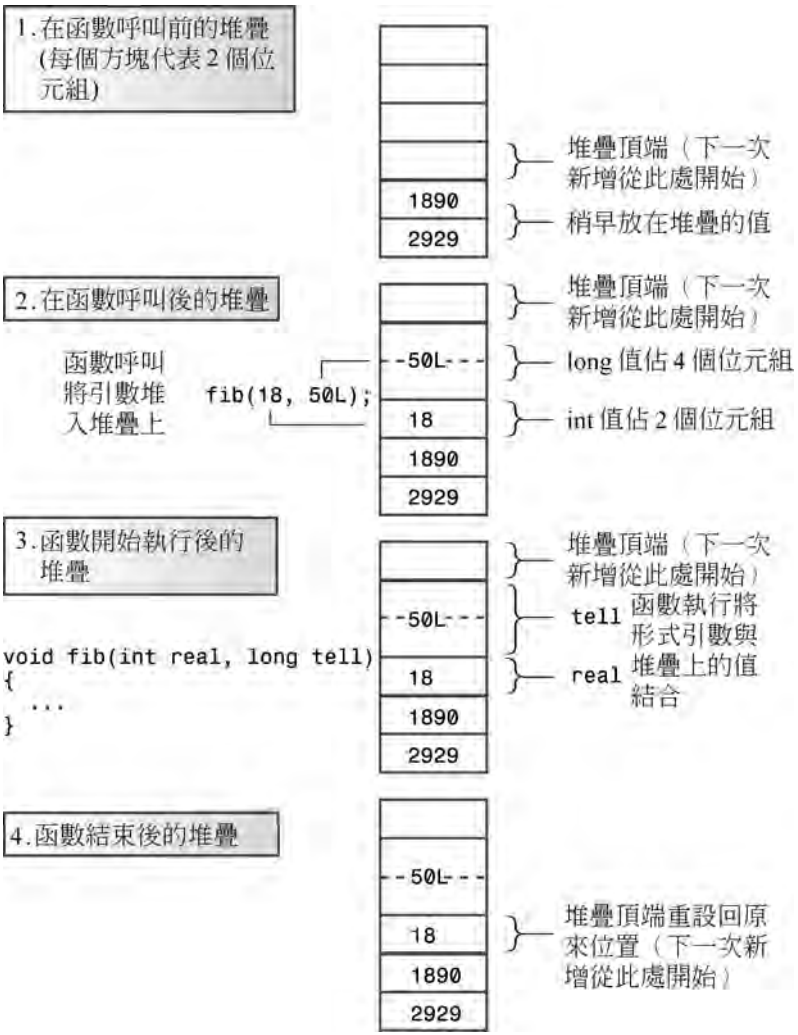

圖 9.3 以堆疊傳遞引數

早期的 K&RC 並不能初始化自動陣列和結構，但可以初始化靜態的陣列和結構。而 ANSI C 和 C++ 則可以初始化兩種。但有些早期使用 C 編譯程式的 C++ 轉譯程式並不完全遵守 ANSI C，若您用的是這種工具，也許您需要使用這三種靜態儲存類別中的一種，以初始化陣列和結構。

碁峰 www.gotop.com.tw

我們來看如何產生這三種不同的靜態期間變數，之後再看它們的性質。要產生具有外部連結性的靜態期間變數，將它宣告在任何區塊之外。要產生具有內部連結性的靜態期間變數，將它宣告在任何區塊之外並用 static 儲存類別修飾字。要產生沒有連結性的靜態期間變數，將它宣告在區塊之內並用 static 修飾字。下面的程式碼顯示這三種變化：

```
...
int global = 1000;           // static duration, external linkage
static int one_file = 50;    // static duration, internal linkage
int main()
{
...
}
void funct1(int n)
{
    static int count = 0;  // static duration, no linkage
    int llama = 0;
...
}
void funct2(int q)
{
...
}
```

如之前所述，所有的靜態期間變數（global、one_file 和 cout）都會一直存在，從程式開始執行至結束。變數 count 宣告在 funct1() 內具有區域範疇，而且無連結性，這意思是只能在 funct1() 函數中使用，就像自動變數 llama。但是與 llama 不一樣的是 count 在 funct1() 不執行時仍留在記憶體中。global 和 one_file 都有檔案範疇，意思是從它們的宣告點之後，至檔案結束前都可以使用它們。尤其是兩個都可以用在 main()、funct1()和 funct2() 中。因為 one_file 有內部連結性，它只能用在包含它的檔案中。而 global 具有外部連結性，所以也可以用在此程式的其他檔案中。

所有的靜態期間變數都有下列初始化特性：未初始化的靜態變數會將其所有位元都設為 0。此變數稱為 **0 的初始化**（zero-initialized）。

表 9.1 總結在名稱空間之前的時代這些儲存類別的特性。接著我們會更仔細研究靜態期間的各種變化。

注意表 9.1 中，關鍵字 static 有兩個地方具有不同的意義。當他用於區域內宣告時，表示靜態變數沒有連結性，static 表明了變數的儲存期間。當他用於區塊外面時，static 表示為內部連結性，此變數已有靜態的儲存期間。這稱為**關鍵字多載**（keyword overloading），由內文可得知。

碁峰 www.gotop.com.tw

表 9.1 5 種變數的儲存類別

儲存說明	期間	範疇	連結性	如何宣告
自動	自動	區塊	無	宣告在區塊中（可以隨意的使用關鍵字 auto）
暫存器	自動	區塊	無	用關鍵字 register 宣告在區塊中
無連結性的靜態	靜態	區塊	無	用關鍵字 static 宣告在區塊中
具外部連結性的靜態	靜態	檔案	外部	宣告在所有函數之外
具內部連結性的靜態	靜態	檔案	內部	用關鍵字 static 宣告在所有函數之外

靜態變數初始化

靜態變數能以 0 做初始化，可能經由**常數表達式初始化**（constant expression initialization），也可能經由**動態初始化**（dynamic initialization）。如您所推測，以 0 做初始化代表將變數設定為數值零。對於純量型態來說，零會被轉換成適當的型態。比方說，在 C++ 程式碼裡，以 0 代表 null 指標，系統內部可能不是以零表示，也就是說，系統內部裡，指標變數會被初始化成內部所表示的數值。結構成員也能以零做始化，任何填充位元皆被設為零。

以零做初始化，以及常數表達式初始化共同被稱作**靜態初始化**（static initialization）。這代表在編譯器處理檔案（或翻譯單元）時，變數便被初始化。動態初始化則代表變數之後才會被賦予值。

那要怎麼判定程式裡是使用何種初始化形式？首先，無論哪種初始化形式，所有的靜態變數都是以零做初始化。接著，如果變數使用常數表達式初始化，而編譯器可單獨從檔案內容（包括匯入的標頭檔）裡解析該表達式，那麼便可以執行常數表達式初始化。編譯器會在需要的時候預備執行簡單的運算。如果在當下並沒有足夠的資訊，變數便會以動態的方式做初始化。請看以下敘述：

```
#include <cmath>
int x;                              // zero-initialization
int y = 5;                          // constant-expression initialization
long z = 13 * 13;                   // constant-expression initialization
const double pi = 4.0 * atan(1.0);  // dynamic initialization
```

首先，x、y、z，以及 pi 皆先以 0 做初始化。接著，編譯器會解析常數運算式，再將 y 與 z 分別初始化為 5 及 169。但 pi 的初始化牽涉到 atan()函數的呼叫，而這個呼叫必須等到該函數被連結，程式執行的時候才能進行。

碁峰 www.gotop.com.tw