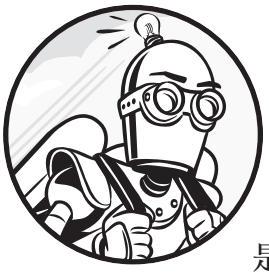
前言

拿起您的畫筆，和我們一起作畫。

——Bob Ross



系統程式設計的需求量非常龐大，隨著網頁瀏覽器、行動裝置、物聯網的普及，現在是成為一名系統程式設計師的最佳時機。高效率、可維護、和正確的程式碼在任何情況下都是迫切需要的，而我堅信 C++ 通常是最適合這項工作的語言。

在經驗豐富的程式設計師手中，C++ 可以產生比地球上任何其他系統程式語言更小、更有效率、更可讀的程式碼。它是最適合用來實現零間接成本抽象機制的語言，不但您的程式執行起來非常快，而且設計程式的時間也很短，並且可以簡單、直接地對應到硬體上，因此您可以在必要的時候進行低階的硬體控制。當您用 C++ 設計程式時，您就像是站在巨人的肩膀上，他們花了幾十年的時間來設計一種大而靈活的語言。

碁峰 www.gotop.com.tw

學習 C++ 有一個非常大的好處，您可以免費存取「C++ 標準函式庫」*stdlib*。stdlib 由三個環環相扣的部分所組成：容器、疊代器和演算法。如果您曾親自寫過自己的快速排序演算法，或者您寫過系統程式碼，並且被緩衝區溢位、懸置指標、釋放後使用、和雙重釋放所困擾，那麼您會很慶幸您熟悉 stdlib。它為您提供了型別安全性、正確性、和效率無與倫比的組合。此外，您會很樂意見到您的程式碼有多麼緊湊並且具有表現力。

C++ 程式設計模型的核心是**物件生命週期**，它為您提供了強而有力的保證，確保程式使用的資源（如檔案、記憶體和網路插槽）能正確釋放，即使在出現錯誤情況時也是如此。當有效地使用時，例外（exception）可以清除程式碼中大量的因錯誤條件檢查而造成的混亂。此外，移動／複製語意提供了安全、高效率和靈活的資源所有權管理方式，而早期的系統程式設計語言（例如 C）根本無法提供這種方式。

C++ 是一個活生生、會呼吸的語言；30 多年來，國際標準組織（International Organization for Standardization，ISO）的 C++ 委員會定期對該語言進行改善。在過去的十年中，已經數次發布了 C++ 標準的更新：分別於 2011 年、2014 年和 2017 年發布了 C++11、C++14 和 C++17。您可以期待在 2020 年有一個新的 C++20。

當我使用**現代 *C++*** 一詞時，我指的是最新標準的 C++ 版本，囊括了這些新增內容中所提供的功能和範式。這些更新對語言進行認真的改進，提高了它的表達能力、效率、安全性和整體可用性。從某種程度上說，這種語言從未像現在這樣廣受歡迎，而且也不會很快消失。如果您決定投資在學習 C++ 上，它將在未來幾年裏帶來回報。

關於本書

儘管現代 C++ 程式設計師可以有許多非常高品質的書籍可供選擇，例如 Scott Meyer 的《*Effective Modern C++* **中文版**》和 Bjarne Stroustrup 的《*The C++ Programming Language* **國際中文版 第四版**》，但

碁峰 www.gotop.com.tw

它們通常都相當進階而難懂。另外還有一些入門的 C++ 書籍可供選擇，但是它們經常略過關鍵的細節，因為它們是針對那些完全沒有程式設計經驗的讀者。對於有經驗的程式設計師來說，往往不清楚該從何處開始學習 C++ 語言。

我比較喜歡刻意地學習複雜的主題，並從它們的基本元素來建立概念。C++ 有著令人望而生畏的名聲，因為它的基本元素緊密地嵌套在一起，很難對這個語言進行完整的描述。當我學習 C++ 時，我大部分的心思花在書籍、影片和精疲力竭的同事之間疲於奔命，所以我寫了這本我希望五年前就有的書。

誰該閱讀本書？

這本書是為已經熟悉基本程式設計概念的中級到高級程式設計師所寫的。如果您沒有特別的系統程式設計經驗也沒關係，有經驗的應用程式設計師也很適合閱讀本書。

NOTE

如果您是一個經驗豐富的 C 程式設計師，或者是有理想有抱負的系統程式設計師，而您想知道是否應該投資於學習 C++ 上面，請務必閱讀第 xxxii 頁的「給 C 程式設計師的建言」，以便進行詳細的評估。

本書內容

這本書分為兩個部分，第一部分介紹了 C++ 語言的核心，而不是按照時間順序（從舊的 C++ 98 到現代 C++ 11/14/17）來呈現 C++ 語言，您將直接學習到現代 C++ 的習慣用語。第二部分介紹了「C++ 標準函式庫」（stdlib）的世界，在這部分您將學到最重要和不可或缺的概念。

碁峰 www.gotop.com.tw

第 1 部分：C++ 語言的核心

第 1 章：啟動和執行　這個入門的章節將幫助您建立一個 C++ 開發環境。您將編譯並執行第一個程式，並學習如何為它除錯。

第 2 章：型別　在這裡，您將探索 C++ 型別系統。您將學到基本型別，這是所有其他型別的基礎。接下來，您將瞭解普通舊資料（plain-old-data, POD）型別和功能齊全的類別（class）。您將深入研究建構子（constructor）、初始化和解構子（destructor）的角色。

第 3 章：參照型別　本章將介紹儲存其他物件記憶體位址的物件。這些型別是許多重要程式設計模式的基石，它們允許您產生靈活、高效率的程式碼。

第 4 章：物件生命週期　在儲存持續時間的上下文中繼續討論類別不變量和建構子。在介紹資源獲取即初始化（resource acquisition is initialization, RAII）範式的同時也談到瞭解構子。您將瞭解到例外以及它們如何強制類別不變量和補足 RAII。在討論了 move 和 copy 語意之後，您將探討如何使用建構子和指派運算子來操作它們。

第 5 章：執行時期的多型　這裡將介紹介面，這是一個程式設計的概念，允許您撰寫執行時期多型的程式碼。您將學到繼承和物件組合的基礎知識，這是如何在 C++ 中使用介面的基礎。

第 6 章：編譯時期的多型　本章將介紹範本（template），這是一種能讓您撰寫多型程式碼的語言特性。您還將探索概念（concept），這個語言特性會添加到未來的 C++ 版本中，並命名為轉換函式（conversion function），讓您能夠將物件從一種型別轉換為另一種型別。

第 7 章：運算式　現在您將深入研究運算元和運算子。在徹底理解了型別、物件生命週期和範本之後，您將準備進入 C++ 語言的核心元件，運算式是這個旅程的第一個的航站。

碁峰 www.gotop.com.tw

第 8 章：敘述　本章將探討組成函式的元素，您將學習到運算式敘述、複合式敘述、宣告敘述、反覆運算敘述和跳躍敘述。

第 9 章：函式　第 1 部分的最後一章將討論如何把敘述安排成工作單元。您將瞭解函式定義、傳回型別、多載解析、不定參數函式、不定參數範本和函式指標的細節。您還將學會如何使用函式呼叫運算子和 lambda 運算式建立可調用的使用者定義型別。您還將探索 `std::function`：一個提供制式容器來儲存可調用物件的類別。

第 2 部分：C++ 函式庫和框架

第 10 章：測試　本章向您介紹單元測試和模擬框架的奇妙世界。在學習 Boost Test、Google Test、Google Mock 等框架的同時，您還將練習測試驅動開發，並實作一個自動駕駛系統。

第 11 章：智慧型指標　本章解釋了 stdlib 所提供的處理動態物件所有權的特殊實用工具程式類別。

第 12 章：實用工具程式　在這裡，您將看到 stdlib 和 Boost 函式庫中用於處理常見程式設計問題的型別、類別和函式的概述。您還將學到資料結構、數值函式和亂數產生器。

第 13 章：容器　本章將介紹 Boost 函式庫和 stdlib 中幫助您組織資料的許多特殊資料結構。您將瞭解什麼是序列容器、關聯容器和無序關聯容器。

第 14 章：疊代器　這是您在上一章中所學到的容器和下一章的字串之間的介面。您將瞭解不同類型的疊代器，以及它們的設計如何為您提供難以置信的靈活性。

第 15 章：字串　本章將教您如何在單一容器家族中處理人類的語言資料。您還將瞭解字串中內建的特殊功能，這些功能可讓您執行一些常見任務。

碁峰 www.gotop.com.tw

第 16 章：串流　這裡將向您介紹支援輸入和輸出操作的主要概念。您將學到如何使用格式化和非格式化操作處理輸入和輸出串流，以及如何使用操縱器。您還將學習如何在檔案中讀寫資料。

第 17 章：檔案系統　在這裡，您將獲得 stdlib 中用於操作檔案系統功能的概述。您將學習如何建構和操縱路徑、檢查檔案和目錄以及列舉目錄結構。

第 18 章：演算法　這是一個數十個問題的快速參照，您可以輕鬆地利用 stdlib 內建的演算法來解決這些問題。您也將瞭解到您可使用的高品質演算法的強大範圍。

第 19 章：協作與並行　本章將教您一些簡單的多執行緒程式設計方法，這些方法是 stdlib 的一部分。您將瞭解什麼是未來（future）、互斥體（mutatex）、條件變數和原子。

第 20 章：Boost Asio 網路程式設計　在這裡，您將學習如何打造透過網路進行通信的高性能程式。您將看到如何使用 Boost Asio 與區塊式和非區塊式輸入和輸出。

第 21 章：撰寫應用程式　最後一章以幾個重要主題的討論來充實這本書。您將瞭解程式支援工具，這些工具允許您與應用程式生命週期掛鉤。您還將瞭解 Boost ProgrampOptions，這是一個函式庫，它讓撰寫接受使用者輸入的控制台應用程式變得簡單易懂。

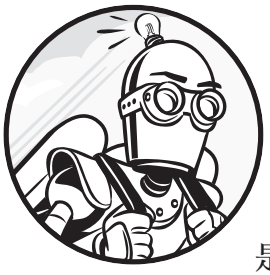
NOTE

造訪隨附網站 *https://ccc.codes/* 以取得本書中的程式碼列表。

碁峰 www.gotop.com.tw

給 C 程式設計師的開場序

亞瑟·丹特（*Arthur Dent*）：「他怎麼了？」
希格·赫特弗斯特（Hig Hurtenflurst）：
「他的鞋子不合腳。」
——道格拉斯·亞當斯（*Douglas Adams*），
《銀河便車指南》，“第十一集”



這篇開場序是為正在考慮是否要閱讀這本書，有經驗的 C 程式設計師所準備的，不是 C 程式設計師的讀者可以跳過這篇前奏。

比樣·史特勞斯特普（Bjarne Stroustrup）從 C 語言中發展出了 C++，雖然 C++ 與 C 完全不相容，但是 C 程式如果寫得很好的話通常也會是有效的 C++ 程式，舉例來說，在布萊恩·柯林漢（Brian Kernighan）和鄧尼斯·裡奇（Dennis Ritchie）的《*The Programming Language*》一書中，所有的範例都是合法的 C++ 程式。

C 語言能普遍存在於系統程式族群中的一個主要原因，是 C 語言允許程式設計師用比組合語言更高的抽象化層級來撰寫程式碼。這樣可以寫出更清晰、更不容易出錯、更易於維護的程式碼。

碁峰 www.gotop.com.tw

一般而言，系統程式設計師不願意為程式設計方便而支付額外的成本，因此 C 語言堅守了零額外開銷原則：**沒有用到的東西就不用支付**。強型別系統是零額外開銷抽象化的最典型例子，型別檢查只在編譯時用來檢查程式的正確性。編譯之後，型別將會消失，而編譯器所產生出的組合語言程式碼將找不到任何型別系統的蹤跡。

作為 C 的後裔，C++ 對於零開銷抽象化和硬體直接映射也非常重視。這種承諾不僅僅是 C++ 支援的 C 語言特性而已，在 C++ 上所建構的一切，包括新的語言特色，都支援這些原則，而且兩者之間的差異都是刻意設計的。事實上，有一些 C++ 的功能甚至比相對應的 C 程式碼帶來更少的間接開銷。關鍵字 `constexpr` 就是一個很好的例子，它指示編譯器在編譯時就（儘可能）把運算式的結果計算出來，如列表 1 中的程式所示。

列表 1：一個說明常數運算式（constexpr）的程式

```
#include <cstdio>

constexpr int isqrt(int n) {
  int i=1;
  while (i*i<n) ++i;
  return i-(i*i!=n);
}

int main() {
  constexpr int x = isqrt(1764); ❶
  printf("%d", x);
}
```

`isqrt` 函式會計算參數 `n` 的平方根，`isqrt` 函式中的迴圈會將區域變數 `i` 從 `1` 開始遞增，直到 `i*i` 大於或等於 `n` 為止。如果 `i*i==n`，則傳回 `i`；否則返回 `i-1`。注意，本例呼叫 `isqrt` 時的參數值是一個固定的字面值（如 ❶ 所示的 `1764`），因此理論上在編譯時就可以先幫您把結果計算出來。

碁峰 www.gotop.com.tw

在 GCC 8.3 上用 -O2 選項以 x86-64 為目標編譯列表 1，將產生列表 2 的組合語言程式。

列表 2：編譯列表 1 後產生的組合語言程式

```
.LC0:
        .string "%d"
main:
        sub     rsp, 8
        mov     esi, 42 ❶
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

這裡很明顯的結果是 main 中的第二行指令 ❶ ；編譯器不是在執行時才計算 1764 的平方根，而是在編譯時就計算好，並在輸出組合語言指令時，直接將 x 視為 42。當然，您可以用計算機算出平方根並手動插入結果，但是使用 constexpr 可以提供很多好處，這種方式可以減少許多因手動複製貼上而造成的錯誤，並且讓程式碼更具有表達能力。

NOTE

如果您不熟悉 x86 組合語言，請參閱 Randall Hyde 所著的《*The Art of Assembly Language*》第二版，以及 Richard Blum 所著的《*Professional Assembly Language*》。

升等到超級 C

現代 C++ 編譯器將滿足大多數 C 程式設計習慣，這讓我們得以輕鬆地接受 C++ 語言提供的一些策略細節，同時刻意避開語言更深層次的主題。這種 C++ 風格，我們稱之為**超級 C**（*Super C*），基於幾個理

碁峰 www.gotop.com.tw

由有進一步討論的必要。首先，經驗豐富的 C 程式設計師可以立即受益於將簡單、戰術等級的 C++ 概念應用到程式中。第二，超級 C 不是 C++ 的習慣用語。單純地在 C 程式中使用零散的 auto 參照和實例也許能夠讓程式碼更加可靠並增加一些可讀性，但是您需要學習其他概念才能充分利用它。第三，在一些嚴峻的環境（例如，嵌入式軟體、一些作業系統核心和異質架構的計算）中，可用的工具鏈並不完全支援 C++。在這種情況下，至少可以受益於一些 C++ 的技巧，而且很可能也可以支援超級 C。本節將介紹一些可以立即應用到程式碼中的超級 C 概念。

NOTE

有一些 C 語言所支援的寫法在 C++ 中並不能正常運作，請參閱本書附屬網站 *https://ccc.codes* 的連結部分。

函式多載

考慮以下標準 C 函式庫中的轉換函式：

```
char* itoa(int value, char* str, int base);
char* ltoa(long value, char* buffer, int base);
char* ultoa(unsigned long value, char* buffer, int base);
```

這些函式實現了相同的目標：將整數型別轉換為 C 風格的字串。在 C 語言中，每個函式都必須有一個唯一的名稱。但是對於 C++ 函式而言，只要它們的引數不同，就可以共用名稱，這就是所謂的**函式多載**（*function overloading*）。您可以利用函式多載來創建自己的轉換函式，如列表 3 所示。

列表 3：呼叫多載函式

```
char* toa(int value, char* buffer, int base) {
  --snip--
}
```


碁峰 www.gotop.com.tw

```
char* toa(long value, char* buffer, int base)
  --snip--
}

char* toa(unsigned long value, char* buffer, int base) {
  --snip--
}

int main() {
  char buff[10];
  int a = 1; ❶
  long b = 2; ❷
  unsigned long c = 3; ❸
  toa(a, buff, 10);
  toa(b, buff, 10);
  toa(c, buff, 10);
}
```

由於每個函式中第一個引數的資料型別不同，因此 C++ 編譯器從傳遞到 toa 的引數中可獲得足夠的資訊來呼叫正確的函式，而每個 toa 的呼叫都指向一個唯一的函式。在這裡，您建立了變數 a ❶、b ❷ 和 c ❸，它們是與三個 toa 函式之一對應的不同型別的 int 物件。這比定義單獨命名的函式更為方便，因為您只需要記住一個名稱，編譯器就會自動判斷要呼叫哪個函式。

參照

指標是 C 語言（擴展到大多數系統程式設計）的一個重要功能，它們可以讓您透過傳遞資料位址，而不是實際資料，來高效率地處理大量資料。指標對於 C++ 同樣重要，但您可以提供額外的安全性功能來防止參照到空值和不小心重新指派指標。

參照（*reference*）是 C++ 對於指標處理的一大改進，它們有點像指標，但有一些關鍵的區別。在語法上，參照與指標有兩大不同。首先，參照是用 & 而不是用 * 來宣告，如列表 4 所示。

碁峰 www.gotop.com.tw

列表 4：說明如何宣告帶有指標和參照的函式

```
struct HolmesIV {
  bool is_sentient;
  int sense_of_humor_rating;
};
void mannie_service(HolmesIV*); // Takes a pointer to a HolmesIV
void mannie_service(HolmesIV&); // Takes a reference to a HolmesIV
```

其次是使用點（.）運算子與成員互動，而不是用箭頭運算子（->），如列表 5 所示。

列表 5：說明點（.）和箭頭（->）運算子用法的程式

```
void make_sentient(HolmesIV* mike) {
  mike->is_sentient = true;
}

void make_sentient(HolmesIV& mike) {
  mike.is_sentient = true;
}
```

在底層機制上，參照相當於指標，因為它們也是零開銷抽象化的運算。編譯器會產生相似的程式碼。為了說明這一點，考慮使用 -O2 選項在 GCC 8.3 上針對 x86-64 機器編譯 make_sentient 函式的結果，列表 6 為編譯列表 5 所產生的組合語言程式。

列表 6：編譯列表 5 所產生的組合語言程式

```
make_sentient(HolmesIV*):
        mov     BYTE PTR [rdi], 1
        ret
make_sentient(HolmesIV&):
        mov     BYTE PTR [rdi], 1
        ret
```

然而，在編譯時，參照對原始指標提供了一些額外的安全性，因為一般來說，它們不能為 null。

碁峰 www.gotop.com.tw

如果使用指標，您可以添加 nullptr 檢查以確保安全。例如，您可以為 make_sentient 添加一個檢查，如列表 7 所示。

列表 7：重構列表 5 的 make_sentient，加上 nullptr 檢查

```
void make_sentient(HolmesIV* mike) {
  if(mike == nullptr) return;
  mike->is_sentient = true;
}
```

在使用參照時，這樣的檢查是不必要的；但是，這並不意味著參照總是有效的。考慮以下函式：

```
HolmesIV& not_dinkum() {
  HolmesIV mike;
  return mike;
}
```

not_dinkum 函式會傳回一個參照，這個參照保證不會是 null，但是它卻指向垃圾記憶體（可能在 not_dinkum 所傳回的呼叫堆疊中），也就是所謂的「**未定義執行時期行為**（*undefined runtime behavior*）」。您絕對不能這麼做，其結果將會是徹底的痛苦，可能會造成當機，也可能給您一個錯誤，或者可能做一些完全無法預料的事情。

參照的另一個安全性功能是它們不可能被**重新定址**（*reseated*）。換句話說，一旦參照被初始化，它就不能被更改為指向另一個記憶體位址，如列表 8 所示。

列表 8：說明參照不能重新定址的程式

```
int main() {
  int a = 42;
  int& a_ref = a; ❶
  int b = 100;
  a_ref = b; ❷
}
```


碁峰 www.gotop.com.tw

您將一個 a_ref 宣告為對 int a ❶ 的參照，此後 a_ref 即無法重新定址為指向另一個 int 的參照。您可能會試著像 ❷ 那樣用 operator= 來為 a_ref 重新定址，但這實際上是 b 的值指派給 a，而不是將 a_ref 的值改為參照到 b。執行以上程式碼片段後，a 和 b 都將等於 100，而 a_ref 仍然指向 a。列表 9 為使用指標的等效程式碼。

列表 9：列表 8 改為使用指標的等效程式碼

```
int main() {
  int a = 42;
  int* a_ref = &a; ❶
  int b = 100;
  *a_ref = b; ❷
}
```

在這裡，您用一個 * 來宣告指標，而不是 & ❶。您把 b 的值指派給一個由 a_ref ❷ 所指向的記憶體。使用參照的話，無需在等號的左邊加上任何修飾字元。但是如果省略了指標 *a_ref 中的 *，編譯器會抱怨您試圖將 int 指派指標型別。

引用只是帶有額外安全預防措施和少量語法糖的指標。當您把一個參照放在等號的左邊時，您實際上是把該參照所指向的位置設定為等號右邊的值。

自動初始化

C 經常要求您多次重複型別資訊，而 C++ 可以只用一次關鍵字 auto 就能表示變數的型別資訊。由於編譯器知道用來初始化一個變數的值是什麼型別，藉此就能推斷出該變數的型別。考慮以下 C++ 變數的初始化：

```
int x = 42;
auto y = 42;
```


碁峰 www.gotop.com.tw

在這裡，x 和 y 都是 int 型別。當您知道編譯器可以推斷出 y 的型別時可能會感到訝異，但是考慮到 42 是一個整數字面值，使用 auto 時編譯器會推斷等號（=）右邊的型別，並將變數的型別設定為與其相同的型別。由於本例中的整數字面值 42 是 int 型別，編譯器會推斷 y 的型別也是 int。在這麼簡單的例子中，似乎看不出有什麼好處，但是考慮用函式的傳回值來初始化一個變數，如列表 10 所示。

列表 10：一個用函式的傳回值來初始化變數的玩具程式

```
#include <cstdlib>

struct HolmesIV {
  --snip--
};

HolmesIV* make_mike(int sense_of_humor) {
  --snip--
}

int main() {
  auto mike = make_mike(1000);
  free(mike);
}
```

auto 關鍵字比明確宣告變數的型別更容易閱讀，也更有利於程式碼重構。如果您在宣告函式時無限制地使用 auto，那麼以後如果您需要更改 make_mike 的傳回型別時，所要做的工作會更少。至於更複雜的型別那就更適合使用 auto 了，例如 stdlib 中那些充斥著範本的程式碼，使用 auto 關鍵字可以讓編譯器為您完成所有型別推斷的工作。

NOTE

您還可以為 auto 加上 const、volatile、&、* 等限定詞。

碁峰 www.gotop.com.tw

struct、union、enum 的命名空間和隱含式 typedef

C++ 將型別標籤視為隱含式的 typedef 名稱。在 C 語言中，當您想使用 struct、union 或 enum 時，必須以關鍵字 typedef 指定一個名稱給所創建的型別，例如：

```
typedef struct Jabberwocks {
  void* tulgey_wood;
  int is_galumphing;
} Jabberwock;
```

在 C++ 領域，您看到這樣的程式碼會忍不住哈哈大笑。因為關鍵字 typedef 是隱含的，您根本就不用寫出來，C++ 可以讓您像以下這樣宣告 Jabberwock 型別：

```
struct Jabberwock {
  void* tulgey_wood;
  int is_galumphing;
};
```

這樣更方便，而且還可以節省一些打字的時間。如果您還想定義 Jabberwock 函式，會發生什麼事？好吧，您不應該這麼做，因為對一個資料型別和一個函式重複使用相同的名稱，可能會造成混淆。但是如果您真的堅持要這麼做，C++ 允許您宣告一個命名空間（namespace），為識別字建立不同的範圍，這有助於保持使用者型別和函式的整潔，如列表 11 所示。

列表 11：使用 namespace 消除同名函式和型別的歧義

```
#include <cstdio>

namespace Creature { ❶
  struct Jabberwock {
    void* tulgey_wood;
    int is_galumphing;
  };
}
```


碁峰 www.gotop.com.tw

```
namespace Func { ❷
  void Jabberwock() {
    printf("Burble!");
  }
}
```

在本例中，Jabberwock 結構和 Jabberwock 函式現在和諧地共存。透過將每個元素放在其自己的 namespace（Creature 命名空間 ❶ 中的 struct 和 Func 命名空間 ❷ 中的函式）中，可以消除您所指的是哪一個 Jabberwock 的歧義。您可以用幾種方法消除歧義，最簡單的方法是用 namespace 來限定名稱，例如：

```
Creature::Jabberwock x;
Func::Jabberwock();
```

您還可以利用 using 指令來匯入 namespace 中的所有名稱，這樣就不再需要每次使用都要打完整的限定元素名稱，清單 12 使用了 Creature namespace。

列表 12：藉由使用 namespace 來參照 Creature 命名空間中的型別

```
#include <cstdio>

namespace Creature {
  struct Jabberwock {
    void* tulgey_wood;
    int is_galumphing;
  };
}

namespace Func {
  void Jabberwock() {
    printf("Burble!");
  }
}

using namespace Creature; ❶
```


碁峰 www.gotop.com.tw

```
int main() {
  Jabberwock x; ❷
  Func::Jabberwock();
}
```

using namespace ❶ 允許您省略 namespace 限定詞 ❷。但是使用 Func::Jabberwock 時您仍然需要加上限定詞，因為它不是 Creature namespace 的一部分。

namespace 的使用是 C++ 的習慣用法，這是一個零開銷抽象化的語法。就像其他的型別識別字一樣，編譯器在產生出組合語言程式碼時會刪除 namespace。在大型專案中，它對於分隔不同函式庫中的程式碼非常有用。

夾雜 C 和 C++ 物件檔案

如果您小心的話，可以讓 C 和 C++ 程式碼和平共存。有時，C 編譯器有必要連結到由 C++ 編譯器所產生出的目的檔（反之亦然）。雖然這是可能的，但需要做一些額外的工作。

有兩個和連結這些檔案有關的問題。首先，C 和 C++ 程式碼中的呼叫慣例可能會不匹配。例如，呼叫函式時如何設定堆疊和暫存器的協定可能有所不同。這些呼叫慣例的不匹配是屬於語言層級的，通常與撰寫函式的方式無關。第二，C++ 編譯器產生出不同於 C 編譯器的符號。有時連結器必須透過名稱來識別物件，C++ 編譯器可以透過對物件進行裝飾來協助，將一個**裝飾名稱**（*decorated name*）與物件產生關聯。由於函式多載、呼叫慣例、和 namespace 的用法，編譯器必須透過裝飾對函式的附加資訊進行編碼，而不僅僅是其名稱。這樣做是為了確保連結器可以唯一地識別函式。遺憾的是，C++ 中對於這樣的裝飾並沒有制定標準（這就是為什麼在連結不同的翻譯單元時必須使用相同的工具鏈和設定）。C 連結器對 C++ 名稱裝飾一無所知，如果在 C++ 中連結 C 程式碼時沒有抑制裝飾，可能會造成問題（反之亦然）。

碁峰 www.gotop.com.tw

解決辦法很簡單。使用 extern "C" 敘述包裝要用 C 風格連結編譯的程式碼，如列表 13 所示。

列表 13：使用 C 風格的連結

```
// header.h
#ifdef __cplusplus
extern "C" {
#endif
void extract_arkenstone();

struct MistyMountains {
  int goblin_count;
};
#ifdef __cplusplus
}
#endif
```

這個標頭可以在 C 和 C++ 程式碼之間共用。它之所以奏效是因為 __cplusplus 是 C++ 編譯器所定義的一個特殊識別字（但是 C 編譯器則沒有）。因此，在前置處理完成後，C 編譯器會看到剩下的程式碼如表 14 所示。

列表 14：C 環境中前置處理器處理列表 13 後剩下的程式碼

```
void extract_arkenstone();

struct MistyMountains {
  int goblin_count;
};
```

這只是一個簡單的 C 標頭檔。在前置處理期間，#ifdef __cplusplus 敘述之間的程式碼將被刪除，因此看不到 extern "C" 包裝器。對於 C++ 編譯器而言，__cplusplus 是在 header.h 中定義，因此它會看到列表 15 的內容。

碁峰 www.gotop.com.tw

列表 15：前置處理程式在 C++ 環境中處理列表 13 之後的程式碼

```
extern "C" {
  void extract_arkenstone();

  struct MistyMountains {
    int goblin_count;
  };
}
```

extract_arkenstone 和 MistyMountains 現在都用 extern"C" 包裝，因此編譯器知道要使用 C 來連結。這麼一來，您的 C 原始程式碼就可以呼叫編譯過的 C++ 程式碼，而 C++ 原始程式碼也可以呼叫編譯過的 C 程式碼。

C++ 主題

本節將簡短的介紹一些讓 C++ 成為最重要的系統程式語言的核心主題。不用太擔心細節，以下各小節主要是想提供您幾樣開胃小菜。

簡潔地表達想法以及重複使用程式碼

精心撰寫的 C++ 程式碼具有優雅、緊湊的效果。考慮以下簡單的操作從 ANSI-C 進化到現代 C++ 的寫法：用迴圈走訪 n 個元素的陣列 v，如列表 16 所示。

列表 16：用迴圈走訪陣列元素的幾種寫法

```
#include <cstddef>

int main() {
  const size_t n{ 100 };
  int v[n];

  // ANSI-C
  size_t i;
  for (i=0; i<n; i++) v[i] = 0; ❶
```


碁峰 www.gotop.com.tw

```
  // C99
  for (size_t i=0; i<n; i++)  v[i] = 0; ❷

  // C++17
  for (auto& x : v) x = 0; ❸
}
```

這個程式碼片段顯示了在 ANSI-C、C99 和 C++ 中宣告迴圈的不同寫法。ANSI-C ❶ 和 C99 ❷ 中的索引變數 i 是用來協助您走訪陣列 v 的每個元素。而 C++ 版本 ❸ 使用了「**以範圍為基礎的** *for*」（*range-based for*）迴圈，在陣列 v 的範圍內執行迴圈，同時隱藏了反覆運算的實作細節。與 C++ 中的許多零開銷抽象化方法一樣，這種結構讓您能夠把注意力放在想要表達的意思而不是語法上。以範圍為基礎的 for 迴圈可用於許多型別，甚至可以跟使用者定義的型別一起使用。

講到使用者定義的型別，它們可以讓您直接用程式碼來表達想法。假設您想要設計一個函式，navigate_to，用來把假想的機器人導航到給定的 x 和 y 座標位置。考慮以下雛型函式：

```
void navigate_to(double x, double y);
```

x 和 y 是什麼？他們的單位又是什麼？使用者必須閱讀說明檔（或者可能是原始程式碼）才能找到答案。比較以下改進的雛型：

```
struct Position{
--snip--
};
void navigate_to(const Position& p);
```

這個函式要清楚得多，對於 navigate_to 所接受的參數沒有任何模糊之處。只要您有一個有效建構的 Position，您就知道如何呼叫 navigate_to。現在，無論是單位或轉換等問題，都留給建構 Position 類別的人去擔心。

碁峰 www.gotop.com.tw

您也可以在 C99/C11 中利用 `const` 指標來達到像這樣清晰的效果，但是 C++ 還可以讓傳回型別緊湊而富有表達能力。假設您想為 robot 撰寫一個叫做 `get_position` 的推論函式來取得位置資訊。在 C 中，有兩個選項，如列表 17 所示。

列表 17：C 風格的 API，用來傳回使用者定義的型別

```
Position* get_position(); ❶
void get_position(Position* p); ❷
```

在第一個選項中，呼叫者要負責清理傳回值 ❶，這可能需要動用到動態記憶體分配（儘管由程式碼中看不出來這一點）。呼叫者負責把 `Position` 分配在記憶體中某個位址並將其傳給 `get_position` ❷。

後一種方法更像 C 風格的慣用寫法，但是這樣會對您造成困擾：您只是想要取得 Position 物件，但您必須擔心呼叫者或被呼叫函式到底誰應該負責分配和釋放記憶體。C++ 透過直接由函式傳回使用者定義型別的方式，可以簡潔地完成所有這些操作，如列表 18 所示。

列表 18：在 C++ 中以傳值方式傳回使用者定義型別

```
Position❶ get_position() {
  --snip--
}
void navigate() {
  auto p = get_position(); ❷
  // p is now available for use
  --snip--
}
```

由於 `get_position` 傳回的是一個值 ❶，編譯器可以省略複本（*elide the copy*），所以就好像直接建構了一個自動 `Position` 變數 ❷，而且沒有執行時期的額外開銷。就功能上而言，這種做法與列表 17 中 C 風格的傳遞參照（pass-by-reference）方式非常類似。

碁峰 www.gotop.com.tw