附錄 A

可攜性與標準化

Portability and Standarization

C++ 語言正處於一個過渡狀態。

這個語言早期唯一的實作品是 AT&T Cfront 編譯器，最可靠的參考手冊則是 Bjarne Stroustrup 所著的 ***The C++ Programming Language*** [Str86] 第一版。此後，當其它廠商的編譯器紛紛出現，Margaret Ellis 與 Stroustrup 合著的 ***Anotated C++ Reference Manual*** [ES90]（被 C++ 迷暱稱爲 ***ARM***）便成爲實際上的標準。

C++ 目前正處於國際標準化的階段（國際標準最後草稿於 1997 年 11 月 14 日通過），該標準規格目前在國際標準組織（International Standards Organization, ISO）與美國國家標準機構（American National Standard Institute, ANSI）聯合委員會的控制之下。在標準化過程中，這個語言有了些許改變。C++ standard [ISO98] 與 ***ARM*** 之間的差異，就像 ANSI C standard 與 ***The C Programming Language*** [KR78] 之間的差異一樣，極富戲劇性。很多新特色被加入這個語言之中，很多方面則做了淨化。

這些枝枝節節對於一般程式員似乎無關緊要，但它卻造成不同的 C++ 實作品（編譯器）之間不再那麼理所當然地具備移植性。目前還沒有一個編譯器實作出完整的 C++ standard 規格，也很少有編譯器將自己侷限於 ***ARM*** 所描述的特色。幾乎所有 C++ 編譯器所實作出來的，都介於 ***ARM*** C++ 與 standard C++ 之間。

STL 的情況也很類似。原始的 STL 定義於 Alexander Stepanov 和 Meng Lee 所撰寫的技術報告內 [SL95]。然而 STL 現在已成爲 standard C++ library 的一部份了。C++ 標準委員會對 STL 的定義做了些微改變 — 就像它也曾經對 C++ 核心語言定義做了些微改變一樣。

事實上即使 STL 原始的 HP 版本是由設計該程式庫正式規格的人所撰寫，那份實作品也沒有完全遵循正規定義。事實上那個時候也沒辦法遵循正規定義，因爲 STL 的定義之中所需要的語言特性在當時還沒有能夠實現出來。HP 版本是「正規設計」與「1994 年的編譯器技術」之間的一個妥協方案。

到了 1998 年，仍然沒有一個 STL 實作品完全符合 Stepanov-Lee 的原始定義，或是符合 C++ standard 的定義。技術上的一個原因是，那份標準規格仍然是份草稿，而且一直有所變動。試圖跟上變動中的規格腳步，就像愛麗絲（Alice）與紅心皇后（Red Queen）之間的競賽：「在這場賽跑中你所能夠做的，就是保持相同的距離。」大部份程式庫實作者在 C++ standard 正式定案之前，會保持謹慎觀望的態度。

第二個問題是，即使到了今天，也沒有任何一個 C++ 編譯器支援 STL 所用到的每一個語言特性。（是的，編譯器廠商也同樣謹慎得很！）在尚未獲得某些語言特性之前，STL 的某些特性不是暫時空著，就是透過某種非標準的迂迴方式實作出來。不幸的是，這些限制及迂迴法往往罕有文件介紹，或甚至根本沒有文件。

STL 中的絕大部分組件已經完全穩定下來了，這一部份不受編譯器的束縛，標準委員會也未曾更動過它。你可以預期這些部分在每個實作版本中都是一樣的。至於其它部分就比較不穩定了。有時候不同實作版本間的差異性會帶來重要的影響。

幸運的是，這些混亂狀況只出現於邊緣地帶。STL 之中沒有任何變化或任何實作上的困難點會影響到該程式庫的核心設計。泛型程式設計的基本原則以及 STL 程式碼主體在任何編譯器以及任何 STL 實作版本中都是有效的，實作版本之間的差異只集中於某幾個問題。

這一份附錄談的是你必須知道的可攜性議題。包括 (1) 不同實作品中某些組件的不同定義方式、(2) 編譯器帶來的限制（以至無法以正規定義方式來實作 STL 程式庫）、(3) 定義有所更動的某些組件 (4) 某些會影響 STL 運用方式的 C++ 語言變動。

A.1 語言上的變動

A.1.1 Template 編譯模式

自從 C++ 增加 template 特性之後，templates 分離式編譯（separate compilation）就成了語言的一部份。其目的在於讓 function templates 和 class templates 可以如同一般 functions 和 classes 一樣地被使用。也就是說，你可以在某個原始碼檔案中宣告一個函式，然後在不同的原始碼檔案中使用這個函式。這兩個檔案可以分開編譯，只要最後兩個目的檔（object files）被聯結起來即可。

ARM 要求 templates 分離式編譯，C++ standard 亦然 — 儘管某些細節有了些微變動。AT&T Cfront 3.0 編譯器包含受限的 templates 分離式編譯，其它很多編譯器亦具有部份受限的分離式編譯能力。

不幸的是，沒有任何兩種分離式編譯完全相同。某些涉及了程式碼中特別的 `#pragma` 宣告，某些要求程式碼檔名需符合特定的命名規則，甚至於很多編譯器根本就不具有任何 templates 分離式編譯能力。

你的編譯器可能允許某種分離式編譯，但如果你使用那種方式，當移植到其它編譯器時，可能會遇到一些麻煩。或許多年之後，每個編譯器才會支援 C++ standard 所定義的分離式編譯。

目前唯一具可攜性的解決方案就是，遵循一個規則：「function template 的定義式，不只有其函式原型(prototype)，還必須在該函式被使用之前可見(visible)」。這意味 function templates 和 class templates 必須定義在表頭檔（`*.h`）中，而你必須含入適當的表頭檔，才能使用某個 template。這就是爲什麼現有的所有 STL 實作品大部分（或甚至全部）都是一些表頭檔的原因。

A.1.2 有預設值的 Template 參數（Default Template Parameters）

就像 C++ 函式的參數可具有預設值，class templates 也一樣。假設你宣告一個 class template 如下：

```
template <class A, class B = A>
class X {
  ...
};
```

那麼當你具現化(instantiated)`X` 時，可以不指定第二個 template 參數。這麼一來便會採用預設值 `A`。是的，型別 `X<int>` 與型別 `X<int, int>` 完全相同。

所有的 STL container classes 都使用帶有預設值的 template 參數。舉個例子，`vector` 就具有兩個 template 參數，第一個表現出 `vector` 的元素型別，第二個表現 allocator，用來參數化「`vector` 的記憶體配置策略」。第二個 template 參數便帶有預設值，因爲大部份的 `vector` 使用時機都沒有理由使用非標準的記憶體配置策略。

目前，並非所有的 C++ 編譯器都完全支援帶預設值的 template 參數。某些編譯器對此有某種限制（例如要求預設值不能與其它 template 參數相依），某些編譯器則根本不允許。由於 STL 極度仰賴帶預設值的 template 參數，所以它不能夠在那樣的編譯器上精確地依據標準設計實作出來。

當一個編譯器不具備這個語言特性，基本上有兩種方法可以實作出帶預設值的 template 參數：要不就要求使用者必須提供該參數值，要不就全然移除該參數。以 `vector` 爲例，第一種方法要求你總得寫成 `vector<int, allocator<int> >`，即使你用的是預設的 allocator。第二種方法允許你簡單地寫成 `vector<int>`，但你卻也因此無法使用別種 allocator。這些妥協辦法都不完全令人滿意。

HP STL 完成之時，尚未有任何編譯器支援帶預設值的 template 參數。當時它使用上述第二種方法。兩種方法在後繼的 STL 實作品中相繼都被用過。

A.1.3 Member Templates

templates 最初加入 C++ 語言時，只有兩種東西可以是 template：全域的 classes 和全域的 functions。也就是說，template 宣告式不能在 class 範圍內發生。這意味 member functions 不能是 function template。

這個限制是沒有必要的。如今 C++ standard 已允許 non-virtual member functions[1]可以成爲 function templates。Member function templates 的呼叫方式和一般的 function templates 一樣。編譯器可以由函式呼叫所給定的引數推導出 template 參數，並自動產生該 member function 的適當實體（instance）。

Class's constructor 是一個 member function，所以如同其它 member functions 一樣，它可以成爲一個 template。這個結果導致 member templates 最重要的用途之一。例如 class `pair` 便擁有一個經過一般化的 copy constructor：

```
template <class T1, T2> template <class U1, class U2>
pair<T1, T2>::pair(const pair<U1, U2>&);
```

此式允許以任何 `pair<U1, U2>` 建構出任何一個 `pair<T1, T2>`，只要 `U1` 可轉換爲 `T1` 且 `U2` 可轉換爲 `T2`。這種語法看起來很奇怪，但它是正確的。關鍵字 `template` 必須出現兩次，因爲此處有兩個 template 參數列，一個針對 `pair` template 本身，另一個針對 member template constructor。

STL container classes 大量使用 member templates。讓我再說一次，其主要用途之一是在 constructors。你可以根據一個 iterators range 建構出一個 container，member templates 允許該 range 所含之型別爲任何 Input Iterator 之 model。例如你可以這樣產生一個 `vector V`，讓它和 `list L` 包含相同的元素：

```
vector<T> V(L.begin(), L.end());
```

只要 `L` 的 value type 可轉換爲 `T`，這種寫法就有效。同樣地，containers 的 `insert` member functions 也可以藉由 member templates 技術，接受一個 iterators range。

在編譯器支援 member templates 之前，這種功能無法達成。如果你的編譯器尚未支援 member templates，便無法撰寫一個完全一般化的 constructor，允許由某個 Input Iterators range 建構出一個 `vector`。另一個可行辦法是針對某些挑選出來的 iterator types，將 `vector` constructor 多載化。這裡所謂挑選出來的 iterator types 是程式庫實作者預期最有用的幾個 iterator types。通常你應該至少能夠根據一個 pointers range 以及一個 container-A iterators range 建構出一個 container-A。

class `pair` 的 member template constructor 不完全是方便性的問題。舉個例子，標準的 container `map` 是一個 Associative Container，其元素型別爲 `pair<const Key, Data>`。和其它 Associative Container 一樣，`map` 有一個 member function 可讓你將單個元素安插到 container 內。當你寫：

```
M.insert(p)
```

[1] 根據 ***The Design and Evolution of C++*** [Str94] 所描述，由於某種技術原因，virtual member functions 目前仍然不可爲 templates。

此處 p 的型別相同於 M 的元素型別，也就是說 p 的型別是 pair<const Key, Data>。問題在於如何建構出 object p。

如果使用 member template constructor，你可以不必寫出 pair constructor 就將一個元素安插到 map：

```
M.insert(make_pair(k, d))
```

這會產生 pair<Key, Data> object，然後將該 object 轉爲 pair<const Key, Data>，然後將它安插到 map 中。如果沒有 member template constructor，便沒有這種轉換行爲。一旦缺少 member templates 的支援，你就必須使用較不方便的形式：

```
M.insert(pair<const Key, Data>(k, d))
```

A.1.4 「局部特殊化」（Partial Specialization）

當 template 成爲 C++ 語言性質的那一天起，我們就已經可以將一個 class template 特殊化了。假設你有一個 class X<T>，由於某種原因，X 的一般定義不能套用在某個特定型別上，那麼你可以針對該型別給予 X 不同的定義。例如你或許能夠將 X<int> 設計得比一般版本更有效率。X<int> 可以和一般版的 X<T> 完全無關，它可以擁有一組完全不同的 member functions。

Class templates 的特殊化如今進展到了所謂的「局部特殊化（partial specialization）」。就拿以下的例子來說，你可以針對「整個型別範疇（entire category of types）」，而非只針對單一型別，給予 template 一份不同的定義。

```
template <class T> class X
{
  // 版本 1：最一般化。
};
template <class T> class X<T*>
{
  // 版本 2：針對一般指標。
};
template <> class X<void*>
{
  // 版本 3：針對某種特定指標。
};
```

完全特殊化與局部特殊化並不使用相同的語法。這是 C++ 標準化過程中的一項改變。如果你所使用的編譯器遵循 C++ standard，你必須寫

```
template <> class X<void*>
```

如果你使用的編譯器比較舊，你得這麼寫：

```
class X<void*>
```

這個式子並未使用關鍵字 `template`。除了使用前處理器（pre-processor）所提供的巨集，否則再沒有其他方法可以寫出符合新舊規格的完全特殊化語法。

局部特殊化對於「最佳化」常常很有用途。例如有時候你可以針對指標寫出某個特別版本的 class，此一特別版本擁有最佳效率。此外，局部特殊化使得某種程式設計技術變得可能，而 STL 正是架構於這些技術之上。

第一，爲了儲存效率，STL 內含一個特殊化版本的 `vector` container class：`vector<bool>`。乍見之下這不像是一個局部特殊化的例子，但它的確是：就像所有的 STL 序列一樣，`vector` 具有兩個 template 參數：一個是 value type，一個是 allocator。`vector<bool>` class 是一個局部特殊化版本，因爲它給予第一個 template 參數一個特別型別，而讓 allocator 完全一般化。在不支援局部特殊化的編譯器上，STL 實作品通常會宣告一個 class `bit_vector` 取而代之。`bit_vector` 只是一個過渡時期的迂迴作法，並非 C++ standard 的一部份，而且一旦局部特殊化被廣泛支援之後，它就可能消失。

更重要的是，STL 基本要素之一的 `iterator_traits`，非常關鍵性地仰賴局部特殊化技術。

3.1 節曾描述的 `iterator_traits` class，是「存取 iterator 相關型別資訊」的一個重要機制。對於任何一個 iterator type `I`，

```
iterator_traits<I>::value_type
```

便是 `I` 的 value type，而

```
iterator_traits<I>::difference_type
```

便是 `I` 的 difference type。這須得仰賴局部特殊化，因爲面對一般指標，我們無法像面對「以 classes 形態出現的」iterators 那樣地定義 `iterator_traits`。

你可以在自己的程式中使用 `iterator_traits`。它也被使用於 STL 的其它部分。

- Iterator adaptor `reverse_iterator` 接受單一一個 template 參數 `Iter`，這是一個 Bidirectional Iterator。這個 adaptor 使用 `iterator_traits` 機制俾使 `reverse_iterator<Iter>` 具有與 `Iter` 相同的 difference type 和 value type。
- 演算法 `distance`（p.181）、`count`（p.214）和 `count_if`（p.216）都在 input iterators range 上頭動作。這些演算法都傳回型別 `typename iterator_traits<InputIter>::difference_type` 的值。

原始的 HP STL 並不仰賴局部特殊化，也不具備 `iterator_traits` class。它提供較粗陋的機制來存取 iterator 型別資訊，此機制涉及查詢函式 `distance_type`、`value_type`、`iterator_category`。它們並未被納入 C++ standard。大多數 STL 實作品仍然支援這些函式，但它們最終會被移除。

如果你的編譯器不支援局部特殊化，你便不能使用 `iterator_traits` 的完全一般化性質。這時候你

可能得以 HP STL 的舊式機制來取代，也可能得使用舊式的 `count`、`count_if`、`reverse_iterator`。

函式的局部特殊化（partial specialization of functions）

C++ standard 有另一項特色，和局部特殊化非常類似：那就是 function template 的偏序（partial ordering）關係。

Template 函式，如同其他函式一樣，是可被重載的（overloaded）。這提高了面對一個函式呼叫時有多個版本能夠精確吻合（exact match）的機率。假設你宣告了兩個 function templates：

```
template <class T> void f(T)
template <class U> void f(U*)
```

當你寫下 `f((int*) 0)`，你會喚起哪個版本？該呼叫動作可吻合第一版本（當 `T` 爲 `int*`）或第二版本（當 `U` 爲 `int`）。

在 template 初被引進之際，這類函式呼叫是不合法的，因爲它精確吻合了一個以上的函式。但是新規則已經放寬了這項限制。每當某個函式呼叫動作可匹配多個重載的 function templates 時，編譯器會選擇其中最特殊化（most specialized）的一個。此例之中 `f` 的第二版本比第一版本更特殊化。是的，第一版本能匹配任何型別，但第二版本只能匹配指標。

STL 只在一個地方使用函式的這種偏序（partial ordering）關係。它有一個 `swap` 函式（p.237），可交換任何兩個變數內容。STL 針對所有的 container classes 定義了特殊化的 `swap` 版本。當你寫下 `swap(v1, v2)`，如果 `v1` 和 `v2` 都是 `vectors`，你會喚起以下函式：

```
template <class T, class Allocator>
void swap(vector<T, Allocator>&, vector<T, Allocator>&)
```

而非更一般化的版本：

```
template <class T> void swap(T&, T&)
```

這很重要，因爲 `swap` 的一般化版本必須以指派行爲（assignment）來運作，而將一個 `vector` 指派給另一個 `vector` 是很慢的動作（因爲它必須複製該 `vector` 的所有引數）。特殊化版本直接在 `vector` 的內部資料上起作用，非常快速。

在不支援 function templates 之偏序（partial ordering）性質的編譯器上，STL 並未定義出特殊化的 `swap` 版本。但這不會影響你所能撰寫的程式碼 — 你仍然可以使用一般版的 `swap` 來交換兩個 `vectors`，這意味某些程式的執行速度會比它們所應該表現的慢許多。

A.1.5 新加入的關鍵字

C++ 語言新增了許多關鍵字，這些關鍵字是 ***ARM*** 發行之時所沒有的。對於 STL 以及運用 templates 的程式而言，有兩個關鍵字特別重要，那就是 `explicit` 和 `typename`。

關鍵字 explicit

`explicit` 用來抑制某種「自動型別轉換」功能。通常，如果 class `X` 具有一個 constructor，可接受型別爲 `T` 的單一引數，C++ 編譯器便會自動以該 constructor 將型別爲 `T` 的值轉換成型別爲 `X` 的值。因此假設你有一個函式 `f`，需要一個型別爲 `X` 的引數，而 `t` 的型別爲 `T`，你可以直接寫 `f(t)`，不必手動完成轉換。

有時候這是你所想要的。有時候這種自動轉換會導致非常不可預期的結果。舉個例子，如果 `f` 預期其引數爲 `vector<string>`，難道你希望 `f(3)` 有效嗎？但 `vector<string>(3)` 卻是一個完全合理的 constructor call。

將單引數 constructor 宣告成 `explicit`，就可以抑制這種自動轉換行爲。噢是的，你還是可以使用該 constructor，但必須明確寫出才行。C++ standard 將大多數 containers 的單引數 constructors 都宣告爲 `explicit`。

由於 `explicit` 的目的在抑制某些自動發生的事，意味可能有某些程式在舊規則中原本合法，在使用標準編譯器與標準程式庫後便不合法了。如果你真的想要傳入一個新建構的 `vector`，並使它擁有三個空的 `strings`，你得明確寫成：`f(vector<string>(3))`。這個事實亦套用於其他不同的宣告上。不要寫成：

```
vector<string> v = 3;
```

你必須寫爲：

```
vector<string> v(3);
```

或

```
vector<string> v = vector<string>(3);
```

關鍵字 typename

`explicit` 關鍵字只影響少數含糊不清的建構行爲 — 它禁止了某些「如今已不再是個好概念」的行爲。但 `typename` 就不一樣，它出現在 C++ 泛型程式設計的很多基礎地點。如果你曾經使用 templates 寫過任何像樣的 C++ 程式，你必須瞭解如何使用 `typename`。

基本議題屬於技術層面，但不是很複雜。當編譯器看到某些算式涉及 template 參數，它應該將該算式視爲「取用某型別」，或視爲「取用其他某物（例如 member function 或 member variable）」呢？

舉例來說，在這個函式中：

```
template <class X> void f(X) {
  X::T(x);
}
```

編譯器究竟應該將 `T` 解釋爲一個型別，致使 `X::T` 成爲一個 constructor call（也就是建構出一個匿

名的暫時物件，然後再將它丟掉）呢？或者應該將 T 解釋爲一個 static member function 或 static member variable 呢？同樣的情況，以下函式：

```
template <class X> void g(X) {
  X::T t;
}
```

編譯器究竟應該將 T 解釋爲一個型別，致使這個函式使用 T 的 default constructor 來產生 object t 呢？或者應該將 T 解釋爲一個 static member function 或 static member variable，致使 X::T t 形成一個語法錯誤呢？

後一種情況特別麻煩。一旦 g 被具現化（instantiated），編譯器當然可以辨別 X::T 究竟是一個 class、一個 member function、一個 member variable 或其他某物。但如果無法在尚未具現化的時候便檢查出該 template 語法的正確性，實在令人失望。果真如此，template 的分離式編譯也就毫無指望了。

C++ 語言的設計使得 template 的語法檢驗可行。C++ 有一個非常簡單的規則來決定是否要將 X::T 視爲型別：如果某個名稱可能代表型別或代表其他某物，則「除非你以別種方式明白告知編譯器，否則編譯器總是假設該名稱不代表某個型別」。

typename 關鍵字就是用來告知編譯器應該將某特定名稱視爲一個型別。函式 g 的正確寫法是：

```
template <class X> void g(X) {
  typename X::T t;
}
```

在 X::T 之前冠以關鍵字 typename，就是告知編譯器應該將 X::T 解釋爲型別。注意，每當要代表一個型別時，你都必須使用 typename。也就是說你必須寫出：

```
template <class X> void g1(X) {
  typename X::T t;
  typename X::T* pt;
}
```

第二個 typename 也是必要的。

基本規則如下：如果沒有 typename 的修飾編譯器就無法將某物視爲一個型別，那麼你就必須使用 typename。也就是說，如果某個名稱符合以下兩條件，你就必須使用 typename：

1. 它是一個資格受限的名稱（qualified name），亦即其他型別中的巢狀型別。
2. 它取決於 template 參數，也就是取決於 template 被具現化後的 template 引數。

例如，函式 g 內的 X::T 是一個資格受限的名稱（以 X:: 加以限制），並取決於 template 參數 X。同樣道理，算式 Y<X>::T 也有一個資格受限的名稱，取決於 template 參數 X。

這對 STL 應用程式象徵了什麼意義？唔，第一件事是不管你自己正在撰寫新的 STL 組件，或者使

用別人所定義的組件，你得使用很多 typename — 甚至當你正在做某件似乎完全無害的事時，例如撰寫一個作用於 vector<T> 身上的 template function 而且必須取用該 vector 的 iterator：

```
template <class T>
void f(vector<T>& v) {
  typename vector<T>::iterator i = v.begin();
  ...
}
```

這雖然看起來很奇怪，但此地的確需要 typename。對身爲人類的你看來，很顯然 vector<T>::iterator 一定是取用 vector 的巢狀的 iterator type，但對編譯器而言，那只是一種取決於 template 參數的資格受限名稱。除非你明白告知編譯器 vector<T>::iterator 是個型別名稱，否則編譯器無法知道。

這個道理也適用於 iterator_traits。在 3.1.5 節中我們看到，定義 iterator_traits class 時，我們必須使用 typename；同樣道理當我們使用 iterator_traits 時，幾乎亦總得使用 typename。以 count 演算法（p.214）爲例，它具有這樣的形式：

```
template <class InputIterator, class EqualityComparable>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last,
      const EqualityComparable& value)
```

不用說，typename 造成了可攜性的問題。符合 C++ standard 之編譯器需要它，但其他更早（尚未加入關鍵字 typename）之前的編譯器卻不認識它。時值 1998 之際，這兩種編譯器仍然都很常見。

唯一可以通吃上述兩種編譯器的作法，就是利用前處理器技巧。你可以定義一個巨集 TYPENAME，讓它在某些編譯器中展開爲 typename，在另一些編譯器下則什麼也沒展開。

關鍵字 typename 是一個技術細節，是爲了某種特殊技術目的才導入的。但不幸的是你無法忽視這個細節。好消息是，如果你錯用了 typename，編譯器一定會給你某些錯誤訊息。是的，萬一不小心遺漏 typename，並不會讓一個原本做某事情的有效程式搖身變爲一個作不同事情的有效程式。

A.2 程式的變動

A.2.1 Allocators

就像 9.4 節所說，STL containers 都將其記憶體配置方法參數化了。這種作法有時候很有用，但從歷史的角度來看，它卻成爲 STL 最不穩定的一部分。目前大家常用的有四種不同的 allocator 設計：HP STL allocators、SGI STL allocators，以及從 C++ standard 草案衍化而來的兩個不同的 allocators 設計。

由於兩個進一步的理由，使可攜性更成爲問題所在。第一，關於「container 如何使用 allocators」，C++

standard 給予程式庫實作者良好的自由度。第二，許多 STL 實作品並未實際提供這四種設計中的任何一種。HP allocator 在缺乏支援所謂 "template template parameters"[2]的環境下無法實作出來，standard 草案版的 allocators 在缺乏支援 member templates 的環境下也不可行。如果你的編譯器不支援 member templates，你便不能使用 C++ standard 所描述的 allocators 版本，頂多只能使用某家廠商所提供的折衷版本。

如果你關心可攜性議題，最安全的作法是根本就不要使用 allocators。啊，應該說是使用預設的 allocator。Allocator 是一個帶有預設值的 template 參數，所以你毋須指明。如果你確實必須撰寫自己的 allocator，你應該詳讀文件，並找出你的程式庫所使用的 allocator 設計法。更明確地說，如果你的編譯器不支援 member templates，你應該確定對於必要的迂迴作法與限制都有足夠的瞭解。每當使用不同的編譯器或程式庫，你也應該預期必須做某些改變。

定義於 C++ standard 中的 allocators，最顯而易見的創新之處是 allocator ***instances***。Member functions `allocate` 和 `deallocate` 與特定的 allocator objects 相應。如果你透過一個 allocator object `a` 來配置記憶體，你必須透過一個相等於 `a` 的 allocator 來釋放該記憶體，因爲不同的 allocators 可能參考到不同的記憶體區域。

只讓 allocator 成爲 container 型別的一部份是不夠的。每個 container object 必須包含一個特別的 allocator instance。即使兩個 containers 具有相同的型別，它們不見得要以相同的方式配置記憶體。

container 被建構出來後才去改變其 allocator，這種作法是不合理的，必須有某種方式來控制該 container 使用哪一個 allocator instance。如果 container 以 allocators 做爲參數之一，那麼該 container 的所有 constructors 就都必須擁有一個 allocator 參數。

C++ standard 定義的所有 STL containers（`vector`、`list`、`deque`、`set`、`map`、`multiset`、`multimap`）都以這種方式來定義其 constructors。這些 containers 的 constructors 都擁有一個型別爲 `Allocator` 的參數，大部分情況下它都是 constructor 的最後一個參數，並具有預設值 `Allocator()`。你可以不提到它，以避免使用 allocator instances。因此如果我們寫：

```
vector<int> V(100, 0);
```

就相當於寫：

```
vector<int, allocator<int> > V(100, 0, allocator<int>());
```

A.2.2 Container Adaptors

STL 定義三種 container adaptors：`stack`、`queue` 和 `priority_queue`。它們並非 Containers，它們不提供 Container 的整個介面，只提供該介面的有限子集。Container adaptor 只是對其底部

[2] 這項語言特性並未在本書任何地方提及，因爲 STL 完全沒有用到它。請參考 ***The C++ Programming Language*** [Str97] C.13.3 節。

（underlying）containers 的某種外包裝（wrapper）而已，該底部 container 是一個 template 參數。此一參數化的精確形式目前已有所改變。

在原始的 HP STL 中，stack adaptor 需要單一一個 template 參數，做爲該 stack 底部資料的型別。如果你想要產生一個由 ints 組成的 stack 並以 deque 作爲其底部資料組織，你得這麼宣告：

```
stack<deque<int> >
```

這雖然不至於含糊不清，但卻令人困惑。看起來像是說「stack 內的值就是 deques，而非 ints」。此外，它迫使你必須明白宣告某些東西（也就是其底部資料組織），而那應該只歸屬於實作細節。

C++ standard 已經改變這一點。stack 如今需要兩個 template 參數。第一個是儲存於該 stack 內的 object 型別，第二個才是該 stack 的底部資料組織。當然後者是一個帶預設值的 template 參數。對 stack 而言，該預設值爲 deque。因此，由 ints 組成的 stack 可以宣告爲 stack<int>。

這項改變造成了兩個可攜性問題。第一，雖然大多數 STL 實作品都已改變其 container adaptors 以遵循 C++ standard 的規定，但畢竟不是全部都如此。因此就著你手上的程式庫實作品，你可能得使用較舊的 container adaptors 版本。第二，如果你使用的是不支援 default template parameters 的編譯器，那你便不能準確使用 C++ standard 所規範的 container adaptors。你應該詳讀文件，找出你的程式庫實作者所使用的迂迴作法。一種常見的迂迴作法是要求你提供兩個 template 參數，例如你可能得把由 ints 組成的 stack 宣告爲：

```
stack<int, deque<int> >
```

A.2.3 次要的程式可變動

標準化過程中，STL 的介面於某些次要部份亦有些微的改變及擴充。

新的演算法

C++ standard 內含三個並未定義於原始 HP STL 中的演算法：(1) find_first_of，它很類似 C 函式庫函式 strpbrk；(2) find_end，它很類似 STL 演算法 search；(3) search_n。

Standard 還包含了三個泛型演算法：uninitialized_copy、uninitialized_fill 和 uninitialized_fill_n，以及兩個暫時記憶體配置函式：get_temporary_buffer 和 return_temporary_buffer，這些函式都曾出現於 HP 版本中，但未見諸文件。這些特別函式主要用於自行撰寫自己的 container 或 adaptive 演算法。

Interator 介面改變

比起原始的 HP STL，C++ standard 對於 iterators 多定義了一個次要功能。所有 iterators（Output Iterator

除外，因爲以下敘述對於它並不合理）如今都定義了 member function `operator->`。如同你對指標的預期，`i->m` 和 `(*i).m` 應該代表相同的事情（動作）。

這對於 `map` 和 `multimap` 特別方便，因爲這些 containers 的元素都是 `pairs`。如果 `i` 是一個 iterator，其 value type 是 `pair<const T, X>`，你可以寫 `i->first` 或 `i->second`，就好像 `i` 就是一般指標一樣。

Sequence 介面改變

所有的 Sequences 都包含了並未列於原始 HP STL 中的三個新的 member functions：(1) `resize`，它會刪除或附加元素於尾端，使 Sequence 變成所指定的大小；(2) `assign`，它是一種一般化的指派操作行爲；(3) `clear`，它是 `erase(begin(), end())` 的簡略寫法。此外，`erase` member function 的傳回型別也有了改變。在 HP STL 中其傳回型別爲 `void`，但 C++ standard 將它改變爲 iterator。是的，`erase` 的傳回值如今是一個 iterator，指向被移除元素的下一個緊鄰位置。

multiplies Function Object

HP STL 內含一個 function obejct `times`，可用來計算兩個引數的乘積。C++ standard 將這函式改名爲 `multiplies`。

不幸的是沒有任何實作品有什麼好方法可以同時提供新舊名稱。這個名稱之所以被改變，是因爲 `times` 與 UNIX 的某個表頭檔名稱互相衝突，如果同時保留新舊兩個名稱，改變就沒有意義了。如果你正在將程式碼由舊的 STL 版本移植到新的版本，你得留意這個 function object 的名稱。

A.3 命名及包裝（Naming and Packaging）

> *What's in a name? That which we call a rose*
> *By any other word would smell as sweet.*
> 名字有什麼意義？我們稱呼玫瑰爲其他名字，並不會改變其芬芳本質。

命名議題或許很瑣細，但它們是 C++ standard STL 與原始的 HP STL 最明顯的差別。例如在原始的 STL 中，下面是一個完整而正確的程式（雖然不是一個非常有趣的程式）：

```
#include <vector.h>

int main() {
  vector<int> v;
}
```

以 C++ standard 的觀點，這不是一個合法程式。兩個原因：

1. C++ 語言如今具備一個 namespace（命名空間）系統。根據 standard 規範，`vector` 並未定

義於 global namespace 中，而是定義於 namespace std 中。或者，以另一個角度來說，C++ standard 並不擁有一個名爲 vector 的 class，而是擁有一個名爲 std::vector 的 class。

2. 根據 standard 規範，class std::vector 不是宣告於表頭檔 <vector.h>，而是宣告於表頭檔 <vector>。

有很多方法可以將這個程式修改爲符合 C++ standard 的所有規範。其中一種方法是：

```
#include <vector>

int main() {
  std::vector<int> v;
}
```

另一種作法是：

```
#include <vector>

int main() {
  using std::vector;
  vector<int> v;
}
```

第一個版本明白地以全名參考 std::vector。第二個版本中的這一行：

```
using std::vector;   // 此爲一個 "using declaration"
```

意思是編譯器應該將 vector 視爲 std::vector 的縮寫。第三種方式不採用 using declaration，而是採用所謂的 "using directive"：

```
using namespace std;
```

這樣會將 std namespace 內的所有名稱匯入。但是我並不鼓勵此法。

除了少許例外，C++ 標準程式庫的所有組件都定義於 namespace std 之中。每當你要使用 containers、algorithms，或標準程式庫中的任何一個名稱，都必須明白冠以修飾詞 std::，否則就得使用 using declaration 將它匯入。

以運算子 == 和 < 爲基礎，定義出的一組運算子 !=、>、<= 和 >=（都是 templates），是以上規則的例外情況。它們本身並非宣告於 namespace std 中，而是宣告於名爲 std::rel_ops 的一個巢狀命名空間中。

表頭檔的命名方式也比過去稍微複雜了些。C++ standard 將來自 HP STL 的所有表頭檔重新命名。新的表頭檔名稱如 <vector>，不再具有 .h 副檔名。新舊名稱之間並沒有一對一的映射關係，新組織出來的檔案簡直和重新命名的一樣多。甚至 <vector> 就是這種情況。在 HP STL 中，template vector<> 宣告於表頭檔 <vector.h>，class bit_vector（等同於特殊化的 vector<bool>）宣告於 <bvector.h>。C++ standard 則將以上兩者都宣告於表頭檔 <vector>。

表 A.1：C++ Standard 和 HP STL 之間的表頭檔對映關係

C++ Standard 表頭檔	相應的 HP STL 表頭檔
<algorithm>	<algo.h>（大部分）
<deque>	<deque.h> （所有）
<functional>	<function.h>（大部分）
<iterator>	<iterator.h>（所有）
<list>	<list.h>（所有）
<memory>	<defalloc.h>（所有）
	<tempbuf.h>（所有）
	<iterator.h>（部分）
	<algo.h>（部分）
<numeric>	<algo.h>（部分）
<queue>	<stack.h>（部分）
<utility>	<pair.h>（所有）
	<function.h>（部分）
<stack>	<stack.h>（部分）
<vector>	<vector.h>（所有）
	<bvector.h>（所有）
<map>	<map.h>（所有）
	<multimap.h>（所有）
<set>	<set.h>（所有）
	<multiset.h>（所有）

表 A.1 是舊式表頭檔與 C++ standard 表頭檔的概略指引。更詳細的資訊應該查閱本書第三篇。篇中每個 functions 及每個 class 的文件都明確指出它被宣告於哪個 standard 表頭檔及哪個舊式表頭檔。

「C++ standard 表頭檔與舊式名稱全然不同」這一事實，證明對於可攜性是幫助而非阻礙。它給予實作者一個回溯相容的管道。

C++ standard 並未提及諸如 <algo.h> 和 <list.h> 等舊式名稱。程式庫實作者沒有必要提供這類表頭檔，但也沒有被禁止這麼做。是的，實作者可以提供舊式的表頭檔名（而且很多人也的確這麼做），並將其內容封裝在 global namespace 而非 namespace std 之中。通常在這種情況下，表頭檔 <vector> 會包含實際的實作碼：

```
namespace std {
  template <class T, class Allocator>
  class vector {
    ...
  };
}
```

而表頭檔 <vector.h> 只包含以下兩行：

```
#include <vector>
using std::vector;
```

這種作法並不屬於 C++ standard 的規範，但它是被允許的，而且相當常見。從歷史的角度來看，將 STL 表頭檔重新命名，其動機便是希望這種方式行得通。它容許你逐漸過渡到新式表頭檔名，並明白地（explicit）使用 namespaces。