1

C++ 編程基本教練

Basic C++ Programming

這一章，我們將從一個小程式出發，透過它來練習 C++ 程式語言的基本組成。其中包括了：

1. 一些基礎資料型別：布林值（Boolean）、字元（character）、整數（integer）、浮點數（floating point）。
2. 算術運算子、關係運算子、以及邏輯運算子，用以操作上述的基礎資料型別。這些運算子不僅包括一般常見的加法運算子、等號運算子（==）、小於等於（<=）運算子以及指派（assignment，=）運算子，也包含比較特殊的累加（++）運算子、條件運算子（?:）以及複合指派（+=）運算子。
3. 條件分支以及迴圈控制述句，例如 `if` 述句以及 `while` 迴圈，可用來改變程式的控制流程。
4. 一些複合型別，例如指標及陣列。指標可以讓我們間接參考一個已存在的物件，陣列則用來定義一組具有相同資料型別的元素。
5. 一套標準的、共通的抽象化程式庫，例如字串和向量（`vector`）。

1.1 如何撰寫 C++ 程式

此刻，假設我們需要撰寫一個簡易程式，必須能夠將一段訊息送至使用者的終端機（terminal）上。訊息的內容則是要求使用者鍵入自己的名字。然後程式必須讀取使用者所輸入的名字，將這個名字儲存起來，以便接下來的後續動作之用。最後，送出一個訊息，以指名道姓的方式對使用者打招呼。

好，該從何處著手呢？每個 C++ 程式都是從一個名爲 `main` 的函式開始執行，我們就從這個地方著手吧！`main` 是個由使用者自行撰寫的函式，其通用型式如下：

```
int main()
{
    // 我們的程式碼置於此處
}
```

int 是 C++ 程式語言的關鍵字。所謂關鍵字（keywords）乃是程式語言先行定義的一些具有特殊意義的名稱。int 用來表示語言內建的整數資料型別。（下一節我將針對資料型別做更詳細的說明）

函式（function）是一塊獨立的程式碼序列（code sequence），能夠執行一些運算。它包含四個部份：傳回值的型別（return type）、函式名稱、參數列（parameter list）、以及函式主體（function body）。讓我依序簡略介紹每一部份。

函式的傳回值通常用來表示運算結果。main() 函式傳回整數型別。main() 的傳回值用來告訴呼叫者，這個程式是否正確執行。習慣上，程式執行無誤時我們令 main() 傳回零。若傳回一個非零值，表示程式在執行過程中發生了錯誤。

函式的名稱由程式員選定。函式名稱最好能夠提供某些訊息，讓我們容易瞭解函式實際上在做些什麼。舉例來說，min() 和 sort() 便是極佳的命名。f() 和 g() 就沒有那麼好了。爲什麼？因爲後兩個名稱相形之下無法告訴我們函式的實際執行動作。

main 並非是程式語言定義的關鍵字。但是，執行起我們這個 C++ 程式的編譯系統，會假設程式中定義有 main() 函式。如果我們沒有定義，程式將無法執行。

函式的參數列（parameter list）由兩個括號括住，置於函式名稱之後。空的參數列如 main()，表示函式不接受任何參數。

參數列用來表示「函式執行時，呼叫者可以傳給函式的型別列表」。列表之中以逗號隔開各個型別（通常我們會說使用者「呼叫（*called*）」或是「喚起（*invoked*）」某個函式）。舉例來說，如果我們撰寫 main() 函式，使其傳回兩數中較小者，那麼它的參數列應該註明兩個即將被拿來比較的數值的型別。這樣一個用來比較兩整數值的 min() 函式，可能會以如下形式加以定義：

```
int min(int val1, int val2)
{
    // 程式碼置於此處
}
```

函式的主體（body）由大括號標出（{ }），其中含有「提供此函式之運算」的程式碼。雙斜線（//）表示該行內容爲註解，也就是程式員對程式碼所下的某些說明。註解的撰寫是爲了便利閱讀者更容易理解程式。編譯過程中，註解會被忽略掉。雙斜線之後直至行末的所有內容，都會被當做程式註解。

我們的第一件工作就是要將訊息送至使用者終端機上。資料的輸入與輸出，並非 C++ 程式語言本身定義的一部份（譯註：此精神同 C 語言，見 K&R 第七章），而是由 C++ 的一套物件導向類別體系（classes hierarchy）提供支援，並作爲 C++ 標準程式庫（standard library）的一員。

所謂類別（class），是使用者自定的資料型別（user-defined data type）。class 機制讓我們得以將資

料型別加入我們的程式中，並有能力識別它們。物件導向的類別體系（class hierachy）定義了一整個家族體系的各相關型別，例如終端機與檔案輸入裝置、終端機與檔案輸出裝置等等。關於類別以及物件導向編程（object-oriented programming）這兩個課題，本書還有許多篇幅會提到它們。

C++ 事先定義了一些基礎資料型別：布林值（Boolean）、字元（character）、整數（integer）、浮點數（floating point）。 雖然它們爲我們的編程任務提供了基石，但它們並非程式的重心所在。舉個例子，照相機具有一個性質：空間位置。這個位置通常可以用三個浮點數表示。照相機還具備另一個性質：視角方向，同樣也可以用三個浮點數表示。通常我們還會用所謂 aspect ratio 來描述照相機視窗的寬/高比，這只需單一浮點數即可表示。

最原始最基本的情況下，照相機可以用七個浮點數來表示，其中六個分別組成了兩組 x、y、z 座標。以這麼低階的方式來進行編程，我們勢必得讓我們的思考不斷地在「照相機抽象性質」和「相應於照相機的七個浮點數值」之間反覆來回。

class 機制，賦予我們「增加程式內之型別抽象化層次」的能力。我們可以定義一個 `Point3d` class，用來表示「空間位置」和「視角方向」兩性質。同樣道理，我們可以定義一個 `Camera` class，其中包含兩個 `Point3d` 物件，和一個浮點數。以這種方式，我們同樣使用七個浮點數值來表示照相機的性質，不同的是我們的思考不再直接面對七個浮點數，而是轉爲對 `Camera` class 的操作。

class 的定義，一般來說分爲兩部份，分別寫於不同的檔案。其中之一是所謂的「表頭檔（header file）」，用來宣告該 class 所提供的各種操作行爲（operations）。另一個檔案，程式本文檔（program text），則包含這些操作行爲的實作內容（implementation）。

欲使用 class，我們必須先在程式中含入其表頭檔。表頭檔可以讓程式知道 class 的定義。C++ 標準的「輸入/輸出 程式庫」名爲 iostream，其中包含了相關的整套 classes，用以支援對終端機和檔案的輸入與輸出。我們必須含入 iostream 程式庫的相關標頭檔，才能夠使用它：

```
#include <iostream>
```

我將利用已定義好的 `cout`（讀作 see out）物件，將訊息寫到使用者的終端機上。output 運算子（`<<`）可以將資料導至 `cout`，像這樣：

```
cout << "Please enter your first name：";
```

上述這行便是 C++ 所謂的「述句（statement）」。述句是 C++ 程式的最小獨立單元。就像自然語言中的句子一樣。述句以分號做爲結束。以上述句將常數字串（string literal，封裝於雙引號內）寫到使用者的終端機。在那之後，使用者便會看到如下訊息：

```
Please enter your first name：
```

接下來我們要讀取使用者的輸入內容。讀取之前，我們必須先定義一個物件，用以儲存資料。欲定義一個物件，必須指定其資料型別，再給定其識別名稱。截至目前，我們已經用過 int 資料型別。但是要用它來儲存某人的名字，幾乎是不可能的事。更適當的資料型別是標準程式庫的 string class ：

```
string user_name;
```

如此一來我們便定義了一個名爲 user_name 的物件，它隸屬於 string class。這樣的定義有個特別的名稱，稱作「宣告述句（declaration statement）」。單只寫下這行述句還不行，因爲我們還必須讓程式知道 string class 的定義。因此還必須在程式中含入 string class 的表頭檔：

```
#include <string>
```

接下來便可利用已定義好的 cin（讀作 see in）物件來讀取使用者於終端機上的輸入內容。透過 input 運算子（>>）將輸入內容導至具有適當型別的物件身上：

```
cin >> user_name;
```

以上所描述的輸出和輸入動作，於使用者終端機上顯示如下（輸入部份以粗體表示）：

```
Please enter your firzst name：anna
```

剩下的工作就是印出向使用者打招呼的訊息了。我們希望獲得這樣的輸出結果：

```
Hello, anna ... and goodbye!
```

當然啦，這樣子打招呼稍嫌怠慢。但這不過才第一章而已。本書結束之前我會有更具創意的招呼方式。

爲了產生上述訊息，我們的第一個步驟便是將輸出位置（螢幕上的游標）調到下一行起始處。將換行（newline）字元常數寫至 cout，便可達到這個目的：

```
cout << '\n';
```

所謂字元常數（character literal）係由一組單引號括住。字元常數分爲兩類：第一類是可列印字元，例如英文字母（'a', 'A', 等等）、數字、標點符號（';', '-', 等等）。另一類是不可列印字元，例如換行字元（'\n'）或跳格字元（tab，'\t'）。由於不可列印字元並無直接的表示法（譯註：這表示我們無法使用單一而可顯示的字元來獨立表示），所以必須以兩個字元所組成的字元序列表示之。

現在，我們已經將輸出位置調整到下一行起始處，接著要產生 Hello 訊息：

```
cout << "Hello, ";
```

接下來應該在此處輸出使用者的名字。這個名字已經儲存在 user_name 這個 string 物件中。我們應當如何進行呢？其實就和處理其它資料型別一樣，只要：

```
cout << user_name;
```

便大功告成。最後我們以道別來結束這段招呼訊息（注意，字串常數內可以同時包含可列印字元和不可列印字元）：

```
cout << " ... and goodbye!\n";
```

一般而言，所有內建資料型別都可以用同樣的方式來輸出 — 也就是說只須換掉 output 運算子右方的值即可。例如：

```
cout << "3 + 4 = ";
cout << 3 + 4;
cout << '\n';
```

會產生如下輸出結果：

```
3 + 4 = 7
```

當我們在自己的應用程式中定義了新的 classes，我們也應該爲每一個 class 提供它們自己的 output 運算子（第 4 章會告訴你如何辦到這件事情）。這麼一來便可以讓那些 classes 的使用者得以像面對內建型別一樣地以相同方式輸出物件內容。

如果嫌連續數行的輸出述句太煩人，也可以將數段內容連結成爲單一輸出述句：

```
cout << '\n'
     << "Hello, "
     << user_name
     << " ... and goodbye!\n";
```

最後，我們以 `return` 述句清楚地表示 `main()` 到此結束：

```
return 0;
```

`return` 是 C++ 的關鍵字。此例中的 0 是緊接於 `return` 之後的算式（expression），也就是此函式的傳回值。先前我曾說過，`main()` 傳回 0 即表示程式執行成功[1]。

將所有程式片段組合在一起，便是我們的第一個完整的 C++ 程式：

```
#include <iostream>
#include <string>
using namespace std;   // 此行目前尚未解釋...

int main()
{
    string user_name;
    cout << "Please enter your first name: ";
    cin >> user_name;
    cout << '\n'
         << "Hello, "
```

[1] 如果沒有在 `main()` 的末尾寫下 `return` 述句，此一述句會被自動加上。本書各程式範例中，我將不再明確寫出 `return` 述句。

```
            << user_name
            << " ... and goodbye!\n";

    return 0;
}
```

編譯並執行後，上述程式碼會產生如下的輸出結果（輸入部份以粗體字表示）：

```
Please enter your first name: anna
Hello, anna ... and goodbye!
```

整個程式中還有一行述句我尚未解釋：

```
using namespace std;
```

讓我們瞧瞧，如果試著這樣解釋會不會嚇到你（此刻，我建議你深吸一口氣）。using 和 namespace 都是 C++ 關鍵字。std 是標準程式庫所駐之命名空間（namespace）的名稱。標準程式庫所提供的任何事物（諸如 string class 以及 cout, cin 這兩個 iostream 類別物件）都被封裝在命名空間 std 內。當然啦，或許你接下來會問，什麼是命名空間？

所謂命名空間（namespace）是一種將程式庫名稱封裝起來的方法。透過這種方法，可以避免和應用程式發生命名衝突的問題（所謂命名衝突是指在應用程式內兩個不同的物體（entity）具有相同名稱，導致程式無法區分兩者。命名衝突發生時，程式必須等到該命名衝突獲得決議（*resolved*）之後，才得以繼續執行）。命名空間像是在眾多名稱的可見範圍之間豎起的一道道圍牆。

若要在程式中使用 string class 以及 cin、cout 這兩個 iostream 類別物件，我們不僅得含入 <string> 及 <iostream> 表頭檔，還得讓命名空間 std 內的名稱曝光。而所謂的 using directive：

```
using namespace std;
```

便是讓命名空間中的名稱曝光的最簡單方法。（[LIPPMAN98] 8.5 節和 [STROUSTRUP97] 8.2 節有命名空間（namespace）的更多相關資訊）

練習 1.1

將先前介紹的 main() 程式依樣畫葫蘆地鍵入。你可以直接鍵入程式碼，或是從網路下載。本書前言已經告訴你如何獲得書中範例程式的原始碼檔案以及練習題解答。試著在你的系統上編譯並執行這個程式。

練習 1.2

將 string 表頭檔改爲註解：

```
// #include <string>
```

重新編譯這個程式，看看會發生什麼事。然後取消對 string 標頭檔的註解動作，再將下一行註銷：

```
// using namespace std;
```

又會發生什麼事情？

練習 1.3

將 main() 函式名稱改爲 my_main()，然後重新編譯。有什麼結果？

練習 1.4

試著擴充這個程式的內容：(1) 要求使用者同時輸入名稱（first name）和姓氏（last name），(2) 修改輸出結果，同時列印姓氏和名稱。

1.2 物件的定義與初始化

現在，我們的程式引起了使用者的注意。讓我們出個小題目來考考他。我要顯示某數列中的兩個數字，然後要求使用者回答下一個數字爲何。例如：

```
The value 2, 3 for two consecutive
    elements of a numerical sequence.
What is the next value?
```

這兩個數字事實上是「費氏數列（Fibonacci sequence）」中的第三和第四個元素。費氏數列的前數個值分別是：1, 1, 2, 3, 5, 8, 13...。費氏數列的最前兩個數設定爲 1，接下來的每個數值都是前兩個數值的總和。（第二章會示範一個計算費氏數列的函式）

如果使用者輸入 5，我們就印出訊息，恭喜他答對，並詢問他是否願意試試另一個數列。如果使用者輸入不正確的值，我們就詢問他是否願意再試一次。

爲了提昇程式的趣味性，我們將使用者答對的次數除以其回答總次數，以此作爲評量標準。

這樣一來，我們的程式至少需要五個物件：一個 string 物件用來記錄使用者的名字，三個整數物件分別儲存使用者回答的數值、使用者回答的次數、以及使用者答對的次數；此外還需一個浮點數，記錄使用者得到的評分。

爲了定義物件，我們必須爲它命名，並賦予它資料型別。物件名稱可以是任何字母、數字、底線（underscore）的組合。大小寫字母是有所區分的，user_name, User_name, uSeR_nAmE, user_Name 所代表的都不是同一個物件。

物件名稱不能以數字爲首。舉例來說，1_name 是不合法的名稱，name_1 則合法。當然，任何命名都不能和程式語言本身的關鍵字雷同。例如 delete 是語言關鍵字，我們不可以將它用於程式內的

命名。這也就是爲什麼 string class 寧願採用 erase()（而非 delete()）來表示「刪去一個字元」的原因。

每個物件都隸屬某個特定的資料型別。物件名稱如果設計得好，可以讓我們直接聯想該物件的屬性。資料型別決定了物件所能持有的數值範圍，同時也決定了物件應該佔用多少記憶體空間。

前一節中我們已經看過 user_name 的定義。 新程式中我們再一次使用相同的定義：

```
#include <string>
string user_name;
```

所謂 class，便是程式員自行定義的資料型別。除此之外，C++ 還提供了一組內建的資料型別，包括布林值（Boolean）、整數（integer）、浮點數（floating point）、字元（character）。每一個內建資料型別都有一個相應的關鍵字，讓我們用來指定該型別。例如，爲了儲存使用者鍵入的值，我們定義一個整數物件：

```
int usr_val;
```

int 是 C++ 關鍵字，此處用來指示 usr_val 是個整數物件。使用者的「回答次數」以及「總共答對次數」也都是整數，唯一差別是，我們希望爲這兩個物件設定初值 0。下面這兩行可以辦到：

```
int num_tries = 0;
int num_right = 0;
```

在單一宣告述句中一併定義多個物件，其間以逗號區隔，也可以：

```
int num_tries = 0, num_right = 0;
```

一般來說，將每個物件初始化，是個好主意 — 即使初值只用來表示該物件尚未具有真正意義的值。我之所以不爲 usr_val設初值，因爲其值必須直接根據使用者的輸入加以設定，然後程式才能用之。

另外還有一種不同的初始化語法，稱爲「建構式語法（constructor syntax）」：

```
int num_tries(0);
```

我知道你心中有疑問：爲什麼需要兩種不同的初始化語法呢？爲什麼直到此刻才提起呢？唔，讓我們看看下面這個解釋能否回答其中一個問題，或甚至能同時回答兩個問題。

「以 assignment 運算子（=）進行初始化」這個動作係衍襲 C 語言而來。如果物件屬於內建型別，或者物件可以單一值加以初始化，這種方式就沒有問題。例如以下的 string class：

```
string sequence_name = "Fibonacci";
```

但是如果物件需要多個初始值，這種方式就沒辦法完成任務了。以標準程式庫中的複數（complex number）類別爲例，它就需要兩個初始值，一爲實部，一爲虛部。以下便是用來處理「多值初始化」的建構式初始化語法（constructor initialization syntax）：

```
#include <complex>
complex<double> purei(0, 7);
```

出現於 complex 之後的角括號，表示 complex 是一個 template class（範本類別）。本書對於 template class 另有更詳盡的討論。template class 允許我們在「不必指明 data members 之型別」的情況下定義 class。

舉個例子，複數類別內含兩個 member data object。 其一表示複數的實數部份，其二表示虛數部份。兩者都需要以浮點數來表現，但我們應該採用那一種浮點數型別呢？C++ 支援三種浮點數型別，分別是以關鍵字 float 表示的單精度（single precision）浮點數，以關鍵字 double 表示的倍精度（double precision）浮點數，以及連續兩個關鍵字 long double 表示的擴充精度（extended precison）浮點數。

template class 機制使程式員得以直到使用 template class 時才決定真正的資料型別。程式員可以先安插一個代名，稍後才繫結至實際的資料型別。上例便是將 complex 類別的成員繫結至 double 型別。

我知道，這些說明帶給你的只怕是疑問多過回答。然而，這是因爲，當「內建資料型別」與「程式員自行定義之 class 型別」具備不同的初始化語法時，我們無法撰寫出一個 template 使它同時支援「內建型別」與「class 型別」。讓語法統一，可以簡化 template 的設計。不幸的是，解釋這些語法似乎使事情益發複雜！

使用者獲得的評分可能是某個比值，所以我們必須以浮點數來表示。我以 double 型別定義之：

```
double usr_score = 0.0;
```

當我們詢問「再試一次？」以及「你是否願意回答其它型態的數列問題？」時，我們還必須將使用者的回答（yes 或 no）記錄下來。這個時候使用字元（char）物件就綽綽有餘了：

```
char usr_more;
cout << "Try another sequence? Y/N? ";
cin >> usr_more;
```

關鍵字 char 表示字元（character）型別。單引號括住的字元代表所謂的字面常數，例如 'a', '7', ';'。此外尚有一些特別的內建字元常數（有時也稱爲「逸出序列，escape sequence」），例如：

```
'\n'    換行字元（newline）
'\t'    跳格（定位）字元（tab）
'\0'    null
'\''    單引號（single quote）
'\"'    雙引號（double quote）
'\\'    反斜線（backslash）
```

舉個例子，我們想要在列印使用者姓名之前，先換行並跳一個定位（tab）距離，下面這行可以辦到：

```
cout << '\n' << '\t' << user_name;
```

另一種寫法是將兩個不同的字元合併成爲一個字串：

```
cout << "\n\t" << user_name;
```

我們常常會在字面常數中使用這些特殊字元。例如在 Windows 作業系統下以常數字串表示檔案路徑時，必須以「逸出序列，escape sequence）」表示反斜線字元

```
"F：\\essential\\programs\\chapter1\\ch1_main.cpp";
```

（譯註：由於反斜線字元已被用來做爲逸出序列的起頭字元，所以連續兩個反斜線即表示一個真正的反斜線字元）

C++ 提供內建的 Boolean 型別，用以表示真假值（true/false）。我們的程式中可以定義 Boolean 物件來控制是否要顯示下一組數列：

```
bool go_for_it = true;
```

Boolean 物件係由關鍵字 `bool` 指出。其值可爲 `true` 或 `false`（兩者都是常數）。

截至目前我們所定義出來的物件，其值都會在程式執行過程中改變。例如 `go_for_it` 最後會被設成 `flase`，使用者每次猜完數字之後，`usr_score` 的值也可能更動。

但有時候我們需要一些用來表示常數值的物件：像是使用者最多可猜多少次啦，或者像圓周率這類永恒不變的值。此等物件的內容在程式執行過程中不應有所更動。我們應當如何避免無意間更動此類物件的值呢？C++ 的 `const` 關鍵字可以派上用場：

```
const int max_tries = 3;
const double pi = 3.14159;
```

被定義爲 `const` 的物件，在獲得初值之後，無法再有任何變動。如果你企圖爲 `const` 物件指定新值，會產生編譯期錯誤。例如：

```
max_tries = 42;   // 錯誤：這是個 const 物件
```

1.3 撰寫算式（Expressions）

內建資料型別都可運用一組運算子，包括算術、相對關係、邏輯、複合指派（compound assignment）。各種算術運算子中，除了「整數除法」以及「餘數運算」外，並無出奇之處：

```
// 算術運算子（Arithmetic Operators）
+          加法運算          a + b
-          減法運算          a - b
*          乘法運算          a * b
/          除法運算          a / b
%          取餘數            a % b
```

兩個整數相除會產生另一個整數（譯註：商數）。小數點之後的部份會被完全捨棄；也就是說並沒有四捨五入。如果想要取得除法運算的餘數部份，可以使用 % 運算子：

```
5 / 3 核定爲 1       5 % 3 核定爲 2
5 / 4 核定爲 1       5 % 4 核定爲 1
5 / 5 核定爲 1       5 % 5 核定爲 0
```

嗯，什麼時機之下，我們會運用「餘數運算」呢？假設我們希望列印的資料每行不超過八個字串；尚未滿八個字串時，就在字串之後印出一個空白字元。如果已滿八個字串，就在字串之後輸出換行字元。以下便是實作法：

```
const int line_size = 8;
int cnt = 1;

// 以下述句將被我執行多次，每次 a_string 的內容
// 都不相同；每次執行完後，cnt 的值都會加 1。
cout << a_string
     << ( cnt % line_size ? ' ' : '\n');
```

其中緊接在 output 運算子(<<)之後，以括號括住的算式(expression)，是所謂的條件運算子(coditional operator）。如果餘數運算的結果爲零，則條件運算的結果爲 '\n'；如果餘數運算的結果非零，則條件運算的結果爲 ' '。讓我們看看這到底是什麼意思。

下面這個算式：

```
cnt % line_size
```

在 cnt 恰爲 line_size 的整數倍時，運算結果爲零，反之則爲非零。而條件運算子的一般形式如下：

```
expr
   ? 如果 expr 爲 true，就執行這裡
   : 如果 expr 爲 false，則執行這裡
```

如果 expr 的運算結果爲 true，那麼緊接在 '?' 之後的算式會被執行。如果 expr 的運算結果爲 false，那麼 ':' 之後的算式會被執行。在我們的程式中會導至餵給 output 運算子一個 ' ' 或是一個 '\n'。

條件式的值如果爲 0，會被視爲 false，其他非零值一律被視爲 true。此例中的 cnt 若非 8 的整數倍，條件式的值便不是 0，於是條件運算子中的 '?' 之後的部份就會被核定爲結果。

複合指派（compound assignment）運算子是一種便利的記號。當我們在物件身上施行某個運算子，然後將結果重新指派給該物件時，我們可能會這樣寫：

```
cnt = cnt + 2;
```

但是 C++ 程式員通常會寫成這樣：

```
cnt +=2 ;      // cnt 原值加 2
```

複合指派運算子可以和每個算術運算子結合，形成 +=, -=, *=, /=, 和 %=。

欲使物件值遞增或遞減，C++ 程式員會使用遞增（increment）運算子和遞減（decrement）運算子：

```
cnt++;   // cnt 的值累加 1
cnt--;   // cnt 的值遞減 1
```

遞增運算子和遞減運算子都有前置（prefix）和後置（postfix）兩種形式。前置式中，原值先遞增（或遞減）之後，才被拿來使用：

```
int tries = 0;
cout << "Are you ready for try #"
     << ++tries << "?\n";
```

上例中的 tries 被印出之前就已進行了遞增運算。至於後置式寫法，物件原值會先供給算式進行運算，然後才遞增（或遞減）：

```
int tries = 1;
cout << "Are you ready for try #"
     << tries++ << "?\n";
```

上例中的 tries 被印出之後才進行遞增運算。以上兩例的列印結果皆爲 1。

任何一個相對關係運算子（relational operator）的核定結果不是 true 就是 false。相對關係運算子包括以下六個：

```
==  相等     (equality)                 a == b
!=  不等     (inequality)               a != b
<   小於     (less than)                a < b
>   大於     (greater than)             a > b
<=  小於等於 (less than or equal)       a <= b
>=  大於等於 (greater than or equal)    a >= b
```

我們可利用相等（equality）運算子來檢驗使用者的回答：

```
bool usr_more = true;
char usr_rsp;

// 詢問使用者是否願意繼續下一個問題
// 將使用者的回答讀入 usr_rsp 之中
if ( usr_rsp == 'N' )
     usr_more = false;
```

if 述句之後的算式運算結果爲 true 時，條件成立，於是緊接其後的述句便會被執行。此例中如果 usr_rsp 等於 'N'，則 usr_more 會被設爲 false。反之如果 urs_rsp 不等於 'N'，那就什麼事也不會發生。不等運算子（inequality operator）的邏輯恰恰相反，例如：

```
if ( usr_rsp != 'Y' )
     usr_more = false;
```

麻煩的是，程式只檢驗 usr_rsp 的值是否爲 'N'，而使用者卻可能輸入小寫的 'n'。因此我們必須能夠辨識兩者。解決方法之一就是加上 else 子句：

```
if ( usr_rsp == 'N' )
     usr_more = false;
else
if ( usr_rsp == 'n' )
     usr_more = false;
```

如果 usr_rsp 之值爲 'N' ，usr_more 將被設爲 false，並結束整個 if 述句。如果其值不等於 'N'，那麼便開始評估接下來的 else 子句。換句話說當 usr_rsp 等於 'n'，usr_more 也會被設成 false。如果兩個條件都不符合，usr_more 不會被指派任何值。

新手常犯的錯誤便是將指派（assignment）運算子誤當做相等（equality）測試之用，例如：

```
// 呃，這會造成將常數字元 'N' 指派給 usr_rsp
// 而整個算式的運算結果爲 true
if ( usr_rsp = 'N' )
    // ...
```

OR 邏輯運算子（||）提供了上述問題的另一個解法。它讓我們得以同時檢驗多個算式的結果：

```
if ( usr_rsp == 'N' || usr_rsp == 'n' )
     usr_more = false;
```

只須左右兩個算式中的一個爲 true，OR 邏輯運算子的評估結果便爲 true。左側算式會先被評估，如果其值爲 true，剩下的另一個算式就不需再被評估（譯註：此爲所謂驟死式評估法）。本例之中，只有當 usr_rsp 不等於 'N' 時，才會再檢驗其值是否爲 'n'。

AND 邏輯運算子（&&）係在左右兩個算式的結果皆爲 true 時，其評估結果方爲 true。舉個例子：

```
if ( password &&
     validate( password ) &&
     ( acct = retrieve_acct_info( password ) ))
        // 處理帳戶（account）相關事務
```

最上一道算式會被先評估。其結果若爲 false，則 AND 運算子的評估結果即爲 false，其餘算式不會（不需要）再被評估。此例之中，當密碼已設定，而且密碼有效之後，帳戶的相關資訊才會被取出。

如果有一個算式的運算結果爲 false，那麼將 NOT 邏輯運算子施行於其上，結果爲 true。例如，我們可以把以下式子：

```
if ( usr_more == false )
     cout << "Your score for this session is "
          << usr_score << " Bye!\n";
```

寫成：

```
if ( !usr_more ) ...
```

運算子的優先序（precedence）

使用內建運算子時，你得明白一件事：如果同一個算式中使用多個運算子，其核定（*evaulate*）順序是由每一個運算子事先定義的優先等級來決定。例如 `5+2*10` 的運算結果是 25 而非 70，這是因爲乘法的優先等級比加法高，因此 2 先和 10 相乘，再加上 5。

如果想要改變內建的運算子優先序，可利用小括號。例如 `(5+2)*10` 的運算結果便是 70。

我將截至目前介紹過的運算子優先序簡列於下。位置在上者的優先序高於位置在下者。同一行的各種運算子具有相同的優先序，其評估次序視出現於算式中的位置而定（由左至右）。

邏輯運算子 NOT
算術運算子（`*, /, %`）
算術運算子（`+, -`）
相對關係運算子（`<, >, <=, >=`）
相對關係運算子（`==, !=`）
邏輯運算子 AND
邏輯運算子 OR
指派（assignment）運算子

舉個例子，當我們想判斷 `ival` 是否爲偶數時，可能會這麼寫：

```
! ival % 2   // 不完全正確
```

我們的想法是利用餘數運算子（`%`）來檢驗其結果。如果 `ival` 是偶數，則餘數運算的結果爲零，施行 NOT 邏輯運算之後，結果爲 `true`。如果餘數運算的結果不爲零，施行 NOT 邏輯運算之後，結果便爲 `false`。

不幸的是上述算式的結果和我們的想像大相逕庭。除非 `ival` 等於零，否則算式結果通通都是 `false`。

爲什麼？因爲邏輯運算 NOT 具備了較高的優先序，使得它最先被評估。 因此它先被施行於 `ival` 之上：如果 `ival` 不爲零，則運算結果爲 `false`，反之則爲 `true`。這個結果再成爲餘數運算的左運算元。而你知道，`false` 在算術算式中被視爲 0，`true` 被視爲 1，所以，根據 C++ 預設的運算優先序，除非 `ival` 的值爲零，否則上述算式便成了 `0%2`。

雖然這樣的結果並不是我們想要的，但也不能算是錯誤，嗯，至少不能算是語言上的錯誤。只能說是我們的程式邏輯的一種不正確的表達方式。編譯器不可能知道你的程式的邏輯。運算優先序是 C++ 編

程之所以複雜的原因之一。如果希望正確評估以上算式，我們應該明白地加上小括號，用以完成我們希望的運算優先序：

```
! ( ival % 2 )   // 正確的寫法
```

要避免此類問題，你必須坐下來好好地、深入地熟悉 C++ 運算子優先序。我並不打算在這裡介紹所有運算子，以及所有的運算優先序，以上介紹對於正在起步的初學者而言，應已十分足夠。如果想獲得更完整的說明，請參考 [LIPPMAN98] 第四章或 [STROUSTRUP97] 第六章。

1.4 條件（Conditaional）述句和迴圈（Loop）述句

基本上，從 `main()` 的第一行述句（statement）開始，程式裡的每行述句都只會被依序執行一次。我們已經在先前小節中對 `if` 述句做了驚鴻一瞥。`if` 述句讓我們依據某個算式的結果（視爲真假值）來決定是否執行一個或多個連續述句。可有可無的 `else` 子句，更可讓我們連續檢驗多個測試條件。至於迴圈（loop）述句，可以讓我們根據某個算式結果（視爲真僞條件），重覆執行單一或連續多個述句。下面以虛擬碼（pseudocode）所表示的程式中，使用了兩組迴圈述句（#1, #2），一組 `if` 述句（#5），一組 `if-else` 述句（#3）以及一組稱爲 `switch` 述句的條件述句（#4）。

```
// 虛擬碼（pseudocode）：程式運作邏輯的一般化表示
while 使用者想要猜測某個數列時
{ #1
    顯示該數列
    while 使用者所猜的答案並不正確 and
          使用者想要再猜一次
    {
        讀取使用者所猜的答案
        將 number_of_tries 數值加一
        if 答案正確
        { #3
            將 correct_guess 數值加一
            將 got_it 的值設爲 true
        } else {
            使用者答錯了，在此對他表示遺憾，並
                根據使用者已猜過的總數，產生不同的
                回應結果。      // #4
            詢問使用者是否願意再試一次
            讀取使用者的意願
            if 使用者回應 no   // #5
                將 go_for_it 的值設爲 false
        }
    }
}
```

條件述句（Conditional Statements）

if 述句中的條件算式（conditaion expression）必須寫在括號內。如果此算式的運算結果爲 true，那麼緊接在 if 之後的那一個述句便會被執行。

```
// #5
if ( usr_rsp == 'N' || usr_rsp == 'n' )
     go_for_it = false;
```

如果想要執行多個述句，那麼必須在 if 之後以大括號將這些述句括住（這樣便稱爲一個述句區段）：

```
// #3
if ( usr_guess == next_elem )
{ // 述句區段由此開始
   num_right++;
   got_it = true;
} // 述句區段在此結束
```

初學者常見的錯誤是忘了加上述句區段：

```
// 呃：忘了加上述句區段
// 兩個述句之中，只有 num_cor++ 是在 if 述句的控制範圍內。
// 至於 got_it = true; 這一行，不管條件是否成立都會被執行。

if ( usr_guess == next_elem )
     num_cor++;
     got_it = true;
```

got_it 之前的縮排反映出程式員的意圖。但是這並不會改變程式本身的行爲。是的，num_cor 的遞增與否與 if 述句相關，而且只有在使用者猜測的值（usr_guess）和下一元素（next_elem）之值相等時，它才會被執行。但是接下來的 got_it 述句卻和 if 述句絲毫無關，因爲我們忘了將這兩個述句同置於述句區段中。因此，本例之中不論使用者所猜的值究竟爲何，got_it 一定會被設成 true。

if 述句也可以配合 else 子句來使用。else 子句用以表示一旦 if 述句之測試條件不成立時，我們希望執行的單行述句或述句區段。

```
if (usr_guess == next_elem)
{
   // 使用者猜對了
}
else
{
   // 使用者猜錯了
}
```

使用 else 子句的第二種方式，便是將它和兩個（或更多）if 述句結合。舉例來說，如果使用者猜錯了，我們希望輸出的回應能夠依照使用者所猜過的總次數有所變化，這時候我們可以撰寫連續三個獨立的 if 述句來加以檢驗：

```
if ( num_tries == 1 )
   cout << "Oops! Nice guess but not quite it.\n";

if ( num_tries == 2 )
   cout << "Hmm. Sorry. Wrong a second time.\n";

if ( num_tries == 3 )
   cout << "Ah, this is harder than it looks, isn't it?\n";
```

三個測試條件中僅有一個會成立。如果其中一個 if 述句爲 true，其它兩者必爲 false。因此我們可以結合一連串的 else-if 子句，來反映這些 if 述句間的關係，也就是：

```
if ( num_tries == 1 )
   cout << "Oops! Nice guess but not quite it.\n";
else
if ( num_tries == 2 )
   cout << "Hmm. Sorry. Wrong a second time.\n";
else
if ( num_tries == 3 )
   cout << "Ah, this is harder than it looks, isn't it?\n";
else
   cout << "It must be getting pretty frustrating by now!\n";
```

第一個 if 述句會先被評估。如果其值爲 true，緊接在後的述句便會被執行，接下來的所有 else-if 子句都不會被評估。但如果第一個 if 述句的評估結果爲 false，便會接著評估下一個 if 述句，直到其中有某一個條件成立爲止。如果 num_tries 的值大於 3，也就是說所有條件都不成立，那麼最末的 else 子句會被執行。

巢狀的（nested）if-else 子句有個很容易令人困惑的地方，那就是要正確組織其邏輯其實是滿難的一件事。舉例來說，我們想要利用 if-else 述句將程式的運作分爲兩種情形：(1) 使用者猜對，(2) 使用者猜錯。以下第一種寫法並不會如我們所預期的方式運作：

```
if ( usr_guess == next_elem )
{
   // 使用者猜對了
}
else
if ( num_tries == 1 )
   // 輸出適當的回應結果
else
if ( num_tries == 2 )
   // 輸出適當的回應結果
else
if ( num_tries == 3 )
   // 輸出適當的回應結果
else
   // 輸出適當的回應結果
```

```
// 現在，詢問使用者是否願意再猜一次。
// 但只有在使用者猜錯時我們才應該這麼做。
// 呃，我們應該把程式碼放在哪兒呢？
```

每個 else-if 子句都無意識地形成二選一的命運，於是我們找不到地方安置處理使用者猜錯時的程式碼。以下是正確的組織方式：

```
if ( usr_guess == next_ elem )
{
   // 使用者猜對了
}
else
{
   // 使用者猜錯了
   if ( num_tries == 1 )
       // ...
   else
   if ( num_tries == 2 )
       // ...
   else
   if ( num_tries == 3 )
       // ...
   else // ...

   cout << "Want to try again? (Y/N)";
   char usr_rsp;
   cin >> usr_rsp;

   if ( usr_rsp == 'N' || urs_rsp == 'n' )
      go_for_it = false;
}
```

如果測試條件值屬於整數型別，我們還可以改用 switch 述句來取代一大堆的 if-else-if 子句：

```
// 等同於上述的 if-else-if 子句
switch ( num_tries )
{
  case 1:
      cout << "Oops! Nice guess but not quite it.\n";
      break;

  case 2:
      cout << "Hmm. Sorry. Wrong a second time.\n";
      break;

  case 3:
      cout << "Ah, this is harder than it looks, isn't it?\n";
      break;
```

```
        default:
            cout << "It must be getting pretty frustrating by now!\n";
            break;
    }
```

關鍵字 switch 之後緊接著一個由小括號括住的算式（是的，物件名稱也可視爲算式），該算式的核定值必須是整數型態。關鍵字 switch 之後是一組 case 標籤，每一個標籤之後都指定有一個常數算式。當 switch 之後的算式值被計算出來，便依序和每個 case 標籤的算式值比較。如果找到相符的 case 標籤，便執行該 case 標籤之後的述句。如果找不到吻合者，而 default 標籤有出現，便執行 default 標籤之後的述句。如果 default 標籤沒有出現，就不執行任何動作。

爲什麼我在每個 case 標籤的最末加上 break 述句呢？要知道，每個 case 標籤的算式值都會依序和 switch 算式值相比較，不吻合者便依序跳過。當某個 case 標籤之算式值吻合，便開始執行該 case 標籤之後的述句 — 這個執行動作會一直貫徹到 switch 述句的最底端。如果我們沒有加上 break 述句，而 num_tries 的值爲 2，那麼程式的輸出結果會是：

```
// 如果 num_tries 之值爲 2，而我們又忘了加上 break 述句，輸出結果如下
Hmm. Sorry. Wrong again.
Ah, this is harder than it looks, isn't it?
It must be getting pretty frustrating by now!
```

是的，當某個標籤和 switch 的算式值吻合時，該 case 標籤之後的所有 case 標籤也都會被執行，除非我們明確使用 break 來結束執行動作。這也正是 break 述句的用途。或許你心中升起疑問，爲什麼要把 switch 述句設計成這樣呢？下面的例子可以解釋這種「向下穿越」的行爲模式是正確的：

```
switch( next_char )
{
  case 'a': case 'A':
  case 'e': case 'E':
  case 'i': case 'I':
  case 'o': case 'O':
  case 'u': case 'U':
      ++vowel_cnt;
      break;
  // ...
}
```

迴圈述句（Loop Statement）

只要條件式不斷成立（亦即其運算結果爲 true），迴圈述句便會不斷地執行單一述句或整個述句區段。我們的程式需要用到兩個迴圈述句，其中一個被巢狀地置於另一個迴圈之中：

```
while 使用者想要猜數列的下一個數
{
    顯示數列
    while 使用者猜的答案並不正確 and
          使用者想要再猜一次
}
```

C++ 的 while 迴圈可以符合我們的需求：

```
bool next_sg = true;     // 顯示下一組數列
bool go_for_it = true;   // 使用者想再猜一次
bool got_it = false;     // 使用者是否猜對
int  num_tries = 0;      // 使用者猜過的總次數
int  num_right = 0;      // 使用者答對的總次數

while ( next_seq == true )
{
  // 爲使用者顯示數列
  while (( got_it == false) &&
        ( go_for_it == true ))
  {
     int usr_guess;
     cin >> usr_guess;
     num_tries++;
     if ( usr_guess == next_elem )
     {
        got_it = true;
        num_cor++;
     }
     else
     {  // 使用者猜錯了
         // 告訴使用者答案是錯的
         // 詢問使用者是否願意再試一次
         if ( usr_rsp == 'N' || usr_rsp == 'n' )
             go_for_it = false;
     }
  } // 內層的 while 迴圈結束

  cout << "Want to try another sequence? (Y/N)";
  char try_again;
  cin >> try_again;

  if ( try_again == 'N' || try_again == 'n' )
      next_seq = false;
} // while( mext_seq == true ) 結束
```

while 迴圈即將開始之際，會先評估括號內的條件式。如果其值爲 true，則緊接在 while 迴圈之後的述句或述句區段便會被執行起來。執行完畢之後，條件式會再被重新評估一次。這種評估/執行

的運作模式不斷循環，直到條件式的值變成 false。通常，述句區段在某種狀態下會將該條件式的值設爲 false。如果條件式的值永遠不爲 false，我們便陷入了一個無窮迴圈之中 — 這樣的程式邏輯當然不正確。

以上程式的外圍 while 迴圈會一直執行到使用者希望結束爲止。

```
bool next_seq = true;
while ( next_seq == true )
{
       // ...
       if ( try_again == 'N' || try_again == 'n' )
           next_seq = false;
}
```

如果 next_seq 初值爲 false，後面的述句區段都不會被執行起來。至於內層的 while 迴圈則讓使用者得以連續猜好幾次。

如果在執行迴圈內的述句時遇上 break 述句，迴圈便會結束。以下程式片段爲例，while 迴圈會一直執行到 tries_cnt 和 max_tries 相等爲止。一旦使用者猜對了，程式便以 break 述句來結束迴圈：

```
int max_tries = 3;
int tries_cnt = 0;
while ( tries_cnt < max_tries )
{
  // 讀取使用者的答案
  if ( usr_guess == next_elem )
      break; // 結束迴圈
  tries_cnt++;
  // 其它程式碼
}
```

我們也可以利用 continue 述句來遽然終止迴圈的現行迭代（current iteration）。例如以下程式片段，所有長度小於四個字元的字彙都會被捨棄掉：

```
string word;
const int min_size = 4;
while ( cin >> word )
{
  if ( word.size() < min_size )
       // 結束此次迭代
       continue;
  // 程式執行到此處，則使用者所輸入的字彙長度
  // 必然大於或等於 min_size 個字元 ...
  process_text(word);
}
```

倘若 word 的長度小於 min_size，便執行 continue 述句，於是結束迴圈的現行迭代動作，也就是說 while 迴圈中的剩餘部份（此例爲 process_text()）便不會被執行。迴圈會重新再來過，而條件式會被再一次評估 — 讀取另一個字串並儲入 word 之中。如果 word 的長度大於或等於 min_size，整個 while 迴圈內容便都會被執行。在此運作模式下，所有長度小於四個字元的字彙都會被捨棄掉。

1.5 如何運用 Arrays（陣列）和 Vectors（伸量）

以下是六種數列的前八個元素值：

```
Fibonacci：        1, 1, 2, 3, 5, 8, 13, 21
Lucas：            1, 3, 4, 7, 11, 18, 29, 47
Pell：             1, 2, 5, 12, 29, 70, 169, 408
Triangular：       1, 3, 6, 10, 15, 21, 28, 36
Square：           1, 4, 9, 16, 25, 36, 49, 64
Pentagonal：       1, 5, 12, 22, 35, 51, 70, 92
```

我們的程式必須能夠顯示數列中的任兩個元素值，讓使用者猜測下一個元素值是什麼。如果使用者猜對了，並且願意繼續下去，程式便接續顯示第二組、第三組…元素值。這要如何辦到呢？

如果接續出現的每組元素都出自於同一數列，那麼一旦使用者找出其中一組答案，他就能夠找出所有答案。這就喪失了趣味性。所以，我們應該在程式主迴圈的每次迭代中，挑選不同的數列。

現在，我們要顯示最多六組元素對（element pairs）：每一組來自不同的數列。我們希望在顯示每組元素的同時，不必知道正在顯示的是哪一種數列。每次迭代都必須存取三個數：「元素對」中的兩個元素值，以及數列中出現的第三個元素值。

本節所討論的問題，解決之道就是使用可存放連續整數值的容器（container）型別。這種型別不僅允許我們以名稱（name）取用容器中的元素，也允許我們以容器中的位置來取用元素。我打算在容器內放入 18 個數值，分爲六組。每一組的前兩個數值用於顯示，第三個數值表示數列中的下一元素值。在迴圈每次迭代過程中，我們令索引（index）每次增加 3，這樣就可以依序走訪六組數據。

C++ 允許我們以內建的 array（陣列）型別或標準程式庫提供的 vector 類別來定義容器。一般而言我建議使用 vector 甚於 array。不過，大量現存的程式碼都使用 array。因此，了解如何善用這兩種方式，便相當重要。

要定義 array，我們必須指定 array 的元素型別，還得給予 array 一個名稱，並指定其尺度大小 — 亦即 array 所能儲存的元素個數。array 的尺度必須是個常數算式（constant expression），也就是一個

不需要在執行期計算其值的算式。以下程式碼是個例子，其中宣告 pell_seq 是個 array，內含 18 個整數元素。

```
const int seq_size = 18;
int pell_seq[ seq_size ];
```

至於定義 vector object， 我們首先必須含入 vector 表頭檔。vector 是個 class template，所以我們必須在類別名稱之後的角括號內指定其元素型別，其尺度則寫在小括號中; 此處所給予的尺度並不一定得是個常數算式。下列程式碼將 pell_seq 定義爲一個 vector object，可儲存 18 個 int 元素，每個元素的初值爲 0。

```
#include <vector>
vector<int> pell_seq( seq_size );
```

無論 array 或 vector，我們都可以指定容器中的某個位置，進而存取該位置上的元素。索引動作（*indexing*）係透過下標運算子（[]）達成。這裡必須注意的小技巧是，容器的第一個元素位置爲 0 而非 1。因此容器的最後一個元素必須以容器的 size-1 加以索引。以 pell_seq 爲例，正確的索引值是 0~17，而非 1~18（此類錯誤極爲常見，因此這種錯誤有個聲名狼籍的名稱，叫做 off-by-one）。舉個例子，要指定 Pell 數列的前兩個元素值，可以這麼寫：

```
pell_seq[ 0 ] = 1; // 指定第一元素值爲 1
pell_seq[ 1 ] = 2; // 指定第二元素值爲 2
```

現在我們來計算 Pell 數列中接下來的 10 個元素。要依序迭代 vector 或 array 中的多個元素時，我們通常會使用 C++ 的另一個重要的迴圈述句，也就是 for 迴圈。例如：

```
for ( int ix = 2; ix < seq_size; ++ix )
     pell_seq[ ix ] = pell_seq[ ix-2 ] + 2*pell_seq[ ix-1 ];
```

for 迴圈包含以下數個組成：

```
for ( init-statement ; condition ; expression )
      statement
```

其中的 init-statement 會在迴圈開始執行之前被執行一次。在我們的例子中，ix 在迴圈開始之前，被設初值爲 2。

condition 做爲迴圈控制之用，其值會在每次迴圈迭代之前被評估出來。如果 condition 爲 true，statement 便會被執行起來。statement 可以是單一述句，也可以是個述句區段。如果 condition 第一次評估值即爲 false，那麼 statement 一次也不會執行。在我們的例子中，condition 用來檢驗 ix 的值是否小於 seq_size。

expression 會在迴圈每次迭代結束之後被評估。通常它用來更改兩種物件的值。一個是在 init-statement 中被初始化之物件，另一個是在 condition 中被檢驗的物件。如果 condition 第一次評估值即爲

false，那麼 expression 不會被執行。在我們的例子中，每次迴圈迭代結束之後 ix 的值便累加 1。

如果想印出每一個元素值，我們可以迭代（走訪，*iterate*）整個集合：

```
cout << "The first " << seq_size
     << " elements of the Pell Series:\n\t";

for （int ix = 0; ix < seq_size; ++ix）
     cout << pell_seq[ ix ] << ' ';

cout << '\n';
```

如果我們願意，也可以在 init-statement 或 expression 或甚至 condition 處（較罕見）留白，不寫任何東西。例如我們可以將上述的 for 迴圈改爲：

```
int ix = 0;
// ...

for ( ; ix < seq_size; ++ix )
     // ...
```

其中的分號是必要的，因爲必須利用它來表示 init-statement 留白。

我們的容器儲存了六個數列中的每一個數列的第二、第三、第四元素。我們應該如何將適當的值填入此容器中呢？如果是 array，我們可以指定初始化序列（initialization list），藉由逗號區隔每一個欲指定的值，這些值便成爲 array 的全部元素或部份元素：

```
int elem_seq[ seq_size ] = {
  1, 2, 3,     // Fibonacci
  3, 4, 7,     // Lucas
  2, 5, 12,    // Pell
  3, 6, 10,    // Triangular
  4, 9, 16,    // Square
  5, 12, 22    // Pentagonal
};
```

初始化序列內的元素個數，不能超過 array 的尺度。如果前者的元素數量小於 array 的大小，其餘的元素值會被初始化爲 0。如果我們願意的話，可以讓編譯器根據初值的數量，自行計算出 array 的容量：

```
// 編譯器會算出此 array 包含了 18 個元素
int elem_seq[] = {
  1, 2, 3,  3, 4, 7,  2, 5,  12,
  3, 6, 10, 4, 9, 16, 5, 12, 22
};
```

vector 不支援上述這種初始化序列。有個冗長的寫法可以爲每個元素指定其值：

```
vector<int> elem_seq( seq_size );
elem_seq[ 0 ] = 1;
elem_seq[ 1 ] = 2;
// ...
elem_seq[ 17 ] = 22;
```

另一個方法都是利用一個已初始化的 array 做爲 vector 的初值：

```
int elem_vals[ seq_size ] = {
  1, 2, 3,  3, 4, 7,  2, 5,  12,
  3, 6, 10, 4, 9, 16, 5, 12, 22
};

// 以 elem_vals 的值來初始化 elem_seq
vector<int> elem_seq( elem_vals, elem_vals+seq_size );
```

上例中我們傳入兩個值給 elem_seq。這兩個值都是實際記憶體位置。它們標示出「用以將 vector 初始化」的元素範圍。此例中我們標示出 elem_vals 內的 18 個元素，並將它們複製到 elem_seq。第三章會說明這種運作方式的細節部份。

現在，讓我們看看如何使用 elem_seq。array 和 vector 之間存在一個差異，那就是 vector 知道自己的大小爲何。之前我們以 for 迴圈迭代 array 的作法，如果應用於 vector 之上，情況稍有不同：

```
// elem_seq.size() 會傳回 elem_seq 這個 vector
// 所包含的元素個數
cout << " The fitst " << elem_seq.size()
     << " elements of the Pell Series:\n\t";

for ( int ix = 0; ix < elem_seq.size(); ++ix )
     cout << pell_seq[ ix ] << ' ';
```

下面以 cur_tuple 表示欲顯示之元素的索引值。首先將它初始化爲 0。每次迴圈迭代之中，我們將其值累加 3，使它能夠索引到下一個數列的第一元素。

```
int cur_tuple = 0;

while ( next_seq == true &&
       cur_tuple < seq_size )
{
       cout << "The first two elements of the sequence are: "
            << elem_seq[ cur_tuple ] << ", "
            << elem_seq[ cur_tuple+1 ]
            << "\nWhat is the next element?";

       // ...
       if ( usr_guess == elem_seq[ cur_tuple+2 ] )
           // correct!

       // ...
```

```
        if ( usr_rsp == 'N' || usr_rsp == 'n' )
            next_seq = false;
        else cur_tuple += 3;
    }
```

將目前進行中的數列名目記錄下來，應該頗有用處。首先我們將每個數列的名稱都用 string 儲存起來：

```
    const int max_seq = 6;
    string seq_names[ max_seq ] = {
      "Fibonacci",
      "Lucas",
      "Pell",
      "Triangular",
      "Square",
      "Pentagonal"
    };
```

我們可以採用以下作法來運用 seq_names：

```
    if ( usr_guess == elem_seq[ cur_tuple+2 ] )
    {
        ++num_cor;
        cout << "Very good. Yes, "
             << elem_seq[ cur_tuple+2 ]
             << " is the next element in the "
             << seq_names[ cur_tuple/3 ] << "sequence.\n";
    }
```

cur_tuple/3 這一算式會依序產生 0、1、2、3、4、5，用來索引出一個字串，代表目前正在進行的猜數遊戲中的數列種類。

1.6 指標帶來彈性

前一節所顯示的解法有兩大缺點。第一，其上限是六個數列；如果使用者猜完了這六個數列，程式會無預期地結束。第二，這個方法每次都以同樣的順序顯示六組元素。如何才能夠拓展程式的彈性呢？

一種可能的解法便是同時維護六個 vectors，每個數列使用一個。每個 vector 儲存某一數量的元素值。每一次迴圈迭代，我們從不同的 vector 取出一組元素值。當第二次用到相同的 vector 時，便以不同的索引值取出 vector 內的元素。這個方法可以解決上述缺點。

就如同早先的解法一樣，我們希望透通地存取不同的 vector。前一節係透過索引（而非名稱）來存取每個元素，藉此達到透通化的目的。每次迴圈迭代，我們將索引加 3。如若不然，程式的執行結果就不會改變。

這一節我們將透過指標（pointer）的運用，捨棄以名稱指定的方式，間接地存取每個 vector，藉此達到透通化的目的。指標爲程式引入了一層間接性。我們可以操控指標（代表某持定記憶體位址），而不再直接操控物件。在我們的程式中，我定義了一個可以對整數 vector 定址的指標。每一次迴圈迭代，便更改指標值，使它定址到不同的 vector。隨後的指標操控行爲不需更動。

在我們的程式中，指標主要形成兩件事情。它可以增加程式本身的彈性，但同時也增加了直接操控物件時所沒有的複雜度。本節內容會令你相信這兩個說法。

我們早已了解如何定義物件。以下述句將 ival 定義爲一個 int 物件，並給予初值 1,204：

```
int ival = 1024;
```

指標內含某特定型別之物件的記憶體位址。當我們要定義某個特定型別的指標時，必須在型別名稱之後加上 * 號：

```
int *pi;  // pi 是個 int 型別的物件的指標
```

pi 是個 int 型別的物件的指標。我們應當如何爲指標設定初值呢？如果以物件名稱來執行核定動作，例如：

```
ival;     // 核估 ival 之值
```

會得到 ival 所存之值。當我們希望取得物件所在的記憶體位址，而非物件的值時，應該使用取址運算子（&）來達成：

```
&ival;    // 核估 ival 所在的記憶體位址
```

下述寫法可將 pi 的初值設爲 ival 所在的記憶體位址：

```
int *pi = &ival;
```

如果要存取一個由指標定址的物件，我們必須對該指標進行提領（*dereference*）動作 — 也就是取得「位於該指標所指之記憶體位址上」的物件。在指標之前使用 * 號，便可以達到這個目的：

```
// 提領 pi，藉以存取它所定址的物件
if ( *pi != 1024 )  // 讀取 ival 的值
     *pi = 1024;     // 寫值至 ival
```

指標的複雜度，如你所見，源於其令人困惑的語法。此例中可能令人感到複雜的地方，便是指標所具有的雙重性質：既可以讓我們操控指標內含的記憶體位址，也可以讓我們操控指標所指的物件值。當我們這麼寫：

```
pi;  // 核定 pi 所持有的記憶體位址
```

此舉形同操控「指標物件」本身。而當我們寫：

```
*pi; // 核定 ival 之值
```

等於是操控 pi 所指之物件。

指標的第二個可能令人感到複雜的地方是，指標可能並不指向任何物件。當我們寫 *pi 時，這種寫法可能會（也可能不會）使程式在執行期形成錯誤。如果 pi 定址到某個物件，則對 pi 進行提領（*dereference*）動作並沒有錯誤。但如果 pi 不指向任何物件，提領 pi 會導致未知的執行結果。這意謂當我們使用指標時，必須在提領它之前先確定它的確指向某物件。該怎麼做呢？

一個未指向任何物件的指標，其內含位址爲 0。有時候我們稱之爲 null 指標。任何指標都可以被初始化，或是令其值爲 0。

```
// 初始化每個指標，使它們不指向任何物件

int *pi = 0;
double *pd = 0;
string *ps = 0;
```

爲了防止對 null 指標進行提領動作，我們可以檢驗該指標所持有的位址是否爲 0。例如：

```
if ( pi && *pi != 1024 )
     *pi = 1024;
```

以下這個算式：

```
if ( pi && ... )
```

只有在 pi 持有一個非零值時，其核定結果方才爲 true。如果核定結果爲 false，那麼 AND 運算子就不會評估其第二算式。欲檢驗某指標是否爲 null，我們通常使用邏輯運算子 NOT：

```
if ( !pi )  // 當 pi 之值爲 0，此算式方爲 true
```

以下便是我們的六個 vector 物件（代表六種數列）：

```
vector<int> fibonacci, lucas, pell, triangular, square, pentagonal;
```

當我們需要一個指標，指向一個「元素型別爲 int」的 vector，該指標應該是什麼模樣呢？通常，指標符合以下形式：

```
type_of_object_pointed_to * name_of_pointer_object
```

由於我們所要的指標係用來指向 vector<int>，我把它命名爲 pv，並給定初值 0：

```
vector<int> *pv = 0;
```

pv 可以依序指向每一個用以表示數列的 vector。當然啦，我們也可以明白地將數列的記憶體位址指派給它：

```
pv = &fibonacci;
// ...
pv = &lucas;
```

但這種指派方式會犧牲程式的透通性。另一種解法是將每個數列的記憶體位址存入某個 vector 中，於是我們可以透過索引的方式，透明地存取這些數列：

```
const int seq_cnt = 6;

// 一個指標陣列，容量爲 seq_cnt，
// 每個指標都指向 vector<int> 物件
vector<int> *seq_addrs[ seq_cnt ] = {
    &fibonacci,  &lucas,  &pell，
    &triangular, &square, &pentagonal
};
```

seq_addrs 是個 array， 其元素型別爲 vector<int> *。seq_addrs[0] 所持有的值是 fibonacci vector 的位址，seq_addrs[1] 的值是 lucas vector 的位址，依此類推。我們透過一個索引值（而非透過其名稱）來存取個別的 vector：

```
vector<int> *current_vec = 0;
// ...

for ( int ix = 0; ix < seq_cnt; ++ix )
{
     current_vec = seq_addrs[ ix ];
     // 所有欲顯示的元素都透過 current_vec 間接存取而得
}
```

最後，剩下一個問題懸而未決。在這種實作方式下，程式的執行結果完全可以預測。要給使用者猜測的數列，總是依 Fibonacci, Lucas, Pell、... 的順序出現。我們希望讓數列的出現順序隨機化（*randomize*）。這可透過 C 語言標準程式庫中的 rand() 和 srand() 兩個函式達成：

```
#include <cstdlib>

srand( seq_cnt );
seq_index = rand() & seq_cnt;
current_vec = seq_addrs[ seq_index ];
```

rand() 和 srand() 都是標準程式庫提供的所謂虛擬亂數（pseudo-random number）產生器。srand() 的參數是所謂亂數產生器種子（seed）。要知道，每次呼叫 rand()，都會傳回一個介於 0 和「int 所能表示之最大整數」間的一個整數。現在，將亂數產生器的種子（seed）設爲 5，我們就可以將 rand() 的傳回值限制在 0 和 5 之間，以便成爲本例的一個有效索引。這兩個函式的宣告式位於 cstdlib 表頭檔。

使用 class object 的指標，和使用內建型別的指標略有不同。這是因爲 class object 聯結到一組我們可以喚起（*invoke*）的操作行爲（operations）。舉例來說，欲檢查 fibonacci vector 的第一個元素是否爲 1，我們可能會這麼寫：

```
if ( ! fibonacci.empty() &&
     ( fibonacci[1] == 1 ) )
```

如何才能間接透過 pv 達到同樣的作用呢？上例中的 fibonacci 和 empty() 兩字之間的逗號，稱爲 dot 成員選擇運算子（member selection operator），用來選擇我們想要施行的操作行爲。如果要透過指標來選擇操作行爲，必須改用 arrow（而非 dot）成員選擇運算子：

```
! pv->empty()
```

由於指標可能並未指向任何物件，所以在我們喚起 empty() 之前，應該先檢驗 pv 是否爲非零值：

```
pv && ! pv->empty()
```

最後，如果要使用下標運算子（subscript operator），我們必須先提領 pv。由於下標運算子的優先序較高，因此 pv 提領動作的兩旁必須加上小括號：

```
if ( pv && ! pv->empty() && ( ( *pv )[1] == 1 ) )
```

我將在第三章深入討論 Standard Template Library（STL），並在第六章設計二元樹（binary tree）時，回頭討論指標相關議題。[LIPPMAN98] 3.3 節對於指標有更深入的討論。

1.7 檔案的讀寫

使用者可能會一再執行這個程式。我們應該讓使用者的分數可以在不同的「執行期間（session）」累計使用。爲了達到這個目的，我們必須 (1) 每次執行結束時，將使用者的姓名及執行期間的某些資料寫入檔案，(2) 在程式開啓另一個執行期間之際，將資料從檔案中讀回。讓我們看看這要怎麼辦到。

欲對檔案進行讀寫動作，首先得含入 fstream 表頭檔：

```
#include <fstream>
```

爲了開啓一個可供輸出的檔案，我們定義一個 ostream（供輸出用的 file stream）物件，並將檔名傳入：

```
// 以輸出模式開啓 seq_data.txt
ofstream outfile( "seq_data.txt" );
```

宣告 outfile 的同時，會發生什麼事？如果指定的檔案並不存在，便會有一個檔案被產生出來並開啓做爲輸出之用。如果指定的檔案已經存在，這個檔案會被開啓做爲輸出之用，而檔案中原已存在的資料會被拋棄。

如果檔案已經存在，而我們並不希望拋棄其原有內容，而是希望增加新資料到檔案中，那麼我們必須以附加模式（append mode）開啓這個檔案。爲此，我們提供第二個參數 ios_base::app 傳給 ostream 物件。（此刻你最好暫時先放手運用它們，不必對於它們的艱深本質探問過深）

```
// 以附加模式（append mode）開啓 seq_data.txt
// 新資料會被加到檔案尾端
ofstream outfile( "seq_data.txt", ios_base::app );
```

檔案有可能開啓失敗。在進行寫入動作之前，我們必須確定檔案的確開啓成功。最簡單的方法便是檢驗 class object 的真僞：

```
// 如果 outfile 的評估結果爲 false，表示此檔案並未開啓成功
if ( ! outfile )
```

如果檔案未能成功開啓，ofstream 物件會被評估爲 false。本例中我們將訊息寫入 cerr，告知使用者這個狀況。cerr 代表標準示誤設備（standard error）。和 cout 一樣，cerr 將其輸出結果導至使用者的終端機。兩者的唯一差別是，cerr 的輸出結果並無緩衝（*bufferred*）情形 — 它會立即顯示於使用者終端機上。

```
if ( ! outfile )
    // 因爲某種原因，檔案無法開啓
    cerr << "Oops! Unable to save session data!\n";

else
    // ok：outfile 開啓成功，接下來將資料寫入
    outfile << usr_name  << ' '
            << num_tries << ' '
            << num_right << endl;
```

如果檔案順利開啓，我們便將輸出訊息導至該檔案，就像將訊息寫入 cout 及 cerr 這兩個 ostream 物件一樣。本例之中，我們將三個數值寫入 outfile，並以空白字元區分後兩個數值。endl 是事先定義好的所謂操控器（manipulator），由 iostream library 提供。

操控器並不會將資料寫到 iostream，也不會從中讀取資料，其作用是在 iostream 上執行某些動作。endl 會插入一個換行字元，並清除輸出緩衝區（output buffer）的內容。除了 endl，另有一些事先定義好的操控器，例如 hex（以 16 進位顯示整數）、oct（以 8 進位顯示整數）、setprecision(n)（設定浮點數顯示精度爲 n）。如果你想知道 iostream 提供的所有操控器，請參考 [LIPPMAN98] 20.9 節。

如果要開啓一個可供寫入的檔案，我們定義一個 istream（供輸入的 file stream）物件，並將檔名傳入。如果檔案未能開啓成功，ifstream 物件會被核定爲 false。如果成功，該檔的寫入位置會被設定在起始處。

```
// 以寫入模式（input mode）開啓 infile
ifstream infile( "seq_data.txt" )

int num_tries = 0;
int num_cor = 0;

if ( ! infile )
{
    // 由於某種原因，檔案無法開啓…
    // 我們將假設這是一位新的使用者…
}
```

```
else
{
    // ok：讀取檔案中的每一行
    //     檢查這個使用者是否曾經玩過這個程式
    // 每一行的格式是：
    //     name num_tires num_correct
    // nt：猜過的總次數（num_tries ）
    // nc：猜對的總次數（num_correct）

    string name;
    int nt;
    int nc;

    while ( infile >> name )
    {
        infile >> nt >> nc;
        if ( name == usr_name )
        {
            // 找到他了
            cout << "Welcome back, " << usr_name
                 << "\nYour current score is " << nc
                 << " out of " << nt << "\nGood Luck!\n";

            num_tries = nt;
            num_cor = nc;
        }
    }
}
```

while 迴圈的每次迭代都會讀取檔案的下一行內容。這樣的動作會持續到檔案尾端才結束。當我們寫下：

```
infile >> name
```

這個述句的傳回值即是從 infile 讀到的 class object。一旦讀到檔案尾端，讀入的 class object 會被核定爲 false。因此我們可以在 while 迴圈的條件算式中，以此做爲結束條件：

```
while ( infile >> name )
```

檔案的每一行都包含一個字串，其後接著兩個整數。形式如下：

```
anna 24 19
danny 16 12...
```

下面這一行：

```
infile >> nt >> nc;
```

會先將使用者猜過的總次數讀到 nt 之中，再將使用者猜對的總次數讀到 nc 之中。

如果想要同時讀寫同一個檔案，我們得定義一個 fstream 物件。爲了以附加模式（append mode）開啓，我們得傳入第二參數值 ios_base:in|ios_base::app[2]：

```
fstream iofile( "seq_data.txt",
                ios_base::in|ios_base::app );

if ( ! iofile )
    // 由於某種原因，檔案無法開啓... 真糟！
else
{
    // 開始讀取之前，將檔案重新定位至起始處
    iofile.seekg( 0 );

    // 其它部份都和先前討論的相同
}
```

當我們以附加模式來開啓檔案，檔案位置會位於尾端。如果我們沒有先重新定位，就試著讀取檔案內容，那麼立刻就會遇上「讀到檔尾」的狀況。seekg() 可將檔案重新定位至檔案的起始處。由於此檔是以附加模式開啓，因此任何寫入動作都會將資料附加於檔案最末端。

iostream library 提供的功能相當豐富，許多細節無法在此一一述及。如果想對 iostream library 有更多的了解，請參考 [LIPPMAN98] 20 章或 [STROUSTRUP97] 21 章。

練習 1.5

撰寫一個程式，能夠詢問使用者的姓名，並讀取使用者所輸入的內容。請確保使用者輸入的名稱長度大於兩個字元。如果使用者的確輸入了有效名稱，就回應一些訊息。請以兩種方式實作：第一種使用 C-style 字元字串，第二種使用 string 物件。

練習 1.6

撰寫一個程式，從標準輸入裝置讀取一串整數，並將讀入的整數依序置入 array 及 vector，然後走訪這兩種容器，求取數值總合。將總和及平均值輸出至標準輸出裝置。

練習 1.7

使用你最稱手的編輯工具，輸入兩行（或更多）文字並存檔。然後撰寫一個程式，開啓該文字檔，將其中每個字都讀取到一個 vector<string> 物件中。走訪該 vector，將內容顯示到 cout。然後利用泛型演算法 sort()，對所有文字排序：

---

[2] 既然稍早我已決定不對 ios_base::app 加以解釋，當然我也不會在這裡解釋這個更爲複雜的東西！請參閱 [LIPPMAN98] 20.6 節，其中有完整的說明及詳盡的範例。

```
#include <algorithm>
sort( container.begin(), container.end() );
```

再將排序後的結果輸出到另一個檔案。

練習 1.8

1.4 節的 `switch` 述句讓我們得以根據使用者答錯的次數提供不同的安慰語句。請以 array 儲存四種不同的字串訊息，並以使用者答錯次數做爲 array 的索引值，以此方式來顯示安慰語句。