Appendix I

轉換成 ANSI/ISO 標準的 C++

你會有一些以 C 發展的程式（或程式設計習慣），或是較舊版本的 C++ 程式想要轉成 Standard C++。這附錄提供一些方向，有些是與 C 移至 C++ 相關，有些則與較舊之 C++ 移至標準的 C++ 相關。

使用一些前處理指令

C/C++ 前處理程式提供一系列的指令。一般而言，C++ 的習慣是用這些指令管理編譯的過程，並避免將指令作為取代程式碼之用。例如，#include 指令是管理程式檔案的基本元件。其他指令如 #ifndef 和 #endif，可以控制特定的區塊程式碼是否已經編譯。#pragma 指令可以控制與編譯程式有關的編譯選項。這些都是有用的，有時是必要的工具。但是在使用 #define 指令時要特別小心。

定義常數使用 const，不要用 #define

符號常數（symbolic constant）可使程式碼更容易閱讀和管理。常數的名稱表示其意義，而且若需要修改其值，只要修改其定義，然後重新編譯即可。對此目的，C 使用前處理程式來對常數建立符號名稱：

```
#define MAX_LENGTH 100
```

然後前處理程式會對原始程式碼作文字取代，在編譯之前將 MAX_LENGTH 取代成 100。

C++ 的方法是用 const 修飾變數的宣告：

```
const int MAX_LENGTH = 100;
```

這將 MAX_LENGTH 視為唯讀的 int。

const 方法有數個優點。首先，這宣告明確的指出型態。對於 #define，必須在數字後用各種字尾表示型態，而不是用 char，int，或 double，如 100L 表示 long

型態，3.14F 表示 float 型態。更重要的是，const 方法可以容易的用在衍生型態，好比這裡的範例：

```
const int base_value[5] = {1000, 2000, 3500, 6000, 10000};
const string ans[3] = {"yes", "no", "maybe"};
```

最後，const 識別字與變數一樣遵守相同的範疇規則。因此，你可以產生全域範疇，具名的名稱空間範疇，和區塊範疇的常數。若在某一函數中定義常數，你並不用擔心，與用於程式其他地方的全域常數產生定義衝突。例如：

```
#define n 5
const int dz = 12;
...
void fizzle()
{
    int n;
    int dz;
...
}
```

前處理程式會將程式碼

```
int n;
```

取代成

```
int 5;
```

因而產生編譯錯誤。而定義在 fizzle() 中的 dz 會是區域變數。而且，若 fizzle() 在必要時可以用範疇運算子（::）存取常數，寫法是 ::dz。

C 從 C++ 中借用關鍵字 const，但是 C++ 版本的 const 更有用。例如，C++ 版本對於外部 const 值具有內部連結性，不像變數和 C const 所用的預設外部連結性。這意思是，程式中每個使用 const 的檔案，需要該 const 定義在特定的檔案中。這似乎會有額外的工作，但是事實上會使工作較簡單。因為內部連結性，所以可將 const 定義置於標頭檔，專案的其他檔案即可使用之。對於外部連結性這是編譯程式的錯誤，但對內部連結性則否。因為 const 必須定義在使用它的檔案中（定義成標頭檔，再交給檔案使用可滿足此需求），所以可用 const 值作為陣列大小的引數：

```
const int MAX_LENGTH = 100;
...
double loads[MAX_LENGTH];
```

```
for (int i = 0; i < MAX_LENGTH; i++)
    loads[i] = 50;
```

這在 C 是不可行的，因為 MAX_LENGTH 的定義宣告可能是在不同的檔案中，而且在編譯含此陣列的檔案時，含 MAX_LENGTH 定義的檔案是不可用的。在 C 可用 static 產生具內部連結性的常數。在 C++中，static 是預設的方式，你可以少擔心一件事情。

修正過的 C 標準（C99）允許你用 const 作為陣列的大小，但是此陣列會視為新格式的陣列，稱為**可變動陣列**（variable array），這不是 C++ 標準的一部份。

#define 指令仍是控制標頭檔編譯的標準作法：

```
// blooper.h
#ifndef _BLOOPER_H_
#define _BLOOPER_H_
// code goes here
#endif
```

對於典型的符號常數要習慣使用 const，而不要用 #define。另一個不錯的方法是，當有一組相關的整數常數時，使用 enum：

```
enum {LEVEL1 = 1, LEVEL2 = 2, LEVEL3 = 4, LEVEL4 = 8};
```

定義簡短的函數用 inline，不要用 #define

在傳統 C 中，產生幾近內嵌函數的方法是用 #define 巨集定義：

```
#define Cube(X) X*X*X
```

這會使前處理程式執行文字取代，其 x 被取代成 Cube() 對應的引數：

```
y = Cube(x);       // replaced with y = x*x*x;
y = Cube(x + z++);  // replaced with x + z++*x + z++*x + z++;
```

因為前處理程式是用文字取代，而不是真正傳遞引數，使用這種巨集會導致意料之外的不正確結果。可在巨集中用大量的括號確保操作的正確順序，而減少這種錯誤：

```
#define Cube(X) ((X)*(X)*(X))
```

即使如此，這巨集仍無法處理，如 z++這樣的情況。

C++ 的方法是用關鍵字 inline 標示內嵌函數，這是更可靠的，因為它使用真正的引數傳遞。而且 C++ 的內嵌函數，可以是一般的函數或是類別成員函數：

```
class dormant
{
private:
    int period;
    ...
public:
    int Period() const { return period; } // automatically inline
    ...
};
```

#define 巨集的一個特性是它無型態，所以它可以用在此操作有意義的任何型態。在 C++ 中可產生內嵌樣版達到與型態無關的函數，而仍保留引數傳遞。

簡而言之，要用 C++ 的內嵌而不要用 C 的 #define 巨集。

使用函數原型

實際上，你沒有選擇。雖然在 C 中函數原型是選擇性的，但在 C++ 中則是必要的。注意，函數定義在它的第一次使用之前，如內嵌函數，都可以作為自己的函數原型。

要在適當時候於函數原型和標頭檔中使用 const。尤其是在表示不能修改的指標參數和 reference 參數時，要用 const。不只是編譯程式可以抓到修改資料的錯誤，也可以使函數更具一般性。也就是說具有 const 指標或 reference 的函數，可以處理 const 和非-const 的資料，而不用 const 指標或 reference 的函數只能處理非 const 的資料。

型態轉換

Stroustrup 對 C 不滿的一點是，其型態轉換運算子沒有規範。型態轉換常常是必須的，但是標準的型態轉換太過鬆散。例如：

```
struct Doof
{
    double feeb;
    double steeb;
    char sgif[10];
};
Doof leam;
short * ps = (short *) & leam;  // old syntax
int * pi = int * (&leam);       // new syntax
```

在 C 語言中，可以任意轉換指標型態為完全不相干的型態。

在某一方面，這狀況類似於 goto 敘述。goto 敘述的問題是彈性太大，而導致奇怪的程式碼。這解決方法是提供較有限制，結構化的 goto 版本處理 goto 需要處理的最常見工作。這是語言元素的起源，如 for 和 while 迴圈以及 if else 敘述。Standard C++ 對於無節制之型態轉換的問題，提供類似的解決方法，那就是將型態轉換限制在處理最常見的型態轉換。下面是第 15 章討論的型態轉換運算子：

```
dynamic_cast
static_cast
const_cast
reinterpret_cast
```

所以若要執行含有指標的型態轉換，盡可能使用這些運算子。這樣既可以說明轉型的目的，也可以提供轉型的檢查。

熟悉 C++ 的特性

若你已經使用 malloc() 和 free()，轉換成使用 new 和 delete。若你用 setjmp() 和 longjmp() 來處理錯誤，則要改用 try，throw，和 catch。對於表示 true 和 false 之值要用 bool 型態。

使用新的標頭檔

C++ 標準中，規定新的標頭檔案名稱，如第 2 章所述。若你是使用舊式的標頭檔，應修改使用新式的名稱。這不只是表面的修改，因為新的版本會加入新的特性。例如，ostream 標頭檔支援寬字元的輸入和輸出，也提供新的運作子，如 boolalpha 和 fixed（如第 17 章所述）。相較於指定許多格式化選項的 setf() 或是 iomanip 函數，這些提供較簡單的介面。若你是用 setf()，在指定常數時要用 ios_base 取代 ios，也就是說用 ios_base::fixed，而非 ios::fixed。還有新的標頭檔整合了名稱空間。

使用名稱空間

名稱空間有助於組織程式所使用的識別字，避免名稱衝突。因為標準函式庫是以新的標頭檔組織實作，將名稱置於 std 名稱空間中，使用這些標頭檔需要處理名稱空間。

為了簡化起見，本書的範例利用 using 指令，使 std 名稱空間中的所有名稱都是可用的：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;                    // a using-directive
```

但是整個輸出名稱空間中的所有名稱，不管是否需要，都違背名稱空間的目的。

比較好的方式是，將 using 指令放在函數裡面；使得這些名稱只能在函數裡面使用。

另外更好且建議的用法是，用 using 宣告或是範疇運算子（::），使程式所需的名稱變成可用。例如：

```
#include <iostream>
using std::cin; // a using-declaration
using std::cout;
using std::endl;
```

在檔案的其餘部分都可以使用 cin，cout，和 endl。範疇運算子只有讓使用此運算子的運算式可以使用此名稱：

```
cout << std::fixed << x << endl;      //using the scope resolution operator
```

這有點麻煩，但是你可以收集常用的 using 宣告置於標頭檔中：

```
// mynames -- a header file
using std::cin; // a using-declaration
using std::cout;
using std::endl;
```

而且你可將 using 宣告收集在名稱空間中：

```
// mynames -- a header file
#include <iostream>
namespace io
{
    using std::cin;
    using std::cout;
    using std::endl;
}

namespace formats
{
   using std::fixed;
   using std::scientific;
   using std:boolalpha;
}
```

然後程式要引入這檔案，並使用所需的名稱空間：

```
#include "mynames"
using namespace io;
```

使用聰明指標

每次使用 new 時就應成對的使用 delete。若使用 new 的函數因丟出異常而提早結束，則會產生問題。如第 15 章的討論，使用 autoptr 物件記錄 new 產生的物件會自動啟動 delete。C++11 還附加 unique_ptr 與 shared_ptr，提供更好的替代方案。

使用 string 類別

傳統 C-格式的字串並不是真正的型態。你可以將字串儲存在字元陣列中，用字串初始化字元陣列。但是你無法用指定運算子將字串指定給字元陣列，而是要用 strcpy() 或 strncpy()。你不能用關係運算子比較 C-格式字串，而是要用 strcmp()（若你忘了並使用如 > 運算子，這不會產生語法錯誤，相反的程式會比較字串位址而不是字串內容）。

string 類別（第 16 章和附錄 F）是用物件表示字串。指定，關係運算子，和加法運算子（連結）都已經定義。而且 string 類別提供自動的記憶體管理，所以你通常不用擔心輸入的字串超過陣列大小或是字串在儲存前被截斷。

string 類別提供許多便利的成員函數。例如，你可以將 string 物件附加在另一個物件之後，然而也可以附加 C-格式的字串或是 char 值至 string 物件。對於需要 C-格式字串引數的函數，你可用 c_str() 回傳合適的 char 指標。

string 類別不只是提供一組設計完整的成員函數可以處理字串的相關工作，如尋找子字串，而且其設計的特色是可以與 Standard Template Library（STL）相容，如此你可以用 STL 演算法處理 string 物件。

使用 STL

STL（第 16 章和附錄 G）對許多程式設計需求，提供一套準備好的解決方法，所以就使用它。例如，不要宣告 double 或 string 物件的陣列，而是產生 vector<double> 物件或是 vector<string> 物件。這優點類似於使用 string 物件而不使用 C-格式字串。已經定義指定，所以你可以用指定運算子將一個 vector 物件指定給另一個 vector 物件。可以傳遞 vector 物件的 reference，而接收這種物件的函數可以用 size() 成員函數，決定在 vector 物件中的元素個數。當使用 pushback() 將元素新增至 vector 物件時，內建的記憶體管理可以自動調整大小。當然還有數個有用的類別成員函數，及通用的演算法可以使用。

若你需要 list，雙向的佇列（或是 deque），stack，一般的佇列，set，或是 map，STL 都提供有用的收納器樣版。演算法函式庫的設計是為了使你容易地複製 vector 的內容至 list 中，或是比較 set 和 vector 的內容。這設計使 STL 變成工具，提供撰寫程式的基本元素。

廣泛的演算法函式庫其設計都以效率為主要目標之一，所以你可以用很少的程式設計心力而得到不錯結果。而且用來實作演算法的迭代器觀念，代表它們不限於使用 STL 收納器。尤其是它們也可以應用在一般的陣列。