<p.8> 第2-3行

原為：我會在某處使用Shell搜尋法，但實際上quicksort搜尋法可能會比較好。

改為：我會在某處使用Shell排序法，但實際上quicksort排序法可能會比較好。

<p.8> 第4行

原為：在真正的程式碼中，函式庫函式的呼叫行為通常更適合用來解說語言功能。

改為：在真正的程式碼中，通常更適當的作法是呼叫函式庫函式，而不是採用此處用來解說語言功能的程式碼。

<p.9> 倒數第12行

原為：演算法的運算式

改為：演算法的表達性

<p.10> 第18行

原為：不使用比C++更低階的語言（特殊情況下，組合語言是例外）。假如你能夠用低階語言寫出高效率的程式碼，那麼C++語言將會是系統程式設計語言的首選。

改為：不讓比C++更低階的語言有任何容身之處（在某些特殊情況下，組合語言是例外）。假如你能夠用更低階的語言寫出更高效率的程式碼，那麼該語言多半會成為系統程式設計語言的首選。

<p.17> 第5行

原為：在實際的程式設計中，「知道最難懂的語言功能」或是「使用大量的語言功能」應該會有些幫助。

改為：「知道最難懂的語言功能」或是「使用大量的語言功能」，在實際的程式設計中，用處並不大。

<p.23> 第2行

原為：就像在大型專案上經常見到的情況一樣，我們對完成的日期非常具有自信。

改為：就像在大型專案上經常見到的情況一樣，我們對完成的日期太過樂觀。

<p.29> 第8行

原為：Boost是一個志工組織，成立的目的是為了提供有用的函式庫元件（以STL[Boost]為基礎）。

改為：Boost是一個志工組織，成立的目的是為了提供有用的函式庫元件（以STL為基礎）[Boost]。

<p.31> 第5行

原為：C++被各種函式庫（ISO C++標準函式庫以外）和工具組所支援…

改為：C++被各種函式庫（除了ISO C++標準函式庫之外）和工具組所支援…

<p.49> 第10行

原為：也就是說，這個函式可用來建立類別的物件

改為：也就是說，這個函式是用來建立類別的物件

<p.49> 第12行

原為：不像一般的函式，系統保證建構式一定可用來初始化類別物件。

改為：不像一般的函式，系統保證一定會用建構式來初始化類別物件。

<p.71> 第4行

原為：由於並沒有指標指向user()以外的Shape物件…

改為：由於user()以外之處並沒有指標指向這些Shape物件…

<p.111> 第3行

原為：諷刺的是，我們只能藉由不使用指標和new運算式來解決問題…

改為：諷刺的是，我們能簡單地藉由不使用指標和new運算式來解決問題…

<p.111> 第5行

原為：很遺憾地，過度使用new運算式（以及過度使用指標和reference）似乎會逐漸升高問題的嚴重性。

改為：很遺憾地，過度使用new運算式（以及過度使用指標和reference）的問題似乎日益嚴重。

<p.113> 倒數第6行

原為：輸出結果將會是不可預測的，而且每次的執行結果都不會一樣…

改為：輸出結果將會是不可預測的，而且每次的執行結果可能都不會一樣…

<p.116> 倒數第4行

原為：因此，請不要選擇共享資料的方式來傳遞資訊，原因只有一個，那就是「效率」（這不需思索，最好實際量測看看）。

改為：因此，請不要直覺以為（甚至不實際量測就以為）效率會比較好，就選擇以共享資料的方式來傳遞資訊。

<p.117> 程式碼

原為：mqueue .pop()

改為：mqueue.pop()

<p.120> 倒數第6行

原為：基本上，sync()將函式呼叫的「呼叫部分」和「取得結果值部分」分離開來…

改為：基本上，async()將函式呼叫的「呼叫部分」和「取得結果值部分」分離開來…

<p.124> 程式碼

原為：cout << \*p;

改為： cout << \*p;

<p.129> 第7行

原為：只有第二個assert函式是有作用的，因為numeric\_limits<int>::max()是一個constexpr函式（§2.2.3、§10.4）。

改為：由於numeric\_limits<int>::max()是一個constexpr函式（§2.2.3、§10.4），所以第二個斷言（才）會有作用。

<p.129> 倒數第9行

原為：不要以共享資料的方式來傳遞資訊，原因只有一個，那就是「效率」（這不需思索，最好實際量測看看）；

改為：請不要直覺以為（甚至不實際量測就以為）效率會比較好，便選擇以共享資料的方式來傳遞資訊。

<p.143> 第4行

原為：很像plain int

改為：如同plain int

<p.143> 第11行

原為：如果字面常數太長而無法完全顯示，…

改為：如果字面常數太長而無法完全表述，…

<p.143> 倒數第11行

原為：利用這些表示法來顯示「真實的數字」會讓你有意想不到的收穫。

改為：利用這些表示法來表達「真實的數字」有可能會導致意想不到的情況。

<p.143> 倒數第9行

原為：由於有比較多的位元來表示整數，因此十進位正整數最大可達到65535。

改為：如果用更多的位元來表示整數，那麼結果會是十進位正整數65535。

<p.143> 倒數第2行

原為：我建議盡量少用意義不明的常數，並限制它們只能出現在定義清楚的地方，例如：const（§7.5）、constexpr（§10.4）、以及列舉元初值列（enumerator initializer）（§8.4）。

改為：我建議盡量少用意義不明的常數，並限制它們只能出現於註解清楚的const（§7.5）、constexpr（§10.4）、以及列舉元（enumerator）（§8.4）初值列中。

<p.144> 倒數第8行

原為：浮點數佔固定的記憶體大小，而且其顯示的數值只是一種近似值而已。

改為：浮點數佔固定的記憶體大小，而且其表述的數值只是一種近似值而已。

<p.158> 第2行

原為：利用{}來進行初始化行為並不會比較好，而且當我們用auto來取得初值列所決定的型別時，會有一個陷阱。

改為：當我們用auto來取得初值列所決定的型別時，利用{}來進行初始化行為並不會比較好，而且會有一個陷阱。

<p.166> 第11行

原為：請不要以型別的表象來為型別命名，而要以型別的目的來替它命名…

改為：以型別的內部表述來為型別命名，而不以型別的目的來替它命名，未必是一個好主意…

<p.171> 第8行

原為：使用void∗指標的函式通常都應用於非常低階的系統中…

改為：使用void∗指標的函式通常都應用於系統中極為低階之處…

<p.171> 第9行

原為：如果在高階系統中看到void∗型別，便應該對它們感到存疑…

改為：如果在系統中較高階之處看到void∗型別，便應該對它們感到存疑…

<p.179> 第4行

原為：如果使用最新的編譯器，那麼這兩個例子應該（通常）會產生出相同的程式碼。

改為：如果使用最新的編譯器，那麼這兩個例子應該（通常）會產生出相同的機器碼。

<p.199> 程式碼

原為：address prev = current;

改為：Address prev = current;

<p.208> 第9行、倒數第4行

原為：二進位變數

改為：二元變數

<p.211> 第4行

原為：union的使用效能通常已達最佳狀態

改為：使用union通常是為了最佳化

<p.217> 第4行

原為：在特定的列舉型別中，大部分的整數值都沒有相應的內部表述。

改為：在一個列舉型別中，大部分的整數值都沒有相應的內部表述。

<p.227> 第3行

原為：switch述句通常也會產生出更好的程式碼，因為重複地檢驗個別的數值是毫無道理的。然而，如果真有這方面的需求，其實可以考慮使用跳轉表（jump table）。

改為：switch述句通常也會產生出更好的程式碼，因為重複地檢驗個別的數值是毫無道理的──編譯器會視情況為switch述句使用跳轉表（jump table）。

<p.242> 第7行

原為：要不是忽略get()回傳值的事情困擾著我…

改為：要是忽略get()回傳值會讓我感到困擾的話…

<p.243> 倒數第7行

原為：使用者在呼叫current()之前，必須先呼叫get()，如此一來才能獲得一個定義良好（well-defined）的Token。

改為：使用者在呼叫current()之前，應該先呼叫get()，不過，即使沒這麼做，也能獲得一個定義良好（well-defined）的Token。

<p.256> 程式碼註解

原為：不論f()或g()哪一個先呼叫，都是undefined行為

改為：undefined行為，不論f()或g()哪一個先呼叫

<p.256> 第7行

原為：然而，若不對運算式評估順序施加限制，便會導致undefined結果。

改為：然而，若不對運算式評估順序施加限制，有可能會導致undefined結果。

<p.256> 第15行

原為：「||」與&&

改為：「||」與「&&」

<p.256> 倒數第3行

原為：如果使用者不覺有異的話

改為：只有在使用者不會覺得有異的情況下

<p.257> 倒數第9行

原為：這是理所當然的

改為：這很自然

<p.278> 第4行

原為：當我們將許多物件配置成單一陣列或多個大型物件時，上述額外配置行為並沒有很大的意義；不過當我們在自由空間上配置出許多小型物件（例如：int或Point）時，上述額外配置行為便極具意義。

改為：當我們將許多物件配置成單一陣列或多個大型物件時，並不會造成很大的額外負擔；不過當我們在自由空間上配置出許多小型物件（例如：int或Point）時，便會造成巨大的額外負擔。

<p.981> 第10行

原為：如果「disguised pointer」的位元型態（bitpattern）是以錯誤的型別（例如：long或char[4]）儲存於記憶體中，那麼它仍然可被優質的垃圾回收器找到，而且也仍然會對它進行適切的對齊操作。

改為：如果「disguised pointer」的位元型態（bitpattern）是以錯誤的型別（例如：long或char[4]）儲存於記憶體中，而且仍然有適切地對齊，那麼它依然可被優質的垃圾回收器找到。

<p.1250> 第4行

原為：有時候，之所以會發生不相容的情況，主要是因為使用到一些經常出現於C++程式碼但卻不相容於C的功能，例如：將「T\*」指派給「void\*」…

改為：有時候，之所以會發生不相容的情況，主要是因為某些C++習用已久的特性並不相容於C，例如：將「T\*」指派給「void\*」的能力…

<p.1250> 第6行

原為：有時候，即使某項功能成為了ISO C++標準成員，它甚至也不會相容於C…

改為：有時候，ISO C++標準的功能即使被C所採用，也會造成不相容的情況…