序

Preface

無數的程式設計師使用前幾版的 *C++ Primer* 學會了 C++。在那段時間中，C++ 成熟了很多：它聚焦的重點，也就是其程式設計師社群所注重的，已經從**機器**（*machine*）效率拓展到投注了更多的心力在**程式設計師**（*programmer*）的效率上。

2011 年 C++ 標準委員會（C++ standards committee）發行了 ISO C++ 標準的重大改版，這次改版後的標準是 C++ 演化最新的一步，並繼續強調程式設計師效率這個重點。新標準的主要目標是：

- 讓此語言更統一，更容易教授與學習
- 讓標準程式庫（standard libraries）的使用更容易、安全，並且更有效率
- 讓撰寫抽象層（abstractions）和程式庫（libraries）的工作更簡單一些

在這一版中，我們全面地修訂了 *C++ Primer*，使用最新的標準。只要看看前面的新功能目錄（特別列出了涵蓋新功能的章節），你就可以大略了解這個新的標準是如何廣泛影響 C++ 的各個面向。

新標準所增添的某些功能，例如用於型別推論（type inference）的 `auto`，是無處不在的。這些新機能讓此版中的程式碼更容易閱讀和理解。程式（以及程式設計師！）可以忽略型別的細節，讓人更容易專注於程式原本的意圖。其他的新功能，例如智慧指標（smart pointers）和可移動的容器（move-enabled containers），能讓我們寫出更精密的類別，而且不用去處理錯綜複雜的資源管理議題。結果就是，我們能夠以比本書的第四版（Fourth Edition）簡單得多的方式教你如何撰寫自己的類別。我們，還有你，不再需要去擔心之前標準中妨礙我們學習的諸多細節。

我們有用圖示在書頁邊緣標出了涵蓋新標準所定義之功能的章節。我們希望已經熟悉 C++ 核心的讀者會覺得這些提示有用，可以幫助他們知道要把注意力放在哪裡。我們也希望這些圖示有助於解釋可能還沒每個新功能都支援的編譯器（compilers）所產生的錯誤訊息。雖然本書中幾乎所有的範例都曾在當前版本的 GNU 編譯器底下成功編譯過，我們也理解某些讀者有可能還無法取用這些最新的編譯器。儘管最新的標準加入了很多功能，核心語言仍然沒變，並構成了本書大部分的篇幅。讀者可以使用這些圖示來提醒自己哪些功能在他們的編譯器上可能尚無法使用。

C++ 11

碁峰 www.gotop.com.tw

為何要撰寫此書？

現代 C++ 主要由三個部分構成：

- 低階的語言（low-level language），其中很多都是從 C 繼承而來
- 能讓我們定義自己的型別（types）並組織大規模程式與系統的進階語言功能
- 標準程式庫（standard library），它們使用了上述的進階功能來提供實用的資料結構與演算法

大部分的書籍都是依照其演進的順序來介紹 C++ 的。它們先教授 C++ 的 C 子集，然後在書末提及 C++ 較為抽象的功能，作為進階主題來講解。這種做法有兩個問題：讀者可能會深陷低階程式設計中固有的繁複細節，而沮喪放棄。能夠堅持下去的人學會的卻是之後必須學著改掉的壞習慣。

我們採取相反的做法：從一開始我們就使用可以讓程式設計師忽略低階程式細節的那些功能。舉例來說，介紹內建的算術（arithmetic）和陣列（array）型別時，我們就一起介紹了程式庫的 `string` 與 `vector` 型別。使用這些程式庫型別的程式更容易撰寫、更容易理解，也較不容易出錯多了。

很常看到的情況是，程式庫被當作「進階」主題來教授。許多書籍使用基於字元陣列指標和動態記憶體管理的低階程式技巧，而不使用程式庫。要讓那些使用低階技巧的程式正確運作，比使用程式庫撰寫相應的 C++ 程式碼還要難得多。

在 *C++ Primer* 中，我們從頭到尾強調的都是良好的程式寫作風格：我們希望幫助你，也就是讀者，從一開始就培養出好習慣，才不用在學習更精深的知識時還得學著改掉壞習慣。我們標示出了特別棘手的部分，並對常見的誤解和陷阱做出警告。

我們也會說明那些規則之所以存在的原因，我們會解釋為什麼（why），而非只是有什麼（what）。我們相信，學習事情的運作原理能幫助讀者更快鞏固對這個語言的掌握。

儘管你不必先懂 C 就能理解本書，我們還是假設你對程式設計有一定的了解，知道如何使用至少一種現代的區塊結構語言（modern block-structured language）來撰寫、編譯並執行程式。特別是，我們假設你有用過變數、寫過並呼叫過函式，也用過編譯器。

第五版的改變

這個新版的 *C++ Primer* 在書頁邊緣標有圖示，幫忙引導讀者。C++ 是一個大型語言，提供了專為各種特殊程式設計問題量身訂做的功能。其中某些功能對經手大型專案的團隊大有助益，但對較小的案子而言或許並非必要。結論就是，並不是每一個程式設計師都必須懂得每個功能的每個細節。我們標注在書頁邊緣的圖示能幫助讀者找出哪些部分可以之後再學，而哪些主題是較為重要的基礎。

碁峰 www.gotop.com.tw

我們將涵蓋此語言基本要素的章節用一個人在讀書的圖案標示。以這種方式標注的章節所涵蓋的主題構成了這個語言的核心部分。每個人都應該閱讀並理解這些章節。

我們也標示了涵蓋進階或特殊用途主題的那些章節。初次閱讀時，這種章節可以跳過或快速瀏覽就好。我們以一疊書的圖案來標示它們，代表此時你可以安全地略過它們。先大略讀過這種章節，了解有哪些功能存在，或許會是個好主意。然而，除非你實際需要在你自己的程式中用到它們，不然沒必要花時間研讀那些主題。

為了進一步導引讀者的注意力，我們用放大鏡的圖案標注出了特別棘手的概念。我們希望讀者會花多一點時間透徹的理解有此標示的章節。在這些章節中，有部分主題一開始會讓你搞不清楚為何要先了解它們，但我們認為你最終會發現，這些章節所涵蓋的主題對於理解此語言來說，是不可或缺的。

另一個幫助你閱讀本書的輔助工具，是廣泛使用的交互參考（cross-references）。我們希望這些參考能幫助讀者更輕易找到他們需要的地方，並且能夠輕鬆地跳回前面，參照後面範例所仰賴的內容。

依舊不變的是，*C++ Primer* 仍然是 C++ 有條理、正確且詳盡的入門指南。我們提供一系列難度遞增的巧妙範例來教導此語言，解說語言功能並展示如何以最好的方式使用 C++。

本書的結構

我們首先在第一篇和第二篇中一起介紹了此語言與程式庫的基礎。這些部分所涵蓋的內容足以讓你，也就是讀者，寫出有用的程式。幾乎所有的 C++ 程式設計都得好好了解在本書這些部分所涵蓋的知識。

除了教導 C++ 的基礎知識，第一篇和第二篇中的內容還有另外一個重要的用途：藉由使用程式庫所定義的抽象機能，你將會變得更善於使用高階程式技巧。通常，程式庫所提供的機能本身就是以 C++ 撰寫的抽象資料型別（abstract data types）。你也能用任何 C++ 程式設計都能取用的同一種類別建構功能來定義程式庫。我們從教授 C++ 的經驗學到的是，先用過設計良好的抽象型別，讀者會比較容易理解如何建置他們自己的型別。

只有在徹底掌握程式庫的使用方法之後，也就是懂得如何寫出程式庫讓你有辦法撰寫的那種抽象程式後，我們才會繼續介紹讓你能夠撰寫你自己的抽象層之 C++ 功能。第三篇和第四篇的焦點是如何以類別（classes）的形式寫出抽象層（abstractions）。第三篇涵蓋基礎知識，而第四篇則包含特殊用途的功能。

在第三篇中，我們涵蓋了拷貝控制（copy control）的議題，以及助你製作跟內建型別（built-in types）一樣好用的類別之其他技巧。類別是物件導向（object-oriented）和泛型（generic）程式設計的基石，也會涵蓋在第三篇中。*C++ Primer* 在第四篇收尾，其中涵蓋的功能最適合用來組織大型且複雜的系統。我們也在附錄 A 中總結了程式庫所提供的演算法。

碁峰 www.gotop.com.tw

1

快速入門

Getting S tarted

本章目錄

1.1 撰寫一個簡單的 C++ 程式 2
1.2 初步認識輸入 / 輸出 5
1.3 關於註解 9
1.4 流程控制 11
1.5 類別簡介 19
1.6 Bookstore 程式 24
本章總結 26
定義的詞彙 26

本章介紹 C++ 主要的基本元素：型別（types）、變數（variables）、運算式（expressions）、述句（statements）與函式（functions）。過程中，我們會簡短解說如何編譯並執行程式。

讀過本章並做過習題後，你應該就能撰寫、編譯及執行簡單的程式。後續章節假設你懂得使用本章所介紹的功能，並且會更詳細說明那些功能。

碁峰 www.gotop.com.tw

學習新程式語言的**唯一方法**是撰寫程式。在本章中，我們會編寫一個程式來幫一家書店（bookstore）解決一個簡單的問題。

這家書店有一個儲存交易明細的檔案，每個交易都是單一本書賣出多少本的販售記錄。每筆交易含有三個資料元素：

```
0-201-70353-X 4 24.99
```

第一個元素是 ISBN（International Standard Book Number，國際標準書號，即書籍的唯一識別號碼），第二個元素是賣出的本數，最後則是每一本的售出單價。書店老闆不時會查看這個檔案，並計算各本書賣出多少本、該本書所帶來的總收入，以及平均銷售價格。

為了撰寫這個程式，我們得涵蓋 C++ 的幾個基本功能。此外，我們得知道如何編譯和執行程式。

雖然我們尚未設計此程式，不過很容易就可以看出它必須

- 定義變數
- 進行輸入與輸出
- 使用一種資料結構來保存資料
- 測試兩筆記錄是否有相同的 ISBN
- 包含一個迴圈（loop）來處理交易檔案中的每一筆記錄

我們會先看看如何用 C++ 解決這些子問題，然後再寫出我們的書店程式。

1.1 撰寫一個簡單的 C++ 程式

每個 C++ 程式都包含一或多個**函式**（*functions*），其中一個必須被命名為 **`main`**。作業系統（operating system）會呼叫 `main` 來執行 C++ 程式。這裡有一個簡單的 `main`，除了回傳一個值給作業系統外，它什麼都不做：

```
int main()
{
    return 0;
}
```

一個函式定義（function definition）有四個元素：一個**回傳型別**（*return type*）、一個**函式名稱**（*function name*）、一個包在括弧（parentheses）中（可能為空）的**參數列**（*parameter list*），以及一個**函式主體**（*function body*）。儘管 `main` 有其特殊之處，我們定義它的方式與其他任何函式都一樣。

在這個例子中，`main` 有一個空的參數列（以其中沒有任何東西的 `()` 表示）。§6.2.5 會討論我們可以為 `main` 定義的其他參數型別。

碁峰 www.gotop.com.tw

`main` 函式必須有 `int` 的回傳型別，它是代表整數（integers）的一個型別。型別 `int` 是一個**內建型別（built-in type）**，這表示它是該語言所定義的型別之一。

函式定義最後的一個部分，也就是函式主體，是由述句所構成的一個區塊（*a block of statements*），以一個左**大括號**（open **curly brace**）開頭，並以一個右大括號結尾：

```
{
    return 0;
}
```

這個區塊中唯一的述句是一個 `return`，它是終止函式的一個述句。在此，`return` 也會送回一個值（value）給該函式的呼叫者（caller）。當 `return` 述句包含一個值，所回傳的這個值之型別必須與函式的回傳型別相容。在此，`main` 的回傳型別為 `int`，而回傳值是 `0`，也就是一個 `int`。

請注意到 `return` 述句最後的分號（semicolon）。分號標示 C++ 中大部分述句的結尾。它們很容易被忽略，若被忘記，就可能導致神秘難解的編譯器錯誤訊息。

在多數系統上，從 `main` 回傳的值是一個狀態指示器（status indicator）。回傳值 `0` 表示成功。非零的回傳值之意義則是由系統所定義。一般來說，非零回傳值指示發生的錯誤種類。

關鍵概念：型別

型別（types）是程式設計中最基本的概念之一，也是在本書中會一再出現的一個概念。一個型別所定義的是一個資料元素的內容，以及可以在那些資料上進行的運算。

我們的程式所操作的資料儲存在變數中，而每個變數都有一個型別。當名為 `v` 的變數之型別為 `T`，我們通常會說「`v` 有型別 `T`」或可以互換使用的「`v` 是一個 `T`」。

1.1.1 編譯並執行我們的程式

寫好程式之後，我們就得編譯（compile）它。編譯程式的方式取決於你的作業系統和編譯器（compiler）。要了解特定的編譯器如何運作，請查閱參考手冊或詢問知情的同事。

PC 上的許多編譯器都是從 IDE（integrated development environment，整合式開發環境）中執行的，它們內含編譯器與建置（build）用和分析用的工具。這些環境是開發大型程式時的絕佳資產，但要學會如何有效運用，得花不少時間。學習如何使用這種環境已遠超出本書涵蓋範圍。

大部分的編譯器，包括 IDE 內附的，都提供命令列介面（command-line interface）。除非你已經會用 IDE，否則你會發現從命令列介面開始著手會比較容易。這麼做可以讓你先專注在學習 C++ 上。此外，只要了解語言部分，IDE 對你來說就很可能更容易學習。

碁峰 www.gotop.com.tw

程式原始碼檔案命名慣例

無論你用的是命令列介面或 IDE，大多數的編譯器都預期你把程式原始碼（source code）儲存在一或多個檔案中。這些程式檔案通常被稱為**源碼檔**（*source files*）。在多數系統上，源碼檔名稱會以一個後綴（suffix）結尾，這會是一個句號（period）後面接著一或多個字元（characters）。這後綴告訴系統該檔案是一個 C++ 程式。不同的編譯器使用不同的後綴慣例，最常見的包括 `.cc`、`.cxx`、`.cpp`、`.cp` 與 `.C`。

從命令列執行編譯器

如果我們用的是命令列介面，我們通常會在一個主控台視窗（console window，例如 UNIX 系統上的 shell 視窗或 Windows 上的命令提示字元視窗）中編譯程式。假設我們的 `main` 程式位於一個名為 `prog1.cc` 的檔案中，我們可以用像這樣的命令來編譯它

```
$ CC prog1.cc
```

其中 `CC` 指的是編譯器，而 `$` 是系統提示字元（system prompt）。編譯器會產生一個可執行檔（executable file）。在 Windows 系統上，這個執行檔的名稱會是 `prog1.exe`。UNIX 編譯器通常會把可執行檔命名為 `a.out`。

要在 Windows 上執行一個可執行檔，我們只需提供執行檔名稱，而且可以省略 `.exe` 延伸檔名（file extension）：

```
$ prog1
```

在某些系統上，你必須明確指出該檔案的位置，即使檔案位於目前目錄或資料夾中也一樣。在這種情況中，我們會輸入：

```
$ .\prog1
```

後面接著一個反斜線（backslash）的點字元 `.` 表示該檔案位於目前的目錄中。

要在 UNIX 上執行一個可執行檔，我們使用檔案全名，包括延伸檔名：

```
$ a.out
```

如果需要指定檔案位置，我們會使用後面接著斜線（forward slash）的點字元 `.` 來表示我們的可執行檔位於目前的目錄中：

```
$ ./a.out
```

從 `main` 回傳的值可依隨著系統而變的方式取用。在 UNIX 及 Windows 系統上，執行程式後，你必須發出適當的 `echo` 命令。

在 UNIX 系統上，我們以下列命令取得狀態：

```
$ echo $?
```

要在 Windows 系統上檢視狀態，就使用：

```
$ echo %ERRORLEVEL%
```

碁峰 www.gotop.com.tw

執行 GNU 或 Microsoft 編譯器

執行 C++ 編譯器的命令隨著編譯器和作業系統而變。最常見的編譯器是 GNU 編譯器和 Microsoft Visual Studio 編譯器。預設情況下，執行 GNU 編譯器的命令是 `g++`：

```
$ g++ -o prog1 prog1.cc
```

這裡的 `$` 是系統提示字元。`-o prog1` 是編譯器的一個引數（argument），指出要放可執行檔的檔案名稱。這道命令會產生一個叫做 `prog1` 或 `prog1.exe` 的可執行檔，視作業系統而定。在 UNIX 上，可執行檔沒有後綴，而在 Windows 上，後綴是 `.exe`。如果省略 `-o prog1`，編譯器會在 UNIX 系統上產生一個名為 `a.out` 的可執行檔，Windows 下則是一個 `a.exe` 檔案。（注意：取決於你所用的 GNU 編譯器版本，你可能得指定 `-std=c++0x` 以開啟對 C++ 11 的支援。）

執行 Microsoft Visual Studio 2010 編譯器的命令是 `cl`：

```
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

在此 `C:\Users\me\Programs>` 是系統提示列，而 `\Users\me\Programs` 是目前目錄的名稱（即目前資料夾）。`cl` 命令會調用編譯器，而 `/EHsc` 是編譯器選項（compiler option），用以開啟標準例外處理（standard exception handling）。Microsoft 編譯器會自動產生一個名稱對應於第一個源碼檔名稱的可執行檔。這個可執行檔具有後綴 `.exe`，而且名稱與源碼檔檔名相同。在此例中，可執行檔的名稱為 `prog1.exe`。

編譯器通常會有一些選項可用來產生警告，點出有問題的地方。使用這些選項通常會是好主意。我們偏好使用 GNU 編譯器的 `-Wall` 和 Microsoft 編譯器的 `/W4`。

進一步的資訊，請參閱你編譯器的使用者指南。

習題章節 1.1.1

練習 1.1：查閱你編譯器的說明文件，找出它所使用的檔案命名慣例。編譯並執行前面的 `main` 程式。

練習 1.2：把程式改成回傳 `-1`。回傳值 `-1` 經常被用來表示程式執行失敗。重新編譯並再次執行你的程式，看看你的系統如何處理來自 `main` 的這種錯誤失敗指示器。

1.2 初步認識輸入 / 輸出

C++ 語言並沒有定義用來進行輸入或輸出（IO）的任何述句。取而代之，C++ 包含了一個擴充性的**標準程式庫（standard library）**來提供 IO（以及其他許多的）機能。就很多用途而言，包括本書中的範例，你只需要認識 IO 程式庫的幾個基本概念和運算就行了。

本書中的多數範例都使用 `iostream` 程式庫。**`iostream`** 程式庫的基礎是名為 **`istream`** 和 **`ostream`** 的兩個型別，分別代表輸入和輸出資料流。一個資料流（stream）是從一個 IO 裝置讀出或寫入的一個字元序列（a sequence of characters）。資料流（*stream*）這個名稱主要是用來指出那些字元是隨著時間循序產生或消耗的。

碁峰 www.gotop.com.tw

標準輸入與輸出物件

此程式庫定義了四個 IO 物件。要處理輸入，我們使用型別為 `istream` 名稱是 **`cin`**（唸做 *see-in*）的一個物件。這個物件也被稱為**標準輸入（standard input）**。對於輸出，我們使用名為 **`cout`**（唸做 *see-out*）的一個 `ostream` 物件。這個物件也被稱為**標準輸出**（*standard output*）。這個程式庫也定義了其他兩個 `ostream` 物件，叫做 **`cerr`** 和 **`clog`**（分別唸做 *see-err* 和 *see-log*）。我們通常使用被稱為**標準錯誤（standard error）**的 `cerr` 來發出警告和錯誤訊息，而 `clog` 用來記錄程式執行過程中的一般資訊。

通常，系統會將每個這種物件關聯至程式在其中執行的視窗。所以，當我們從 `cin` 讀取，資料就是從正在執行該程式的視窗讀出，而寫入 `cout`、`cerr` 或 `clog` 的時候，輸出就會被寫到相同的視窗中。

用到 IO 程式庫的一個程式

在我們的書店問題中，我們有多筆想用來計算總和的記錄。舉個較為簡單且相關的問題，讓我們先看一下如何將兩個數字加起來。使用 IO 程式庫，我們可以擴充我們的 `main` 程式，讓它提示使用者提供兩個數字，然後印出它們的總和：

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

這個程式啟動時會先把

```
Enter two numbers:
```

印到使用者的螢幕上，然後等候使用者的輸入。如果使用者輸入了

```
3 7
```

後面接著一個 newline（新行字元），然後程式產生下列輸出：

```
The sum of 3 and 7 is 10
```

我們程式的第一行是

```
#include <iostream>
```

告訴編譯器我們想要使用 `iostream` 程式庫。角括號（angle brackets）內的名稱（在此為 `iostream`）指的是一個**標頭（header）**。用到某個程式庫機能的每個程式都必須引入（include）它關聯的標頭。`#include` 指示詞必須寫在單一行上，也就是說，標頭的名稱和

碁峰 www.gotop.com.tw

`#include` 都必須出現在同一行上。一般來說，`#include` 指示詞必須放在任何函式之外。通常，我們會將一個程式所有的 `#include` 指示詞都放在原始碼檔案的開頭。

寫入資料流

`main` 主體中的第一個述句執行了一個**運算式（expression）**。在 C++ 中，一個運算式會產出一個結果，並且是由一或更多個運算元（operands）及（通常是）一個運算子（operator）所構成。這個述句中的運算式使用輸出運算子（`<<` **運算子**）在標準輸出上印出一個訊息：

```
std::cout << "Enter two numbers:" << std::endl;
```

`<<` 運算子接受兩個運算元：左手邊的運算元必須是一個 `ostream` 物件，而右手邊的運算元則是要印出的一個值。此運算子會把給定的值印在給定的 `ostream` 上。輸出運算子的結果是其左手邊的運算元，也就是說，結果是我們在其上寫入給定值的 `ostream`。

我們的輸出述句用了 `<<` 運算子兩次。因為此運算子會回傳其左邊的運算元，第一個運算子的結果就變成第二個運算子的左運算元。結論就是，我們能將輸出請求鏈串起來。因此，我們的運算式等同於

```
(std::cout << "Enter two numbers:") << std::endl;
```

串鏈中的每個運算子都有與其左運算元相同的物件，在此例中即為 `std::cout`。又或者是，我們能夠使用兩個述句來產生相同的輸出：

```
std::cout << "Enter two numbers:";
std::cout << std::endl;
```

第一個輸出運算子印出了一個訊息給使用者，該訊息為一個**字串字面值（string literal）**，也就是包在雙引號（double quotation marks）中的字元序列。兩個引號之間的文字會被印到標準輸出。

第二個運算子印出 `endl`，它是一個特殊值，叫做一個 **manipulator（操作符）**。寫入 `endl` 有結束目前文字行並且排清（flush）關聯裝置*緩衝區*（*buffer*）的效果。排清緩衝區的動作確保程式到目前為止產生的所有輸出都實際被寫入到輸出資料流，而非待在記憶體中等候被寫入。

WARNING

程式設計師經常在除錯（debugging）過程中加入列印述句（print statements）。這種述句應該*總是*排清資料流。否則的話，如果程式當掉，輸出可能會留在緩衝區中，導致推論出來的程式故障點不正確。

使用來自標準程式庫的名稱

細心的讀者會注意到此程式使用了 `std::cout` 與 `std::endl` 而非單純的 `cout` 與 `endl`。`std::` 這個前綴（prefix）指出名稱 `cout` 和 `endl` 定義在名為 **`std`** 的**命名空間（namespace）**

碁峰 www.gotop.com.tw

中。命名空間能讓我們避免我們所定義的名稱和程式庫中相同的名稱不慎產生衝突。標準程式庫定義的所有名稱都在 `std` 命名空間中。

程式庫使用命名空間的一個副作用是，當我們使用來自程式庫的名稱，必須明確指出我們想要使用來自 `std` 命名空間的名稱。使用範疇運算子（scope operator，即 **`::` 運算子**）寫成 `std::cout` 來表達我們想要使用定義於命名空間 `std` 中的名稱 `cout`。§3.1 會展示取用程式庫名稱更簡單的方式。

讀取自一個資料流

向使用者請求輸入之後，接下來我們想要讀取那個輸入。我們先定義名為 `v1` 和 `v2` 的兩個變數（*variables*）來存放輸入：

```
int v1 = 0, v2 = 0;
```

我們將這些變數的型別定義為 `int`，這是代表整數的一個內建型別。我們也會將它們*初始化*（*initialize*）為 `0`。當我們初始化一個變數，就是在創建該變數的同時，賦予了它所指示的值。

下一個述句

```
std::cin >> v1 >> v2;
```

讀取輸入。輸入運算子（**`>>` 運算子**）的行為與輸出運算子類似。它接受一個 `istream` 作為其左運算元，以及一個物件作為其右運算元。它從給定的 `istream` 讀取資料，並將所讀到的東西儲存在給定的物件中。就像輸出運算子，輸入運算子會回傳其左運算元作為結果。因此，這個運算式等同於

```
(std::cin >> v1) >> v2;
```

因為此運算子會回傳其左運算元，我們可以結合一序列輸入請求成為單一個述句。我們的輸入運算從 `std::cin` 讀取兩個值，並將第一個儲存在 `v1`、第二個存在 `v2`。換句話說，我們的輸入運算會像這樣執行

```
std::cin >> v1;
std::cin >> v2;
```

完成此程式

剩下的就是印出我們的結果：

```
std::cout << "The sum of " << v1 << " and " << v2
          << " is " << v1 + v2 << std::endl;
```

這個述句，雖然比向使用者請求輸入的那個還要長，但在概念上是相似的。它在標準輸出上印出它的每一個運算元。這個範例最有趣的地方是，運算元並非全都是同一種值。某些運算元是字串字面值，例如 `"The sum of "`。其他的則是 `int` 值，例如 `v1`、`v2`，以及估算

碁峰 www.gotop.com.tw

（evaluate）算術運算式 `v1 + v2` 得到的結果。程式庫定義了負責處理這每一個不同型別的輸入與輸出運算子版本。

習題章節 1.2

練習 1.3：寫一個程式在標準輸出印出 `Hello, World`。

練習 1.4：我們的程式用到加法運算子（addition operator）`+` 來相加兩個數字。寫一個程式，使用乘法運算子（multiplication operator）`*` 印出乘積（product）。

練習 1.5：我們將輸出寫為一個大型述句。改寫這個程式，使用分開的述句來印出各個運算元。

練習 1.6：解釋下列程式片段是否合法。

```
std::cout   << "The sum of " << v1;
            << " and " << v2;
            << " is " << v1 + v2 << std::endl;
```

如果這個程式是合法的，它會做什麼？如果不合法，那是為什麼呢？你會如何修正它？

1.3 關於註解

在我們的程式變得更複雜之前，我們應該看看 C++ 如何處理註解（*comments*）。註解輔助的是我們程式的讀者。它們通常用來總結演算法的運作、標示變數的用途，或說明可能很難懂的程式片段。編譯器會忽略註解，所以它們對程式的行為或效能沒有影響。

雖然編譯器會忽略註解，但我們程式碼的讀者不會。即使其他部分的系統說明文件已經過時了，程式設計師還是會傾向於相信註解。一個錯誤的註解比完全沒註解還要糟糕，因為它可能誤導讀者。當你更改你的程式碼，請確定你也有更新註解！

C++ 中的註解種類

C++ 中有兩種註解：單行（single-line）和成對（paired）註解。一個單行註解以一個雙斜線（double slash，`//`）開始，以一個 newline 結尾。目前這行在斜線右方的所有東西都會被編譯器忽略。這種註解可以包含任何文字，包括額外的雙斜線。

另一種註解使用兩個分隔符號（`/*` 與 `*/`），這是從 C 繼承而來的。這種註解以一個 `/*` 開頭，並以下一個 `*/` 結尾。這些註解可以含有不是 `*/` 的任何東西，包括 newlines。編譯器會把 `/*` 和 `*/` 之間的所有東西視為註解的一部分。

這樣的成對註解可以放在允許 tab、空格（space）或 newline 的任何地方。成對註解可以跨越一個程式的多個文字行，但不一定要這樣。當一個成對註解有跨越多行，通常比較好的做法是在視覺上凸顯內部文字行，讓讀者知道它們是一個多行註解的一部分。我們的風格是以一個星號（asterisk）作為註解每一行的開頭，指出整個範圍都是屬於同一個多行註解。

碁峰 www.gotop.com.tw

通常，程式中這兩種形式的註解都會有。成對註解一般用於多行的說明，而雙斜線註解常用來作為半行或單行的注記：

```
#include <iostream>
/*
 * 簡單的 main 函式：
 * 讀取兩個數字並寫出它們的總和
 */
int main()
{
    // 提示使用者輸入兩個數字
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0; // 存放我們所讀取的輸入的變數
    std::cin >> v1 >> v2; // 讀取輸入
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

Note

在這本書中，我們用斜體標示註解，以在一般的程式文字中凸顯它們。在實際的程式中，註解文字是否與程式碼文字有不同呈現方式則取決於你所用的程式設計環境。

成對註解不能內嵌

以 /* 開頭的一個註解會以下一個 */ 結尾。結果就是，一個成對註解不能出現在另一個之中。因為這種失誤而導致的編譯器錯誤訊息可能會很神秘且令人困惑。舉個例子，請在你的系統上編譯下列程式：

```
/*
 * comment pairs /* */ cannot nest
 * ''cannot nest'' 會被視為原始碼，
 * 就跟程式的其他部分一樣
 */
int main()
{
    return 0;
}
```

我們常需要在除錯時將一個區塊的程式碼註解掉（comment out）。因為那段程式碼可能含有內嵌的成對註解，所以註解一個程式碼區塊最好的方式就是在我們想要忽略的片段的每一行開頭插入單行註解：

```
// /*
// * 一個單行註解內的所有東西都會被忽略
// * 包括內嵌的成對註解
// */
```

碁峰 www.gotop.com.tw

習題章節 1.3

練習 1.7：編譯錯誤地內嵌了註解的一個程式。

練習 1.8：指出下列輸出述句何者是合法的（如果有的話）：

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
std::cout << /* "*/" /* "/*" */;
```

預測了會發生什麼事之後，編譯含有上述各個述句的一個程式來驗證你的答案。更正你所遇到的任何錯誤。

1.4 流程控制

述句通常都是循序地執行：一個區塊中的第一個述句會先執行，接著是第二個，依此類推。當然，很少有程式（包括解決我們書店問題的那一個）能單以循序執行寫成。取而代之，程式語言提供了各種流程控制述句（flow-of-control statements），讓我們能寫出更為複雜的執行路徑。

1.4.1 while 述句

只要一個給定的條件為 true（真），**while 述句**就會重複執行一段程式碼。我們可以使用一個 while 來寫出加總數字 1 到 10 （包括兩端）的程式，如下：

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // 只要 val 小於或等於 10，就持續執行這個 while
    while (val <= 10) {
        sum += val; // 指定 sum + val 給 sum
        ++val;      // 加 1 到 val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

當我們編譯並執行這個程式，它會印出

```
Sum of 1 to 10 inclusive is 55
```

跟之前一樣，一開始我們先引入 iostream 標頭，並定義 main。在 main 中，我們定義了兩個 int 變數：存放總和的 sum，以及用來代表從 1 到 10 各個值的 val。我們賦予 sum 一個初始值 0，並讓 val 從 1 這個值開始。

碁峰 www.gotop.com.tw

這個程式新出現的部分是那個 while 述句。一個 while 有這樣的形式

```
while (condition)
    statement
```

一個 while 執行的方式是（反覆地）測試 *condition*，並執行關聯的 *statement* 直到 *condition* 變為 false（假）為止。一個**條件（condition）**是產出的結果要不是 true 就是 false 的一個運算式。只要 *condition* 為 true，*statement* 就會執行。執行 *statement* 之後，*condition* 就會再次被測試。如果 *condition* 同樣還是 true，那麼 *statement* 就會再次執行。while 就這樣反覆測試 *condition* 並據此執行 *statement* 直到 *condition* 變為 false 為止。

在這個程式中，while 述句是

```
// 只要 val 小於或等於 10 就重複執行這個 while
while (val <= 10) {
    sum += val; // 指定 sum + val 給 sum
    ++val;      // 加 1 給 val
}
```

其中的條件用到小於或等於運算子（less-than-or-equal operator，即 **<= 運算子**）來比較 val 目前的值和 10。只要 val 小於或等於 10，條件就為 true。如果條件為 true，這個 while 的主體就會被執行。在此，主體是由兩個述句構成的一個區塊：

```
{
    sum += val; // 指定 sum + val 給 sum
    ++val;      // 加 1 給 val
}
```

一個區塊（block）是包在大括號（curly braces）中的零或多個述句所構成的一個序列。一個區塊算是一個述句，可以被用在需要述句的任何地方。這個區塊中第一個述句使用複合指定運算子（compound assignment operator，即 **+= 運算子**）。這個運算子會將其右手邊的運算元加到左手邊的運算元，然後將結果儲存在左運算元。它的效果基本上就跟一個加法運算及一個**指定（assignment）**運算相同：

```
sum = sum + val; // 指定 sum + val 到 sum
```

因此，區塊中的第一個述句會把 val 的值加到 sum 目前的值上，並將結果存回 sum。

下一個述句

```
++val; // 加 1 到 val
```

使用前綴的遞增運算子（prefix increment operator，即 **++ 運算子**）。遞增運算子會加 *1* 到其運算元。寫 ++val 的意思跟 val = val + 1 相同。

執行 while 主體之後，迴圈（loop）會再次估算（evaluate）其條件。如果（遞增過後的）val 值仍然小於或等於 10，那麼 while 的主體就會再次執行。迴圈持續進行，測試條件並執行主體，直到 val 不再小於或等於 10。

碁峰 www.gotop.com.tw

只要 val 變得大於 10，程式就會跳出這個 while 迴圈，並繼續執行接在 while 之後的述句。在此，這個述句會印出我們的輸出，後面再接著 return，結束了我們的 main 程式。

習題章節 1.4.1

練習 1.9：寫一個程式用 while 將 50 到 100 的數字加總。

練習 1.10：除了會加 1 到其運算元的 ++ 運算子，另外還有一個會減 1 的遞減運算子（--）。使用遞減運算子寫一個 while，印出從十遞減到零的數字。

練習 1.11：寫個程式提示使用者輸入兩個整數。印出由這兩個整數所指定的範圍內每一個數字。

1.4.2 for 述句

在我們的 while 迴圈中，我們使用變數 val 來控制要執行迴圈多少次。我們在條件中測試 val 的值，並在 while 的主體中遞增 val。

這種模式，也就是在條件中使用一個變數，並在主體內遞增該變數，因為太常出現了，所以此語言定義了第二種迴圈 **for 述句**，能夠縮減依循此模式的程式碼。我們可以改寫這個程式，改用一個 for 迴圈來加總 1 到 10 的數字，如下：

```
#include <iostream>
int main()
{
    int sum = 0;
    // 加總從 1 到 10 的值，包括兩端
    for (int val = 1; val <= 10; ++val)
        sum += val; // 等同於 sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

跟之前一樣，我們定義 sum 並將之初始化為零。在這個版本中，我們將 val 定義為 for 述句本身的一部分：

```
for (int val = 1; val <= 10; ++val)
    sum += val;
```

每個 for 述句都有兩個部分：一個標頭（header）及一個主體（body）。標頭控制主體執行的次數。標頭本身由三個部分構成：一個 *init-statement*（初始述句）、一個 *condition*（條件）以及一個 *expression*（運算式）。在此，*init-statement*

```
int val = 1;
```

定義名為 val 的一個 int，並賦予它初始值 1。變數 val 只存在於 for 之內，你無法在此迴圈結束後使用 val。*init-statement* 僅執行一次，也就是最初進入 for 的那次。*condition*

```
val <= 10
```

碁峰 www.gotop.com.tw

將目前的值與 10 做比較。*condition* 每次跑迴圈時都會被測試。只要 val 小於或等於 10，for 主體就會被執行。*expression* 會在 for 主體後執行。這裡，*expression*

```
++val
```

使用前綴遞增運算子，它會加 1 到 val 的值。執行 *expression* 之後，for 會重新測試 *condition*。如果新的 val 值仍然小於或等於 10，for 的迴圈主體就會再次執行。主體執行後，val 會再次被遞增。迴圈會持續進行到 *condition* 不成立為止。

在這個迴圈中，for 主體進行加總運算

```
sum += val; // 等同於 sum = sum + val
```

回顧一下，這個 for 整體的執行流程為：

1. 創建 val 並將之初始化為 1。
2. 測試 val 是否小於或等於 10。如果測試通過，就執行 for 主體。如果測試失敗，就退出迴圈，繼續執行接在 for 主體後的第一個述句。
3. 遞增 val。
4. 重複第 2 步中的測試，只要條件為 true，就持續進行剩下的步驟。

習題章節 1.4.2

練習 1.12：下列 for 迴圈會做什麼呢？ sum 的最終值是什麼呢？

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

練習 1.13：使用 for 迴圈改寫 §1.4.1 的頭兩個練習。

練習 1.14：對照比較使用 for 的迴圈與使用 while 的迴圈。它們各有什麼優缺點呢？

練習 1.15：寫一個程式，其中含有後面補充說明「重訪編譯」中所討論的常見錯誤。讓自己熟悉編譯器產生的那些訊息。

1.4.3 讀取未知數量的輸入

在前面的章節中，我們寫了從數字 1 加總到 10 的程式。這個程式的邏輯延伸會是請使用者輸入一組要加總的數字。在這種情況中，我們並不知道要相加的數字有多少個。取而代之，我們會持續讀取數字，直到沒有數字可讀為止：

碁峰 www.gotop.com.tw

```
#include <iostream>
int main()
{
    int sum = 0, value = 0;
    // 持續讀到檔案結尾（end-of-file）為止，以讀取的所有值計算出累計總和
    while (std::cin >> value)
        sum += value; // 等同於 sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

如果我們給此程式這樣的輸入

```
3 4 5 6
```

那麼我們的輸出會是

```
Sum is: 18
```

`main` 內的第一行定義了兩個 `int` 變數，名為 `sum` 和 `value`，我們將它們初始化為 `0`。我們會用 `value` 在讀取輸入的過程中存放每個數字。我們在 `while` 的條件內讀取資料：

```
while (std::cin >> value)
```

估算這個 `while` 條件會執行運算式

```
std::cin >> value
```

這個運算式從標準輸入讀取下一個數字，並將那個數字儲存在 `value` 中。輸入運算子（§1.2）回傳它的左運算元，在此為 `std::cin`。因此，這個條件會測試 `std::cin`。

當我們使用一個 `istream` 作為條件，效果就等於測試該資料流（stream）的狀態。若該資料流是有效的，也就是說，該資料流尚未遭遇錯誤，那麼測試就成功通過。一個 `istream` 會在碰到 *end-of-file*（檔案結尾）或遇到無效輸入（例如讀取一個不是整數的值）時變得無效。處於無效狀態的 `istream` 會導致條件產出 false。

所以，我們的 `while` 會持續執行，直到碰到 end-of-file（或輸入錯誤）為止。這個 `while` 的主體使用複合指定運算子（compound assignment operator）來將目前的值加到累計的和。一旦條件測試失敗，`while` 就會結束。我們會落到下一個述句繼續執行，這會印出後面接著 `endl` 的 `sum`。

從鍵盤輸入 END-OF-FILE

當我們用鍵盤輸入資料給程式，不同的作業系統會使用不同的慣例來讓我們輸入檔案結尾（end-of-file）。在 Windows 系統上，我們使用 control-z（按住 Ctrl 鍵再按 z）並按下 Enter 或 Return 鍵。在 UNIX 系統上，包括 Mac OS X 的機器，通常以 control-d 代表檔案結尾。

碁峰 www.gotop.com.tw

重訪編譯

編譯器的工作之一是找出程式文字中的錯誤。編譯器無法偵測程式是否有依照程式作者的意圖進行，但它可以偵測出程式形式（*form*）上的錯誤。以下是編譯器最常偵測到的錯誤種類。

語法錯誤（*Syntax errors*）：程式設計師犯了 C++ 語言中的文法錯誤（grammatical error）。下列程式展示常見的語法錯誤，每個註解描述了下一行中的錯誤：

```
// error: missing ) in parameter list for main（main 的參數列少了）
int main ( {
    // error: used colon, not a semicolon, after endl（endl 後用了冒號而非一個分號）
    std::cout << "Read each file." << std::endl:
    // error: missing quotes around string literal（字串字面值少了引號包圍）
    std::cout << Update master. << std::endl;
    // error: second output operator is missing（缺少第二個輸出運算子）
    std::cout << "Write new master." std::endl;
    // error: missing ; on return statement（return 述句上少了 ;）
    return 0
}
```

型別錯誤（*Type errors*）：C++ 中的每個資料項目都有一個關聯的型別。舉例來說，10 這個值的型別為 `int`（或者，更通俗的說法：「是一個 `int`」）。`"hello"` 這個詞，包括雙引號，是一個字串字面值（string literal）。型別錯誤的例子之一是傳入一個字串字面值給預期一個 `int` 引數（argument）的函式。

宣告錯誤（*Declaration errors*）：C++ 程式中所用到的每個名稱都必須在使用前先宣告。名稱沒有宣告通常會導致錯誤訊息產生。最常見的兩種宣告錯誤是忘記為來自程式庫的名稱使用 `std::`，以及拼錯了一個識別字（identifier）的名稱：

```
#include <iostream>
int main()
{
    int v1 = 0, v2 = 0;
    std::cin >> v >> v2; // 錯誤：用了 "v" 而非 "v1"
    // 錯誤：cout 沒有定義；應該為 std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

錯誤訊息通常含有行號（line number）以及簡短的描述，說明編譯器認為的出錯原因。以回報的順序來更正錯誤，是良好的實務做法。通常，單一個錯誤可能導致連鎖效應，使得編譯器回報了比實際上更多的錯誤。每次修正後就重新編譯，也是個好主意，或者一次最多只做出少少幾個明顯的更正。這種循環叫作 *edit-compile-debug*（編輯 - 編譯 - 除錯）。

碁峰 www.gotop.com.tw