自序

本書問世以來受到廣大的迴響，不斷有讀者經由 email 寫信鼓勵及提問。這次大幅改版反映了這些年來讀者的建議和指正，同時也因應 Microsoft Visual Studio 與日俱增的重要性，以及使用 Dev-C++ 的實際需要，做了必要的更新。

C++ 是第一個被廣泛使用的物件導向程式語言 (Object-Oriented Programming Language)，本書提供一個使用 C++ 程式語言的堅實基礎，內容不僅涵蓋了最基本的語法，也深入探討了物件導向的主要精神。

本書的撰寫動機，在於希望貢獻一本「如果當年我學習 C++ 時，有這麼一本書該多好」的經典。雖然目前國內已有為數不少關於 C++ 的中文著作和翻譯書籍，但這些書籍在敘述的方式，以及對於內容的取捨等與讀者最相關的本質上，並不切合實際的需要。許多讀者半途而廢，甚至在 C++開發環境下捨棄 C++ 的強大功能，繼續延用較易理解的 C 語言。也有讀者雖然勉強讀完，卻發現說明和舉例過於簡略、避重就輕，除了書上的例題外，仍然舉步維艱，無法用來解決自己實際面對的問題。

為了切合國內讀者的需要，本書採取簡明易懂的敘述方式，並透過精心的安排的大量例題，每學完一章都可據以完成實用的程式。例如：如何避免語法和語意的錯誤，如何使用前處理指令，如何產生亂數，如何估計程式運算所花費的時間，如何從既有檔案讀取資料，如何將執行結果存檔，如何使用物件來模擬實際的互動關係，等等問題都可在本書內找到清楚解答。本書內附超過 180 個完整的範例程式，全部經過符合最新 ANSI / ISO 標準的 C++編譯器測試，包括：Microsoft Visual C++ Community 2015，Dev-C++ 5.11，以及 Borland C++ Compiler 5.5.1，並能正確執行。

除了清楚完整的範例程式之外，我們對於程式實際的運作機制，譬如函數間互相呼叫的詳細過程，宣告 (declaration) 和定義 (definition) 的區別，做為多型 (polymorphism) 基礎的晚期聯結 (late binding)，虛擬函數，VPTR 以及 VTABLE 等等主要的觀念都有清楚說明。

此外，在開發一個龐大的應用程式時，基於降低開發難度，提高再利用率，利於爾後程式的修訂維護，以及使架構合理化等因素，實際上無法將全部程式寫成一個檔案。本書詳細說明如何將程式區分為許多小檔案的技巧。

中文科技圖書中的英文名詞翻譯經常造成學習的困擾。由於 C++ 語言的主體原本就是用英文字彙或是其縮寫所構成，強加翻譯常造成閱讀的困難，也不利於程式的撰寫。因此，本書對於 C++ 的關鍵字在本文中儘量中英文並列，便於學習。為了查閱方便，我們在附錄中提供主要名詞的中英翻譯對照表，並在書末索引中列出所有的專有名詞、C++ 的關鍵字和特殊符號，並標示其所在的頁數，可以順暢的進行學習。

本書共分四篇：第一篇「C++程式語言基礎」，介紹電腦的基本架構、C++的基本語法、資料型態和使用環境，選擇和重複處理兩種程式流程控制語法，以及函數和陣列等等主題。學習完這個部份，就可以使用程序式程式設計（procedural programming）的語法處理許多問題。

第二篇「進階 C++ 程式語言」，介紹指標、字串、函數的進階應用、前處理指令、資料流與檔案的存取、輸出格式、程式計時、struct 與資料結構、名稱空間以及異常處理等主題。學習完這個部份，就可以具備使用檔案存取資料，自由設定資料格式，並將大型程式區分為許多小檔案，以解決實際問題的能力。在第九章「字串」中，我們有一個關於編碼的有趣程式，讀者可以用來把電子郵件轉成只有擁有破解碼才能理解的文字，便於機密文件的傳遞。

第三篇「物件導向程式設計」，討論類別與物件、組合與繼承、多型與虛擬函數、運算子重載、樣版類別，和泛型程式設計等主題，以循序漸進的方式介紹封裝 (encapsulation) 、繼承 (inheritance) 和多型 (polymorphism) 三種物件導向語言的主要技術。在這個部份中，我們藉助許多有意義的範例程式解說了向上轉型 (upcast)，抽象化，衍生類別所定義的物件之建構和解構次序，混合組合和繼承以建立新的類別，重載虛擬函數，虛擬解構函數，等等被大部份介紹 C++ 的書籍所忽略的重要主題。在第 21 章中，我們藉由複數演算 (特別是交流電路阻抗的計算) 的實例展示了「運算子重載」在簡化程式上的強大功能。

第四篇「數值運算的應用」，我們使用 C++ 以實際的範例來處理最佳化問題和常微分方程式的數值解。許多 C++ 的使用者到目前為止都沒有使用 C++ 物件導向的功能以處理數值問題的經驗。對於需要耗費大量時間的數值問題，最通常的情況是使用 C、FORTRAN 或是 MATLAB。我們在這兩章中分別透過 simplex 最佳化計算，以及常微分方程式的數值解，充份顯示 C++ 不僅可以有 MATLAB 的方便性，同時具有 C 和 FORTRAN 的效率。

本書所附的光碟中載有書內所有範例程式的原始程式碼，以章節編號分別存放在易於搜尋的檔案夾中。這些原始程式碼可以自由修改編譯，以切合自己的需要。除了原始程式碼外，光碟中還附有已編譯過的執行檔，可以馬上在個人電腦上體驗執行的情況。從光碟中，你可以找到：(1) Borland C++ Compiler 5.5.1，和 (2) Dev-C++ 5.11 兩種完整而合法的 C++ 程式開發工具。只要有一台個人電腦，在不需要額外添購任何配備的情況下，立即可以展開 C++ 的學習之旅。

這次改版承蒙研究助理楊曜鴻的大力協助，以及洪彗文小姐、謝天揚先生多年來的鼓勵，機電整合實驗室的同學：許茗晞、曹逸祥、張祐誠、彭明杰、翁茂耘、吳昱成、莊舜中、林聖華，更是貢獻良多。本書也得力於海內外的熱心讀者甚多，例如: 施佩汝、唐平波、李小兵、黃園斌、周國旗，以及很多未具名的朋友。特別感謝吳俊仲和王能治兩位教授的仔細校閱，同事孫明宗教授的討論，以及侯榮富博士和蔡志仁博士在校稿和版面上的熱心協助。研究生邱俊傑、楊文裳、黃裕暉、陳信誌和黃銘宏，以及許建豐先後在校稿和繪圖上也提供了很多協助。

本書在教學研究之餘寫成，難免疏漏，讀者有任何指正，盼望透過電子郵件與我聯絡，當在再版時更正。

張耀仁 謹誌

Email：zen@mail.cgu.edu.tw

CHAPTER

6

函數

故用兵之法，十則圍之，五則攻之，倍則分之，
敵則能戰之，少則能逃之，不若則能避之。
故小敵之堅，大敵之擒也。

《孫子兵法．謀攻篇》

函數的使用是**模組化程式寫作** (modularized programming) 的重要方式。以硬體為例，我們已熟知一部汽車可以分為引擎、傳動、剎車、懸吊、ABS、空調和轉向等**模組** (module)，各模組各司其職，互相協調，而能共同組成一部安全舒適的交通工具。採用相同的想法，我們也可以把許多重要的資料處理程序區分出來，分別包裝成**函數** (function) 的型式，獨立進行開發和測試，再以呼叫的方式使用，降低程式開發的難度。

6.1 函數的基本概念

使用函數的目的

「把許多相關的程式指令放在一起，並加以命名，自成一個執行單元」是個非常有用的程式寫作方式。C++ 稱呼這種執行單元為 **function** (**函數**，或稱為**函式**)。其他的語言，例如 FORTRAN 和 BASIC 則分別稱之為 subroutine 和 subprogram (**副程式**)。使用 function 可使程式區分出許多功能明確的組成部分，有系統地進行程式寫作，減少錯誤。

事實上，我們在第三章 3.4 節裏已初步使用過 pow()、abs()和 fabs()等數學運算上不可或缺的**函數**，而主程式 main()本身便是一個標準的函數。函數使用有兩大目的：

1. 減少重複撰寫類似功能的程式 - 將常用的演算法包裝成函數，以利重複使用，簡化程式的撰寫。
2. 易於除錯和維護 - 把程式的各主要部份區分成幾個模組，可以分別進行開發和測試。

函數的語法

函數間可以互相呼叫。發起呼叫動作的函數稱為**呼叫函數** (calling function)，被呼叫的函數稱為**被呼叫函數** (called function) 。以程式開發的觀點來看，函數的語法可以分為下列三個部份：

1. **函數的宣告**
   建立一個函數的**原型** (prototype)，以告知編譯器本程式即將使用的函數名稱，以及進出這個函數的資料之型態和數量。
2. **函數的定義**
   將函數的內容具體地寫成程式。
3. **函數的呼叫**
   使用已經定義過的函數。

範例程式

下列程式 TempConv.cpp 是一個使用自定函數的簡單程式，包括函數 C2F () 的宣告、定義及使用。這個程式改寫自 5.5 節的程式 Temp.cpp，其功能為產生一個攝氏與華氏溫度的對照表：

```
// TempConv.cpp
#include <iomanip>
#include <iostream>
using namespace std;
// --- 函數 C2F () 的宣告 -------------------------
double C2F (double) ;
```

```
//------ 主程式 -------------------------------
int main ()
{
   double CTemp;
   cout << "  攝氏   華氏 " << endl ;
   cout << "-----------------"  << endl ;
   for ( int i = 1 ; i <= 10 ; i++ )
   {
     CTemp = 10.0*i;
     cout << setw (5) << CTemp << "  "
          << setw (5) << C2F (CTemp) << endl ;
   }
   cout << "-----------------"  << endl ;
   system("PAUSE");
   return 0;
}
// -------- 函數 C2F () 的定義 --------------------
double C2F (double C)
{
   double F;
   F = C*9.0/5.0 + 32.0;
   return F;
}
```

程式操作結果

```
  攝氏   華氏
-------------
   10    50
   20    68
   30    86
   40   104
   50   122
   60   140
   70   158
   80   176
   90   194
  100   212
-------------
```

從程式 Tempconv.cpp，我們可以清楚區分函數 C2F 的宣告、定義和使用：

1. 函數 C2F () 的宣告

函數在呼叫使用前要先宣告，這點和變數在使用前需宣告的情形非常類似。函數的**宣告** (declaration) 又稱為函數的**原型** (prototype)，它宣告了函數在呼叫時應該給予的資料 (稱為**引數**) 之**資料型態** (如果資料的數目不止一個，則各資料的給予次序也必須一致)，以及執行後，所返回的資料型態和次序。例如，Tempconv. cpp 中的函數原型

```
double C2F (double) ;
```

宣告了函數 C2F() 期望收到一個 double 數值，且將返回一個 double 數值，具體地表示成以下的圖形：

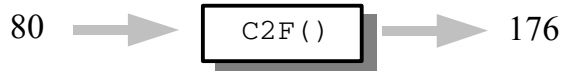

圖 6.1.1 函數使用示意圖

函數原型的通式為：

返回資料型態 **函數名稱** (參數資料型態列)；

其中的小括號 () 稱為**函數呼叫運算子** (function call operator)。**函數原型**的置放位置通常在主程式 main() 之前，可供主程式和同檔案的其他函數呼叫，如果放在 main() 的主體大括號 {} 內，則只能從 main() 內呼叫。

以下列出幾種常見的函數原型：

```
int Warning ();
int MaxInt (int, int, int);
int Area (int width, int Length);
Area_2 (int, int) ;
void Swap (double x, double y);
```

從上述五例中，我們可以發現**參數資料型態列** (list of parameter data types) 中可以沒有任何資訊或符號。例如

```
int Warning ();
```

代表不需要在呼叫時給予任何資訊。但是它也可以有一個以上的參數。例如：

```
int MaxInt (int, int, int);
```

在參數資料型態列中的各個資料型態後面也可以加入**參數名稱**以幫助使用者了解。以本例而言，函數 C2F() 的原型可以寫成

```
double C2F (double C);
```

返回資料型態只能有一個。如果執行後不返回任何資料，則以 void 標示。例如：

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

此外，某些編譯器 (例如：Borland C++)，int 為預設的**返回資料型態**；如果返回的資料型態為 int，就可以忽略不寫。例如：

```
Area_2 (int, int);
```


提示

由於主程式 main () 本身亦為函數，因此在 Borland C++ 中可以忽略 main 前面的 int 字樣。

2. 函數 C2F () 的定義

將函數處理資料的細節寫成程式稱為**函數的定義** (definition of a function)。函數一旦完成定義就可以隨時呼叫使用。函數定義的語法和函數的宣告一樣，只是把原先**函數呼叫運算子 ()** 後面的**空敘述**「;」(null statement) 換成具有完整內容的**複合敘述** (compound statement)，而且把只有資料型態的參數資料型態列加入完整的引數名稱成為**參數列** (parameter list)。

此外，每一個函數，包括 main()，地位都是相等的，因此，不可以把函數的定義置於另一個函數的定義內。

函數定義的一般語法如下所示：

```
返回資料形態  函數名稱 (參數列)              // 函數的標頭列
{
      主體敘述
```

```
        return 傳回值;
    }
```

「返回資料形態 函數名稱 (參數列)」的部份稱為函數的**標頭列** (header line)，而「{ 主敘述體 return 傳回值;}」的部份稱為函數的**本體** (function body)。

例如，程式 TempConv.cpp 中的 C2F() 函數定義寫成：

```
double C2F (double C)          // 函數的標頭列
{
   double F;
   F = C*9.0/5.0 + 32.0;          函數的本體
   return F;
}
```

這個函數非常簡單，甚至可以不用定義變數 **F**，而直接將函數定義進一步簡化為

```
double C2F (double C)  { return C*9.0/5.0 + 32.0; }
```

提示 由於 2 的英文發音與 to 相同，因此函數名稱通常採用簡化的命名方式，以 C2F 表示 C 和 F 之間的轉換 (代表 Celsius to Fahrenheit)。

3. 函數 C2F () 的呼叫和使用

呼叫函數時，必須給予合於參數列規定的資料，稱為**引數** (arguments)。從返回數值的有無，函數的呼叫可以分為下列兩類：

(1) 沒有返回資料時，單是**函數名稱**及**函數呼叫運算子** () 以及呼叫運算子內的**引數**，就可自成一個完整的敘述。例如：

```
swap (a, b) ;
```

(2) 如果有返回資料，則可以將函數的呼叫放在任何適合其資料型態的敘述中。例如，**指派敘述** (assignment statements) 或**算術敘述** (mathematical statements)：

```
N = MaxInt (P, q, r) ;      // 指派敘述
M = 2.5 * Area (W,L) ;      // 算術敘述
cout << "Temperature is" << C2F (Temp)
     << " Fahrenheit"      << endl;
```

討論

參數 (parameter) 和 引數 (argument)

在函數的**宣告**和**定義**中，參數列中所宣告的變數稱為**參數** (參數也稱為「**型參**」)。一般而言，參數是函數本體內的**局部變數**。

當使用函數時，在函數的**呼叫敘述**中所傳遞的值稱為**引數** (引數也稱為「**實參**」)。引數可以是數值或常數，也可以是已經有確定值的變數或敘述。在進行函數的呼叫時，各參數分別獲得各引數的值。例如，在上述的例子中：

swap (a,b);

此呼叫敘述中的 a 和 b 都是**引數**，而

double C2F (double C) {...}

函數定義式中的 C 是**參數**。

傳值呼叫

當函數被呼叫時，**引數**首先被複製到記憶體的一個特殊區域，稱為**堆疊** (stack) (下圖中的步驟 1)，且程式的執行次序和控制權都轉交給被呼叫的函數。如下圖所示 (在這裏 main() 為**呼叫函數**，C2F() 為**被呼叫函數**)：

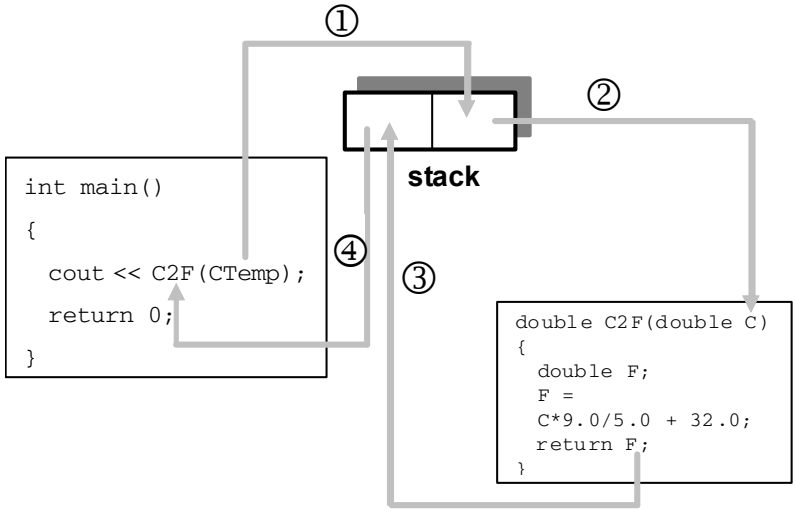

圖 6.1.2 函數引數的傳遞方式 (傳值)

這種引數傳遞的方式叫做**傳值** (pass by value) 或**以值呼叫** (call by value)。被呼叫的函數先將**參數** (double C) 初始化為**堆疊**內的值 (上圖中的步驟 2)，才開始進行運算。在運算完畢後 (執行完 return F; 或已經到達被呼叫的函數結尾的地方)，才將結果複製一份到堆疊內 (上圖中的步驟 3)，接著將控制權交還給呼叫函數。呼叫函數這時再把堆疊內的結果複製一份回來 (上圖中的步驟 4)，並恢復原先的執行次序。在本例中呼叫函數為 main()，但也可以是別的函數，也就是說**函數可以呼叫函數**，執行的步驟和上述程序相同。

由於這種傳值的呼叫方式，**引數**的值被層層複製，因此**被呼叫函數**內對**參數**的任何更動都不會影響**呼叫函數**內的**引數**。例如，把程式改寫成：

```
double C2F (double C)
 {
    double F;
    F = C*9.0/5.0 + 32.0;
    C = 3.0;        // 更動 C 的值
    return F;
 }
```

則主程式內的 CTemp 並不會跟著改變。也就是說，藉由傳值的機制，所有函數的輸出和輸入都在嚴格的控制之下，不會造成意料之外的更動。

不使用函數原型（prototype）的語法

函數原型 (prototype) 的目的在於**宣告**一個尚未定義但即將使用的函數。如果我們把函數定義置於函數原型的位置 (亦即在 main() 之前)，則此函數定義就兼具**宣告**和**定義**的功能，不需要再使用函數原型。

例如以下的程式 TempConv2.cpp 和原先程式 TempConv.cpp 在功能上沒有任何差異。

範例程式 檔案 TempConv2.cpp

```
// TempConv2.cpp
#include <iomanip>
#include <iostream>
using namespace std;
double C2F (double C)  //函數 C2F () 的定義，兼具宣告的功能
{ double F;
   F = C*9.0/5.0 + 32.0;
   return F;
}
//----- 以下為主程式 --------------------
int main ()
 {
   double CTemp;
   cout << "  攝氏   華氏 " << endl ;

   cout << "-----------------"  << endl ;
   for ( int i = 1 ; i <= 10 ; i++ )
   { CTemp = 10.0*i;
      cout << setw (5) << CTemp << "  "
           << setw (5) << C2F (CTemp) << endl ;
   }
   cout << "-----------------"  << endl ;
   system("PAUSE");
   return 0;
 }
```

函數內 return 的功能

我們在這裏藉由另一個範例程式討論函數內「return 敘述」(return statement) 的功能。下列完整程式 SeasonsFnc.cpp 改寫自第四章 Seasons.cpp。這個程式把計算季節的部份寫成函數 CheckSeason ()：

範例程式 檔案 SeasonsFnc.cpp

```
// SeasonsFnc.cpp
#include <iostream>
using namespace std;
// 以下為函數 CheckSeason () 的宣告
void CheckSeason (int) ;
//------ 主程式 ----------------------------------
```

```
int main ()
{
  int M;
  cout << "\n"
       << "請輸入一個月份  :  " << endl;
  cin   >> M;
  CheckSeason (M) ;
  system("PAUSE");
  return 0;
}
// --- 以下為函數 CheckSeason () 的定義 ------------
void CheckSeason (int Month)
{
  if (Month < 1 || Month >12)
    { cout <<  "您輸入的月份沒有意義!";    return;   }
  cout << "\n" << Month << "月是";
  switch ( (Month%12) /3)
  {
    case 0 :
      cout << "冬季" << endl;
      break;
    case 1 :
      cout << "春季" << endl;
      break;
    case 2 :
      cout << "夏季" << endl;
      break;
    case 3 :
      cout << "秋季" << endl;
      break;
    default :
      cout << "程式有問題!"  << endl;
   }
   return;   // 此指令可以省略
}
```

討論

函數 CheckSeason () 中使用了兩個「return;」。第一個 return 放在 if 敘述中，一旦成立，(Month < 1) 或 (Month >12)，便返回呼叫函數 main()。這種寫法避免了第四章 Seasons.cpp 中使用 if-else 敘述將 switch 敘述包起來的做法，也不需要如 SeasonsGoTo.cpp 中使用 goto 敘述的寫法，但可以獲得完全一樣的結果。

此外，如果函數的返回資料型態為 void，則此 return 指令可以省略。

6.2 以參照的方式呼叫（call by reference）

使用上一節的方式，我們已經可以寫出很多實用的程式。但是，基本的函數受到只能傳回一個數值的限制。此外，有些時候我們希望**引數**的數值也能跟著計算結果而改變。要突破這些限制，可以使用**參照** (reference) 的技巧。

參照是同一個變數的**別名** (alias)。例如，下列兩個敘述：

```
int N;
int& M = N;
```

定義了變數 N，以及 N 的**參照** M。此外這裏 & 稱為**參照運算子** (reference operator)，表示 M 和 N 所代表的變數位於記憶體裏面的同一個位址。對於 M 所做的變化都等同於直接作用在 N 上，反之亦然。「`int& M = N;`」這個定義**參照**的敘述也可以寫成：

```
int &M = N;
```

也就是說，下面兩個函數的**標頭列** (header line) 是一樣的：

```
int Fnc (int& N; double& x)
int Fnc (int &N; double &x)
```

提示

上面介紹的**參照運算子**和**取址運算子** (address operator，將在第 8 章中介紹) 都使用相同的符號「&」，但是它們的用法和意義都不一樣。例如：

```
int N;
int M = int (&N) ;
```

所代表的意義是：「把 N 的位址 (也是 32 位元大小) 存放到整數 M 裏」。注意，此處不能寫成

```
int N;
int M = &N;
```

因為 &N 的資料型態是 int*，而 C++ 沒有從 int* 到 int 的自動轉換，必須採取**顯式資料型態轉換** (explicit type conversions)，讀者可參考 3.3 節。

在下列程式 Alias.cpp 中，我們定義了變數 N，以及 N 的參照 M，以了解它們之間的關係。

範例程式 檔案 Alias.cpp

```
// Alias.cpp
#include <iostream>
using namespace std;
// ------ 主程式 ------------------------
int main ()
 {
    int N = 10;
    int& M = N;
    cout << "M 的值原來是："  << M << endl;
    cout << "N 的值原來是："  << N << endl;
    N = 5;
    Cout << "執行 「N = 5;」 之後" << endl;
    Cout << "M 的值目前是："  << M << endl;
    M = 2;
    Cout << "執行 「M = 2;」 之後" << endl;
    Cout << "N 的值目前是："  << N << endl;
    system("PAUSE");
    return 0;
 }
```

操作結果

```
M 的值原來是：10
N 的值原來是：10
執行「N = 5;」之後
M 的值目前是：5
執行「M = 2;」之後
N 的值目前是：2
```

由於參照所代表的是同一個變數，因此，6.1 節程式 TempConv.cpp 中的函數 C2F() 也可以進一步改寫，使用參照將數值傳回。如下列程式 TempConv3.cpp 所示 (執行結果完全相同，不再列出)：

範例程式 檔案 TempConv3.cpp

```
// TempConv3.cpp
#include <iomanip>
#include <iostream>
using namespace std;
```

```
void C2F (double, double&) ;    // 函數 C2F () 的原型
//-------- 主程式 ------------------------------
int main ()
{
   double CTemp, FTemp;
   cout << "  攝氏   華氏 " << endl ;
   cout << "-----------------"  << endl ;
   for ( int i = 1 ; i <= 10 ; i++ )
   {
     CTemp = 10.0*i;
     C2F (CTemp, FTemp) ;
     cout << setw (5) << CTemp << "  "
          << setw (5) << FTemp << endl ;
   }
   cout << "-----------------"  << endl ;
   system("PAUSE");
   return 0;
}
//-------- 函數 C2F () 的定義 ---------------------
void C2F (double C, double& F)
{
   F = C*9.0/5.0 + 32.0;
   return;              // 此指令可以省略
}
```

討論

1. 在程式 TempConv3.cpp 中，函數 C2F() 中的第二個參數 F 為主程式中變數 FTemp 的**參照** (reference)，因此不需要回傳任何數值，只要在函數 C2F ( ) 中改變 F 的值，就可以直接反應在主程式內的變數 Ftemp 上。
2. 函數 C2F() 定義中的「return;」可以省略。程式執行到被呼叫函數結尾的地方，會自動返回呼叫函數。

   一般而言，為了避免在不經意的情況下更動到呼叫函數內的變數，我們並不鼓勵這樣的作法，而將參照的使用限制在下面三種情況：

   (1) 引數本身必須改變。
   (2) 要傳回兩個以上的值。
   (3) 有大量數值需要傳遞。如果不使用參照，則將耗費時間在數值的複製上。例如，引數為大型向量或矩陣的情況。

程式 Swap.cpp 是一個典型的程式，利用參照來交換兩個變數的值。

範例程式　檔案 Swap.cpp

```
// Swap.cpp
#include <iostream>
using namespace std;

void Swap (int&, int&) ;
void Swap2 (int, int) ;
// ------ 主程式 ------------------------
int main ()
 {
   int A = 5, B = 10;
   Swap (A, B) ;
   cout << "執行過 Swap () \n";
   cout << " A 的值是："    <<  A
        << "\n B 的值是：" <<  B    << endl;
   cout <<  endl;

   Swap2 (A, B) ;
   cout << "執行過 Swap2 () \n";
   cout << " A 的值是："    <<  A
        << "\n B 的值是：" <<  B    << endl;
   system("PAUSE");
   return 0;
 }
//----------------------------------------
void Swap (int& x, int& y)
{
   int Temp;
   Temp = x;      x = y;     y = Temp;
}

//----------------------------------------
void Swap2 (int x, int y)
{
   int Temp;
   Temp = x;      x = y;     y = Temp;
}
```

操作結果

```
執行過 Swap ()
 A 的值是：10
 B 的值是：5

執行過 SWAP2 ()
 A 的值是：10
 B 的值是：5
```

討論

1. 函數 Swap()的功能是將原有兩個參數的值對調，這個功能需要藉助參照 (reference) 才能完成，如下圖所示：

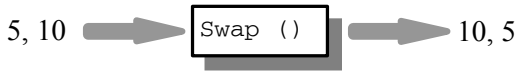

圖 6.2.1 Swap () 函數使用示意圖

函數 **Swap2**()沒有使用參照，因此呼叫函數內的值並不會改變，呼叫前後的結果都一樣。

2. 對於呼叫函數 main()而言，呼叫 Swap()和呼叫 **Swap2**()的語法完全一樣，無法區別。

6.3 inline 函數

依照 6.1 節所介紹的函數語法，函數可以彼此呼叫，但基本上每個函數具有同等地位，不可以把函數的定義放在另一個函數的內部。這一點和**複合敘述內可以放置入其他的複合敘述**，形成**巢狀判斷敘述**和**巢狀迴圈敘述**是不一樣的。

當函數被呼叫時，程式的主控權就會交給**被呼叫函數**，執行完畢後才把主控權交還**呼叫函數**。在這個過程裏，必須在堆疊裏記錄每個階段的執行指令程式碼的位置以待稍後接著繼續執行；這個記錄執行指令程式碼位置的變數稱為**程式指標** (instruction pointer, IP) 或**程式記數器** (program counter，簡稱為

PC)。主控權的交接，以及參數的複製在在都需要時間。對於這樣的額外負擔，C++ 提供一個適用於函數的特殊語法，稱為**行內函數** (inline function)。

Inline 函數的實際運作和一般的函數非常不同。在編譯時，其程式碼在每個呼叫的地方直接展開加入，因此，不需要浪費時間進行主控權的轉換，直接變成呼叫程式的一段複合敘述。Inline 函數的定義非常簡單，只要直接在函數的定義前加上關鍵字「inline」就可以。

我們可以把 6.1 節程式 TempConv.cpp 中的函數 C2F () 改寫成 inline 函數：

範例程式 檔案 Inline.cpp

```
// Inline.cpp
#include <iomanip>
#include <iostream>
using namespace std;
// ---inline 函數 C2F () 的定義 -----------
inline double C2F (double C)
   { return C*9.0/5.0 + 32.0; }
//--------- 主程式 ---------------------
int main ()
{
   double CTemp;  int i;
   cout << "  攝氏   華氏 " << endl ;
   cout << "-----------------"  << endl ;
   for ( i = 1 ; i <= 10 ; i++ )
   {
     CTemp = 10.0*i;
     cout << setw (5) << CTemp << "  "
          << setw (5) << C2F (CTemp) << endl ;
   }
   cout << "-----------------"  << endl ;
   system("PAUSE");
   return 0;
}
```

6.5 常犯的錯誤

1. **呼叫函數時使用錯誤的資料型態**

   即使是非常有經驗的人也常犯這個錯誤。通常，這是由於急著把自己的想法趕快寫成程式，而沒有耐心去仔細檢查參數的數量，資料型態和次序等細節。解決之道，可以先寫一個沒有具體內容，只是把引數原原本本的輸出到螢幕上的函數，編譯沒有問題，再逐步將該有的演算程序加入。

   我們在附錄 D 中，將函數的參數傳遞格式列表整理，方便在使用函數時參考。

2. **忘記加入函數的原型 (prototype)**

   如果在**呼叫函數** (calling function) 前放入函數定義，則函數原型可以忽略。一般的做法還是應該把所有的函數原型置於檔案開頭的部份，或是集中置於**標頭檔** (header filer) 內，再以#include 的方式插入。我們將在第十一章進一步討論標頭檔的使用。

3. 在函數內使用和**全域變數**相同的名稱定義**局部變數**，或是在呼叫函數和被呼叫函數內都以相同的名稱定義變數；一旦誤以為可以互相傳值，常造成錯誤的結果。這時候，編譯器無法發現任何語法上的問題，但具有相同名稱的變數實際上是完全獨立，不會互相影響的，

6.6 本章重點

1. 函數使用有兩大目的：

   (1) 減少重複撰寫功能類似的程式碼。

   (2) 易於除錯和維護。

   函數的使用是**模組化程式寫作** (modularized programming) 的重要方式。

2. 發起呼叫動作的函數稱為「呼叫函數」 (calling function)，被呼叫的函數稱為「被呼叫函數」 (called function)。以程式開發的觀點來看，函數的語法可以分為下列三個部份：
   (1) 函數的宣告
   (2) 函數的定義
   (3) 函數的呼叫

3. **函數的宣告**又稱為**函數的原型** (prototype)，它宣告了函數在呼叫時應該給予的資料型態，以及執行後所返回的資料型態和次序。函數原型的置放位置通常在主程式 `main()` 之前，可供主程式和同檔案的其他函數呼叫。

4. 如果我們把函數定義置於函數原型的位置，則此函數定義就兼具宣告和定義的功能，不需要再使用函數原型了。

5. 函數只能有一個返回資料型態。

6. 將函數處理資料的細節寫成程式稱之為函數的定義。不可以把函數的定義置於另一個函數的定義內。函數定義的一般語法如下所示：

```
返回資料形態  函數名稱 (參數列)
{
  主體敘述
  return 傳回值;
}
```

7. 函數呼叫時，參數的傳遞方式有**傳值** (call by value) 和**傳參照** (call by reference) 兩種：
   - 使用**傳值**的方式呼叫時，引數的值被層層複製，因此被呼叫函數內對參數的任何更動都不會影響到呼叫函數內的參數。
   - **參照** (reference) 是同一個變數的**別名** (alias)。以參照的方式呼叫 (call by reference) 時，被呼叫函數和呼叫函數作用的參數雖然名稱不同，但位於記憶體內的相同位置，因此被呼叫函數內對參數的任何更動都直接影響呼叫函數內的引數。

8. 為了避免在不經意的情況下更動到呼叫函數內的變數，應將參照的使用限制在下面三種情況：
   (1) 參數本身必須改變。
   (2) 要傳回兩個以上的值。
   (3) 有大量參數需要傳遞。

9. Inline 函數的實際運作和一般的函數非常不同。在編譯時，其程式碼在每個呼叫的地方直接展開加入，因此，不需要浪費時間進行主控權的轉換，形同呼叫程式的一段複合敘述。

10. Inline 函數雖然提昇了計算的效率，但如果呼叫次數不只一次，則因為是直接展開加入的關係，程式碼會比一般的寫法還要長。因此，inline 函數通常只用在下列情況下：
    (1) 函數本身非常簡短，但值得包裝起來使用。
    (2) 函數被不同位置呼叫的次數不多時。

11. 局部變數的宣告又分為 auto、static 和 register 三個儲存種類：
    - auto 是所有函數內變數的預設儲存方式，只在函數被呼叫時才存在，該函數結束後就消失，不會保留最後的值。
    - static 局部變數並不隨函數呼叫結束而消失，每一次函數被呼叫時，static 變數的值是上一次呼叫結束時的值。
    - register 變數可以留在 CPU 內，提昇函數的執行速度。

12. 如果把變數的宣告放在所有的函數之前，則這些變數值變數稱為**全域變數** (global variables) 可以讓其後的所有函數共用。

13. 如果沒有在宣告的同時給予初值，所有的全域變數和 static 局部變數在編譯時都會被自動初始化為零。

14. 全域變數固然好用，但它也破壞了函數原先所具有的獨立性和嚴格的資料出入控制。特別是當程式很龐大時，如果對全域變數的使用不加以節制，將使除錯變得非常困難。因此，全域變數的數量常常在撰寫大型程

式時，嚴格的自行限制在 10 個以下，而且往往全部使用大寫字母，或故意使用很長的名稱來凸顯。

15. 為了規範檔案間全域變數的適用範圍，有 extern 和 static 兩種儲存種類的宣告：
    - extern 全域變數宣告這個全域變數已經在其他檔案內定義過了。
    - static 全域變數宣告這個全域變數只能在自己的檔案內才適用。

6.7 本章練習

1. 寫一個叫做 Sum() 的函數，它可以指定要累加的數字數量，並把結果傳回來。例如，在主程式中執行以下敘述：

```
int x;
x = Sum (5);
```

會讓程式發出要求使用者輸入 5 個整數的訊息，並把使用者輸入的 5 個整數加總後存到變數 x 裏面。

2.

(1) 寫一個程式，它包括兩個分別叫做 Square() 和 Cubic() 的 inline 函數，以自動產生下列對照表 (各行都是整數)：

N	N^2	N^3
0	0	0
5	25	125
10	100	1000
15	225	3375
20	400	8000
25	625	15625
30	900	27000
35	1225	42875
40	1600	64000
45	2025	91125
50	2500	125000

其中 Square() 用來計算輸入值的平方，而 Cubic() 用來計算輸入值的三次方。(提示：可以參考第五章練習 4。)

(2) 在函數 Square()和 Cubic()中分別加入一個 static 變數 (各叫做 CountSquare 和 CountCubic)，讓各函數能夠計算被呼叫的次數。

3. 寫一個函數，它的原型如下：

```
void FindRoot (double a, double b, double c) ;
```

它的功能是將二次方程式

$$ax^2 + bx + c = 0$$

的兩個根計算並顯示出來。

提示 可以參考第四章 4.4 節程式 Root2.cpp

4. 寫一個程式，它呼叫一個叫做 CircleArea() 的函數來計算圓的面積 (圓面積 $= \pi r^2$, r 為半徑)。這個函數的原型如下：

```
double CircleArea (double r) ;
```

5. 寫一個程式，它包括一個叫做 Max() 的函數，以找出兩個值中較大的那一個。例如：

```
double x = 5, y = 8;
double z = Max(x, y);
```

則 z 的值為 8。

6. 改寫第四章 4.4 節程式 Tax.cpp，它包括一個叫做 Tax() 的函數，能從使用者輸入的綜合所得淨額計算綜合所得稅。