第 9 章

變數範圍與函式

But in the gross and scope of my opinion,
This bodes some strange eruption to our state.
（依我來看，這可能是我國將有突變發生的預警。）
— 莎士比亞，《哈姆雷特》第一幕第一場

目前為止，你用的都是**全域變數**（global variable），這些變數可在程式中的任何地方使用。本章將會介紹另一種變數，也會告訴你如何將程式拆成多個函式，討論一些關於函式的細節，包含重載、程式結構化，以及遞迴函式的呼叫方式。

範圍及儲存類型

變數具有兩種屬性，分別是**範圍**（scope）與**儲存類型**（storage class）。所謂變數範圍，是指變數在程式中可以使用的位置，全域變數的有效範圍是從宣告開始，一直到程式結束。**區域變數**（local variable）的有效範圍只有宣告部分所在的區塊，而且不能在區塊外面進行存取（設定內容值或讀取）。**區塊**（block）是指以大括號（{}）包含的程式碼。圖 9-1 說明區域及全域變數兩者之間的差異。

你可宣告一個與全域變數名稱相同的區域變數。一般來說，第一個宣告的 count 變數其有效範圍（在圖 9-2 中宣告）是整個程式；第二次宣告的區域 count 變數，於宣告所在的區塊裡有較高優先權。該區塊裡全域的 count 會被遮蓋掉。你可以再加一層區域的宣告並隱藏區域變數，內層區域變數比區域變數的有效範圍更小（前一句告訴你的是使用內層區域變數會使你的程式更難了解）。圖 9-2 說明隱藏變數。

變數 count 同時宣告為區域變數及全域變數。一般情形下 count（全域）的有效範圍是整個程式，但是在區塊裡宣告另一個新變數 count 時，該變數會成為這區塊中優先權最高的。全域的 count 在這個區塊範圍裡會被區域的 count 遮蓋。圖中陰影部分指出 count（全域）被遮蔽的範圍。

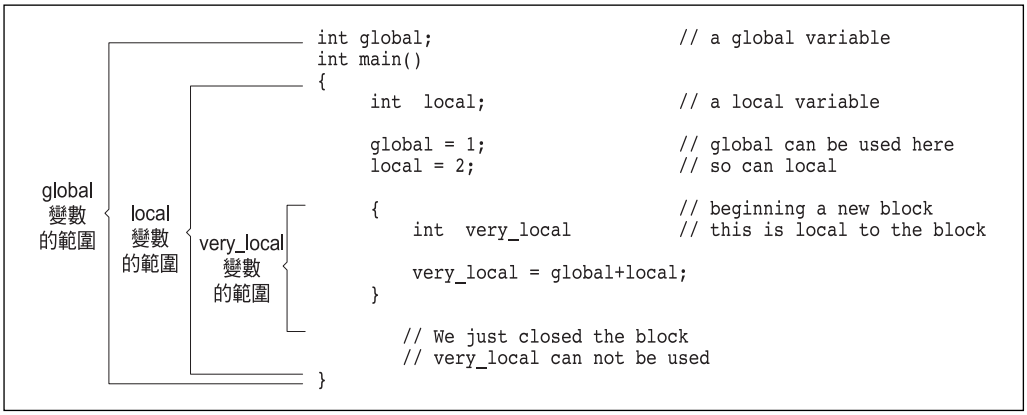

圖 9-1：區域與全域變數

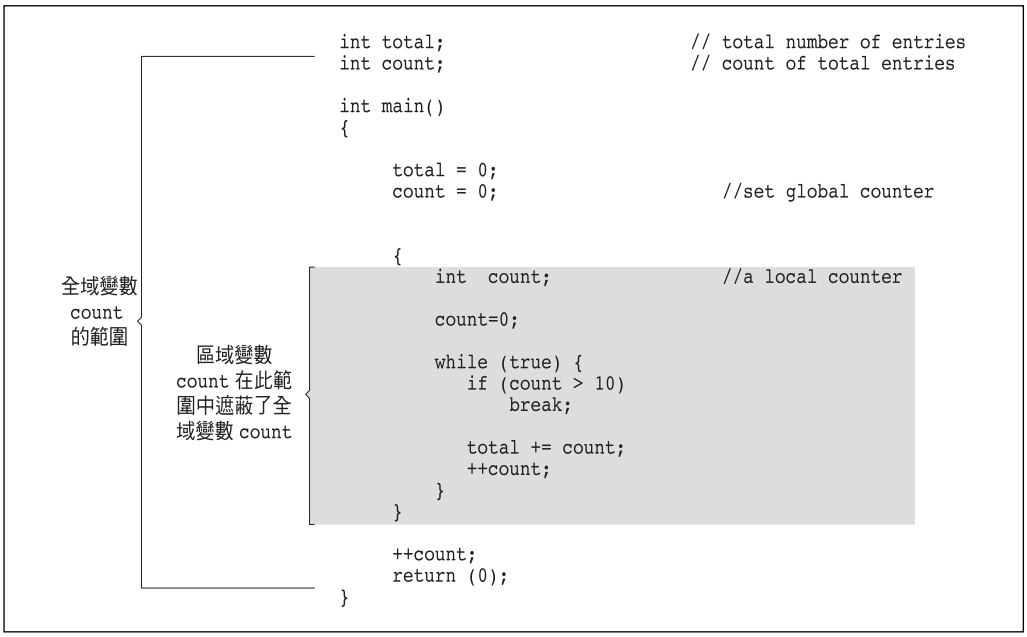

圖 9-2：隱藏的變數

其實，在程式中遮蔽變數並不妥當，下面這個敘述就會出問題：

```
count = 1;
```

此時很難分辨使用的是那個 count！是 main 上方宣告的 count 全域變數，還是 while 迴圈中的 count 區域變數？如果使用不同的名稱，像是 total_count、current_count 及 item_count，就非常清楚了。

變數的儲存類型可為**永久性**（**permanent**）或是**暫時性**（**temporary**），全域變數一定是永久性。它們在程式開始之前會建立及設定初值，在結束之前都會存在。暫時性變數是在區塊開始執行時，從**堆疊**（**stack**）中取得記憶體的配置。若是你使用太多的暫時性變數，就會導致**堆疊上溢**（**stack overflow**）的錯誤。暫時性變數所使用的空間，在區塊執行完畢時會清除，而每次進入區塊時，就會設定暫時性變數。

堆疊的大小依系統和編譯器而定。在許多 UNIX 系統上，程式會自動配置可用的最大堆疊。在其它系統中，預設的堆疊大小可用編譯選項加以設定。

區域變數是暫時性的，除非用 static 進行宣告。

static 在全域變數中的意義完全不同（指出這是該檔案中的區域變數），請參考第二十三章。static 這個字眼的完整討論請參考表 14-1。

範例 9-1 說明了永久性及暫時性變數的差異。我們選用的字可代表該變數的含意：像是 temporary 是暫時性變數，permanent 則永久性變數。C++ 會在建立 temporary 時先設定其內容值（在 for 敘述區塊中的開始），而 permanent 只會在程式開始時設定一次。

迴圈中的兩個變數都會增加，然而在迴圈的頂端，temporary 會設定為 1。

範例 9-1：perm/perm.cpp

```
#include <iostream>

int main() {
    int counter;    // 迴圈計數器

    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1;
        static int permanent = 1;

        std::cout << "Temporary " << temporary <<
                " Permanent " << permanent << '\n';
        ++temporary;
        ++permanent;
    }
    return (0);
}
```

程式的輸出結果如下：

```
Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3
```

暫時性變數有時可稱為**自動變數**（automatic variable），因為它們的空間是自動配置。auto 修飾子可用來指定暫時性變數；在實際情況下，auto 極為少用。

表 9-1 說明了各種變數的宣告方式。

表 9-1：變數宣告的修飾子

宣告	範圍	儲存類型	設定
區塊的外部	全域	永久性	一次
在區塊外部的 **static**	全域	永久性	一次
區塊的內部	區域	暫時性	每次進入區塊時
在區塊內的 **static**	區域	永久性	一次

C++ 中的關鍵字 static 是被重載（overload）最多的關鍵字。在不同的地方代表不同的意義，請參考表 14-1 的內容。

for 範圍

for 敘述類似於一對大括弧，你可以在其中的敘述中宣告變數，變數的範圍從 for 的開始到敘述的結尾（這包括由 for 控制的敘述或區塊）。在下面的敘述中：

```
for (int count = 0; count < MAX; ++count)
    sum += count;
// 從這裡開始，就超過 count 範圍
```

count 變數是在 for 內部宣告的，其範圍到敘述的結尾，以 for 後面的第一個分號結束。

名稱空間

隨著程式不斷地增大，程式碼的行數不斷增多，全域變數也越來越多。因此，全域名稱空間變得非常擁擠。

解決辦法是使用格式化（stylized）的變數名稱。例如，資料處理模組中的所有變數都以 dp_（data processing）開始，而儲存模組中的所有變數則以 st_（storage）開始。

這個辦法勉強能夠應付，但是隨著加入的模組越來越多，就變得越麻煩。在處理核心軟體系統、用戶界面模組、遊戲組和西洋雙陸棋模組時，就必須將所有變數加上字首 core_ui_games_back_，這可麻煩了。

此問題的 C++ 解決方案是將程式分成名稱空間。在現實生活中，你每天都在處理名稱空間。例如，你叫自己的家族成員時只叫他們的名，比如 Steve、Bill、Sandra 和 Fred，而不叫他們的姓。而對於不是家族成員的外人，你就會叫他們比較正式的名字，像 Steve Smith、Bill Smith，等等。

在 C++ 中可以定義 **namespace**。在 **namespace** 中宣告的變數會被當作同一家族（名稱空間）的成員。例如，下面的程式碼宣告了三個整數，它們都是名稱空間 display 的成員：

```
namespace display {
    int width;        // 顯示幕的寬度
    int height;       // 顯示幕的高度（以行為單位）
    bool visible;     // 顯示幕可見嗎？
};
```

家族成員的全名可能是 Stephen Douglas Smith。而 C++ 對等的全名叫做**完整合格名稱**（**fully qualified name**）在這個例子中，width 變數的完整合格名稱是 display::width。而家族內的成員（亦即，屬於名稱空間 display 的成員）可以用較不正式的名稱 width。

名稱空間 std

本書一開始將物件 std::cout 用於輸出，實際上，這表示在名稱空間 std 中使用變數 cout。C++ 使用該名稱空間來定義標準程式庫的物件和函式。

讀者應該還記得，之前大多數的程式都以下面的敘述開始：

```
#include <iostream>
```

第一個敘述使編譯器讀入 *iostream* 檔，該檔案中包含 C++ 標準變數的定義。例如，下面是一個簡化的 *iostream*：

```
namespace std {
    istream cin;   // 定義輸入串流 cin
    ostream cout;  // 定義輸出串流 cout
    ostream cerr;  // 定義標準錯誤串流
   // 其他的定義
}
```

編譯器一旦看到這些定義，std::cin、std::cout 和 std::cerr 就可用了。

全域名稱空間

如果不將程式碼或變數放在任何名稱空間中，編譯器會給它們一個空的名稱空間。例如，運算式：

```
::global = 45;
```

將 45 指定給變數 global，該變數是在任何名稱空間宣告之外宣告的。

檔案專屬的名稱空間

假設你想要定義一個模組，並想讓檔案中大多數的函式和變數放在各自唯一的名稱空間中。

可將以下敘述放在檔案的開頭：

```
namespace my_□le_vars {
```

如果其他人定義同名的名稱空間，會發生什麼事呢？結果是：造成名稱空間衝突。

要避免這一點，C++ 發明了未命名的名稱空間。以下宣告：

```
namespace {
```

沒有指定名稱，將所有包含的宣告放在檔案惟一的名稱空間中。

套疊的名稱空間

名稱空間可以層層套疊。例如，如下宣告變數：

```
namespace core {
    namespace games {
       namespace dice {
          int roll;    // 最後一次擲骰子的值
```

這樣套疊有點繁雜（core::games::dice::roll），但是當需要組織大量的程式碼時，套疊名稱空間將有其用處。

using 敘述

假設程式有一個命令模組（其名稱空間為 command）和一個命令解析模組（名稱空間為 command_parser）。這兩個模組關係密切，命令模組經常引用解析模組中的變數（這樣緊密的關聯是不良的程式設計，但暫且忽略之）。

在編寫命令模組時，如果想要引用解析模組中的變數，必須於前面加上名稱空間識別字：

```
if (command_parser::□rst_argument == "ShowAll")
```

因為這些模組緊密的關聯，所以必須一直用 command_parser::□rst_argument；如此一會兒就讓會人厭煩了。

但是你可以告訴 C++：「我知道 □rst_argument 位在 command_parser 模組中，但請假裝它也在我的模組中。」以上只要利用 using 敘述即可達成：

```
using command_parser::□rst_argument;
```

C++ 現在能讓你使用 □rst_argument 代替 command_parser::□rst_argument 了。

```
using command_parser::□rst_argument;

if (□rst_argument == "ShowAll")
```

using 宣告的範圍與其他變數宣告的範圍相同：於宣告區塊的結尾處結束。

現在假設我們希望從 command_parser 模組匯入大量的變數。可以在程式中為每個變數放入一個 using 敘述，但這需要加入大量的敘述。或者我們可以整批處理，並告訴 C++，名稱空間 command_parser 中的所有名稱都將匯入我們的模組中，如以下敘述所示：

```
using namespace command_parser;
```

using 敘述的問題

在多數情況下應盡量避免使用 using 敘述。本節示範的例子，因為這兩個名稱空間之間有許多關聯，才有必要使用 using 敘述。但是如此的緊密關聯並不是優良的程式設計。

using 敘述也會引起名稱空間衝突。通常看到無範圍宣告的變數時（例如，signal_curve），你可以假設它屬於目前的名稱空間。如果在程式中有 using 敘述，以上假設就不再有效，而問題也將變得更加複雜。程式已經夠複雜了，因此別再讓程式複雜化。

函式

函式（functions） 的作用是將常用的程式碼放在一起，重覆地進行呼叫。其實你已經用過一個函式了，就是 main。它是在程式開始執行時所呼叫的一個特殊函式。其它的函式都是直接或間接由 main 所呼叫。

假設你要寫一個程式計算 3 個三角形的面積，可以將公式重覆寫三次；或是，建立一個函式處理這件工作，重複呼叫這個函式三次。每個函式前面應該加上註解，放入下列內容：

名稱

函式的名稱。

說明

該函式的功能描述。

參數

該函式中每個參數的說明。

傳回值

函式傳回值的說明。

當然還可以加入其它事項，例如檔案格式、參考資料或說明等等。

以計算三角形面積的函式來說，其註解部分為：

```
/*****************************************
 * triangle -- 計算三角形的面積              *
 *                                         *
 * 參數                                    *
 *  width -- 三角形的寬                    *
 *  height -- 三角形的高                   *
 *                                         *
 * 傳回值                                  *
 *  三角形的面積                           *
 *****************************************/
```

這個函式開頭可以這樣寫：

```
□oat triangle(□oat width, □oat height)
```

float 是函式的型態，定義該函式傳回的資料型態；width 及 height 是該函式的參數，也就是函式中的區域變數，用來將資料傳送到函式中。

首先要檢查所傳入的參數。大家都知道三角形的寬和高不可能是負數。但是程式設計有自己的領域，你不能相信任何東西。所以需利用兩個 assert 敘述來檢查輸入：

```
assert(width >= 0.0);
assert(height >= 0.0);
```

這種謹慎的做法在大型程式除錯時相當有用。此類的 assert 敘述在追蹤不良的程式碼時能有很大的幫助；它們可以盡早停止程式，這樣能省下許多追溯錯誤資料來源的時間。請記住：你的小心謹慎，並不代表你就能避免程式除錯。

這個函式會用以下敘述來計算面積：

```
area = width * height / 2.0;
```

接下來是要如何將結果傳回給呼叫方，可以透過 **return** 敘述：

```
return (area)
```

完整的 triangle 函式請參考範例 9-2。

範例 9-2：tri/tri-sub.cpp

```
#include <assert.h>

/*****************************************
 * triangle -- 計算三角形的面積                *
 *                                         *
 * 參數                                     *
 *  width -- 三角形的寬                      *
 *  height -- 三角形的高                     *
 *                                         *
 * 傳回值                                   *
 *  三角形的面積                             *
 *****************************************/
□oat triangle(□oat width, □oat height)
{
    □oat area; // 三角形的面積

    assert(width >= 0.0);
    assert(height >= 0.0);
    area = width * height / 2.0;
    return (area);
}
```

這一行：

```
size = triangle(1.3, 8.3);
```

會呼叫 triangle 函式。C++ 見到這個呼叫函式時，會採取下面的動作：

三角形的寬度 = 1.3
三角形的高度 = 8.3

開始執行 triangle 函式的第一行。

這種參數傳遞方式的正式名稱是**傳值呼叫**（call by value）。只有在函式被呼叫時，才進行指定（assignment）的動作，因此資料是透過參數單向（one way）傳入。

return 敘述將資料傳出該函式。在三角形的例子中，函式將區域變數 area 的值設定為 5.4，然後執行 return (area)，因此該函式的傳回值是 5.4。這個值會指定給 size 變數。

```
return(area);       5.4（面積的值）
// . . . .
size = triangle(1.3, 8.3)
```

範例 9-3 可計算 3 個三角形的面積。

範例 9-3：tri/tri.cpp

```
#include <iostream>
#include <assert.h>

int main()
{
    // 計算三角形面積的函式
    oat triangle(oat width, oat height);

    std::cout << "Triangle #1 " << triangle(1.3, 8.3) << '\n';
    std::cout << "Triangle #2 " << triangle(4.8, 9.8) << '\n';
    std::cout << "Triangle #3 " << triangle(1.2, 2.0) << '\n';
    return (0);
}
/*****************************************
 * triangle -- 計算三角形的面積              *
 *                                         *
 * 參數                                    *
 *  width -- 三角形的寬                    *
 *  height -- 三角形的高                   *
 *                                         *
 * 傳回值                                  *
 *  三角形的面積                           *
 *****************************************/
oat triangle(oat width, oat height)
{
    oat area; // 三角形的面積

    assert(width >= 0.0);
    assert(height >= 0.0);
    area = width * height / 2.0;
    return (area);
}
```

函式的宣告與變數一樣必須加以宣告，宣告的作用是告訴 C++ 編譯器關於函式的傳回值和參數。函式的宣告有兩種方式，第一是在使用函式之前先宣告，另一種是定義函式的原型（prototype），讓編譯器有足夠的資訊來呼叫這個函式。函式原型看起來像是函式的第一行，但沒有函式主體。例如，triangle 的原型如下：

```
oat triangle(oat width, oat height);
```

注意這一行的最後面的分號，它是告訴 C++，這是一個函式原型，而非實際的函式。

在宣告原型時，C++ 允許你不必指定參數名稱。這個函式可以寫成：

```
□oat triangle(□oat, □oat);
```

然而，這種方式不常用，反正函式的原型很容易寫（在第二十三章中，我們都放在標頭檔內）；況且，在原型中加入變數名稱，可讓別人在閱讀時取得更多的資訊。

若函式沒有參數，宣告時的參數列即是 ()。例如：

```
int get_value();
```

你也可以用參數列 (void)，它算是早期 C 語言的遺產，當時空串列 () 是用來表示舊式的 K&R-style C 函式原型。實際上，C++ 可接受空串列或 void 宣告。

幾乎所有的 C++ 程式設計師都喜歡用空串列。然 (void) 形式的優點有：

- 提供一個明顯的指示，表明沒有參數列（換句話說，如果加入 void，就表明了「該函式真的沒有參數，不是我忘了」）。
- 可相容於舊的 C 語言。

空串列的優點有：

- () 比 (void) 在語法與宣告參數的方式上更加健全和一致。
- void 是用來解決因空串列另做他用而導致的語法問題，而加入 C 中的歷史產物。為了相容性，仍將它從 C 移植到 C++。
- 我們是用 C++ 寫程式而不是 C，既然如此，幹麻還要在程式中使用這些舊語法呢？

基於這些原因，多數人都選擇了空串列。本文作者是一個例外。筆者偏愛 (void) 架構，但是當有三位審閱和一位編輯同聲說「你錯了！」時，就得重新思考你的選擇了。全書都使用空串列。

傳回 void

void 也可用來指出函式沒有傳回值（類似 FORTRAN 的 SUBROUTINE 和 PASCAL 的 Procedure）。例如，這個函式純粹印出訊息，不傳回任何值：

```
void print_answer(int answer)
{
    if (answer < 0) {
        std::cout << "Answer corrupt\n";
        return;
    }
    std::cout << "The answer is " << answer << '\n';
}
```

名稱空間和函式

名稱空間不僅影響變數，還影響函式。函式屬於其宣告時所在的名稱空間。例如：

```
namespace math {

int square(const int i) {
    return (i * i);
}

} // 結束名稱空間

namespace body {

int print_value()
{
    std::cout << "5 squared is " << math::square(5) << '\n';
}

}
```

名稱空間中的所有函式可以直接存取該名稱空間中的變數，而不需使用 **using** 子句或 **namespace** 完整合格名稱。例如：

```
namespace math {

const double PI = 3.14159;

double area(const double radius)
{
    return (2.0 * PI * radius);
}

}
```

const 參數及傳回數值

如果參數宣告為 **const**，表示這個參數在函式中是不可改變的。一般的變數在函式中是可以改變的，但是這個改變無法傳回給其呼叫者。

例如，在 `triangle` 函式中，我們沒有改變 `width` 和 `height` 的值，所以能夠宣告為 **const**；因為傳回值也是不能改變的，所以也可以宣告為 **const**。**const** 宣告可以讓你知道，在函式內不會改變參數內容；如果你改變了 **const** 參數，編譯器會產生錯誤訊息。`triangle` 函式經過修改之後如下：

範例 9-4：tri/tri-sub2.cpp

```
const ◘oat triangle(const ◘oat width, const ◘oat height)
{
    ◘oat area; // 三角形的面積

    assert(width >= 0.0);
    assert(height >= 0.0);
    area = width * height / 2.0;
    return (area);
}
```

目前為止，傳回值的 const 宣告似乎只是裝飾。下一節將會告訴你如何傳回參考（reference），以及使 const 的宣告能夠派上用場。

參考參數與傳回值

第五章曾經談過「參考變數」（reference variables）。參考變數是宣告變數名稱的另一種方式。對全域及區域變數來說，參考變數的用處不大；但當作參數使用時，情形就完全不一樣了。

假如要寫一個副程式來增加計數器的值，用範例 9-5 的方式，結果會是錯的。

範例 9-5：value/value.cpp

```
#include <iostream>

// 這個函式不會正確運作
void inc_counter(int counter)
{
    ++counter;
}

int main()
{
   int a_count = 0; // 計數器

   inc_counter(a_count);
   std::cout << a_count << '\n';
   return (0);
}
```

為什麼不行呢？因為 C++ 的預設方式是傳值呼叫，所以傳進去的值不會傳回來。

如果將參數 counter 設定成參考指標呢？透過參考指標，可以讓同一個變數有兩個名稱。當呼叫 inc_counter 時，counter 會變成指到 a_count 的參考指標。範例 9-6 就是透過參考指標，結果是正確的。

範例 9-6：value/ref.cpp

```
#include <iostream>

// 正常運作
void inc_counter(int& counter)
{
    ++counter;
}

int main()
{
   int a_count = 0;    // 計數器

   inc_counter(a_count);
   std::cout << a_count << '\n';
   return (0);
}
```

參考指標宣告也可以用於傳回值。例如範例 9-7，可找出陣列中最大的元素。

範例 9-7：value/big.cpp

```
int& biggest(int array[], int n_elements)
{
    int index;    // 目前元素的索引
    int biggest;  // 最大元素的索引

    // 假設第一個是最大的元素
    biggest = 0;
    for (index = 1; index < n_elements; ++index) {
        if (array[biggest] < array[index])
            biggest = index;
    }

    return (array[biggest]);
}
```

如果要印出陣列中最大的元素，只需用下面的方式：

```
int item_array[5] = {1, 2, 5000, 3, 4}; // 陣列

std::cout << "The biggest element is " <<
         biggest(item_array, 5) << '\n';
```

讓我們看詳細一點。首先，建立參考變數時思考一下它的作用：

```
int& big_reference = item_array[2]; // element #2 的參考指標
```

參考變數 big_reference 是 item_array[2] 的另一個名稱。你可以透過這個參考印出一個值：

```
std::cout << big_reference << '\n'; // 印出 element #2
```

但由於它是一個參考指標，你也可以讓它放在指定敘述的左側（放在指定敘述 = 號左側的運算式通稱為 lvalue）。

```
big_reference = 0; // 將陣列最大的值設為零
```

biggest 函式會傳回一個參考指標。請記住，在下面的程式碼中，biggest() 就是 item_array[2]。下面三段程式碼會進行相同的處理。實際的變數 item_array[2] 本身並沒有改變，只是我們採用別的方式來指定。

```
// 使用實際的變數
std::cout << item_array[2] << '\n';
item_array[2] = 0;

// 使用簡單的參考指標
int big_reference = &item_array[2];
std::cout << big_reference << '\n';
big_reference = 0;

// 使用傳回參考指標的函式
std::cout << biggest(item_array, 5) << '\n';
biggest(item_array, 5) = 0;
```

由於 biggest 會傳回一個參考指標，它可以放在指定敘述（=）的左側；若是你不希望這樣子，可以傳回一個 const 參考指標：

```
const int& biggest(int array[], int n_elements);
```

這會告訴 C++，傳回的是一個參考指標，其結果不能改變。因此下面的程式是錯誤的：

```
biggest() = 0; // 會產生錯誤
```

懸盪參考（Dangling Renferences）

使用參考指標傳回值時應特別小心，若使用不當，則會參考到一個根本不存在的變數。範例 9-8 說明了這個問題。

範例 9-8：ref/ref.cpp

```
1: const int& min(const int& i1, const int& i2)
2: {
3:     if (i1 < i2)
4:     return (i1);
```

範例 9-8：ref/ref.cpp（續）

```
 5:     return (i2);
 6: }
 7:
 8: int main()
 9: {
10:    const int& i = min(1+2, 3+4);
11:
12:    return (0);
13: }
```

第 1 行是 min 函式的定義，它會傳回兩個整數之中較小的。

第 10 行是呼叫該函式。在呼叫 min 之前，C++ 建立一個暫時變數來存放 1 + 2 這個算式的內容。指向暫時變數的參考指標會以 i1 參數的方式傳給 min 函式。C++ 會替 i2 參數建立另一個暫時變數。

接著呼叫 min 函式同時傳回指向 i1 的參考指標，但是 i1 的參考指標是指到哪裡？它指到 main 之中所建立的一個暫時變數。當敘述結束的時候，C++ 會清除所有的暫時變數。

讓我們仔細觀察呼叫 min（第 10 行）的過程。下面是第 10 行的虛擬碼 (pseudo-code)，包含了一些 C++ 隱藏的內部運作：

```
建立整數 tmp1，指定其值為 1 + 2
建立整數 tmp2，指定其值為 3 + 4
將參數 i1 指向 tmp1
將參數 i2 指向 tmp2
呼叫 min 函式
將 main 的變數 i 指向傳回值（指向 tmp1 的 i1）
// 此時 i 參考到 tmp1
刪除 tmp1
刪除 tmp2

// 此時 i 仍指向 tmp1
// 實際上已不存在，但 i 仍是參考到它
```

第 10 行的結尾有狀況：i 參考到一個已被清除的暫時變數；換言之，i 指向一個根本不存在的位置。這稱為**懸盪參考**，你必須極力避免這種情況。

陣列參數

你已經知道如何面對單純的參數傳遞，但 C++ 對於陣列參數有些不同的做法。首先，你不需要在原型宣告中指定陣列的大小。例如：

```
int sum(int array[]);
```

C++ 利用**傳址呼叫**（call by address）方式的來傳遞陣列。你可以用另一種角度來思考：C++ 會自動將所有的陣列參數換成參考指標，所以它能傳遞的陣列大小毫無限制。剛才宣告的 sum 函式，可接受任何大小的整數陣列。

然而，即使要放入陣列大小也是可行的。但 C++ 根本不理會你所放入的數字。這種方式唯一的作用，在於告訴程式使用者：這個函式只能處理固定大小的陣列。

```
int sum(int array[3]);
```

如果是多維陣列，除了最後一個陣列之外，你必須指定每一維陣列的大小。因為 C++ 需要這些維度的資料，才能算出陣列中每個元素的位置。

```
int sum_matrix(int matrix1[10][10]);  // 合法
int sum_matrix(int matrix1[10][]);    // 合法
int sum_matrix(int matrix1[][]);      // 不合法
```

問題 9-1：範例 9-9 中的函式可計算字串【註】的長度，但結果卻是長度為 0，為什麼？

範例 9-9：length/length.cpp

```
/*******************************************************
 * length -- 計算字串的長度                            *
 *                                                     *
 * 參數                                                *
 *      string -- 要計算長度的字串                     *
 *                                                     *
 * 傳回值                                              *
 *      字串的長度                                     *
 *******************************************************/
int  length(char string[])
{
    int index;      // 字串的索引

    /*
     * 一直進行處理到字串結束的字元
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* 沒事 */
    return (index);
}
```

註　這個函式的作用和標準函式 strlen 是一樣的。

函式重載

下面是一個傳回整數平方的函式：

```
int square(int value) {
    return (value * value);
}
```

我們也要算浮點數的平方：

```
float square(float value) {
    return (value * value);
}
```

現在我們有兩個同名的函式，這是否會導致無效？在 C 及 PASCAL 語言中，這種方式的確不行，而 C++ 卻允許這種方式，稱為**函式重載**（function overloading），也就是多個函式可具有相同的名稱。你可以將 square 定義為接受各種型態的參數：int、float、short int、double 甚至 char。

為了維持程式碼的一致，所有同名的函式應該是執行相同的功能。例如，你可以定義下面兩個函式：

```
// 整數的平方
int square(int value);

// 在螢幕上畫一個方形
void square(int top, int bottom, int left, int right);
```

這是合法的 C++ 程式，但也讓許多人大惑不解。

函式重載有一個限制：C++ 必須能判斷應採用那一個函式。例如，下面的方式是錯誤的：

```
int get_number();
float get_number();  // 無效
```

問題在於：C++ 透過參數列來分辨函式；但是，上面兩個 get_number 函式的參數列完全相同：都是 ()，使得 C++ 無法分辨，因而認為第二個宣告是錯的。

預設參數

若是要定義在螢幕上畫方形的函式，這個函式必須要有方形的大小比例。函式的定義為：

```
void draw(const int width, const int height, double scale)
```

使用一陣子之後，你發現到有九成的機會你用不到大小比例，因為大部分都是用 1.0 的比例來畫出正方形。

這時你可以指定比例的預設值：

```
void draw(const int width, const int height, double scale = 1.0)
```

意思是說：「若沒有指定比例，就採用 1.0」。因為下面兩者是相等的：

```
draw(3, 5, 1.0);     // 指定尺度比例
draw(3, 5);          // 採用預設的 1.0 尺度比例
```

使用預設參數會有些風格上的缺陷，例如：

```
draw(3, 5);
```

你是否能判斷出程式設計師是用 1.0 的比例，還是忘了指定？雖然這項功能有時很方便，但切勿濫用。

未使用的參數

若是定義一個參數而沒有用到，大多數的編譯器會出現警告訊息。例如：

```
void exit_button(Widget& button) {
    std::cout << "Shutting down\n";
    exit (0);
}
```

編譯時會產生下面的訊息：

```
Warning: line 1. Unused paraemter "button"
```

如果你真的不想使用參數時會如何？是否可以要求編譯器不要騷擾你？可以，你乾脆不要指定參數的名稱。

```
// 沒有警告出現，但在風格上還需要改進
void exit_button(Widget&) {
    std::cout << "Shutting down\n";
    exit (0);
}
```

以上固然 OK，但一般人可能很難意會。exit_button 利用 Widget& 當作參數，但這個參數是什麼？解決方案是將參數名稱以註解的方式表示。

```
// 較佳的方式
void exit_button(Widget& /*button*/) {
    std::cout << "Shutting down\n";
    exit (0);
}
```

可能有些人覺得這種方式不好看，沒錯，看起來真的很礙眼。我也希望能找出更好的方式。

你可能會納悶：「我為什麼要把程式寫成這種樣子？為什麼不能去掉參數？」

許多系統會用到回呼函式（callback function）。例如，你可通知 X Windows，在按下 Exit 按鈕時呼叫 exit_button，你的回呼函式可處理各種按鈕，重點是知道哪個按鈕被按下。所以，X Window 提供了 button 參數給函式。

如果你知道只有 button 會讓 X 呼叫 exit_button 呢？X 仍會傳給你，你根本不用去管它。所以某些函式裡面會存在一些不用的參數。

inline 函式

回頭看看上面整數的 square 函式，它是很短的函式（只有一行）。但在 C++ 呼叫函式時，會有許多的工作要處理，像是將各個參數放入堆疊，進入及離開函式，以及在函式結束時將堆疊內容還原。

例如下面的程式碼：

```
int square(int value) {
    return (value * value);
}
int main() {
    // . . . .
    x = square(x);
```

在 68000 系列 CPU（譯註：舊型的 Mac 都是採用這類 CPU）的機器上執行，可能會產生下面的組合語言程式：

```
label "int square(int value)"
        link a6, #0          // 設定區域變數

        // 下二行是進行運算的處理
        movel a6@(8), d1     // d1 = value
        mulsl a6@(8), d1     // d1 = value * d1
        move d1, d0          // 將傳回值放到 d0
        unlk a6              // 恢復堆疊內容
        rts                  // 傳回（d0）

label "main"
// . . . .
//      x = square(x)
//
        movel a6@(-4), sp@-  // 將數字 x 放到堆疊
        jbsr "void square(int value)"
                             // 呼叫函式
```

```
        addqw #4, sp          // 還原堆疊
        movel d0, a6@(-4)     // 將傳回值放到 x
// . . . .
```

你可以看到，兩行的程式碼會產生 8 行的處理步驟。C++ 允許你透過 inline 函式，來減少這些額外的工作。**inline** 關鍵字指出這個函式很小，編譯器可以將整個函式主體放進程式碼，而不要產生對於這個函式的呼叫：

```
inline int square(int value) {
    return (value * value);
}
```

修改後的 square 函式，其組合程式碼也變小了：

```
lable "main"
// ...
//      x = square(x)
//
        movel d1, a6@(-4)     // d1 = x
        movel a6@(-4), d0     // d0 = x
        mulsl d0, d0          // d0 = (x * x)

        movel d0, a6@(-4)     // 儲存結果
```

將該函式展開後，可減少 8 行額外的工作，可得到較快的執行速度。

在產生程式時，**inline** 提供 C++ 一個非常重要的提示。它會通知編譯器，這個程式碼很小，而且很簡單。和 **register** 很類似，**inline** 修飾子只是一種提示，如果 C++ 編譯器無法產生 inline 函式，就採用一般函式的呼叫。

參數類型的摘要說明

表 9-2 列出各種不同的參數類型。

表 9-2：參數類型

型態	宣告
傳值呼叫	`function(int var)` 值會傳給函式，並且在函式中可以改變其內容，但最後的結果不會傳回給呼叫方。
傳值的常數呼叫	`function(const int var)` 數值可傳入函式中，而且不可以改變。
參考指標	`function(int& var)` 參考指標會傳送到該函式中。任何的改變會傳回到呼叫方。

表 9-2：參數型態（續）

型態	宣告
常數參考指標	`function(const int& var)` 在函式中不可改變其值。對於複雜資料型態，這種參數比「傳值的常數呼叫」更有效率。
陣列	`function(int array[])` 值可傳入和修改。C++ 會自動將陣列轉換為參考指標。
傳址呼叫	`function(int *var)` 傳入一個指標。關於指標的說明，請參閱第十五章。

遞迴

所謂遞迴，就是一個函式直接或間接的自我呼叫。某些函式天生就適合採用遞迴演算法，例如階乘（factorial）。

遞迴函式必須遵循兩個基本原則：

- 必須有一個終點。
- 必須可簡化問題。

階乘的定義是：

```
fact(0) = 1
fact(n) = n * fact(n-1)
```

在 C++ 中，該定義轉換成：

```
int fact(int number)
{
    if (number == 0)
        return (1);
    /* 否則 */
    return (number * fact(number-1));
}
```

它滿足了以上兩個準則。首先，它的定義有終點（`number == 0`）；第二，它能簡化問題，因為 `fact(number - 1)` 比 `fact(number)` 更單純。

階乘只有在 `number >= 0` 情況下才成立。如果要算出 `fact(-3)`，結果會是什麼？程式會因為堆疊上溢（stack overflow）而中斷執行，或是出現類似的錯誤訊息。`fact(-3)` 呼叫 `fact(-4)` 呼叫 `fact(-5)` 等等，根本就沒有終點。這稱為**無窮的遞迴錯誤**

(infinite recursion error)。在這個例子中是由不良的參數所引起的。我們應該檢查參數：

```
int fact(int number)
{
    assert(number >= 0);
    if (number == 0)
        return (1);
    /* 否則 */
    return (number * fact(number-1));
}
```

許多重覆的處理都可透過遞迴的方式進行，例如將陣列中的元素加總。你可以定義一個函式，把陣列的每個元素相加：

若是只有一個元素，就不用說了。
否則，將第一個元素和其它元素的總合相加。

以 C++ 來表示：

```
int sum(const int ▫rst, const int last, const int array[],
        const int array_size)
{
    assert((▫rst > 0) && (▫rst < array_size));
    assert((last > 0) && (last < array_size));

    if (▫rst == last)
        return (array[▫rst]);
    /* 否則 */
        return (array[▫rst] + sum(▫rst + 1, last, array));
}
```

例如：

```
Sum(1 8 3 2) =
   1 + Sum(8 3 2) =
      8 + Sum(3 2) =
         3 + Sum(2) =
            2
         3 + 2 = 5
      8 + 5 = 13
   1 + 13 = 14
Answer = 14
```

不過這未必是陣列加總最簡單或最快速的方法。在這個例子中，使用迴圈會快得多。以上只是示範如何用遞迴建立非傳統的解法。

結構化程式設計

電腦科學家竭盡心力研究程式設計的方法，終於發現了最有效率、最完美的方法，但每個月就發現一種！這些方式包括流程圖、由上而下的方式、由下而上的方式、結構化程式設計，以及物件導向的方式等等。

你已經學過了函式，讓我們接下來討論**結構化程式設計**（structured programming）的技巧。它的方式很簡單，就是將程式拆成許多小型且具備獨立功能的函式，使得程式更容易編寫，也更容易了解。我個人不認為這是完美的方式，但在某些情況下的確很管用。如果你有更適合的方式，就儘管採用。

結構化程式設計的重點，在於程式碼本身。之後會談到如何將程式與資料結合，以組成類別（class），然後才開始討論物件導向。

程式設計的第一步，是決定你要做什麼；接下來，是決定如何組織你的資料。

最後才進入編寫程式的階段。在寫一個報告時，你會先擬出報告的大綱，將每個段落以一個句子闡明重點；程式設計也毫無二致：先擬出大綱，這個大綱就是 main 函式；細節則是由各個函式分別處理。例如，範例 9-10 的程式可一舉解決世界上所有的問題。

範例 9-10：終極解決方案

```
int main()
{
    init();
    solve_problems();
    □nish_up();
    return (0);
}
```

當然，其中還有許多的細節根本沒有考慮。

main 函式不應該超過兩頁。函式的大小應在 3 頁以內，因為人類在短時間內所能記憶的範圍大約也是這個篇幅。main 完成之後，可開始進行其它函式。結構化程式設計也稱為**由上而下的設計方式**（top-down programming）。從頂端部分（main）開始設計，依序往下處理。

另一種方式是**由下而上**（bottom-up programming）。先寫最底層的函式，接著進行測試及建立完整的工作群組。如果是使用一個從未用過的標準函式，我就採用這種方式，先寫一個小函式，確使它能正確運作之後，才繼續下去。第七章的計算機程式就是採用這個方法。

稍後的第十三章，我們將學習物件導向的程式設計，這種方法可將資料和對資料的處理放在所謂的類別（class）之中。

實質的程式設計

過去幾年，筆者使用過許多不同的程式設計技巧。所用的方法端視欲解決的問題而定；也發現了建立完美程式的一些秘訣。

第一步是在著手去做之前先考慮你將要做什麼。先忍住動手編寫程式的衝動，先坐下來規劃一下，讓問題越簡單越好。程式碼越簡單，出錯的可能就越小。

另外，盡量讓你的設計更富彈性。畢竟，你今天不瞭解的東西，也許明天就瞭解了。

接下來，以清楚的方法組織程式所需的資訊。視情況而定，這種方法可能需要建立文件說明、圖表，等等。全都取決於問題本身以及你的思路邏輯；該怎麼做就怎麼做。

編寫程式時，請確定在每一階段都能夠測試程式。緩步地往正確的方向邁進絕對會比偏離正軌的大躍進更快到達終點。

最後，請體認一點：沒有所謂「正確的」程式編寫技巧。不同的問題會有不同方法可行。使用最適合你的方法就對了。

非傳統的程式編寫技巧

傳統的程式編寫技巧描述應如何順利地編寫。但是回歸現實後，你很快就會發現「理論的做法」和「實際的做法」之間有差別。在現實生活中，經常會使用到許多非傳統的程式編寫技巧：

創意抄寫法

當程式設計師發現某個程式大致能符合他的需要時，就會抄寫這個程式，並做適當的修改，以滿足其目的。現有大量的開放原始碼，多數情況下，都是可「借用」的。

實驗觀察法

常常我們會碰到文件說明很差或不正確的系統。此技巧需要建立一套實驗，觀察該系統實際的運作情形。越瞭解系統，更能精練研究成果。當終於明白系統的運作方式時，即可刪掉除錯的程式碼，並將該系統應用到產品中。

千錘百鍊法

從小而簡單的程式開始，然後一再地編輯它，上百次，甚至上千次。每次都做一些小修改，增加功能並改良程式。有些程式是經過百千次的編輯後才能運作的。

讀者練習

練習 9-1：寫一個程序來算出字串中的字數（用文件說明計算字數的方式）。再寫一個程式來測試這個程序。

練習 9-2：寫一個函式 `begins (string1, string2)`，若 `string2` 以 `string1` 開頭，就傳回 `true`。寫一個程式來測試這個函式。

練習 9-3：寫一個函式 `count (number, array, length)`，計算 `array` 中出現 `number` 的次數，這個陣列大小為 `length`（元素數目），請用遞迴來處理。寫一個程式測試此函式。

練習 9-4：寫一個函式讀取一個字串，傳回字串中每個字元相加後的數值。

練習 9-5：寫一個函式傳回一個數值陣列中的最大值。

練習 9-6：寫一個函式將字串中的 "-" 字元以 "_" 字元取代。

問題解答

答案 9-1：程式設計師往往搞不懂，for 迴圈為什麼不做事（除了將索引值遞增）；然而，for 迴圈的結束沒有分號，C++ 會繼續讀取，直到它看到下一個敘述（這裡是 `return(index)`），並且將它放進 for 迴圈。範例 9-11 才是正確的做法。

範例 9-11：length/rlen.cpp

```
int  length(char string[])
{
    int index;       // 字串的索引

    /*
     * 一直進行處理到字串結束的字元
     */
    for (index = 0; string[index] != '\0'; ++index)
        /* 沒事 */ ;
    return (index);
}
```