前言

在一個幾乎每天都會有新的 JavaScript 框架來來去去的世界裡，您為什麼要深入研究像 C 這樣陳舊的、準系統的語言呢？好吧，首先，如果您希望跟上所有這些框架時（哎呀，僅屬個人意見），您可能需要這樣的背景，因為這些技術為許多「現代」語言提供了基礎。您是否正在 TIOBE（*https://oreil.ly/ZTdwN*）之類的網站上尋找流行的程式設計語言並始終發現 C 是位於頂端呢？也許您對令人驚訝的高級視訊卡感興趣，並想了解驅動它們的軟體是如何工作的。或者，也許您正在探索像 Arduino 這樣更新 —— 且更小 —— 的小工具，並聽說 C 是適合這項工作的工具。

不管是什麼原因，您在這裡真是太好了。順便說一句，所有這些理由都是正當的。C 是一種基礎語言，理解它的語法和奇特之處會給您一個非常持久的電腦語言素養，這可以幫助您更輕鬆地學習新的語言和風格。在為裝置驅動程式（device driver）或作業系統編寫低階程式碼時，C（以及它的近親 C++）仍然被廣泛使用。物聯網正在為資源有限的微控制器注入新的活力。而 C 非常適合處理這些微小的環境。

雖然我將聚焦於為小型、有限的機器編寫乾淨、緊湊的程式碼的這個想法，但我仍會從電腦程式設計的基礎知識開始，並涵蓋各種情況下的 C 所適用的各種規則和樣式。

如何使用本書

本書旨在涵蓋上述的任何情況下如何進行良好的 C 程式設計的所有基礎知識。我們將研究 C 語法的控制結構、運算子、函數和其他元素，以及可以把編譯後的程式大小減少一些位元組的替代樣式的範例。我們還會把 Arduino 環境視為可以精實 C 程式碼的出色應用。為了最好地享受 Arduino 部分，您應該具備一些建構簡單電路和使用 LED 和電阻器等組件的基本經驗。

以下是章節預覽：

第 1 章，C 的基礎知識

簡要介紹 C 語言的歷史和設定開發環境的步驟。

第 2 章，儲存和敘述

介紹 C 中的敘述，包括基本 I/O、變數和運算子。

第 3 章，控制流程

在這裡，我將介紹分支和迴圈敘述，並更深入地了解變數及其作用域。

第 4 章，位元和（許多）位元組

快速回顧資料的儲存。我們會向您展示 C 中處理單一位元和在陣列中儲存大量更大的東西的工具。

第 5 章，函數

我會看看如何把您的程式碼分解為可管理的區塊。

第 6 章，指標和參照

更進階一點，我建立了更複雜的資料結構，並學習如何把它們傳遞給函數並從函數傳回它們。

第 7 章，程式庫

了解如何查找和使用可以幫助您完成常見或複雜任務的流行程式碼。

第 8 章，真實世界的 C 與 Arduino

真正的樂趣開始了！我們會設定 Arduino 開發環境並讓一些 LED 閃爍。

第 9 章，較小的系統

使用完整的 Arduino 專案來嘗試幾種電子週邊裝置，包括感測器、按鈕和 LCD 顯示器。

第 10 章，更快的程式碼

學習一些編寫程式碼的技巧，這些技巧專門用來幫助小型處理器充分利用其資源。

第 11 章，客製化程式庫

借助編寫容易，文件說明齊全且與 Arduino 相容之程式庫的提示和技巧來建立您的 C 程式庫技能。

第 12 章，下個下一步

嘗試一個快速的物聯網專案，其中包含一些臨別想法和一些關於下一步的嘗試的想法，因為您將繼續提高您的精實程式設計技能。

附錄包括我使用的一些硬體和軟體的便捷連結，以及有關如何下載和配置本書中所展示的 C 和 Arduino 範例的資訊。

本書編排慣例

本書使用以下印刷慣例：

斜體（*Italic*）

表示新的術語、URL、電子郵件地址、檔名和延伸檔名。（中文使用楷體字。）

定寬（`Constant width`）

用於程式列表，以及在段落中參照的程式元素，例如變數或函數名稱、資料庫、資料型別、環境變數、敘述和關鍵字。

定寬粗體（**`Constant width bold`**）

顯示命令或其他應由使用者輸入的文字。

定寬斜體（***`Constant width italic`***）

顯示應該被使用者提供的值或根據前後文決定的值所取代的文字。

第二章

儲存和敘述

程式設計的本質是對資料的操作。程式設計語言為人類提供了一個介面，用於告訴電腦該資料是什麼，以及您想對該資料做什麼。為了強大的機器所設計的語言可能會隱藏（或推斷）很多關於儲存資料的細節，但 C 在這方面仍然相當簡單。也許用簡單這個詞是錯誤的，但它的資料儲存方法相當直接，但同時仍然允許複雜的操作。正如我們將在第 6 章中看到的，C 還為程式設計師提供了一扇窗戶，可以了解資料在電腦記憶體中的儲存位置。當我們在本書後半部分開始直接使用微控制器時，這種存取將變得更加重要。

不過現在，我想解決 C 語法的一些基礎知識，以便我們可以開始編寫原始程式，而不僅僅是從書中複製程式碼行。這一章有很多這樣的程式碼行，我們非常鼓勵您在閱讀時複製它們！但希望我們能夠讓您為您自己的程式設計挑戰創造出新穎的答案。

如果您已經有另一種語言的經驗而對程式設計感到滿意，請隨意瀏覽本章。您應該閱讀第 34 頁上有關 `printf()` 和 `scanf()` 函數的「printf() 和 scanf()」小節，但其他小節也可能很相似。

C 中的敘述

您會聽到的另一個作為程式設計的基本元素概念是*演算法*（*algorithm*）的概念。演算法是在電腦上處理資料並通常會完成任務的指令集。演算法的一個經典類比是廚房的食譜。給定一組原料，食譜是您把這些原料變成蛋糕之類的東西的個別步驟。在程式設計中，那些「個別步驟」就是敘述（statement）。

在 C 中，敘述有多種形式。在本章中，我將介紹宣告（declaration）敘述、初始化（initialization）敘述、函數呼叫（function call）和註解（comment）。後面的章節會處理控制敘述和不完全像敘述的敘述，例如建立您自己的函數和前置處理器（preprocessor）命令。

敘述分隔字元

敘述之間使用分號來分隔。C 語言中的分號和英語中的句點非常相似。英語中的長句子可能會跨越好幾行，但您知道應該要繼續下去，直到看到句號為止；同樣的，您可能會在一行中把幾個短句子組合在一起，但您可以根據這些句點輕鬆地區分它們。您可能會很容易忘記敘述末尾的分號。如果每個敘述都佔了一行，那麼我們很容易就會假設編譯器「看到」了那種人類可以輕鬆識別的相同結構。不幸的是，編譯器並不能。即使用了第 15 頁「建立 C 的 'Hello, World'」中我們的第一個非常簡單的程式，我們用來在終端機視窗中印出一些文字的那個敘述，也需要以分號結尾。如果您好奇的話，請嘗試刪除該分號、儲存您的檔案，然後重新編譯它。您最終會得到這樣的東西：

```
$ gcc hello.c
hello.c:4:27: error: expected ';' after expression
  printf("Hello, world\n")
                          ^
                          ;
1 error generated.
```

可惡，有一個錯誤。但至少錯誤資訊是有用的。它告訴我們兩件關鍵的事情：出了什麼問題（「expected ';' after expression」）以及編譯器在哪裡發現問題（「hello.c:4:27」，也就是 *hello.c* 檔案的第 4 行、第 27 欄）。我不想在您探索 C 的早期就用錯誤訊息來嚇跑您，但您肯定會遇到它們很多次。令人高興的是，這只是意味著您需要仔細查看原始碼，然後再試一次。

敘述流

分隔字元告訴編譯器一個敘述在哪裡結束，下一個會從哪裡開始。這個順序也很重要。敘述流是從上到下，如果多個敘述在同一行時是從左到右。並且多個敘述是絕對被允許的！我們可以快速擴展我們簡單的「Hello, World」程式，讓它更加冗長。

如果您有時間和精力，我強烈建議您手動轉錄原始碼。這將為您提供更多 C 語法的練習。您會經常犯一兩個錯誤。發現並糾正這些錯誤是學習的好方法！即使這些錯誤有時會讓人有點沮喪。

考慮以下程式 *ch02/verbose.c*（*https://oreil.ly/wqnYC*）：

```
#include <stdio.h>

int main() {
  printf("Ahem!\n");                                  ❶
  printf("May I have your attention, please?\n");     ❷
  printf("I would like to extend the warmest of\n"); ❸
  printf("greetings to the world.\n");
  printf("Thank you.\n");
}
```

❶ 我們從一個和我們在 *hello.c* 中所使用的敘述非常相似的敘述開始。唯一真正的區別是我們印出的文字。請注意，我們會以分號分隔字元來結束該行。

❷ 我們有和第一個 `printf()` 敘述類似的第二個敘述。它確實會第二個執行。

❸ 只是為了充分表達想法，第三個敘述會在前兩個敘述之後呼叫。最後兩個呼叫將在此呼叫之後進行。

這是我們簡單的多行升級版本的輸出：

```
$ gcc verbose.c
$ ./a.out
Ahem!
May I have your attention, please?
I would like to extend the warmest of
greetings to the world.
Thank you.
```

不錯。您可以看到輸出是如何精確地遵循我們程式中敘述的順序。請嘗試切換它們的順序，並自己確認程式的流程是從上到下的，或者嘗試把兩個 `printf()` 呼叫放在同一行。這並不是要整您。我只是希望您盡可能多地練習編寫、執行和編譯程式碼。您嘗試的範例越多，您就越能避免簡單的錯誤，並且越容易遵循新的程式碼範例。

變數和型別

當然，我們可以做的不僅僅是印出文字。我們還可以在實作演算法或執行任務時儲存和操作資料。在 C（以及大多數語言）中，您會把資料儲存在**變數**（*variable*）中，這是解決問題的強大工具。這些變數具有**型別**（*type*），這些型別決定了您可以儲存哪些類型的資料。這兩個概念都在我提到的兩種敘述型式中佔有重要地位：宣告和初始化。

變數是值的佔位符。變數可以儲存簡單的值，例如數字（班上有多少學生？我的購物車中物品的總成本是多少？）或更複雜的東西（這個特定學生的名字是什麼？每個學生的成績是多少？甚至是一個實際的複數值，例如 –1 的平方根）。變數可以儲存從使用者那裡收到的資料，它們允許您編寫可以解決一般性問題的程式，而無須重寫程式本身。

獲取使用者輸入

我們很快就會探索定義和初始化變數的細節，但讓我們首先以獲取一些輸入來為使用者建立動態的輸出，而不用每次都重新編譯程式的這個想法來進行。我們將回到我們的「Hello, World」程式並稍微升級一下。我們可以要求使用者提供他們的名字，然後直接和他們打招呼！

到目前為止，您已經看到了一個輸出敘述，也就是我們用來問候世界的 `printf()` 函數呼叫。還有一個對應的輸入函數：`scanf()`。您可以使用列印 / 掃描（print/scan）配對來提示使用者，然後等待他們輸入答案。我們再把該答案儲存在一個變數中。如果您用其他語言做過一些程式設計，接下來的程式應該看起來很熟悉。如果您是程式設計和 C 的新手，程式列表可能有點密集或奇怪 —— 沒關係！輸入這些程式並在修正任何錯誤後，讓它們執行是一種有用的學習方式。

很多程式設計只是深思熟慮的抄襲。這是一個小笑話，但不完全是笑話。您開始的方式很像人類學習口語的方式：重複您看到(或聽到)的東西，而不必理解它的一切。如果您重複進行足夠多次，您就會發掘語言中固有的樣式，並了解您可以在哪裡進行有用的更改。做足這些有用的更改，您就會發現如何從頭開始創造新的、有意義的東西。這就是我們的目標。

這個 *ch02/hello2.c*（*https://oreil.ly/OrUqu*）程式，只是另一個您在開始程式設計發掘路徑時可以複製的一小段程式碼：

```
#include <stdio.h>

int main() {
  char name[20];

  printf("Enter your name: ");
  scanf("%s", name);
  printf("Well hello, %s!\n", name);
}
```

希望這個程式的結構看起來很熟悉。我們包含了我們的標準 I/O 程式庫、我們有一個 `main()` 函數、該函數有一個主體、在一對大括號中包含多個敘述。但是，該主體包含幾個新的項目。讓我們看看每一行。

```
char name[20];
```

這是我們宣告變數的第一個範例。變數的名稱就是「name」。它的型別是 `char`，這在 C 中指的是單一（ASCII）字元[1]。它也是一個**陣列**（*array*），意味著它按順序儲存了多個 `char` 值。在我們的範例中，可以儲存 20 個這樣的值。第 4 章中有更多關於陣列的介紹。現在，請注意這個變數可以保留一個人的名字，只要它少於 20 個字元。

```
printf("Enter your name: ");
```

這是一個相當標準的 `printf()` 呼叫 —— 和我們在第 15 頁的「建立 C 的 'Hello, World'」中的第一個程式中使用的非常相似。唯一有意義的差別是雙引號組中的最後一個字元。如果您查看 *hello.c* 或 *verbose.c*，您會注意到最後兩個字元是反斜線和字母「n」。這兩個字元（`\n`）的組合表示單一「換行（newline）」字元。如果在末尾添加了 `\n`，則會列印一行，隨後對 `printf()` 的任何呼叫都會在下一行進行。相反的，如果省略 `\n`，終端機中的游標（cursor）會停留在目前這行。如果您想做一些事情，比如列印一張表格，但一次只列印表格的一個單元格（cell），這會很方便。或者像在我們的案例中，如果您想提示使用者輸入一些內容，然後允許他們在和問題相同的那行上輸入他們的回答。

```
scanf("%s", name);
```

---

1 雖然在 90 年代 C 添加了一些對寬字元的支援，但 C 通常不能很好地處理更流行的 UTF 字元編碼，例如 UTF-8、UTF-16 等。這些編碼允許使用多位元組字元，而 C 的 `char` 型別在建構時只考慮了單位元組。（有關型別的更多資訊，請參見第 27 頁的「字串和字元」。）如果您使用國際或本地化文本，您需要研究一些程式庫來提供幫助。雖然我不會詳細介紹本地化，但我會在第 7 章中更深入地介紹程式庫。

這是我在本節開頭提到的新功能。`scanf()` 函數會「掃描」字元並把它們轉換為 C 的資料型別，例如數字，或者在本例中為字元陣列。轉換後，`scanf()` 會期望把每個「事物」儲存在一個變數中。那麼，在這一行中，我們正在掃描一堆字元，並把它們儲存在我們的 `name` 變數中。我們會在第 34 頁的「printf() 和 scanf()」中，看看括號內那個具有非常奇怪的語法的東西。

```
printf("Well hello, %s!\n", name);
```

最後，我們要列印我們的問候語。同樣的，這看起來應該很熟悉，但現在我們有了更奇怪的語法。如果 `%s` 像呼叫 `scanf()` 時一樣吸引了您的注意，恭喜！您剛剛發現了一個非常有用的樣式。這對字元正是 C 在列印或掃描字元陣列時會使用的。字元陣列是 C 語言中的一種常見型別，它有一個更簡單的名稱：字串（string）。因此在這個字元對中使用了「s」。

那麼 `name` 會發生什麼事呢？ `scanf()` 呼叫會採用您輸入的任何名稱（不包括您按下的 Return 鍵[2]）並把它儲存在記憶體中。我們的 `name` 變數包含了這些字元的記憶體位置。當我們呼叫 `printf()` 時，我們的第一個參數（`"Well hello, %s!\n"` 部分）包含一些文字字元，例如單字「Well」中的字元，以及字串的佔位符（`%s` 部分）。變數非常適合填充佔位符。您輸入的任何名稱現在都會顯示給您！

另請注意，我們確實在問候語中包含了特殊的 `\n` 換行字元。這意味著我們會列印問候語，然後「按 Return 鍵」，以便讓終端機中顯示的任何其他內容都會顯示在下一行。

讓我們繼續執行程式，看看事情是如何運作的。您可以使用 VS Code 底部的「終端機（Terminal）」頁籤，或者您的平台的 Terminal 或 Command 應用程式。您需要先使用 `gcc` 來編譯它，然後執行 `a.out` 或使用 `-o` 選項所選擇的任何名稱。您應該會得到類似於圖 2-1 的內容。

請注意，當您輸入名稱時，它會和要求您輸入的提示出現在同一行。當我們去掉換行字元（`\n`）時，這正是我們想要的效果。但是嘗試再次執行它並輸入不同的名稱。您得到您預期的結果了嗎？再試第三次。這種回應使用者輸入的動態行為，使得變數在電腦程式設計中非常寶貴。同一個程式可以根據不同的輸入產生不同的輸出，而不用重新編譯。反過來，這種能力有助於讓電腦程式對我們的日常生活變得無價。

---

2 您可能仍會在網上看到有關包含或排除「carriage return」的討論，這只是用於行尾標記的舊程式設計術語。這是一個繼承自早期打字機的術語，它具有把送紙滾筒（paper carriage）返回到起始位置的文字機制，因此您可以開始輸入下一行文字。

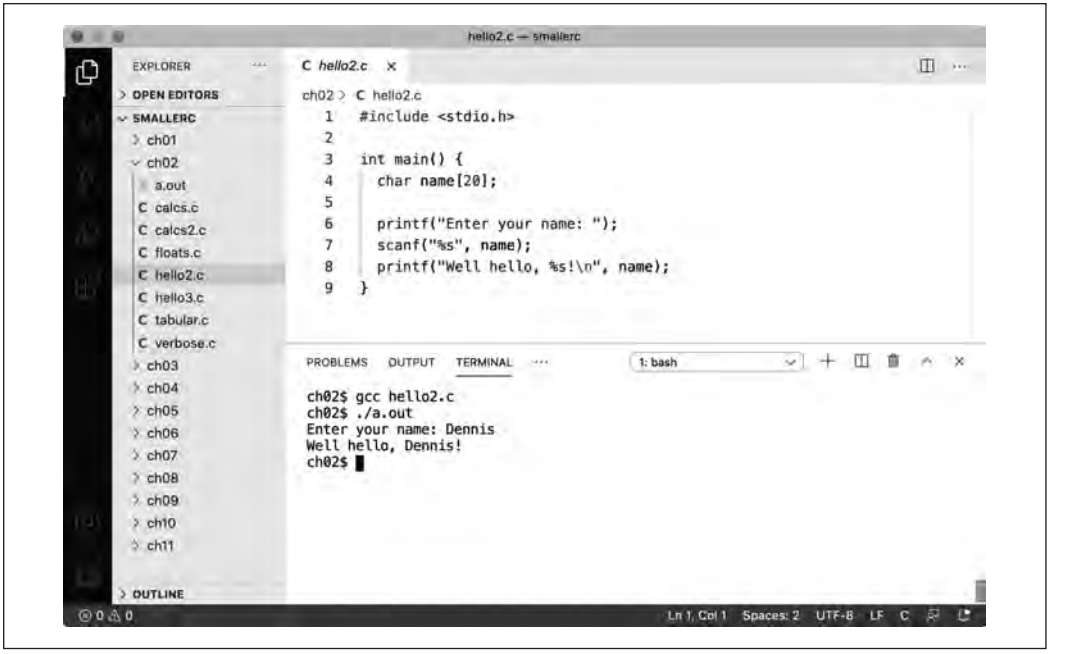

圖 2-1　我們量身定製的 Hello World 輸出

字串和字元

讓我們仔細看看 `char` 型別以及它的近親，字元陣列 —— `char[]` ——更為人知的被稱為字串。當您在 C 中宣告一個變數時，您要給它一個名稱和一個型別。最簡單的宣告看起來會像這個樣子：

```
char response;
```

在這裡，我們建立了一個名為 `response` 的變數，其型別為 `char`。`char` 型別會保存一個字元。例如，我們可以儲存「y」或「n」。第 5 章會介紹記憶體地址（memory address）和參照（reference）的詳細資訊，但現在，請記住，變數宣告會在記憶體中留出一個位置，該位置有足夠的空間來儲存您指定的任何型別。如果我們有一系列問題要問，那麼我們可以建立一系列的變數：

```
char response1;
char response2;
char finalanswer;
```

這些變數中的每一個都可以保存一個字元。但同樣的，當您使用變數時，您不必提前預測或決定該字元會是什麼。內容可能會有所變化（變化…變數…明白了嗎？）

C 編譯器會決定您的來源字元使用哪種編碼方式。較舊的編譯器會使用較舊的 ASCII[3] 格式，而較新的編譯器通常使用 UTF-8。兩種編碼方式都包含了小寫和大寫字母、數字以及您在鍵盤上看到的大多數符號。如果要講的是特定字元而不是 `char` 型別的變數的話，請使用單引號對它進行分隔。例如，`'a'`、`'A'`、`'8'` 和 `'@'` 都是有效的。

特殊字元

一個字元也可以是特殊的。C 支援定位字元（tab）和換行字元之類的東西。我們已經看到了換行字元（`\n`），但表 2-1 中還列出了一些其他的特殊字元。這些特殊字元使用了「逸出序列（escape sequence）」程式設計，反斜線被稱為「逸出字元（escape character）」。

表 2-1　C 中的逸出序列

字元	ASCII	名稱	說明
`\a`	7	BEL	使終端機在列印時發出「嗶」聲
`\n`	10	LF	換行 (Mac 和 Linux 上的標準行結束字元)
`\r`	13	CR	歸位 (和 `\n` 一起使用時，是在 Windows 上通用的行結束字元)
`\t`	15	HT	(水平) 定位字元
`\\`	92		用於在字串或字元中放置文字反斜線
`\'`	39		用於在 char 中放置文字單引號 (在字串中不需要逸出)
`\"`	34		用於在字串中放置文字雙引號 (char 中不需要逸出)

這不是一個詳盡的列表，但涵蓋了我們將在本書中使用的字元。

這些名稱捷徑只涵蓋最流行的字元。如果您必須使用其他特殊字元，例如來自數據機（modem）的傳輸結束（end of transmission）（EOT，ASCII 值為 4）信號，您可以使用反斜線以八進位來給出字元的 ASCII 值。那麼，我們的 EOT 字元將會是 `'\4'`，或者有時您會看到三個數字：`'\004'`。（由於 ASCII 是 7 位元編碼，三個八進位數字可以涵

3　美國資訊交換標準碼（American Standard Code for Information Interchange），最初是為電傳打字機而建構的 7 位元編碼。雖然現在有了 8 位元的變體，但它仍然是基於英語的。其他更可擴展的編碼（例如 Unicode 及其 UTF-8 選項）已成為規範。

蓋最高的 ASCII 字元。如果您好奇，這個字元就是刪除（delete）（DEL，ASCII 值為 127）或八進位逸出序列 `'\177'`。有些人更喜歡始終看到三個數字的這種一致性。）

您可能不需要很多這些捷徑，但由於 Windows 路徑名稱使用了反斜線字元，因此請務必記住某些字元需要這個特殊字首。當然，換行字元將繼續出現在許多列印敘述中。您可能已經注意到八進位逸出序列，字首的反斜線包含在單引號內。所以定位字元是 `'\t'`，反斜線是 `'\\'`。

字串

字串是一系列的 `char`，不過是一個非常正規的系列。很多程式設計語言都支援這樣的系列，稱為陣列。第 4 章會更詳細地介紹陣列，但是 char 型別的陣列 —— C 語法中的 `char[]` —— 十分常見，我想單獨提及它。

我們一直在處理字串，但並沒有非常明確的說明它們。在我們的第一個 hello 程式中，我們使用字串參數來呼叫 `printf()`。C 中的字串是零個或多個字元的集合，帶有一個特殊的最終「空（null）」字元 `\0`（ASCII 值為 0）。您通常會在程式碼中把這些字元包含在雙引號之間，例如我們的 `"Hello, world!\n"` 參數。令人高興的是，當您使用這些雙引號時，您不必自己添加 `\0`。它隱含在字串文字的定義中。

宣告字串變數就像宣告 `char` 變數一樣簡單：

```
char firstname[20];
char lastname[20];
char jobtitle[50];
```

這些變數中的每一個都可以儲存簡單的內容，例如名稱，或更複雜的內容，例如多部分職稱，例如「資深程式碼和美味餡餅的開發人員」。字串也可以為空：`""`。這可能看起來很傻，但想想您輸入名字之類的表單。如果您碰巧是一個非常成功的流行歌星，只有一個名字，上面的 `lastname` 變數可以被賦予有效值 `""`（也就是只有終止的 `'\0'`），以指出 Drake（德瑞克）和 Cher（雪兒）在沒有姓的情況下也是可以的。

數字

毫不意外的，C 也有可以儲存數值的型別。或者更準確地說，C 具有用來儲存比一般適合 `char` 型別的變數更大的數字的型別。（儘管到目前為止本章中的範例都使用 `char` 來儲存實際字元，但它仍然是一種數字型別，並且也適用於儲存和字元編碼無關的小數字。）C 把這些數字型別分為兩個子類別：整數和浮點數（即小數）。

整數型別

整數型別會儲存簡單的數字。主要型別稱為 `int`，但有許多變體。變體的主要差別在於可以儲存在給定型別的變數中的最大數字的大小。表 2-2 總結了這些型別及其儲存容量。

表 2-2　整數型別及其典型大小

型別	位元組	範圍	備註
char	1	–127 到 +127 或 0 到 255	通常用於字母；也可以儲存小數字
short	2	–32,767 到 +32,767	
int	2 或 4	–32,767 到 +32,767 或 –2,147,483,647 到 +2,147,483,647	依實作而異
long	4	–2,147,483,647 到 +2,147,483,647	
long long	8	–9,223,372,036,854,775,807 到 +9,223,372,036,854,775,807	在 C99 中引入

雖然 `char` 被定義為一個位元組，但其他型別的大小取決於系統。

上面的大多數型別都是**有正負號**（*signed*）型別[4]，這意味著它們可以儲存小於零的值。所有五種型別也都有一個明確的**無正負號**（*unsigned*）變體（例如，`unsigned int` 或 `unsigned char`），它的位元 / 位元組大小相同，但並不儲存負值。它們的範圍會從零開始，大約在有正負號範圍最上面的兩倍處結束，如表 2-3 所示。

表 2-3　無正負號整數型別及其典型大小

型別	位元組	範圍
unsigned char	1	0 到 255
unsigned short	2	0 到 65535
unsigned int	2 或 4	0 到 65535 或 0 到 4,294,967,295
unsigned long	4	0 到 4,294,967,295
unsigned long long	8	0 到 18,446,744,073,709,551,615

4　`char` 型別實際上可以是有符號或無符號的，具體取決於您的編譯器。

以下是一些整數型別宣告範例。注意 x 和 y 變數的宣告。您經常會看到以「x 和 y」形式討論的網格或圖形上的坐標。C 允許您同時宣告多個具有相同型別的變數名稱，並使用逗號分隔它們。這種格式沒有什麼特別之處，但如果您有一些簡短的相關變數名，這可能是一個不錯的選擇。

```
int studentcount;
long total;
int x, y;
short volume, chapter, page;
unsigned long long nationaldebt;
```

如果您有小的值要儲存，例如「最多一打」或「前 100 個」，請記住可以使用 char 型別。它的長度只有 1 個位元組，編譯器並不關心您是否把值列印為一個實際字元或一個簡單數字。

浮點型別

如果要儲存小數或財務數字，則可以使用 float 或 double 型別。這些都是小數點不固定（例如，它可以浮動）的浮點型別，能夠儲存像 999.9 或 3.14 這樣的值。但是因為我們談論的是以離散形式進行思考的電腦，所以浮點型別會儲存以 1 和 0 編碼的值的近似值，就像 int 一樣。float 型別是一種 32 位元編碼方式，可以儲存範圍很廣的值，從非常小的分數到非常大的指數。但浮點數在大約 –32k 到 32k 和小數點後六個有效位之間的窄區中最為準確。

double 型別的精準度是 float 的「雙倍」[5]。這意味著大約可以準確地表達 15 位的小數。我們在一些地方會看到這種近似可能會導致問題，但對於一般用途而言，例如收據總額或溫度感測器的讀數，這些型別就足夠了。

和其他型別一樣，您要把型別放在名稱之前：

```
float balance;
float average;
double microns;
```

5 這些格式由 IEEE（電機與電子工程師協會）指定。32 位元版本稱為「單精準度」，64 位元版本稱為「雙精準度」。還存在著更高的精準度，其規範（IEEE 754（*https://oreil.ly/rxm00*））仍在繼續開發中。

由於普通小數也可以儲存整數值，例如 6（表達為 6.0），因此我們可能會很容易就把 `float` 當作是預設數值型別。但是在像 Arduino 這樣的微型 CPU 上處理用小數點編碼的數字可能會很昂貴。即使在大晶片上，它仍然比使用簡單整數更昂貴。出於效能和準確性的原因，大多數 C 程式設計師都堅持使用 `int`，除非他們有明確的理由不這樣做。

變數名稱

不管變數是什麼型別，它都有一個名稱。在大多數情況下，您可以隨意使用任何您想要的名稱，但您必須遵守一些規則。

在 C 中，變數名稱可以用任何字母或底線字元（「_」）來開頭。在該初始字元之後，名稱可以包含更多字母、更多底線或數字。變數名稱會區分大小寫（`total` 和 `Total` 不是同一個變數）並且（通常）會限制為 31 個字元長[6]，儘管慣例是讓它們更短。

C 也有幾個保留給 C 語言本身使用的**關鍵字**（*keyword*）。因為表 2-4 中的關鍵字已經對 C 有意義，所以它們不能被用來作為變數名。一些實作版本可能會保留其他詞（例如 `asm`、`typeof` 和 `inline`），但大多數的替代性關鍵字會以一或兩個底線開頭，以避免和您自己的變數名稱發生衝突。

表 2-4　C 的關鍵字

保留字			
_Bool	default	if	static
_Complex	do	int	struct
_Imaginary	double	long	switch
auto	else	register	typedef
break	enum	restrict	union
case	extern	return	unsigned
char	float	short	void
const	for	signed	volatile
continue	goto	sizeof	while

6　例如，GNU C 編譯器沒有施加任何限制。但是為了相容性和一致性，堅持少於 31 個字元仍然是較受歡迎的。

如果您在宣告變數時偶然發現和關鍵字發生衝突，您會看到和使用了無效變數名稱（例如以數字開頭的變數名稱）類似的錯誤：

```
badname.c: In function ‘main’:
badname.c:4:9: error: expected identifier or ‘(’ before ‘do’
    4 |   float do;
      |         ^~
badname.c:5:7: error: expected identifier or ‘(’ before numeric constant
    5 |   int 5r;
      |       ^~
```

「expected identifier」這個詞是一個強有力的指標，指出您的變數是錯誤的原因。編譯器期待一個變數名稱，但它找到了一個關鍵字。

變數賦值

在我們的 *hello2.c* 範例中，我們依賴於對 `name` 變數的相當內隱式的賦值（assignment）。作為 `scanf()` 函數的參數，無論使用者輸入如何，都會儲存在該變數中。但是我們可以（並且經常這樣做）對變數進行直接賦值。您可以使用等號（「=」）來表達這樣的賦值：

```
int total;
total = 7;
```

您現在已成功地把值 7 儲存在變數 `total` 中。恭喜！

您也可以隨時覆寫該值：

```
int total;
total = 7;
total = 42;
```

雖然連續的賦值有點浪費，但這段 C 程式碼片段並沒有錯。變數 `total` 只會保留一個整數值，因此最近的賦值將是獲勝者，在本例中為 42。

您經常會看到同時定義變數並為其賦值初始值（用程式設計師的話說是*初始化*（*initialize*））：

```
int total = 7;
char answer = 'y';
```

現在 `total` 和 `answer` 都有可以使用的值了，但還是可以根據需要再進行更改。這正是變數的作用。

文字

我們在這些範例中插入變數的那些簡單值稱為**文字**（*literal*）。文字就只是一個不需要解釋的值。數字、單引號內的字元或雙引號內的字串都算作文字：

```
int count = 12;
char suffix = 's';
char label[] = "Description";
```

希望前兩個變數的定義您會看起來很熟悉。但是請注意，當我們初始化名為 `label` 的字串時，我們並不會給陣列指定長度。C 編譯器會從我們在初始化過程中所使用的文字來推斷出大小。那麼，在這個案例中，`label` 的長度是 12 個字元；其中 11 個用於「Description」這個單字中的字母，另外一個用於結尾的 `'\0'`。如果您知道稍後會在程式碼中需要它，您可以給字串變數更多的空間，但您不應該指定太小的空間。

```
char automatic[] = "A string variable with just the right length";
char jobtitle[50] = "Chief Acceptable Length Officer";
char warning[5] = "This is a bad idea.";
```

如果您嘗試賦值一個對於它的 `char[]` 變數來說太長的字串文字，您可能會看到來自編譯器的警告：

```
toolong.c: In function ‘main’:
toolong.c:6:21: warning: initializer-string for array of chars is too long
    6 |   char warning[5] = "This is a bad idea.";
      |                     ^~~~~~~~~~~~~~~~~~~~
```

這是一個相當明確的錯誤，所以希望您會發現它很容易修正。順便說一句，您的程式仍然可以執行。請注意，編譯器給了您一個**警告**（*warning*），而不是我們在前面的一些編譯器問題範例中所看到的**錯誤**（*error*）。警告通常意味著編譯器認為您犯了一個錯誤，不過您可以從這個懷疑得到好處。無論如何，通常最好解決警告，但這不是必要的。

printf() 和 scanf()

我們已經看到了如何使用 `printf()` 來列印資訊以及如何使用 `scanf()` 來接受使用者的輸入，但我省略了這兩個函數的許多細節。現在讓我們來看看其中的一些細節。

printf() 格式

`printf()` 函數是 C 的主要輸出函數。我們已經用它來列印簡單的字串，比如「`Hello, world\n`」。我們還在第 24 頁的「獲取使用者輸入」中偷看了怎麼使用它來列印變數。它可以列印所有變數型別，您只需要提供正確的**格式字串**（*format string*）即可。

當我們呼叫 `printf()` 時，我們首先要提供的通常是字串文字。第一個引數稱為格式字串。您可以把簡單的字串「按原樣」回顯（echo）到終端機，也可以列印（和格式化）變數的值。您使用格式字串來讓 `printf()` 知道接下來會發生什麼。您可以透過包含**格式說明符**（*format specifier*）來做到這一點，例如來自 *ch02/hello2.c*（*https://oreil.ly/DcU5k*）的 `%s`。讓我們列印一些我們在討論宣告和賦值時所建立的變數。看一下 *ch02/hello3.c*（*https://oreil.ly/qhIIT*）：

```
#include <stdio.h>

int main() {
  int count = 12;
  int total = 7;
  char answer = 'y';
  char jobtitle[50] = "Chief Acceptable Length Officer";
  // char warning[5] = "This is a bad idea.";

  printf("You can have %d, you currently have %d.\n", count, total);
  printf("You answered: %c\n", answer);
  printf("Please welcome our newest %s!\n", jobtitle);
}
```

結果如下：

```
ch02$ gcc hello3.c
ch02$ ./a.out
You can have 12, you currently have 7.
You answered: y
Please welcome our newest Chief Acceptable Length Officer!
```

把輸出與原始碼進行比較。您可以看到，我們大多按原樣列印出格式字串中的字元。但是當我們遇到格式說明符時，我們會替換格式字串後面的引數之一的值。仔細看看我們對 `printf()` 的第一次呼叫。我們在格式字串中有兩個格式說明符。在該字串之後，我們提供兩個變數。變數會按從左到右的順序填入格式說明符。如果您檢查輸出，您可以看到輸出的第一行確實首先包含了 `count` 的值，然後是 `total` 的值。酷！我們也得到了 `char` 和字串變數的輸出。

如果您注意到每種型別會使用不同的說明符，那麼恭喜您！您找到了這些敘述中的重要差異。（如果這一切看起來仍然有點像胡言亂語，請不要放棄！樣式 —— 以及不符合樣式的東西 —— 會隨著您閱讀和練習的更多而浮現出來。）事實上，`printf()` 有相當多的格式說明符，如表 2-5 所示。有些是顯而易見的，而且明顯和特定型別相關聯。其他則較深奧，但這就是書籍的用途。您會記住您最常使用的幾個說明符，並且可以在需要時隨時查找不太常用的說明符。

表 2-5　printf() 的常用格式說明符類型

說明符	型別	描述
`%c`	`char`	列印出單一字元
`%d`	`int, short, long`	列印以 10 為底的整數值（「十進位」）
`%f`	`float, double`	列印浮點值
`%i`	`int, short`	列印以 10 為底的整數值
`%li, %lli`	`long, long long`	列印以 10 為底的長整數值
`%s`	`char[]`(字串)	把 char 陣列列印為文本

還有其他格式，但我會把這些留到以後我們需要列印出奇數或資料中的特殊位元的地方。這些格式將會涵蓋您日常所需的絕大多數內容。附錄 B 更詳細地討論了本書中使用的所有格式。

定製輸出

但是如何格式化這些值呢？畢竟，C 使用了「格式字串」和「格式說明符」等詞。您把資訊添加到格式說明符來達成此目標。最常見的範例之一是列印浮點數，如銀行帳戶餘額或類比感測器的讀數。讓我們給自己一些有趣的小數，然後試著把它們列印出來。

```
#include <stdio.h>

int main() {
  float one_half = 0.5;
  double two_thirds = 0.666666667;
  double pi = 3.14159265358979323846264338327;

  printf("1/2: %f\n", one_half);
  printf("2/3: %f\n", two_thirds);
  printf("pi:  %f\n", pi);
}
```

我們宣告了三個變數，一個 `float` 型別和兩個 `double` 型別。我們在 `printf()` 敘述中使用了 `%f` 格式說明符。太棒了！這是我們在編譯和執行程式後得到的：

```
1/2: 0.500000
2/3: 0.666667
pi:  3.141593
```

嗯，它們都有六位小數，儘管我們沒有指定我們想要多少位，而且我們的變數都不是剛好有六位小數。為了獲得恰到好處的資訊量，您需要為格式說明符提供一些額外的細節。所有說明符都可以接受寬度和精準度引數。兩者都是可選的，您可以提供其中一個或兩個都提供。額外的細節看起來像一個小數：*`width.precision`*，這些細節介於百分號和型別字元之間，如圖 2-2 所示。

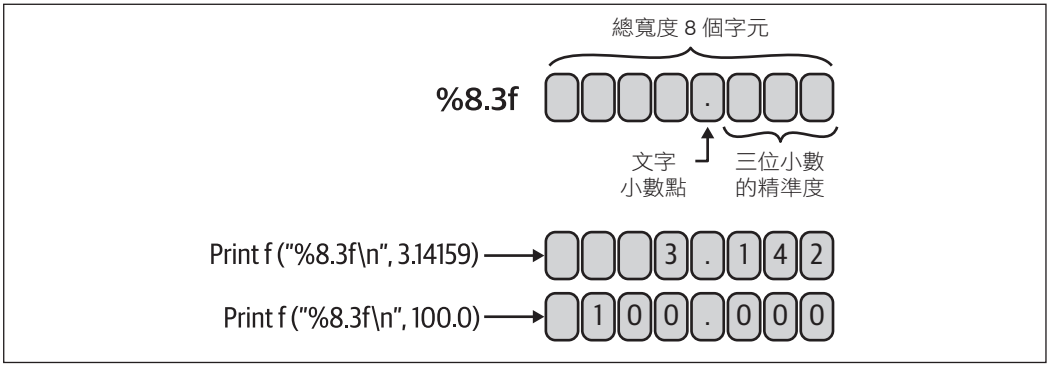

圖 2-2　內隱式轉換階層

使用這兩個選項對浮點數很有意義。我們現在可以要求更多或更少的數字。嘗試更改 *ch02/floats.c*（*https://oreil.ly/Os37q*）中的三個 `printf()` 呼叫，如下所示：

```
printf("1/2: |%5.2f|\n", one_half);
printf("2/3: |%12f|\n", two_thirds);
printf("pi:  |%12.10f|\n", pi);
```

我在擴展格式說明符之前和之後添加了豎線（vertical bar）或管道（pipe）字元（`|`），以便您可以看到寬度元素會如何影響輸出。看看新的結果：

```
1/2: | 0.50|          ❶
2/3: |    0.666667|   ❷
pi:  |3.1415926536|   ❸
```

❶ 我們的值 `0.5` 在五個字元的總欄位寬度中以小數點後兩位的精準度來顯示。因為我們不需要全部五個位置，所以在開頭會添加一個空白字元。

❷ 在 12 個位置內列印更長的小數。請注意，在沒有指定任何寬度或精準度的情況下，我們得到了相同的六位小數。

❸ 讓更長的小數顯示在 12 個位置內，但包括 10 位數的精準度。注意這裡的 12 是總寬度 —— 它包括了小數點後數字所佔據的位置。

對於 `printf()`，如果有給定寬度時，您請求的精準度和您正在列印的實際值會優先於所給定的寬度。您經常會看到像「`%0.2f`」或「`%.1f`」這樣的浮點格式，它們在所需的確切位置內為您提供正確的小數位數。例如，把這兩種範例格式應用於 π，將分別得到 `3.14` 和 `3.1`。

對於字串或整數等其他型別，寬度選項相當簡單。例如，透過使用相同的寬度而不考慮列印的值，您可以很容易地列印表格資料，如 *ch02/tabular.c*（*https://oreil.ly/nQC7x*）中所示：

```
float root2 = 1.4142;
float phi = 1.618034;
float pi = 3.1415926;
printf("     %10s%10s%10s\n", "Root 2", "phi", "pi");
printf(" 1x  %10.4f%10.4f%10.4f\n", root2, phi, pi);
printf(" 2x  %10.4f%10.4f%10.4f\n", 2 * root2, 2 * phi, 2 * pi);
```

具有出色的欄狀結果：

```
          Root 2       phi        pi
 1x       1.4142    1.6180    3.1416
 2x       2.8284    3.2361    6.2832
```

非常好。並注意我是如何處理欄標籤的。我使用了格式說明符和字串文字，而不是手動加入空白來分隔標籤的單一字串。我這樣做是為了突顯輸出寬度的運用，即使手動操作並不困難。事實上，手動把標籤置中在這幾行上會更容易。如果您準備做一個小練習，請開啟 *tabular.c* 檔案並嘗試調整第一個 `printf()` 看看您是否可以讓標籤置中。

雖然寬度選項對所有型別都很簡單，但對於非浮點格式，添加精準度選項的效果可能不那麼直覺。對於字串而言，指定精準度會導致截斷文本以適合給定的欄位寬度。（對於 `int` 和 `char` 型別來說，它通常沒有效果，但您的編譯器可能會警告您不要依賴這種「典型」行為。）