前言

C 語言是十分重要的程式設言，一直以來都十分受到重視，指標是 C 語言的核心，提供許多彈性與功能；包含動態操作記憶體、高階資料結構操作以及存取硬體等能力。這些強大功能與彈性也同時造成了精通指標的困難。

本書特色

有許多 C 語言相關的書廣泛地介紹 C 語言各個面向，但對指標相關主題僅止於基本概念與使用；少部份走馬看花地帶過堆疊（stack）與堆積（heap）等重要的記憶體管理技術；然而少了這些重要的知識就無法對指標有完整的認識。堆疊與堆積被用於支援函數與動態記憶體配置的記憶體區域。

指標是需要專書介紹的複雜主題，本書以指標為核心，深入介紹 C 語言。部份內容涉及深入瞭解程式堆疊與堆積以及指標在這些環境下的使用方式；理解任何知識領域都有階段性，從走馬看花的概論到深入理解成為直覺的一部份，只有切實了解指標與記憶體管理相關知識，才能提高對 C 語言的理解。

方法

撰寫程式就是操作資料，這些資料通常存放在記憶體；因此愈瞭解 C 語言處理記憶體的方式，就愈能提高程式設計技能。知道 `malloc` 函數會在堆積配置一塊記憶體是一回事，要理解記憶體配置帶來的影響又是另一回事；如果配置一個結構的邏輯大小是 45 位元組，可能會對於實際配置的記憶體大於 45 個位元組（byte）感到驚訝，而且所配置的記憶體還包含了碎片。

呼叫函數時會建立一個椎疊框架（stack frame）並推入程式堆疊（stack）；理解堆疊框架與程式堆疊有助於釐清傳值與傳指標的差異。雖然與指標沒有直接的關係，但瞭解堆疊框架也有助於解釋遞迴的運作。

為了幫助讀者理解指標與記憶體管理相關技術，本書介紹許多不同的記憶體模型，從最簡單的線性記憶體表示法，到描繪特定情況下程式堆疊與堆積狀態的複雜圖表。畫面或書本中的程式碼只能以靜態的方式呈現程式的動態行為，這種抽象的表示法是理解程式動態行為的一大障礙，而記憶體模型則能夠縮短兩者間的鴻溝。

目標讀者

C 語言是種區塊結構式語言，許多現代程式語言都採用它的程序特性，如 C++ 與 Java。這些語言同樣使用了程式堆疊與堆積，也使用了指標，只是有時會偽裝成參考（reference）。若讀者對 C 語言已有基本瞭解，對於正在學習 C 語言的讀者，本書對指標與記憶體等主題提供更完整的介紹，能擴展讀者對 C 語言的認識，加強 C 語言較不被熟悉的部份；對於較有經驗的 C 或 C++ 設計師，本書有助於填補 C 語言及其底層運作原理間的鴻溝，讓讀者成為更好的程式設計師；對於 C# 與 Java 程式設計師，本書能讓讀者更加瞭解 C 語言，對物件導向式語言處理堆疊與堆積的方式有更深入的認識。

本書結構

本書結構依循傳統主題的順序，如陣列、結構與函數；但各章節的內容都圍繞著指標的使用與記憶體的管理方式。例如介紹將指標傳入與傳出函數的方式，同時也說明將指標作為堆疊框架一份子的使用方式，以及參考堆積記憶體的方式。

第一章，入門

本章針對還不熟悉指標或剛接觸指標的讀者，介紹指標基礎知識，包含指標操作與不同型別指標的宣告，如常數指標、函數指標、`NULL` 使用方式與其他相關變化，這些對記憶體配置與使用方式都有很大的影響。

第二章，*C* 語言的動態記憶體管理

第二章的主題是動態記憶體管理，涵蓋標準記憶體配置函數以及釋放記憶體相關技術，對大多數應用程式而言有效釋放記憶體至關重要，記憶體若未妥善釋放，可能導致記憶體洩漏（memory leak）與懸置指標（dangling pointer）等問題。此外還介紹了包含垃圾處理與例外處理函數等其他的釋放技巧。

第三章，指標與函數

函數是程式碼的基本建構單元，新手開發人員常對資料傳入與傳出函數感到困擾，本章將介紹傳遞資料相關技巧、利用指標傳回資訊的常見錯誤，並詳細介紹函數指標。函數指標提供進一步的控制與彈性，可延伸出許多高階程式技巧。

第四章，指標與陣列

雖然陣列表示法與指標表示法並非完全相容，但兩者的關係的確十分密切。本章涵蓋一維與多維陣列以及對應的指標表示法。特別是傳遞陣列以及動態配置陣列時的許多細節，不論是連續或非連續記憶體配置都有詳細解釋，並透過不同的記憶體模型提供詳盡的例子。

第五章，指標與字串

字串是應用程式的重要成員，本章介紹字串基礎以及以指標操作字串。字串常量池（literal pool）及其對指標的影響是 C 語言中另一個常被忽略的特色，本章提供了許多範例以解釋與說明這個主題。

第六章，指標與結構

結構是組織與操作資料十分有用的方式，指標提昇結構彈性，進一步強化結構操作資料的能力。本章介紹結構基礎與結構記憶體配置和指標的關係，透過範例示範將指標應用在不同的資料結構當中。

第七章，安全問題與不當使用指標

指標是強而有力的工具，也是許多安全問題的根源。本章檢視以緩衝區溢位為中心的基本問題以及相關的指標議題，也提供改善問題的方法。

第八章，其他補充

最後一章介紹指標的其他技巧與問題，雖然 C 語言不是物件導向程式語言，仍然能夠在程式中使用包含多型在內的許多物件導向式程式設計的觀念。介紹如何在多執行緒環境中使用指標，也涵蓋了 `restrict` 關鍵字的意義與使用方式。

結語

本書希望能比其他書籍在使用指標主題上提供更深入的討論，書中範例涵蓋指標的主要使用情境與較少見的使用方式，並明確提出常見的指標問題。

本書編排慣例

本書使用的字型、字體慣例，如下所示：

斜體字（*Italic*）

用來表示檔名、副檔名、路徑、網址和電子郵件。對於初次提到或重要的詞彙，中文以楷體字呈現，其對應的英文則以斜體表示。

第一章

入門

對指標的瞭解與掌控能力是資深 C 語言開發人員與新手間最大的不同，指標在 C 語言中隨處可見，也是 C 語言強大彈性的根源，提供了動態記憶體配置的能力，與陣列有密切的關係，當指標指向函數時能進一步操控程式的流向。

長久以來，指標一直是學習 C 語言最大的障礙，指標的基本概念很簡單：就是個存放記憶體位址的變數。然而，當開始使用各種運算子與辨別這些奇怪的符號時，概念很快就變得十分複雜。其實並不需如此，只要從基本概念開始，建立紮實的基礎，就很容易掌握指標的進階使用。

掌握指標的關鍵在於理解 C 語言的記憶體管理方式，指標存放的是記憶體的位址，如果不清楚記憶體組織與管理的方式就很難理解指標的作用。為了解決這個問題，必須在解釋指標概念的適當時機一併說明記憶體的組織結構，一旦掌握了記憶體及其組織結構就很容易理解指標。

本章簡單介紹指標、運算子以及與記憶體間的交互運作方式。第一節介紹各種宣告方式、基本指標運算子以及 null 的概念，C 語言支援很多種不同型別的「null」，詳加瞭解會有很大的幫助。

第二節更深入解釋撰寫 C 語言時必然會遇上的記憶體模型，使用的編譯器與作業系統環境都會影響指標使用，也解釋了幾個與指標相關的內定型別以及記憶體模型。

接下來的章節更深入涵蓋指標運算子（operator），包含指標運算與比較。最後一節則是常數與指標，結合各種不同的宣告方式提供了許多有趣又實用的可能性。

不論是 C 語言新手或資深程式設計師，本章能提供 C 語言紮實的知識，補齊不足的部份。資深程式設計師可以選擇有興趣的章節，新手程式設計師則建議依序閱讀。

指標與記憶體

編譯 C 語言程式時，需要處理三種不同的記憶體：

靜態／全域

宣告為靜態（static）的變數會配置在這塊記憶體，全域變數也使用相同區域；這些變數從程式啟動時配置，持續存在到程式終止，所有的函數都能夠存取全域變數，靜態變數的存取範圍則局限在變數宣告所在的函數當中。

自動（*automatic*）

這些變數在函數中宣告，當函數被呼叫時自動建立，範圍受限於函數，生命週期也局限在函數執行期間。

動態

記憶體配置在堆積（heap），能依應用程式需要配置與釋放，指標參考到配置的記憶體，範圍受限於參考到記憶體的指標，會一直存續直到記憶體被釋放，這部份也是第二章的重點。

表 1-1 是記憶體中變數的範圍與生命週期的整理。

表 1-1　範圍與生命週期

	範圍	生命週期
全域	整個檔案	應用程式的生命週期
靜態	宣告所在的函數	應用程式的生命週期
自動（區域）	宣告所在的函數	函數執行期間
動態	視參考到記憶體的指標而定	直到釋放記憶體

認識不同類型的記憶體有助於理解指標的運作方式，大多數指標都用於操作記憶體中的資料，瞭解記憶體的用途與組織方式能釐清指標操作記憶體的行為。

指標變數存放的是記憶體的位址，這塊記憶體可能代表了另一個變數、物件或函數；指到的記憶體是指透過如 `malloc` 函數等記憶體配置函數配置的記憶體。指標通常會宣告為特定型別（type），代表所指向記憶體內容的資料型別，例如指向 `char` 的指標。目的可能是任何 C 語言資料型別，如整數、字元、字串或結構；但指標本身並不含任何能夠得知所參照記憶體的資料型別所需的資訊，指標內容就只有一個記憶體位址。

為什麼需要精通指標？

指標有許多的用途：

- 建立快速有效率的程式碼
- 提供更方便解決問題的方法
- 動態記憶體配置
- 簡化表示式
- 利用指標傳遞結構資料，避免傳遞大量資料造成的負擔
- 保護以參數傳入函數的資料

指標較貼近硬體，能提高程式速度也更有效率，也就是編譯器更容易將運算轉換為機械碼。這時指標操作的負擔會比起其他操作少了很多。

利用指標能簡化許多資料結構的實作，例如，串列（linked list）有陣列與指標兩種實作方式，指標較容易使用，也能夠直接對應到前一個與下一個連結（link）；使用陣列索引實作的陣列版本，比起指標較不直覺也少了許多彈性。

圖 1-1 以圖形化的方式呈現了串列的陣列與指標兩種實作方式，左半部是使用陣列實作，`head` 變數表示串列中的第一個元素位於陣列索引值為 10 的位置，陣列中的元素包含了代表員工資料的結構，其中 `next` 欄位則是下個員工資料所在的陣列索引，陰影部份表示陣列未使用的位置。

右半部則是使用指標實作的相同結構，`head` 變數持有指向第一個員工節點的指標，每個節點同時持有員工資料以及指向串列下一個元素的指標。

指標表示法不只清楚也更有彈性，陣列的大小通常在建立後就固定不變，這限制了串列的元素個數，指標則沒有這個限制，能夠依需要動態建立新的節點。

指標對 C 語言的動態記憶體配置有很大的影響，`malloc` 與 `free` 函數分別用於配置與釋放動態記憶體，動態記憶體配置能達成動態長度陣列（varaible length array，VLA），以及串列與佇列（queue）等資料結構；在新的 C 語言標準 C11 中，提供了動態長度陣列的功能。

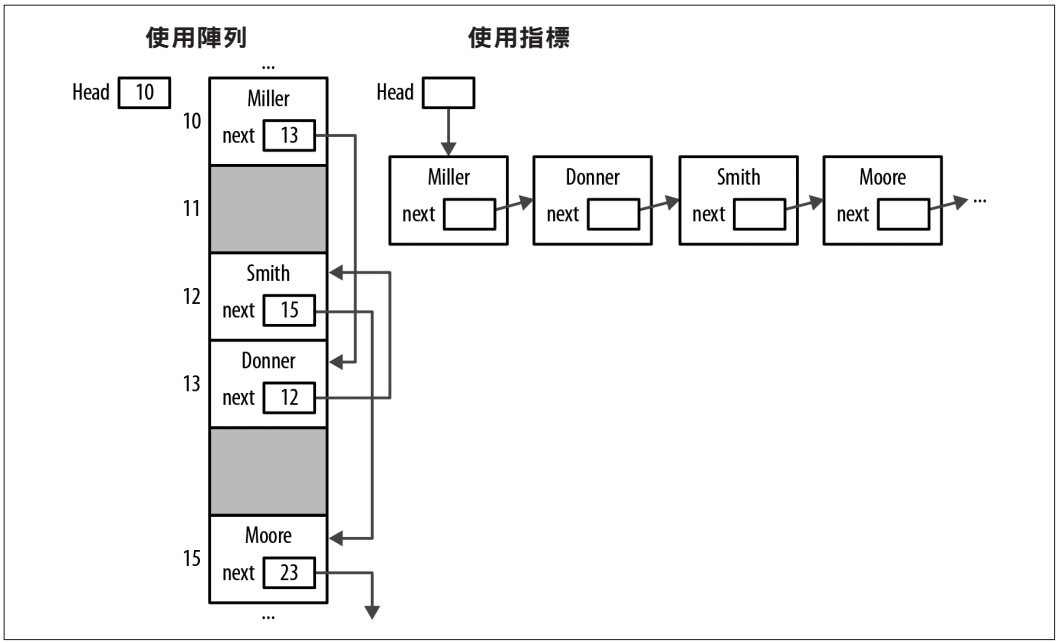

圖 1-1　串列的陣列表示法與指標表示法

簡潔的表示式可以很清楚或十分難懂，就像並非所有的程式設計師都能夠清楚理解指標；簡潔的表示式是為了解決問題，而非為簡潔而簡潔。例如，在以下程式碼使用了兩個不同的 `printf` 函數顯示 `names` 中第二個元素的第三個字元（在第 10 頁《使用間接運算子解參考指標》一節中會詳細說明解參考的運作方式）。雖然這兩種表示方式同樣都能顯示字元 `n`，但使用陣列表示法的方式看起來較為簡潔清楚。

```
char *names[] = {"Miller", "Jones", "Anderson"};
printf("%c\n", *(*(names+1)+2));
printf("%c\n", names[1][2]);
```

指標是能建立與強化應用程式的強力工具，但在使用上可能引發許多的問題：

- 存取陣列或其他資料結構時超出可用範圍
- 在可用範圍之外參考自動變數
- 在記憶體釋放後參考配置在堆積上的記憶體
- 在配置記憶體前解參考指標

第七章將會詳細介紹這些問題。

C 語言規格（*http://bit.ly/173cDxJ*）中清楚定義了指標語法、語義以及使用方式。然而，仍然有規格中未明確定義的行為，在這些情況下的行為可能是：

以實作定義（*implementation-defined*）

提供了特定、有文件說明的行為；整數右移位運算時將高位元傳播到低位元就是以實作定義的例子。

未指定（*unspecified*）

提供實作但無文件定義；`malloc` 函數對參數 0 的處理就是個未指定行為，CERT Secure Coding Appendix DD（*http://bit.ly/YOFY8s*）提供了一份未指定行為的清單。

未定義（*undefined*）

行為沒有規範限制，任何事都可能發生。`free` 函數對釋放後指標內的數值處理就是個未定義行為，CERT Secure Coding Appendix CC（*http://bit.ly/16msOVK*）提供了一份未定義行為清單。

還有些因地區而異（locale-specific）的行為，通常在編譯器廠商提供的文件中有詳細說明，提供因地區而異的行為能讓編譯器開發人員將精力集中於產生更有效率的程式碼。

宣告指標

指標變數的宣告是由資料類型接著星號再加上指標變數名稱組成；以下的範例分別是一個整數以及一個整數指標：

```
int num;
int *pi;
```

星號前後的空白不會影響宣告的意義，以下定義都有相同的效果：

```
int* pi;
int * pi;
int *pi;
int*pi;
```

空白使用取決於使用者的喜好。

星號將變數宣告為指標，這個有多重意義的符號同時作為乘法與解參考指標之用。

圖 1-2 呈現上述宣告在記憶體中的配置方式，圖中的三個長方格表示記憶體位置，長方格左側的數字表示位址，位址旁的名稱則是指派給該位址的變數名稱，為了方便說明，假設位址位於 100，一般情況下並不會知道指標或任何變數真正所在的位址，而且應用程式也不在意實際的位址，長方格中的三個小點表示記憶體尚未初始化。

指向未初始化記憶體的指標會造成問題，解參考未初始化指標時，指標內容可能不是合法的位址，即使是合法的位址，位址的內容也不是合法資料；非法位址（invalid address）是指應用程式無權存取的位址，在大多數平台上會造成程式終止，後果十分嚴重，會造成許多問題，第七章有詳細的討論。

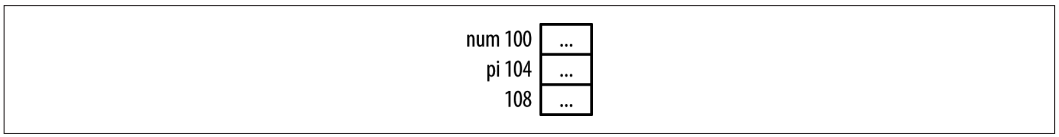

圖 1-2　記憶體示意圖

變數 num 與 pi 分別配置在位址 100 與 104，假設都需要四個位元組的空間，實際大小會隨著系統不同而異，詳細說明參見第 15 頁《指標大小與型別》一節。除非有額外說明，本書的範例都假設整數大小為四個位元組。

本書使用 100 之類的位址說明指標行為以簡化範例內容；執行範例程式時會看不到同的位址，甚至每次執行程式的位址都不相同。

特別提醒以下幾點：

- pi 的內容最終會指派為某個整數變數的位址。
- 這些變數都還沒初始化，內容包含垃圾。
- 指標的實作本身沒有任何能提示所參考的資料型別的資訊，不論其內容是否有意義。
- 然而，宣告指標時必須指定型別，當使用錯誤的型別時，編譯器會發出警告訊息。

垃圾是指記憶體內容可能是任何數值。配置記憶體時並不會清除記憶體內容，可能是之前的任何數值。如果之前存放的資料是浮點數，用整數方式解讀就不會有太大的意義；即使之前的資料是整數，也不是正確的整數，因此用垃圾表示記憶體原有的內容。

雖然指標不需要初始化就能夠使用，但初始化才能確保有正確的行為。

如何閱讀宣告？

接下來要介紹如何閱讀指標宣告才能比較容易理解。秘訣是由後往前讀，雖然還沒有介紹常數指標，請先看以下範例：

```
const int *pci;
```

由後往前讀能就能逐步理解宣告內容（圖 1-3）。

1. pci 是個變數	`const int *pci;`
2. pci 是個指標變數	`const int *pci;`
3. pci 是個指向整數的指標變數	`const int *pci;`
4. pci 是個指向整數常數的指標變數	`const int *pci;`

圖 1-3　由後往前讀宣告

許多程式設計師都認為從由後往前讀宣告簡單得多。

處理複雜的指標表示式時，可如之後的範例使用圖示法。

位址運算子

位子運算子 & 會傳回運算元的位址，可以用以下的程式將 pi 指標初始化為 num 變數的位址：

```
num = 0;
pi = &num;
```

num 變數的數值設為 0，pi 設定成指向 num 的位址，如圖 1-4 所示。

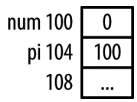

圖 1-4　記憶體指派

也可以在宣告指標的時候就將 pi 指派為 num 的位址：

```
int num;
int *pi = &num;
```

以下程式在大多數編譯器會產生語法錯誤：

```
num = 0;
pi = num;
```

錯誤訊息如下：

```
error: invalid converion from 'int' to 'int*'
```

pi 變數的型別是整數指標，num 變數的型別則是整數，錯誤訊息說明不能將整數轉換成整數指標型別。

將整數指派給指標通常會產生警告或錯誤訊息。

指標與整數不同，雖然兩者在大多數機器上都佔用相同的位元組，但型別並不相同。可以利用轉型將整數指派給整數指標：

```
pi = (int *)num;
```

這樣一來就不會有語法錯誤，但執行時可能造成程式異常終止，因為程式可能解參考位址 0。位址 0 在大多數作業系統不能合法使用，在第 11 頁《Null 的概念》一節會更詳細介紹這個問題。

盡早初始化指標是個好習慣：

```
int num;
int *pi;
pi = &num;
```

顯示指標值

現實中處理的變數很少有像 100 或 104 這樣的位址，變數的位址能夠用以下的方式顯示：

```
int n um = 0;
int *pi = &num;

printf("Address of num: %d  Value: %d\n", &num, num);
printf("Addreess of pi: %d  Value: %d\n", &pi, pi);
```

執行時會產生如下輸出，這個範例使用真實的位址，讀者執行程式時也許會看到不同的位址：

```
Address of num: 4520836  Value: 0
Address of pi: 4520824  Value: 4520836
```

printf 函數有其他的欄位格式，在顯示指標內容時十分方便，參考表 1-2。

表 1-2　欄位格式

格式	意義
%x	以 16 進位數字顯示數值
%o	以 8 進位顯示數值
%p	以實作指定的方式顯示數值；一般是指 16 進位

使用方式如下：

```
printf("Address of pi: %d  Value: %d\n", &pi, pi);
printf("Address of pi: %x  Value: %x\n", &pi, pi);

printf("Address of pi: %o  Value: %o\n", &pi, pi);
printf("Address of pi: %p  Value: %p\n", &pi, pi);
```

會顯示 pi 的位址以及內容，以本例來說，pi 的內容就是 num 的位址：

```
Address of pi: 4520824  Value: 4520836
Address of pi: 44fb78  Value: 44fb84
Address of pi: 21175570  Value: 21175604
Address of pi: 0044FB78  Value: 0044FB84
```

%p 與 %x 的差別在於會以大寫顯示 16 進位數值，之後的範例除非有特別說明，不然都會使用 %p 格式。

在不同平台上用相同的方式顯示指標數值並不容易，一個方式是將指標轉型為指向 void 的指標，再用 %p 格式顯示：

```
printf("Value of pi: %p\n", (void*)pi);
```

稍後在第 13 頁《void 指標》一節中有進一步解釋，為了保特範例的單純化，接下來將使用 %p 格式，但不轉型為 void 指標。

虛擬記憶體與指標

顯示位址的問題並不單純，虛擬作業系統（virtual operating system）中顯示的指標位址與實體記憶位址不同，虛擬作業系統能切割主機的實體記憶體空間給不同程式，應用程式被切割成分頁或框架（page/frame），各分頁代表主記憶體的特定區域；應用程式的分頁被配置在不同甚至不相連的實體記憶體，有時不會同時存在實體記憶體。如果作業系統不需要使用特定分頁的記憶體，可以先將這塊記憶體移到輔助儲存（secondary storage），需要時再載入到不同位置的記憶體中。此功能可提高虛擬作業系統管理記憶體的彈性。

應用程式假設能夠存取主機的整個實體記憶體空間，但事實上並非如此。應用程式使用的位址是虛擬位址，作業系統會將虛擬位址轉換為真正的實體記憶體位址。

這表示分頁中的程式碼與資料在程式執行時可能放置在不同的實體位置，應用程式的虛擬位址，也就是存放在指標中的位址不會改變，虛擬位址透過作業系統轉換為實體位址，應用程式不會察覺到這個過程。

作業系統控制了整個轉換程序，程式設計師無法控制，也不需要擔心這個過程；理解這些細節能夠說明在虛擬作業系統中執行的應用程式所取得的位址。

使用間接運算子解參考指標

間接運算子（*）會傳回指標變數指到的記憶體當中的數值；通常將這個動作稱為解參考（dereference）指標，接下來的範例宣告並初始化了 num 與 pi：

```
int num = 5;
int *pi = &num;
```

下列的間接運算子命令會顯示 5，也就是 num 的數值：

```
printf("%p\n", *pi);      // 顯示 5
```

也可以用解參考後的數值作為 lvalue，lvalue 表示能放在指派運算子左側的運算元，因為被指派了新值，所有的 lvalue 數值都會被修改。

以下透過 pi 將數值指派為 200，因為 pi 指向變數 num，等同於將 num 的數值指派為 200，圖 1-5 是記憶體中的效果：

```
*pi = 200;
printf("%d\n", num);     // 顯示 200
```

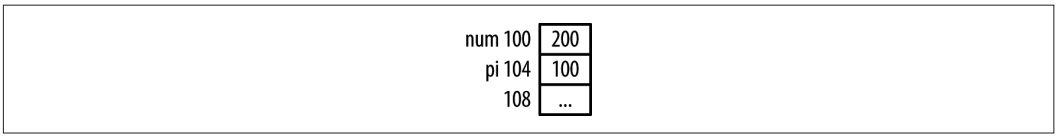

圖 1-5　使用解參考運算子指派新值

函數指標

指標也可以指向函數，這種宣告式有點難懂，以下示範了宣告函數指標的方式，目標函數沒有傳入參數也沒有傳回值，指標變數的名稱是 foo：

```
void (*foo)();
```

函數指標是涵蓋較廣的主題，會在第三章有詳細的介紹。

Null 的概念

null 是個有趣但常被誤解的概念。誤解通常來自於幾個相似但不同的概念，包含：

- null 的概念
- null 指標常數
- NULL 巨集
- ASCII NUL
- null 字串
- null 指令

當 NULL 被指派給指標時表示指標沒有指向任何東西，null 的概念表示指標持有一個不會與其他指標相等的特殊數值，沒有指到記憶體中的任何區域。兩個 null 指標總是相等，但兩者可以各自的指標型別，例如字元指標與整數指標，但這並不常見。

null 是個抽象的概念，在實務上透過 null 指標常數實現，這個常數可能是數值 0。C 語言的程式設計師不需要考量 null 指標的實際數值為何。

NULL 巨集是個轉型為 void 指標的整數常數 0，在許多函數庫中都定義如下：

```
#define NULL  ((void*)0)
```

這也是一般人所認知的 null 指標，這個定義通常存在幾個不同的標頭檔當中，包含 *stddef.h*、*stdlib.h* 以及 *stdio.h*。

如果編譯器使用了非零的位元模式（bit pattern）表示 null，編譯器就必須確保所有使用到 `NULL` 或 0 的指標被作為 null 指標處理，實際上在內部 null 的表示方式是由實作定義（implementation defined），`NULL` 或 0 是語言層級，表示 null 指標的符號。

ASCII `NUL` 是被定義為全部是 0 的位元組，與 null 指標並不相同，C 語言的字串是由以 0 值結尾的字元序列表示，null 字串是個不含任何字元的空字串，最後，null 指令是指只有一個分號的空指令。

稍後將會看到在實作許多資料結構時，null 指標都是個十分方便的功能，例如串列通常會用 null 指標表示串列的結尾。

想要把 `pi` 指派為 null 值時，通常會如以下方式使用 NULL 型別：

```
pi = NULL;
```

null 指標與未初始化指標並不相同，未初始化指標可能持有任何數值，有 NULL 值的指標則不會參照到記憶體的任何位置。

有趣的是可以將 0 值指派給指標，卻無法將其他數值指派給整數，如以下的指令：

```
pi = 0;
pi = NULL;
pi = 100;     // 語法錯誤
pi = num;     // 語法錯誤
```

指標可以用在邏輯表示式中唯一的運算元，例如，可以用以下指令測試指標是否為 NULL 值：

```
if (pi) {
   // 不是 NULL
} else {
  // 是 NULL
}
```

以下兩個表示式都合法但卻多餘，雖然可能會比較清楚，但並不需要特別寫出與 NULL 比較。

如果 pi 已經被指派為 NULL 值，就會被解譯為二進位零值，在 C 語言中這也代表 false，當 pi 是 NULL 值，就會執行 else 部份。

```
if(pi == NULL) ...
if(pi != NULL) ...
```

因為 null 指標不含合法位址，永遠不要解參考 null 指標，否則會造成程式終止。

該不該使用 NULL ？

指標應該使用 NULL 或是 0 值比較好？兩種方式都正確，取決於個人偏好。有些開發人員比較喜歡使用 NULL，因為可以明確提醒正在處理的變數是指標；有些人則覺得沒有必要，0 就已經明確表示沒有任何東西了。

NULL 不該用在非指標的情況下。雖然不見得會有問題，但並非它的目的，在應該使用 ASCII NUL 字元的時候使用 NULL 一定會造成問題，這個字元沒有定義在任何 C 標頭檔，而是相當於字元常量（character literal）'\0'，從數值上與十進位值 0 相等。

0 的意義會隨著使用情境有所不同，在某些情境下代表整數 0，而在另一個情境下表示 null 指標，例如以下範例：

```
int num;
int *pi = 0;          // 0 表示 null 指標，NULL
pi = &num;
*pi = 0;              // 0 表示整數 0
```

開發人員已經很習慣運算子過載（overload），例如 * 號用於宣告指標、解參考指標或乘法。0 也同樣是過載。因為運算元過載比較少見，會覺得不太習慣。

void 指標

void 指標是個可以指向任何資料型別的通用指標，以下是個 void 指標的例子：

```
void *pv;
```

void 指標有幾個特性：

- void 指標與 char 指標有相同的表示方式與記憶體對齊（memory alignemtn）。
- void 指標永遠不會和其他指標相等，然而，兩個指派為 NULL 的 void 指標則會相等。

任何指標都可以指派給 void 指標，之後還可以轉型回原來的型別，轉型後的值會等同於原來指標值。在下列程式中，int 指標先指派給 void 指標，再轉型為 int 指標：

```
int num;
int *pi = &num;
printf("Value of pi: %p\n", pi);
void *pv = pi;
pi = (int*) pv;
printf("Value of pi: %p\n", pi);
```

以上程式的輸出如下，指標值沒有改變：

```
Value of pi: 100
Value of pi: 100
```

void 指標用於資料指標而非函數指標，在第 202 頁《C 語言中的多型》一節中會介紹利用 void 指標處理多型行為。

使用 void 指標時要注意，將指標轉型為 void 指標後，void 指標能夠轉型為任何指標型別，沒有任何限制。

sizeof 運算子能夠用在 void 指標，卻不能用在 void 之上：

```
size_t size = sizeof(void*);          // 合法
size_t size = sizeof(void);           // 不合法
```

size_t 是用於表示大小的型別，在第 16 頁《內建指標型別》一節中會詳細介紹。

全域與靜態指標

當指標宣告為全域（global）或靜態（static）時，會在程式啟始時初始化為 NULL，以下是全域與靜態指標的範例：

```
int *globalpi;

void foo() {
   static int *staticpi;
   ...
}

int main() {
   ...
}
```