Boost 函式庫、Technical Report1 以及 C++0x

23

The danger from computers is not that they will eventually get as smart as men, but we will meanwhile agree to meet them halfway.
—Bernard Avishai

學習目標

在本章中，你將學到：

- C++未來的方向
- 什麼是 Boost 函式庫。
- Boost 開放原始碼專案的沿革簡介，新的資料庫如何加入 Boost，以及如何安裝 Boost。
- 使用 Boost.Regex 來搜尋字串，驗證資料，利用正規表示式替代部分字串。
- 利用 Boost.Smart_ptr 管理動態記憶體配置，以避免記憶體遺漏。
- Technical Report 1 (TR1，對 C++標準函式庫的變更建議) 中包含哪些 Boost 函式庫 (以及其他函式庫) 。
- 新的 C++標準 － C++0x 中包含了哪些核心語言以及標準函式庫的改變。

本章綱要

23.1 簡介

在本章中，我們要討論的是 C++的未來發展。我們將介紹 Boost C++函式庫、Technical Report 1 (TR1) 以及 C++0x.。**Boost C++函式庫** (**Boost C++ Libraries**) 是由 C++社群成員所建立的免費開放原始碼函式庫。Boost 提供了實用的、設計良好的函式庫，可以與目前的 C++標準函式庫一同使用。Boost 函式庫可以在許多平台和編譯器上使用。我們將會簡單介紹一些常用的 Boost 函式庫，並提供正規表示式和聰明指標函式庫的程式碼範例。**Technical Report 1** 是對 C++標準函式庫的變更建議，其中有許多是以 Boost 函式庫爲基礎的。這些函式庫替 C++增加許多有用的功能。**C++0x** 是下一代 C++標準的工作名稱。它包括一些核心語言的變動，並增加許多 TR1 和其他新增函式庫中所描述的函式庫。

23.2　Deitel 線上 C++資源中心及相關資源。

我們會定期在線上資源中心 www.deitel.com/resourcecenters.html 發表程式設計、軟體、Web 2.0 和網際網路商業相關的重要主題。C++0x 尚未定案，Boost 經常加入新的函式庫，兩者都仍持續有變動。我們建立了幾個線上資源中心，提供連結讓讀者可以取得這些重要的資訊。你可以在 C++ Boost Libraries 資源中心 www.deitel.com/CPlusPlusBoostLibraries/找到目前可用的函式庫和新釋出版本的資訊。在 C++資源中心 (www.deitel.com/cplusplus/) 的 C++0x 章節，可以找到 TR1 和 C++0x 的資訊 (點選分類清單中的 C++0x)。我們使用 Visual C++ 2008 Express 版本來編譯本章的範例程式碼。讀者可以拜訪 C++資源中心 (www.deitel.com/VisualCPlusPlus/) 以取得更多關於 Visual C++的資料。

23.3　Boost 函式庫

將免費的開放原始碼 C++函式庫放到線上資料庫這個想法，最先是在 1998 [1] 年由 Beman Dawes 的論文所提出的。他和 Robert Klarer 是在出席 C++標準委員會議時，得到這個靈感。論文中建議設立一個網站，讓 C++程式設計師可以搜尋和分享函式庫，並幫助 C++未來的開發。這個想法最後成爲 Boost 函式庫，其網址爲 www.boost.org。Boost 目前已經超過 100 個函式庫，而且還時常增加。今日的 Boost 社群有數千位程式設計師參與。

將新函式庫加入 Boost

Boost 接受任何人貢獻實用的、設計良好的、具可攜性的函式庫。想要加入 Boost 的函式庫應該要遵守 C++標準，並使用 C++標準函式庫 (或是其他適當的 Boost 函式庫)。爲了確認函式庫符合 Boost 的高品質和高可攜性標準，函式庫需要經過正式的接受程序。

C++社群會發表文章在郵件論壇上並視回應來決定是否將某個函式庫納入考慮。假如決定將某個函式庫納入考慮，則會將此函式庫初步提交，發表在 **Boost Sandbox** (svn.boost.org/svn/boost/sandbox/) 上，這是一個用來存放開發中函式庫的資料庫。Sandbox 讓其他使用者試驗這個函式庫並提出反饋。

當函式庫準備好接受正式的審查時，會將程式碼的提交發表在 Sandbox Vault，並由

---

[1] "Proposal for a C++ Library Repository Web Site," Beman G. Dawes, May 6, 1998, www.boost.org/ more/proposal.pdf.

經認可的志願者中選出審查委員。審查委員會先確認程式碼已經準備好要開始正式審查，設定審查時程，閱讀所有使用者的審查，最後決定是否要接受這個函式庫。審查委員會提出某些需要執行的修正或改進事項，完成這些事項之後，函式庫才能正式加入Boost。一旦函式庫被接受，原作者就要負責它的維護。

Boost 軟體授權條款

Boost 軟體授權條款 (Boost Software License，`www.boost.org/more/license_info.html`)允許複製、修改、使用、發布 Boost 原始碼和執行檔做爲商業和非商業用途。唯一的要求是必須將版權與授權訊息與任何公開的原始碼一起發佈 (雖然原始碼並非一定要釋出)。這些條款讓 Boost 函式庫可以運用在任何應用程式中。每一個 Boost 函式庫都必須遵守這些條款。

安裝 Boost 函式庫

Boost 函式庫可以在許多平台和編譯器上以最小安裝的方式使用。BoostPro Computing 公司提供免費的安裝程式，可以在 Visual Studio 中使用 Boost，網址爲：`www.boostpro.com/download`。大多數的 Linux 版本也提供 Boost 套件，不過有時會標頭檔和函式庫會有獨立的套件。你可以在 `www.boost.org/more/getting_started/index.html` 找到適用於各種平台和編譯器的安裝指南。

23.4 Boost 函式庫概述

Boost 函式庫十分龐大，很難在本書中含括完整的介紹。我們在本節中簡介最常用的一些函式庫。接下來的小節中，我們會示範這些函式庫中的兩個。

Array [2]

`Boost.Array` 是固定大小陣列的包裝器，藉著支援大多數的 STL 容器介面，加強內建陣列的功能。`Boost.Array` 讓你可以在 STL 應用程式中使用固定大小的陣列，而非 `vector` (動態配置陣列)，當我們不需要變動陣列大小時，後者的效率是較差的。

---

[2] 參考文件 Boost.Array, Nicolai Josuttis, `www.boost.org/doc/html/array.html`。

Bind [3]

Boost.Bind 擴充標準函式 std::bind1st 和 std::bind2nd 的功能。bind1st 和 bind2nd 函式是用來轉換二元函式 (binary function，有兩個引數的函式)，以供接收一元函式 (unary function，有一個引數的函式) 的標準演算法使用。Boost.Bind 強化這個功能，允許你轉換多達九個引數的函式。Boost.Bind 也讓你利用佔位符，輕鬆地將傳給函式的引數重新排序。

Function [4]

Boost.Function 讓你在函式包裝器中儲存函式指標、成員函式指標和函式物件。boost::function 可以儲存函式，將它的引數和回傳型別轉換爲符合函式包裝器的簽章。例如，假如函式包裝器所存放的函式接收一個 string 引數，並回傳一個 string，它也可以存放一個接收 char* 並回傳 char* 的函式，因爲 char* 可以利用轉型建構子轉換爲 string。

Random [5]

Boost.Random 讓你可以建立各種各樣的亂數產生器和亂數分布。C++標準函式庫的 std::rand 和 std::srand 可以用來產生虛擬亂數。**虛擬亂數產生器 (pseudo-random number generator)** 利用初始狀態來產生虛擬的亂數，也就是說，使用相同的初始狀態會產生同樣的一串數目。rand 函式總是使用同樣的初始狀態，因此每次都會產生同樣的一串數目。函式 srand 讓我們可以設定初使狀態，以改變這串數目。虛擬亂數通常用來測試，它的可預測性讓我們能夠確認結果。Boost.Random 提供虛擬亂數產生器，也提供**非確定性亂數 (nondeterministic random numbers)** 產生器，後者產生的亂數是不可預測的。這些亂數產生器可以用在不需要可預測性的模擬程式和安全性情境中。

---

[3] 參考文件 Boost.Bind, Peter Dimov,www.boost.org/libs/bind/bind.html。

[4] 參考文件 Boost.Function, Douglas Gregor, www.boost.org/doc/html/function.html。

[5] Jens Maurer, "A Proposal to Add an Extensible Random Number Facility to the Standard Library," Document Number N1452, April 10, 2003, www.open-std.org/jtc1/sc22/wg21/docs/papers/ 2003/n1452.html.

Boost.Random 讓你可以指定亂數的分佈。常用的分佈像是**均勻分佈 (uniform distribution)** 會讓指定範圍內所有的數值都有同樣的機率。這類似投擲骰子或硬幣，每種結果的機率差不多是相同的。你可以在編譯時其設定範圍。Boost.Random 允許你使用任何亂數產生器結合某種分佈，甚至建立你自己的分佈。

Regex [6]

Boost.Regex 支援 C++中的**正規表示式 (regular expression)** 處理 (。正規表示式是用來比對文字中的特定字元樣式。許多現代程式語言都內建支援正規表示式，但是 C++沒有。你可以使用 Boost.Regex 搜尋 string 中的特殊表示式，將符合正規表示式的部份取代爲其他字元，或是使用正規表示式定義分界符號，將 string 切割成字符。這些都是文字處理、剖析和輸入驗證中常用的技巧。我們會在 23.5 節中更詳細討論 Boost.Regex 函式庫。

Smart_ptr [7]

Boost.Smart_ptr 定義聰明指標，可以幫助你管理動態配置的資源 (例如記憶體、檔案和資料庫連結) 。程式設計師經常搞不清楚何時應該釋放記憶體，或是根本就忘記做這件事，特別是當記憶體由多個指標參照的時候。聰明指標會自動幫你做這些事。TR1 包含兩個位於 Boost.Smart_ptr 函式庫的聰明指標。**shared_ptr** 處理動態配置物件的生命週期管理。當沒有任何 shared_ptr 指向某個記憶體時，此記憶體會被釋放。**weak_ptr** 讓你可以觀察 shared_ptr 中的值，而不用負任何管理責任。我們會在第 23.6 節更詳細討論 Boost.Smart_ptr 函式庫。

Tuple [8]

tuple 是一組物件。**Boost.Tuple** 讓你可以使用泛型的方法產生一組物件，並對這組物件執行泛型函式。這個函式庫讓你可以建立至多 10 個物件的 tuple，這個限制還可以再放寬。基本上，Boost.Tuple 是 STL 的 std::pair 類別樣板的延伸。Tuple 通常用

---

[6] 參考文件 Boost.Regex, John Maddock, www.boost.org/libs/regex/doc/index.html。

[7] 參考文件 Boost.Smart_ptr, Greg Colvin and Beman Dawes, www.boost.org/libs/smart_ptr/smart_ptr.htm。

[8] 參考文件 Boost.Tuple, Jaakko Järvi, www.boost.org/libs/tuple/doc/tuple_users_guide.html。

在從函式中回傳一組數值，也可以用來在 STL 容器中儲存一組元素，每一組元素即爲容器中的一個元素。另一個實用的功能，是使用 tuple 來設定變數的值。

Type_traits [9]

Boost.Type_traits 函式庫讓我們能夠擷取不同型別之間的差異點，使一般化的程式設計實作得到最佳化。type_traits 類別讓你判斷某個型別的特色 (它是一個指標或參照型別？這個型別是否擁有 const 修飾詞)，然後執行型別轉換，讓物件使用通用的程式碼。這些資訊讓通用程式碼得以最佳化。例如，有時候我們可以使用 C 函式 memcpy 複製集合中的物件，而不需要使用 STL 的 copy 演算法走訪集合中所有的元件。使用 Boost.Type_traits 函式庫，我們可以確認型別的特質，然後依此執行演算法，使通用演算法得以最佳化。

23.5　正規表示式與 Boost.Regex 函式庫

正規表示式 (Regular expressions) 是格式化的 string，用來搜尋文字中的特定樣式。它們可以用來驗證資料，確認資料的格式無誤。例如，ZIP 碼必須含有五個數字，或者一個人名的姓氏必須以大寫字母開始。

Boost.Regex 函式庫提供了好幾種類別和演算法，可以用來辨識和操作正規表示式。類別樣板 **basic_regex** (在 boost 命名空間) 代表一個正規表示式。Boost.Regex 演算法 **regex_match** 會在符合正規表示式時，回傳 true 值。使用時 regex_match，整個字串都必須符合正規表示式。Boost.Regex 也提供 **regex_search** 演算法，假如 string 的任何一部分符合正規表示式，就會回傳 true 值。使用 Boost.Regex 時，需含括標頭檔 <boost/regex.hpp>。請注意，Boost 標頭檔使用 .hpp 做爲副檔名。

正規表示式字元類別

圖 23.1 的表格列出一些可以在正規表示式中使用的**字元類別 (character classes)**。字元類別並非 C++類別，它只是一個跳脫序列，用來表示可能出現在字串中的一組字元。

---

[9] 參考文件 Boost.Type_traits, Steve Cleary, Beman Dawes, Howard Hinnant and John Maddock, www.boost.org/doc/html/boost_typetraits.html。

字元類別	符合	字元類別	符合
\d	任何十進位數字	\D	任何非十進位數字
\w	任何文字字元	\W	任何非文字字元
\s	任何空白字元	\S	任何非空白字元

圖 23.1　字元類別

文字字元 (word character) 是任何字母數字或是底線字元。**空白字元 (whitespace character)** 爲空白鍵、tab 鍵、換行、新行或表格饋入字元 (form feed)。**數字字元 (digit)** 是所有可以數學操作的字元。正規化表示法不只使用圖 23.1 這些內建的字元類別。在圖 23.2 中，你會看到正規表示式使用其他表示法，在 `string` 中尋找複雜的樣式。

23.5.1　正規表示式範例

圖 23.2 的程式試著比對生日和一個正規表示式。第 11 行會找出不在四月出生而且以"J"爲名字開頭的人的生日。

```
1  // Fig. 23.2: fig23_02.cpp
2  // Demonstrating regular expressions.
3  #include <iostream>
4  #include <string>
5  #include <boost/regex.hpp>
6  using namespace std;
7
8  int main()
9  {
10    // create a regular expression
11    boost::regex expression( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );
12
13    // create a string to be tested
14    string string1 = "Jane's Birthday is 05-12-75\n"
15       "Dave's Birthday is 11-04-68\n"
16       "John's Birthday is 04-28-73\n"
17       "Joe's Birthday is 12-17-77";
18
19    // create a boost::smatch object to hold the search results
20    boost::smatch match;
21
22    // match regular expression to string and print out all matches
23    while ( boost::regex_search( string1, match, expression,
```

圖 23.2　以正規化表示法檢查生日

```
24          boost::match_not_dot_newline ) )
25       {
26          cout << match << endl; // print the matching string
27
28          // remove the matched substring from the string
29          string1 = match.suffix();
30       } // end while
31    } // end function main
```

```
Jane's Birthday is 05-12-75
Joe's Birthday is 12-17-77
```

圖 23.2　以正規化表示法檢查生日 (續)

第 11 行會建立一個 **regex** (basic_regex 類別樣板的 typedef 名稱) 物件，將正規表示式傳遞給 regex 建構子作爲參數。請注意，我們在每個反斜線字元前面加上了另一個反斜線。還記得吧，C++ 將字串常數裡的反斜線當做跳脫序列的開頭。要在字串中插入反斜線，你必須使用另一個反斜線來跳脫。例如，在 C++ 字面字串中，字元類別\d 必須表示成\\d。

正規表示式中的第一個字元 "J" 是一個字面字元。任何符合此正規表示式的 string 都必須以 "J" 開始。在正規表示式中，句點字元 "." 代表任何單一字元。假如句點字元後面接著星號，就像是 ".*"，即代表任意數量的任意字元。一般來說，當運算子 "*" 套用在樣式時，表示這個樣式出現零次以上。相反地，將運算子 "+" 套用在樣式上，表示這個樣式出現一次以上。例如，"A*" 和 "A+" 都會符合 "A"，但是只有 "A*" 符合空字串。

如圖 23.1 所示，"\d" 符合任何十進位數字。除了事先定義的字元類別以外，若要指定某一組字元，可以將這些字元列在方括號[]中。例如，"[aeiou]" 樣式代表任意母音。指定一範圍的字元可以用連字號 (-) 表示。例如，"[0-35-9]" 代表樣式所指定的範圍內的數字，也就是 0 到 3 之間的數字或是 5 到 9 之間的數字，也就是說，它符 4 以外的數字。你也可以將樣式指定爲方括號之外的字元。只要將 ^ 放在方括號中的第一個位置即可。注意，"[^4]" 並不等於 "[0-35-9]"，"[^4]" 符合 4 以外的任何非數字和數字。

雖然在中括號內的 "－" 字元是用來表示範圍，但是在群組表示式以外的 "－" 是被當做字面字元來看待的。因此，第 11 行的正規表示式會搜尋以字母 J 開頭，後面接著任意數量的字元，再接著兩位數字 (第二個數字不能是 4)，再接著一個橫線以及另外兩位數字，以及另外一條橫線和另外兩位數字的字串。

第 20 行會建立一個 smatch (唸做「ess-match」，這是 match_results 的 typedef 名稱) 物件。當**match_results**物件被當成引數傳遞給某個 Boost.Regex 演算法時，會將符合正規表示式的字串儲存在此物件中。**smatch** 儲存 string::const_iterator 型別的物件，你可以用它來存取符合的字串。有 typedef 支援其他的字串表示，像是 char*(cmatch)。

while 敘述 (第 23–30 行) 會搜尋 string1 中是否有符合正規表示式的字串，直到找不到爲止。我們使用 regex_search 呼叫作爲 while 敘述的條件式 (第 23–24 行)。假如 string(string1) 中有符合正規表示式 (expression) 的字串，則 regex_search 會回傳 true。我們也將 smatch 物件傳遞給 regex_search 以存取符合的字串。最後一個引數 **match_not_dot_newline** 會防止 "." 字元比對換行字元。while 敘述的主體會列印出符合正規表示式的子字串 (第 26 行) ，並將它從搜尋的字串中移除 (第 29 行)。呼叫 match_results 的成員函式 **suffix** 會傳回上一個符合字串尾端到搜尋字串尾端的字串。圖 23.2 的輸出顯示 String1 中找到兩個符合的子字串。這兩個子字串都符合正規表示式指定的樣式。

量化詞

圖 23.2 第 11 行中，星號 (*) 的正式名稱爲**量化詞 (quantifier)**。圖 23.3 列出你可以放在正規表示式樣式後方的各種量化詞，以及這些量化詞的功用。

量化詞	符合
*	前方樣式出現零次以上。
+	前方樣式出現一次以上。
?	前方樣式出現零或一次。
{n}	前方樣式恰好出現 n 次。
{n,}	前方樣式至少出現 n 次。
{n,m}	前方樣式出現 n 到 m 次之間 (包括兩端點值)。

圖 23.3　正規化表示式中的量化詞

我們已經介紹過如何使用星號 (*) 和加號 (+)。量化詞標記 (?) 表示該樣式出現零或一次。大括號中間包含一個數字 ({n}) 表示該樣式恰好出現 n 次。我們會在下一個範例中示範這個量化詞。在大括號中含有一個數字後面接著一個逗點表示該樣式至少出

現 n 次以上。大括號中包含兩個數字 ({n, m})，表示該樣式出現次數在 n 到 m 次 (包含兩端點值) 之間。所有的量化詞都是**貪婪的 (greedy)**，這表示只要仍符合，他們會對應到出現最多指定樣式的情況。然而，如果在量化詞後面加上問號 (?)，量化詞將變爲**懶惰的 (lazy)**。表示盡可能的以規定的表示式符合越短的字串。

23.5.2　利用正規表示式驗證使用者輸入

圖 23.4 的程式是一個更複雜的範例，它使用正規表示式來驗證使用者輸入的名稱、位址和電話號碼資訊。

```
1  // Fig. 23.4: fig23_04.cpp
2  // Validating user input with regular expressions.
3  #include <iostream>
4  #include <string>
5  #include <boost/regex.hpp>
6  using namespace std;
7
8  bool validate( const string&, const string& ); // validate prototype
9  string inputData( const string&, const string& ); // inputData prototype
10
11 int main()
12 {
13    // enter the last name
14    string lastName = inputData( "last name", "[A-Z][a-zA-Z]*" );
15
16    // enter the first name
17    string firstName = inputData( "first name", "[A-Z][a-zA-Z]*" );
18
19    // enter the address
20    string address = inputData( "address",
21       "[0-9]+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
22
23    // enter the city
24    string city =
25       inputData( "city", "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
26
27    // enter the state
28    string state = inputData( "state",
29       "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
30
31    // enter the zip code
32    string zipCode = inputData( "zip code", "\\d{5}" );
33
```

圖 23.4　　利用正規表示式驗證使用者輸入

```
34       // enter the phone number
35       string phoneNumber = inputData( "phone number",
36          "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" );
37
38       // display the validated data
39       cout << "\nValidated Data\n\n"
40          << "Last name: " << lastName << endl
41          << "First name: " << firstName << endl
42          << "Address: " << address << endl
43          << "City: " << city << endl
44          << "State: " << state << endl
45          << "Zip code: " << zipCode << endl
46          << "Phone number: " << phoneNumber << endl;
47    } // end of function main
48
49    // validate the data format using a regular expression
50    bool validate( const string &data, const string &expression )
51    {
52       // create a regex to validate the data
53       boost::regex validationExpression = boost::regex( expression );
54       return boost::regex_match( data, validationExpression );
55    } // end of function validate
56
57    // collect input from the user
58    string inputData( const string &fieldName, const string &expression )
59    {
60       string data; // store the data collected
61
62       // request the data from the user
63       cout << "Enter " << fieldName << ": ";
64       getline( cin, data );
65
66       // validate the data
67       while ( !( validate( data, expression ) ) )
68       {
69          cout << "Invalid " << fieldName << ".\n";
70          cout << "Enter " << fieldName << ": ";
71          getline( cin, data );
72       } // end while
73
74       return data;
75    } // end of function inputData
```

```
Enter last name: Doe
Enter first name: John
Enter address: 123 Some Street
Enter city: Some City
Enter state: Some State
Enter zip code: 12345
Enter phone number: 123-456-7890

Validated Data
```

圖 23.4　利用正規表示式驗證使用者輸入 (續 1)

```
Last name: Doe
First name: John
Address: 123 Some Street
City: Some City
State: Some State
Zip code: 12345
Phone number: 123-456-7890
```

圖 23.4　利用正規表示式驗證使用者輸入 (續 2)

程式首先呼叫 inputData 函式，請使用者輸入姓氏 (第 14 行)。inputData 函式 (第 58–75 行) 會接收兩個引數，輸入資料的名稱和要比對的正規表示式。接著，函式會提示使用者輸入指定的資料 (第 63 行)。然後 inputData 會呼叫 validate 函式 (第 50–55 行)，檢查輸入的資料格式是否正確。validate 函式會接收兩個引數：要驗證的 string 和要比對的正規表示式。函式首先會使用正規表示式來建立 regex 物件 (第 53 行)。接著它會呼叫 regex_match，判斷 string 是否符合正規表示式。假如輸入的字串是不合法的，inputData 會提示使用者重新輸入資訊。一旦使用者輸入了合法的資料，就會回傳此 string 資料。程式會重複這個過程，直到所有的資料欄位都驗證完成 (第 14–36 行)。接著會在螢幕上顯示所有的資訊 (第 39–46 行)。

在前一個範例中，我們搜尋 string 中是否有符合正規表示式的子字串。在本範例中，我們想要驗證整個 string 是否符合特定的正規表示式。例如，我們希望 "Smith" 符合人名的姓，但不允許 "9@Smith#"。在這裡，我們使用 regex_match 來替代 regex_search，當整個 string 符合正規表示式時，regex_match 會回傳 true 值。你也可以選擇使用以 "^" 字元開頭，以 "$" 字元結束的正規表示式。字元 "^" 和 "$" 分別代表 string 的開頭和結尾。這些字元合在一起，會強迫正規表示式只能在整個 string 符合正規表示式時，才能回傳符合的字串。

第 14 行的正規表示式會使用方括號加上範圍表示法，表示第一個字必須爲大寫字母，其後的字母則大小寫均可，a-z 表示任何小寫字母，A-Z 表示任何大寫字母。*量化詞表示第二個範圍的字元可以在字串中出現零次以上。因此，這個表示式會比對任何以一個大寫字母開頭，其後跟著零個以上字母的字串。

\s 表示單一空白字元 (第 21、25、29 行)。表示式\d{5}用來比對 zipCode 字串 (第 32 行)，表示任五個數字。字元 "|" (第 21、25、29 行) 表示左右兩邊的表示法**都許可**，例如，"Hi (John|Jane)" 表示 "Hi John" 或 "Hi Jane"。在第 21 行，我們使用 "|" 字元表示地址中包含擁有一個字母以上的單字，**或是**擁有一個字母以上的單字

加上一個空白，再加上擁有一個字母以上的單字。其中的括號用來將正規表示式的部分表示式併列在一起。我們可以將量化詞放在括號中，用來建立更複雜的正規表示式。

`lastName` 和 `firstName` 變數 (第 14 和 17 行) 都接受以大寫字母開頭的任意長度 `string`。`address string` 的正規表示式 (第 21 行) 會比對一串數字 (至少一個)，加上一個空白以及至少一個字元的單字，或是至少一個字元的單字加上一個空白再加上至少一個字元的單字。因此，`"10 Broadway"` 和 `"10Main Street"` 都是合法的地址。目前第 21 行的形式並不符合不以數字開頭的地址，或是超過兩個單字的地址。城市 (第 25 行) 和州 (第 29 行) 的正規表示式也規定爲至少一個字元或以一個空白分隔的兩個單字 (各至少一個字元)。所以 `Waltham` 和 `West Newton` 也符合規定。這些正規表示式不會接受兩個字以上的名稱。`zipCode string` 的正規表示式 (第 32 行) 會驗證郵遞區號爲五位數字。`phoneNumber string` (第 36 行) 的正規表示式規定電話的格式必須是 `xxx-yyy-yyyy`，x 代表區碼，y 代表電話號碼。第一個 x 和第一個 y 不能是 0，以 [1－9] 來表示。

23.5.3　置換以及切割字串

有時候，利用正規表示式將字串的某一部分替換掉或是切割字串是很實用的。`Boost.Regex` 提供 `regex_replace` 演算法以及 `regex_token_iterator` 類別來達成以上目標，我們在圖 23.5 中示範這些功能。

```
1  // Fig. 23.5: fig23_05.cpp
2  // Using regex_replace algorithm.
3  #include <iostream>
4  #include <string>
5  #include <boost/regex.hpp>
6  using namespace std;
7
8  int main()
9  {
10    // create the test strings
11    string testString1 = "This sentence ends in 5 stars *****";
12    string testString2 = "1, 2, 3, 4, 5, 6, 7, 8";
13    string output;
14
15    cout << "Original string: " << testString1 << endl;
16
17    // replace every * with a ^
18    testString1 =
19       boost::regex_replace( testString1, boost::regex( "\\*" ), "^" );
20    cout << "^ substituted for *: " << testString1 << endl;
```

圖 23.5　使用 `regex_replace` 演算法

```
21
22       // replace "stars" with "carets"
23       testString1 = boost::regex_replace(
24          testString1, boost::regex( "stars" ), "carets" );
25       cout << "\"carets\" substituted for \"stars\": "
26          << testString1 << endl;
27
28       // replace every word with "word"
29       testString1 = boost::regex_replace(
30          testString1, boost::regex( "\\w+" ), "word" );
31       cout << "Every word replaced by \"word\": " << testString1 << endl;
32
33       // replace the first three digits with "digit"
34       cout << "\nOriginal string: " << testString2 << endl;
35       string testString2Copy = testString2;
36
37       for ( int i = 0; i < 3; i++ ) // loop three times
38       {
39          testString2Copy = boost::regex_replace( testString2Copy,
40             boost::regex( "\\d" ), "digit", boost::format_first_only );
41       } // end for
42
43       cout << "Replace first 3 digits by \"digit\": "
44          << testString2Copy << endl;
45
46       // split the string at the commas
47       cout << "string split at commas [";
48
49       boost::sregex_token_iterator tokenIterator( testString2.begin(),
50          testString2.end(), boost::regex( ",\\s" ), -1 ); // token iterator
51       boost::sregex_token_iterator end;  // empty iterator
52
53       while ( tokenIterator != end ) // tokenIterator isn't empty
54       {
55          output += "\"" + *tokenIterator + "\", "; // add the token to output
56          tokenIterator++; // advance the iterator
57       } // end while
58
59       // delete the ", " at the end of output string
60       cout << output.substr( 0, output.length() - 2 ) << "]" << endl;
61    } // end of function main
```

```
Original string: This sentence ends in 5 stars *****
^ substituted for *: This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars": This sentence ends in 5 carets ^^^^^
Every word replaced by "word": word word word word word word ^^^^^

Original string: 1, 2, 3, 4, 5, 6, 7, 8
Replace first 3 digits by "digit": digit, digit, digit, 4, 5, 6, 7, 8
string split at commas ["1", "2", "3", "4", "5", "6", "7", "8"]
```

圖 23.5　使用 regex_replace 演算法 (續)

regex_replace 演算法只要原來的字串符合正規化表示法，就會將字串中的文字以新的文字取代。第 19 行中，regex_replace 將 testString1 中的每一個實體 "*" 都以 "^" 取代。請注意正規化表示法 ("*") 的 * 前加上一個反斜線 \。量化詞 * 表示前面的樣式可以出現任意多次，然而在這個範例中，我們希望找出的字面字元是星號 *，所以我們必須將跳脫字元 * 之前加上反斜線。在一個正規化表示法的特殊字元前面加上 \，等於告知正規化表示的搜尋引擎我們所欲搜尋的是原始字元 "*"。第 23-24 行使用 regex_replace 將 testString1 中的 "stars" 以 string "carets" 取代。第 29–30 行中，regex_replace 將 testString1 中的每一個字都以 string "word" 取代。

第 37–41 行將 testString2 中的前三個數字 ("\d") 以 "digit" 取代。我們將引數 **boost::format_first_only** 傳遞給 regex_replace (第 39–40 行)。這個引數告訴 regex_replace，我們只要替換第一個符合正規表示式的子字串就好了。regex_replace 通常會將樣式所出現的每一個地方都替換掉。我們將這個呼叫放入 for 迴圈中執行三次，每一次都會將第一次出現數字的位置替換成文字 "digit"。我們使用的是 testString2 的副本 (第 35 行)，因此我們可以使用原本的 textString2 來進行下一部分的範例。

接下來，我們使用 regex_token_iterator 將一個 string 分割成幾個子字串。**regex_token_iterator** 會走訪 string 中符合正規表示式的部份。第 49 和 51 行使用 sregex_token_iterator，它是一個 typedef 名稱，表示我們使用 string::const_iterator 來操作其結果。我們將兩個循環器 (testString2.begin() 和 testString2.end()) 傳遞給建構子，以建立此循環器 (第 49–50 行)，它們分別代表要比對的 string 之開頭及結尾。在這個範例中，我們想要走訪不符合正規表示式的部份字串，因此將-1 傳遞給建構子。這表示程式應該走訪每一個不符合正規表示式的子字串。原來的字串會在任何符合指定的正規表示式之處被分開。我們利用 while 敘述 (第 53–57 行)，將每個子字串加入 output string 中 (第 13 行)。regex_token_iterator end (第 51 行) 是一個空的循環器。當 tokenIterator 等於 end 的時候 (第 53 行)，就表示我們已經走訪完整個字串了。

23.6　聰明指標與 Boost.Smart_ptr

C 和 C++中有許多常見的錯誤是跟指標有關的。**聰明指標 (Smart pointers)** 提供一些標準指標之外的功能，可以幫助你避免這些錯誤。這些功能會強化記憶體配置和釋放的流程。聰明指標也可以幫助你撰寫出不會有問題的例外程式碼。假如程式在呼叫指標的 delete 之前拋出一個例外，它就會造成記憶體遺漏的問題。在拋出例外之後，聰明指標的解構子仍會被呼叫，它會替你呼叫指標的 delete。

我們在 16.12 節曾提到，C++ 標準函式庫也提供了一個基本的聰明指標：auto_ptr。auto_ptr 負責管理動態配置記憶體，當 auto_ptr 被清除或是離開使用域時，會自動呼叫 delete 來釋放動態記憶體。聰明指標可以用在自動變數上，但是由於轉換動態記憶體所有權的方式，聰明指標無法跟 STL 容器一起使用。

Boost.Smart_ptr 函式庫會提供其他聰明指標，以解決這方面的問題。這些聰明指標並不是用來取代 auto_ptr 的。它們提供不同的功能，以供程式設計師選擇。

23.6.1　參照計數的 shared_ptr

shared_ptr 會替某個需要與程式中其他物件一同分享的資源 (例如動態配置的物件) 儲存一個內部指標。你可以用任意數量的 shared_ptr 指向同一個資源。shared_ptr 會分享這個資 源，假如你用某一個 shared_ptr 改變了這個資源，其他的 shared_ptr 也會「看見」這個改變。假如最後一個指向此資源的 shared_ptr 被清除了，則內部指標就會被刪除。shared_ptr 使用**參照計數 (reference counting)** 來判斷有幾個 shared_ptr 指向此資源。每當建立了新的 shared_ptr 指向此資源時，**參照計數 (reference count)** 就會遞增。當某個 shared_ptr 被清除時，參照計數就會遞減。當參照計數等於零時，內部指標就會被刪除並釋放記憶體。

當我們需要以多個指標指向同一個資源時 (例如在 STL 容器中)，shared_ptr 就會很有用。auto_ptr 不能與 STL 容器一起使用，因爲容器或演算法可能會複製其中的元素。auto_ptr 在經過複製之後，不會等於它的副本，因爲原來的 auto_ptr 會被設定爲 NULL。而 shared_ptr 可以被安全地複製，因此可以使用在 STL 容器中。

shared_ptr 也讓你決定如何刪除資源。對於大多數的動態配置物件來說，我們會使用 delete。然而，某些資源需要更複雜的清除工作。你可以提供自訂的 **deleter** 函式或是函式物件給 shared_ptr 的建構子，來處理這種狀況。我們用 deleter 來決定清除資源的方式。當參照計數等於零時，資源就可以被清除了，此時 shared_ptr 會呼叫自訂的 deleter 函式。這個功能讓 shared_ptr 能夠管理幾乎所有類型的資源。

shared_ptr 使用範例

圖 23.6-23.7 定義一個簡單的 Book 類別，使用一個 string 代表 Book 的名稱。Book 類別 (圖 23.7 第 12-15 行) 的解構子會在螢幕上顯示一個訊息，表示有一個實體被清除了。我們利用這個類別來示範 shared_ptr 常用的功能。

```
1  // Fig. 23.6: Book.h
2  // Declaration of class Book.
3  #ifndef BOOK_H
4  #define BOOK_H
5  #include <string>
6  using namespace std;
7
8  class Book
9  {
10 public:
11    Book( const string &bookTitle ); // constructor
12    ~Book(); // destructor
13    string title; // title of the Book
14 };
15 #endif // BOOK_H
```

圖 23.6　Book 標頭檔

```
1  // Fig. 23.7: Book.cpp
2  // Member-function definitions for class Book.
3  #include <iostream>
4  #include <string>
5  #include "Book.h"
6  using namespace std;
7
8  Book::Book( const string &bookTitle ) : title( bookTitle )
9  {
10 }
11
12 Book::~Book()
13 {
14    cout << "Destroying Book: " << title << endl;
15 } // end of destructor
```

圖 23.7　Book 的成員函式定義

```
1  // Fig. 23.8: fig23_08.cpp
2  // Demonstrate used of shared_ptrs.
3  #include <iostream>
4  #include <vector>
5  #include <boost/shared_ptr.hpp>
6  #include "Book.h"
7  using namespace std;
8
9  typedef boost::shared_ptr< Book > BookPtr; // shared_ptr to a Book
10
11 // a custom delete function for a pointer to a Book
12 void deleteBook( Book* book )
13 {
14    cout << "Custom deleter for a Book, ";
15    delete book; // delete the Book pointer
16 } // end of deleteBook
17
18 // compare the titles of two Books for sorting
19 bool compareTitles( BookPtr bookPtr1, BookPtr bookPtr2 )
20 {
21    return ( bookPtr1->title < bookPtr2->title );
22 } // end of compareTitles
23
24 int main()
25 {
26    // create a shared_ptr to a Book and display the reference count
27    BookPtr bookPtr( new Book( "C++ How to Program" ) );
28    cout << "Reference count for Book " << bookPtr->title << " is: "
29       << bookPtr.use_count() << endl;
30
31    // create another shared_ptr to the Book and display reference count
32    BookPtr bookPtr2( bookPtr );
33    cout << "Reference count for Book " << bookPtr->title << " is: "
34       << bookPtr.use_count() << endl;
35
36    // change the Book's title and access it from both pointers
37    bookPtr2->title = "Java How to Program";
38    cout << "The Book's title changed for both pointers: "
39       << "\nbookPtr: " << bookPtr->title
40       << "\nbookPtr2: " << bookPtr2->title << endl;
41
42    // create a std::vector of shared_ptrs to Books (BookPtrs)
43    vector< BookPtr > books;
44    books.push_back( BookPtr( new Book( "C How to Program" ) ) );
45    books.push_back( BookPtr( new Book( "VB How to Program" ) ) );
46    books.push_back( BookPtr( new Book( "C# How to Program" ) ) );
47    books.push_back( BookPtr( new Book( "C++ How to Program" ) ) );
48
49    // print the Books in the vector
50    cout << "\nBooks before sorting: " << endl;
51    for ( int i = 0; i < books.size(); i++ )
52       cout << ( books[ i ] )->title << "\n";
```

圖 23.8　`shared_ptr` 範例程式

```
53
54      // sort the vector by Book title and print the sorted vector
55      sort( books.begin(), books.end(), compareTitles );
56      cout << "\nBooks after sorting: " << endl;
57      for ( int i = 0; i < books.size(); i++ )
58         cout << ( books[ i ] )->title << "\n";
59
60      // create a shared_ptr with a custom deleter
61      cout << "\nshared_ptr with a custom deleter." << endl;
62      BookPtr bookPtr3( new Book( "Small C++ How to Program" ), deleteBook);
63      bookPtr3.reset(); // release the Book this shared_ptr manages
64
65      // shared_ptrs are going out of scope
66      cout << "\nAll shared_ptr objects are going out of scope." << endl;
67   } // end of main
```

```
Reference count for Book C++ How to Program is: 1
Reference count for Book C++ How to Program is: 2

The Book's title changed for both pointers:
bookPtr: Java How to Program
bookPtr2: Java How to Program

Books before sorting:
C How to Program
VB How to Program
C# How to Program
C++ How to Program

Books after sorting:
C How to Program
C# How to Program
C++ How to Program
VB How to Program

shared_ptr with a custom deleter.
Custom deleter for a Book, Destroying Book: Small C++ How to Program

All shared_ptr objects are going out of scope.
Destroying Book: C How to Program
Destroying Book: C# How to Program
Destroying Book: C++ How to Program
Destroying Book: VB How to Program
Destroying Book: Java How to Program
```

圖 23.8 `shared_ptr` 範例程式 (續)

圖 23.8 的程式使用 `shared_ptr` 來管理 `Book` 類別的幾個實體。我們必需先含括 `<boost/shared_ptr.hpp>` 標頭檔，才能使用 `shared_ptr`。我們也建立了 `typedef BookPtr` 做爲 `boost::shared_ptr <Book>` 的別名 (第 9 行)。第 27 行建立一個指向 `Book` 的 `shared_ptr` (使用 `BookPtr typedef`)，`Book` 的名稱爲 `"C++ How to Program"` `shared_ptr` 建構子接收一個指向物件的指標做爲引數。我們將 `new` 運算子回傳的指標傳遞給它。這樣會建立一個用來管理 `Book` 的 `shared_ptr`，並

將參照計數設定爲 1。建構子也可以接收另一個 shared_ptr，在這種情況下，資源會與另一個 shared_ptr 分享，而且參照計數會遞增 1。第一個指向資源的 shared_ptr 必須使用 new 運算子來建立。使用一般指標所建立的 shared_ptr 會假設它是第一個指向此資源的 shared_ptr，並將參照計數設定爲 1。假如你使用同一個一般指標來建立多個 shared_ptr，則 shared_ptr 不會知道彼此的存在，參照計數會產生錯誤，當 shared_ptr 被刪除時，它們都會呼叫 delete 來清除資源。

第 28–29 行在螢幕上顯示 Book 的名稱以及參照此實體的 shared_ptr 個數。請注意，我們使用 -> 運算子來存取 Book 的資料成員 title，跟一般指標的用法一樣。shared_ptr 提供指標運算子 * 和 ->。我們利用 shared_ptr 的成員函式 **use_count** 取得參照計數，它會回傳指向資源的 shared_ptr 數量。接著我們建立另一個指向類別 Book 的實體的 shared_ptr (第 32 行)。我們傳遞原始的 shared_ptr 給 shared_ptr 建構子。你也可以使用指定運算子 (=) 來建立指向同一個資源的 shared_ptr。第 33-34 行列印出原始 shared_ptr 的計數參照，顯示出建立第二個 shared_ptr 會讓計數遞增 1。如同之前提到的，當 shared_ptr 所指向的資源改變時，會被所有指向該資源的 shared_ptr「看見」。當我們使用 bookPtr2 改變 Book 的名稱時 (第 37 行) ，也可以使用 bookPtr 看見這個改變 (第 38–40 行)。

接下來我們要示範如何在 STL 容器中使用 shared_ptr。我們建立一個 BookPtr 的 vector (第 43 行)，並加入四個元素 (還記得在第 9 行，BookPtr 是 shared_ptr <Book> 的 typedef)。這可以示範使用 shared_ptr 的主要優點，如同我們之前提過的，你不能將 auto_ptr 用在 STL 容器中。第 50–52 行會列印 vector 的內容。接著我們會將 vector 中的 Book 依書名排序 (第 55 行)。我們會使用 sort 演算法中的 compareTitles 函式 (第 19–22 行)，依照字母順序比較每個 title 資料成員。

第 62 行用一個自訂的 deleter 建立 shared_ptr。我們定義自訂的 deleter 函式 deleteBook (第 12–16 行)，將它和指向新 Book 物件的指標一起傳遞給 shared_ptr 建構子。當 shared_ptr 清除 Book 類別的物件時，它會利用內部的 Book* 做爲引數呼叫 deleteBook。請注意，deleteBook 接收 Book*，而不是 shared_ptr。自訂的 deleter 函式必須使用 shared_ptr 的內部指標型別做爲引數。deleteBook 會在螢幕上顯示訊息表示程式呼叫了自訂的 deleter，接著 delete 這個指標。自訂的 deleter 主要是用在第三方 C 函式庫。C++ 函式庫通常會提供類別的建構子和解構子，然而 C 函式庫不同，它們通常用一個函式回傳指向 struct (代表某個資源) 的指標，當該資源不再需要時，則用另一個函式做清除的工作。自訂的 deleter 讓你可以使用 shared_ptr 來追蹤資源，並確保它們被正確地釋放。

我們呼叫 shared_ptr 的成員函式 reset (第 63 行) 以示範自訂的 deleter。**reset** 函式會釋放目前的資源，並將 shared_ptr 設定爲 NULL。假如沒有其他 shared_ptr 指向此資源，就會將它清除。你也可以傳遞一個指向新資源的指標或是 shared_ptr 給 reset 函式，這種情況下，shared_ptr 會管理這個新的資源。但是如果是使用建構子，你只能使用由 new 運算子回傳的一般指標。

在 main 函式的最後，所有的 shared_ptr 和 vector 都會離開使用域。當 vector 被清除時，其中的 shared_ptr 也會被清除。程式的輸出顯示，每一個 Book 類別的物件都會被 shared_ptr 自動清除。你不需要自己 delete vector 中的每一個指標。

23.6.2 weak_ptr：shared_ptr 觀察者

weak_ptr 可以指向 shared_ptr 所管理的資源，但是卻不必負任何管理責任。當 weak_ptr 指向 shared_ptr 所管理的資源時，參照計數不會遞增。這意味著，即使還有 weak_ptr 指向 shared_ptr 的資源，這個資源仍然可以被清除。當最後一個 shared_ptr 被清除時，這個資源就會被刪除，剩餘的 weak_ptr 則會被設爲 NULL。我們會在本節示範，weak_ptr 的用途之一，是避免循環參照所造成的記憶體遺漏。

weak_ptr 沒辦法直接存取它指向的資源，你必須從 weak_ptr 建立一個 shared_ptr 才能存取資源。有兩種方式可以做到這點。你可以將 weak_ptr 傳遞給 shared_ptr 建構子。這樣會替 weak_ptr 所指向的資源建立一個 shared_ptr，並且使參照計數遞增。假如資源已經被刪除了，shared_ptr 建構子會拋出一個 **boost::bad_weak_ptr** 例外。你也可以呼叫 weak_ptr 的成員函式 **lock**，它會回傳一個指向 weak_ptr 的資源的 shared_ptr。假如 weak_ptr 指向已刪除的資源 (也就是 NULL)，則 lock 會回傳一個空的 shared_ptr (也就是指向 NULL 的 shared_ptr)，所以 lock 應該用在空的 shared_ptr 不會造成錯誤的情況下。一旦你擁有 shared_ptr 之後，就可以存取資源了。weak_ptr 是用在你想要觀察資源，卻不想負責管理它的情況下。接下來的範例會示範 weak_ptr 在**循環參照資料 (circularly referential data)** 中的用法，也就是兩個物件在內部互相指向對方的情況。

weak_ptr 使用範例

圖 23.9-23.12 定義 Author 和 Book 類別。每個類別都有一個指標指向另一個類別的實體。這樣會在兩個類別之間建立一個循環參照。請注意，我們同時使用 weak_ptr 和 shared_ptr 兩者來存放每個類別的相互參照 (分別位於圖 23.9 和 23.10 的第 20–21 行)。假如我們使用的是 shared_ptr，程式會產生記憶體遺漏。稍後我們會解釋原因並使用 weak_ptr 來處理這個問題。

```
1  // Fig. 23.9: Author.h
2  // Definition of class Author.
3  #ifndef AUTHOR_H
4  #define AUTHOR_H
5  #include <string>
6  #include <boost/shared_ptr.hpp>
7  #include <boost/weak_ptr.hpp>
8  using namespace std;
9
10 class Book; // forward declaration of class Book
11
12 // Author class definition
13 class Author
14 {
15 public:
16    Author( const string &authorName ); // constructor
17    ~Author(); // destructor
18    void printBookTitle(); // print the title of the Book
19    string name; // name of the Author
20    boost::weak_ptr< Book > weakBookPtr; // Book the Author wrote
21    boost::shared_ptr< Book > sharedBookPtr; // Book the Author wrote
22 };
23 #endif // AUTHOR_H
```

圖 23.9　`Author` 類別定義

```
1  // Fig. 23.10: Book.h
2  // Definition of class Book.
3  #ifndef BOOK_H
4  #define BOOK_H
5  #include <string>
6  #include <boost/shared_ptr.hpp>
7  #include <boost/weak_ptr.hpp>
8  using namespace std;
9
10 class Author; // forward declaration of class Author
11
12 // Book class definition
13 class Book
14 {
15 public:
16    Book( const string &bookTitle ); // constructor
17    ~Book(); // destructor
18    void printAuthorName(); // print the name of the Author
19    string title; // title of the Book
20    boost::weak_ptr< Author > weakAuthorPtr; // Author of the Book
21    boost::shared_ptr< Author > sharedAuthorPtr; // Author of the Book
22 };
23 #endif // BOOK_H
```

圖 23.10　`Book` 類別定義

類別 Author 和 Book 都各自定義了解構子，會在類別物件被清除時，在螢幕上顯示一個訊息 (圖 23.11 和 23.12 第 15-18 行)。每個類別也都定義了會印出 Book 書名以及 Author 名稱的成員函式 (每個圖的 21–34 行)。你應該還記得 weak_ptr 沒辦法直接存取它指向的資源，你必須從 weak_ptr 建立一個 shared_ptr 才能存取資源 (每個圖的第 24 行)。假如 weak_ptr 所指向的資源不存在，則呼叫 lock 函式會回傳一個指向 NULL 的 shared_ptr，此條件式會失敗。否則，新的 shared_ptr 中會包含一個合法的指標指向 weak_ptr 的資源，我們可以藉此存取此資源。假如第 24 行的條件式爲 true (bookPtr 和 authorPtr 不是 NULL)，我們會在螢幕上顯示出參照計數，證明它隨著新的 shared_ptr 而遞增了。接著我們會印出 Book 和 Author 的 name。當程式離開函式時，shared_ptr 會被清除，而參照計數會減一。

```
1  // Fig. 23.11: Author.cpp
2  // Member-function definitions for class Author.
3  #include <iostream>
4  #include <string>
5  #include <boost/shared_ptr.hpp>
6  #include <boost/weak_ptr.hpp>
7  #include "Author.h"
8  #include "Book.h"
9  using namespace std;
10
11 Author::Author( const string &authorName ) : name( authorName )
12 {
13 }
14
15 Author::~Author()
16 {
17    cout << "Destroying Author: " << name << endl;
18 } // end of destructor
19
20 // print the title of the Book this Author wrote
21 void Author::printBookTitle()
22 {
23    // if weakBookPtr.lock() returns a non-empty shared_ptr
24    if ( boost::shared_ptr< Book > bookPtr = weakBookPtr.lock() )
25    {
26       // show the reference count increase and print the Book's title
27       cout << "Reference count for Book " << bookPtr->title
28          << " is " << bookPtr.use_count() << "." << endl;
29       cout << "Author " << name << " wrote the book " << bookPtr->title
30          << "\n" << endl;
31    } // end if
32    else // weakBookPtr points to NULL
33       cout << "This Author has no Book." << endl;
34 } // end of printBookTitle
```

圖 23.11　Author 的成員函式定義

```
1  // Fig. 23.12: Book.cpp
2  // Member-function definitions for class Book.
3  #include <iostream>
4  #include <string>
5  #include <boost/shared_ptr.hpp>
6  #include <boost/weak_ptr.hpp>
7  #include "Author.h"
8  #include "Book.h"
9  using namespace std;
10
11 Book::Book( const string &bookTitle ) : title( bookTitle )
12 {
13 }
14
15 Book::~Book()
16 {
17    cout << "Destroying Book: " << title << endl;
18 } // end of destructor
19
20 // print the name of this Book's Author
21 void Book::printAuthorName()
22 {
23    // if weakAuthorPtr.lock() returns a non-empty shared_ptr
24    if ( boost::shared_ptr< Author > authorPtr = weakAuthorPtr.lock() )
25    {
26       // show the reference count increase and print the Author's name
27       cout << "Reference count for Author " << authorPtr->name
28          << " is " << authorPtr.use_count() << "." << endl;
29       cout << "The book " << title << " was written by "
30          << authorPtr->name << "\n" << endl;
31    } // end if
32    else // weakAuthorPtr points to NULL
33       cout << "This Book has no Author." << endl;
34 } // end of printAuthorName
```

圖 23.12　Book 的成員函式定義

圖 23.13 定義一個 main 函式，示範 Author 和 Book 類別之間的循環參照所導致的記憶體遺漏。第 12–14 行建立指向每個類別物件的 shared_ptr。第 17–18 行設定 weak_ptr 資料成員。第 21–22 行設定每個類別的 shared_ptr 資料成員。現在，Author 類別和 Book 類別的實體成爲互相參照對方的情況。接著，我們印出 shared_ptr 的參照計數，顯示每個實體都被兩個 shared_ptr 參照 (第 25–28 行)：是我們在 main 函式中建立的，另一個是每個實體中的資料成員。還記得吧，weak_ptr 不會影響參照計數。接著我們呼叫每個類別的成員函式以印出儲存在 weak_ptr 資料成員中的資訊 (第 33–34 行)。這個函式也顯示出另一個 shared_ptr 會在函式呼叫的過程中被建立。最後，我們再次列印參照計數，顯示在 printAuthorName 和 printBookTitle 成員函式中建立的 shared_ptr 會在離開函式時被清除掉。

```
1  // Fig. 23.13: fig23_13.cpp
2  // Demonstrate use of weak_ptr.
3  #include <iostream>
4  #include <boost/shared_ptr.hpp>
5  #include "Author.h"
6  #include "Book.h"
7  using namespace std;
8
9  int main()
10 {
11    // create a Book and an Author
12    boost::shared_ptr< Book > bookPtr( new Book( "C++ How to Program" ) );
13    boost::shared_ptr< Author > authorPtr(
14       new Author( "Deitel & Deitel" ) );
15
16    // reference the Book and Author to each other
17    bookPtr->weakAuthorPtr = authorPtr;
18    authorPtr->weakBookPtr = bookPtr;
19
20    // set the shared_ptr data members to create the memory leak
21    bookPtr->sharedAuthorPtr = authorPtr;
22    authorPtr->sharedBookPtr = bookPtr;
23
24    // reference count for bookPtr and authorPtr is one
25    cout << "Reference count for Book " << bookPtr->title << " is "
26       << bookPtr.use_count() << endl;
27    cout << "Reference count for Author " << authorPtr->name << " is "
28       << authorPtr.use_count() << "\n" << endl;
29
30    // access the cross references to print the data they point to
31    cout << "\nAccess the Author's name and the Book's title through "
32       << "weak_ptrs." << endl;
33    bookPtr->printAuthorName();
34    authorPtr->printBookTitle();
35
36    // reference count for each shared_ptr is back to one
37    cout << "Reference count for Book " << bookPtr->title << " is "
38       << bookPtr.use_count() << endl;
39    cout << "Reference count for Author " << authorPtr->name << " is "
40       << authorPtr.use_count() << "\n" << endl;
41
42    // the shared_ptrs go out of scope, the Book and Author are destroyed
43    cout << "The shared_ptrs are going out of scope." << endl;
44 } // end of main
```

圖 23.13　在循環參照資料中的 `shared_ptr` 會導致記憶體遺漏

```
Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

Access the Author's name and the Book's title through weak_ptrs.
Reference count for Author Deitel & Deitel is 3.
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 3.
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

The shared_ptrs are going out of scope.
```

圖 23.13　在循環參照資料中的 shared_ptr 會導致記憶體遺漏 (續)

在 main 函式的最後，指向 Author 和 Book 物件的 shared_ptr 會離開使用域並被清除。注意，我們在輸出中並沒有看到 Author 和 Book 的解構子被執行。這個程式產生了記憶體遺漏，由於 shared_ptr 的關係，Author 和 Book 物件沒有被清除。在 main 函式的最後，當 bookPtr 被清除時，Book 物件的參照計數變成 1，因爲 Author 物件還有一個 shared_ptr 指向 Book 物件，所以它沒有被刪除。而當 authorPtr 離開使用域被清除時，Author 物件的參照計數變成 1，因爲 Book 物件還有一個 shared_ptr 指向 Author 物件。因爲這兩個物件的參照計數都是 1，所以它們都不會被刪除。

現在，請在第 21–22 行的開頭加上 //，將它們變成註解。這樣就可以防止程式碼設定 Author 和 Book 類別的 shared_ptr 資料成員。接著重新編譯程式碼，再執行程式一次。圖 23.14 顯示了這次的輸出。你可以看到，每個物件的參照計數一開始是 1 而不是 2，因爲我們沒有設定 shared_ptr 資料成員。輸出的最後兩行顯示 Author 和 Book 物件在 main 函式的最後被清除了。我們藉著使用 weak_ptr 資料成員替代 shared_ptr 資料成員，消除了記憶體遺漏的狀況。weak_ptr 沒有影響參照計數，但是仍然允許我們存取資源，只要建立暫時的 shared_ptr 指向資源即可。當我們在 main 函式中建立的 shared_ptr 被清除時，參照計數會變成零，Author 和 Book 物件會正確地被刪除。

```
Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

Access the Author's name and the Book's title through weak_ptrs.
Reference count for Author Deitel & Deitel is 2.
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 2.
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

The shared_ptrs are going out of scope.
Destroying Author: Deitel & Deitel
Destroying Book: C++ How to Program
```

圖 23.14 在循環參照資料中的 `weak_ptr` 可用來防止記憶體遺漏

23.7 Technical Report 1

TR1 是對 C++ 標準函式庫的變更及新增建議。TR1 中的許多函式庫會被 C++ 標準委員會接受，但是在下一個版本確認之前，它們仍然不屬於 C++ 標準的一部分。新增的函式庫替許多程式設計上的困難提供了解決方法。這些新增的函式庫大多包含在 11 個 Boost 函式庫中──也就是我們在第 23.4 節介紹的 Boost 函式庫以及其他幾個比較小的函式庫。接下來會介紹另外三個 TR1 函式庫。

Visual Studio 2008 SP1 以及最新版的 GNU C++ 支援大部份的 TR1 功能。Boost 提供了一個相容層，會在編譯器不支援時自動回到 Boost 函式庫實作。[10]

由於時間的限制，許多函式庫沒有放入 TR1 中。**Technical Report 2 (TR2)** 會在 C++0x 之後不久發佈，它包含了 TR1 所沒有的其他建議函式庫。TR2 會讓標準函式庫具有更多功能，不需要等到下一個新的標準發佈。

無序的關聯式容器 (Unordered Associative Container) [11]

無序關聯式容器函式庫定義四種新的容器：`unordered_set`、`unordered_map`、`unordered_multiset` 和 `unordered_multimap`。這些關聯式容器是實作成雜湊

[10] 參考文件 Boost.TR1, John Maddock, `www.boost.org/doc/html/boost_tr1.html`.

[11] Matthew Austern, "A Proposal to Add Hash Tables to the Standard Library," Document Number N1456=03-0039, April 9, 2003, `www.open-std.org/jtc1/sc22/wg21/docs/2003/n1456.html`.

表。**雜湊表 (hash tables)** 分成許多區塊，稱爲**「桶 (buckets)」**。我們使用鍵值來決定元素儲存在容器的哪一個位置。鍵值會被傳遞給**雜湊函式 (hash function)**，然後回傳 size_t。雜湊函式回傳的 size_t 是用來決定要將數值放在哪一個「桶」中，假如兩個數值相等，則 size_t 回傳的值也會相同。我們可以將多個值放在同一個「桶」中，用 set 或 map 的方式，以鍵值從容器中取出元素。鍵值會決定數值存在哪一個「桶」中，接著就要在那個桶中搜尋我們想要的值。

unordered_set 和 **unordered_multiset** 使用元素本身作爲鍵值。**unordered_map** 和 **unordered_multimap** 使用另外的鍵值來決定要將元素放在哪裡 — 引數的形式爲數對 <const Key, Value>。unordered_set 和 unordered_map 要求所有的鍵值必須是唯一的，unordered_multiset 和 unordered_multimap 則沒有此限制。這些容器定義在 <unordered_set> 和 <undordered_map> 標頭檔中。

數學特殊函式 (Mathematical Special Function) [12]

這個函式庫合併了 **C99** 中加入的數學函式，C99 是 1999 年發布的 C 語言標準，不包括在 C++標準中。C99 提供了三角、雙曲線、指數、對數、乘冪和其他的特殊函式。此函式庫將這些函式 (以及其他函式) 加入 C++的 <cmath> 標頭檔中。

增加與 C99 的相容性 [13]

C++來自 C 程式語言。大多數的 C++編譯器也能編譯 C 的程式，但是兩種語言之間還是有一些不相容的地方。此函式庫的目的是增加 C++和 C99 的相容性，在 C++標頭檔中加入支援 C99 功能的項目。這個任務通常是藉由含括 C99 的標頭來達成的。

23.8　C++0x

C++標準委員會目前正在修訂 C++標準。上一個標準是在 1998 年發布的。目前正在修訂的新標準，也就是 C++0x，是從 2003 年開始的。這個新標準預計會在 2010 或 2011 年

---

[12] Walter E. Brown, "A Proposal to Add Mathematical Special Functions to the C++ Standard Library," Document Number N1422=03-0004, February 24, 2003, std.dkuug.dk/jtc1/sc22/wg21/docs/ papers/2003/n1422.html.

[13] P. J. Plauger, "Proposed Additions to TR-1 to Improve Compatibility With C99," Document Number N1568=04-0008, www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1568.htm.

發布，包括 TR1 函式庫以及其他新增的核心語言。在 C++資源中心 (www.deitel.com/cplusplus/) 的 C++0x 章節，可以找到 C++0x 的資訊 (點選分類清單中的 C++0x)。

標準化流程

國際標準化組織 (International Organization for Standardization，ISO) 監督國際程式語言標準的制定，包括 C 和 C++。每一個 C++標準的增加或變更，都必須經過維護 C++ 標準的委員會 ISO/IEC JTC 1/SC 22 Working Group 21 (WG21) 的核准。這個委員會由來自 C++ 程式設計委員會的志願者所組成，他們一年舉行兩次會議，討論 C++ 標準的相關議題。在正式的會議之間，則會舉行較小型的非正式會議以討論提案。ISO 要求新的標準草案之間至少要間隔五年的時間。

C++0x 的目標 [14]

C++ 程式語言的創造者，Bjarne Stroustrup 曾經表達過他對 C++ 未來願景的看法，他認爲新標準的主要目標是要讓 C++ 更容易學習，改善函式庫建構能力，並增加與 C 程式語言的相容性。

23.9 核心語言的改變

你可以在下列網址找到核心語言的變更建議列表：www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2869.html。其中包括每個建議的相關論文連結。我們簡單地討論新標準的草案中已確立的核心語言變更中較重要的一部分。在新標準定案之前，加入工作草案的提案數目應該會再增加。GNU C++ 編譯器有 C++0x 模式可供選擇，讓你能夠試驗許多核心語言的改變 (gcc.gnu.org/projects/cxx0x.html)。Visual Studio 2010 亦支援某些 C++0x 的功能──你可以在 Visual C++ 團隊的部落格 (blogs.msdn.com/ vcblog/) 中找到 C++0x 在 Visual Studio 中的更新狀況。

---

[14] Bjarne Stroustrup, “The Design of C++0x,” May 2005, www.research.att.com/~bs/rules.pdf.

Rvalue 參照 [15]

C++0x 的 **rvalue 參照 (rvalue reference)** 型別允許你將 rvalue (臨時物件) 連結到一個非 `const` 參照。rvalue 參照宣告爲 `T&&` (`T` 是被參照的物件型別)，與一般參照 `T&` 不同 (稱爲 lvalue 參照)。rvalue 參照可以有效地實作搬移語意—物件的狀態可以被搬移 (並非複製) 而遺留下空值。例如，下面的程式碼建立一個暫時的 `string` 物件，並將它傳到 `push_back`，接著再把它複製到 `vector` 中。

```
vector< string > myVector;
myVector.push_back( "message" );
```

假如 `push_back` 被多載爲可以接收 rvalue 參照，則 `string` 暫存物件的儲存空間可以直接被 `vector` 中的 `string` 物件利用。當函式返回時，暫存的物件必然會被摧毀，所以沒有必要保有它的值。

Rvalue 參照也可以用在「轉呼叫函式」(forwarding function) － 轉換函式以接收較少引數的函式物件 (例如：`std::bind1st` 或是使用 `Boost.Bind` 建立的函式物件) 。每個參照參數通常會需要一個 `const` 和一個非 `const` 版本提供給 lvalue、`const` lvalue 以及 rvalue。使用 rvalue 參照，你只需要一個轉呼叫函式。

`static_assert` [16]

`static_assert` 宣告讓你可以在編譯時期測試程式的某些部分。`static_assert` 宣告會接收一個常數整數運算式以及一個字面字串。假如運算式的值等於 `0(false)`，則編譯器會發出錯誤訊息。錯誤訊息中包含了宣告中所提供的字面字串。`static_assert` 宣告可以用在命名空間、類別或區域範圍中。

`static_assert` 的加入讓學習 C++更容易。程式設計新手經常犯的錯誤包括在函式呼叫或樣板具現化中使用錯誤的引數型別，此時斷言可以提供更多有用的錯誤訊息。它們在函式庫開發中也很有用，可以有效地警告錯誤的函式庫使用方式。

---

[15] Howard E. Hinnant, "A Proposal to Add an *rvalue* Reference to the C++ Language," October 19, 2006, Document Number N2118=06-0188, `www.open-std.org/jtc1/sc22/wg21/docs/papers/ 2006/n2118.html`.

[16] Robert Klarer, Dr. John Maddock, Beman Dawes and Howard Hinnant, "Proposal to Add Static Assertions to the Core Language," Document Number N1720, October 20, 2004, `www.openstd.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html`.

增加與 C99 的相容性

C++0x 加入了 1999 C 標準中新增的許多變更，包括了前置處理器 [17]、新增 `long long` 整數型別 [18] 以及新增整數型別 (例如 128 位元的整數型別) 的擴充原則 [19]。讓新的 C 程式能如同 C++一般地正確編譯。

委派建構子 [20]

這個功能讓建構子可以將初始化工作委派給另一個類別的建構子 (呼叫另一個類別的建構子)。這可以讓我們比較容易撰寫多載建構子。目前，多載建構子必須複製其他建構子中相同的程式碼。這樣會導致程式碼重複而且容易出錯。某一個建構子的錯誤可能會造成物件初始化的不一致。藉由呼叫另一個建構子版本，我們不需要重複同樣的程式碼，因而減少了錯誤發生的機率。

右角括號 (Right Angle Brackets) [21]

當我們使用巢狀樣板時，需要在右角括號 (>) 之間加上空白。假如沒有加上空白，編譯器會認爲這兩個角括號是右移運算子 (>>)。也就是說，撰寫 `vector <class T>>` 會導致編譯錯誤。這個敘述應該要寫成 `vector <class T> >`。許多初學者都會犯這個錯誤。在 C++0x 中，C++ 編譯器能辨識出樣板的 >>，不會誤認爲右移運算子。

---

[17] Clark Nelson, "Working Draft Changes for C99 Preprocessor Synchronization," Document Number N1653, July 16, 2004, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1653.htm`.

[18] J. Stephen Adamczyk, "Adding the long long Type to C++," Document Number N1811, April 29, 2005, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1811.pdf`.

[19] J. Stephen Adamczyk, "Adding Extended Integer Types to C++," Document Number N1988, April 19, 2006, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1988.pdf`.

[20] Herb Sutter and Francis Glassborow, "Delegating Constructors," Document Number N1986=060056, April 6, 2006, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1986.pdf`.

[21] Daveed Vandevoorde, "Right Angle Brackets," Document Number N1857=05-0017, January 14, 2005, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html`.

由初始值推斷變數型別 [22]

這個提案定義了關鍵字 auto 的新功能，它會依照變數的初始值運算式，自動決定變數型別。auto 可以用來替代冗長複雜的型別，這些型別很難徒手鍵入。auto 可以與 const 和 volatile 修飾字一起使用。你可以使用 auto 建立指標和參照，就跟你使用完整的型別名稱一樣。auto 支援在單一敘述中宣告多個變數 (例如：auto x = 1, y=2)。auto 關鍵字可以用來節省時間，讓學習過程更輕鬆，以及改良泛型程式設計。下列程式碼會建立 Class<T> 物件的 vector。

```
vector< Class< T > > myVector;
vector< Class< T > >::const_iterator iterator = myVector.begin();
```

利用 auto，循環器的宣告可以寫成：

```
auto iterator = myVector.begin();
```

循環器的型別是 vector<Class T>>::const_iterator。你也可以在單一宣告中，建立兩個型別相同的變數。以下敘述

```
auto iteratorBegin = myVector.begin(), iteratorEnd = myVector.end();
```

會建立兩個型別爲 vector<Class<T>>::constiterator 的變數。你也可以將 auto 與 const 或 volatile 修飾字一同使用，建立指標或參照。以下敘述

```
const auto &iteratorRef = myVector.begin();
```

會建立一個指向 vector<Class<T>>::const_iterator 的 const 參照。auto 自動幫你決定所宣告的變數型別，讓你節省很多時間。

Variadic 樣板 [23]

目前，類別或函式樣板的樣板參數數目是固定的。假如你希望類別或函式樣板具有不同的樣板參數數目，你必須分別定義這些樣板。**variadic 樣板**會接受任意數目的引數，這

---

[22] Jaakko Järvi, Bjarne Stroustrup and Gabriel Dos Reis, "Deducing the Type of Variable From Its Initializer Expression," Document Number N1984=06-0054, April 6, 2006, www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1984.pdf.

[23] Douglas Gregor, Jaakko Järvi and Gary Powell, "Variadic Templates," Document Number N2080=060150, September 9, 2006, www.osl.iu.edu/~dgregor/cpp/variadic-templates.pdf.

樣可以大幅簡化樣板的程式撰寫。你可以提供一個 variadic 函式樣板來替代多個具有不同參數的多載樣板。許多樣板函式庫，像是 Boost.Bind、Boost.Tuple 以及 Boost.Function，運用了大量的重複程式碼或使用複雜的前置處理器巨集來產生所需的樣版定義。Variadic 樣版可以簡化此類函式庫的實作。

樣板別名 (template alias) [24]

函式庫經常會使用具有許多參數的樣板來實作泛型程式。假如我們可以指定樣板中某些不會改變的引數，而讓其他引數仍舊可以變化，將會很有幫助。樣板別名可以達到這個目標。**樣板別名 (template alias)** 類似 `typedef`，它會產生一個用來參照樣板的名稱。在 `typedef` 中，所有的樣板參數都是固定的。當我們使用樣板別名時，可以固定某些參數，並讓其他參數保持其可變動性。你可以使用樣板別名設定某些固定的參數，而讓其他參數在實體化時仍然可以變動。藉此讓一般用途的樣板成爲更特殊化的樣板。例如，下列敘述式宣告樣板 `MyStack`，它使用 `list<T>` 做爲底層的實作以替代 `std::stack` 中預設的 `deque`。

```
template< typename T > using MyStack< T > =
   stack< T, list< T > >;
```

使用者自訂型別的初始值列表 [25]

目前，初始值列表只能用在陣列和 `struct` 中。在 C++0x 中，類別可以讓建構子接收型別爲 **`std::initializer_list<T>`** 的參數。這樣就可以使用初始值列表來初始化某一類別的物件，如下：

```
vector< int > second = { 4, 5, 6 }; // legal in C++0x
```

初始值儲存在 `initializer_list` 物件中，然後傳給類別的建構子。所有的標準函式庫容器類別都會被更新，使建構子能夠接收 `initializer_list`。

---

[24] Gabriel Dos Reis and Mat Marcus, "Proposal to Add Template Aliases to C++," Document Number N1449=03-0032, April 7, 2003, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/ n1449.pdf`.

[25] J. Stephen Adamczyk, Gabriel Dos Reis and Bjarne Stroustrup, "Initializer list WP wording (Revision 2)," Document Number N2531=08-0041, `www.open-std.org/jtc1/sc22/wg21/docs/ papers/2008/n2531.pdf`

基於範圍的 `for` 敘述式 (Range-Based for Statement) [26]

`for` 敘述式經常用來走訪容器中的元素。目前，內建陣列與函式庫容器的語法並不相同，內建陣列使用索引或是原始指標，而容器類別則使用 `begin` 或 `end` 成員函式所傳回的循環器。爲了簡化語法，**基於範圍的 `for` 敘述式 (range-based for statement)** 讓你使用相同的語法來走訪陣列和容器。下列程式碼走訪一組 `int` 值：

```
for ( int &item : items ) // items can be an array or container
   item *= 2;
```

Lambda 運算式 [27]

許多函式庫函式會接收函式指標或函式物件作爲參數。目前，函式或函式物件在傳遞給函式庫函式作爲引數之前必須先被定義。**Lambda 運算式** (或 **lambda 函式**) 讓你可以在傳遞函式物件給函式的同時定義它們。它們可以定義在函式內部，並且「抓到」(以傳值或傳參考的方式) 外圍函式區塊的區域變數，在 lambda 本體內處理這些數值。Visual Studio 2010 中實作了 lambda 運算式，但是 GNU C++以及許多其他編譯器則尚未支援。圖 23.15 示範了簡單的 lambda 運算式，將 `int` 陣列中的每個元素值加倍。

```
1  // Fig. 23.15: fig23_15.cpp
2  // Example of lambda expressions in C++0x.
3  #include <iostream>
4  #include <algorithm>
5  using namespace std;
6
7  int main()
8  {
9     const int size = 4; // size of array values
10    int values[ size ] = { 1, 2, 3, 4 }; // initialize values
11
12    // output each element multiplied by two
13    for_each( values, values + size,
```

圖 23.15　C++0x 的 lambda 運算式範例

---

[26] Thorsten Ottosen, "Wording for range-based for-loop (revision 3)," Document Number N2934=07-0254, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2394.html`

[27] Jaakko Järvi, John Freeman and Lawrence Crowl, "Lambda Expressions and Closures:Wording for Monomorphic Lambdas (Revision 4)," Document Number N2550=08-0060, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf`

```
14          []( int i ) { cout << i * 2 << endl; } );
15
16       int sum = 0; // initialize sum to zero
17
18       // add each element to sum
19       for_each( values, values + size,
20          [ &sum ]( int i ) { sum += i; } );
21
22       cout << "sum is " << sum << endl; // output sum
23    } // end main
```

```
2
4
6
8
sum is 10
```

圖 23.15 C++0x 的 lambda 運算式範例 (續)

第 9 和 10 行宣告並初始化一個小型的整數陣列。第 13-14 行對 `values` 陣列的每個元素呼叫 `for_each` 演算法。`for_each` 的第 3 個引數爲 lambda 運算式。Lambda 運算式以 `[]` (lambda introducer) 開始，後面接著參數列和函式本體。假如函式本體是 `return` expression；型態的單一敘述式，就可以直接推論出回傳型別，否則回傳型別預設爲 `void`。第 14 行的 lambda 運算式接收一個 `int` 值，將它乘以 2，並顯示出結果。`for_each` 演算法將陣列中的每一個元素傳遞給 lambda。

第二次呼叫 `for_each` 演算法 (第 19–20 行) 時則計算陣列元素的總和。[&sum] 表示這個 lambda 運算式以傳參考的方式抓取了區域變數 `sum` (注意我們使用了&)，因此 lambda 可以更改 `sum` 的值。假如沒有使用&，`sum` 會以傳值的方式被抓住，則函式無法更新區域變數的值。`for_each` 演算法將陣列中的每一個元素傳遞給 lambda，將數值加入 `sum`。接著第 22 行會在螢幕上顯示 `sum` 的值。

概念 (Concepts) [28]

錯誤地使用樣板所產生的錯誤訊息通常是冗長而且難以理解的。這種訊息通常會出現在演算法中使用錯誤型別的循環器時。例如 `sort` 演算法必須使用隨機存取循環器。假如

[28] Douglas Gregor, Bjarne Stroustrup, James Widman and Jeremy Siek, "Proposed Wording for Concepts (Revision 9)," Document Number N2773=08-0283, `www.open-std.org/jtc1/sc22/wg21/ docs/papers/2008/n2773.pdf`

你對 list 使用了 sort 演算法，你會收到錯誤訊息，因爲 list 只支援雙向循環器。不幸的是，我們沒有辦法在程式語言中清楚指定 sort 的循環器需求。

概念 (Concepts) 提供了一個解決方法。每個樣板會附加一個概念定義，指定樣板的需求規格。每個概念都會定義對應的樣板使用某個型別時，必須支援哪些運算。例如，LessThanComparable 概念可能會要求某個型別必須提供 operator< 的定義。假如某個要求 LessThanComparable 的樣板使用了不具有 operator< 定義的型別，編譯器會告知需要 LessThanComparable。

23.10　總結

我們在本章討論了許多有關 C++未來的發展。我們介紹了 Boost C++函式庫以及其中最常用的一些函式庫。

我們討論了 Boost.Regex 函式庫以及用在正規表示式中的符號。我們提供了 Boost.Regex 的使用範例，包括 regex, match_results 以及 regex_token_iterator。你學到了如何在字串中找到樣式，以及使用 Boost.Regex 演算法 regex_search 和 regex_match 比對整個字串與樣式。我們介紹了如何使用 regex_replace 替換字串中的字元，以及如何使用 regex_token_iterator 將字串切割成字符。

我們示範了如何使用 Boost.Smart_ptr 函式庫。你學到了如何使用 shared_ptr 和 weak_ptr 類別，在使用動態配置記憶體時，避免記憶體遺漏。我們介紹了如何使用自訂的 deleter 函式，讓 shared_ptr 管理需要特殊清除程序的資源。我們也解釋了如何使用 weak_ptr 來避免循環參照資料中的記憶體遺漏。

我們簡介了即將發表的標準－ C++0x，討論了 TR1 以及核心語言的改變。我們介紹了 TR1 中的函式庫。我們介紹了新的核心語言功能，包括關鍵字 auto，rvalue 參照，與 C99 的相容性，初始值列表以及 lambda 運算式。請記得，Boost，TR1 以及 C++0x 是持續變動的，請連上我們的網路資源中心取得最新的資料。

摘要

23.2　Deitel 線上 C++ 資源中心及相關資源。

- 你可以在 C++ Boost Libraries 資源中心 www.deitel.com/CPlusPlusBoostLibraries/ 找到目前可用的函式庫和新釋出版本的資訊。

- 在 C++資源中心 (www.deitel.com/cplusplus/) 的 C++0x 章節，可以找到 TR1 和 C++0x 的資訊 (點選分類清單中的 C++0x)。
- 想要取得 Visual C++的資訊，可以參考我們的線上 Visual C++資源中心，其網址爲 www.deitel.com/VisualCPlusPlus/。

23.3 Boost 函式庫

- Boost 資料庫 (www.boost.org) 提供免費的同儕評核 C++ 資料庫。
- Boost 的函式庫應該要遵守 C++ 標準，並使用 C++ 標準函式庫 (或是其他適當的 Boost 函式庫)。
- 每個初步提交的 Boost 函式庫會發表在 Boost Sandbox Vault。
- 審查委員會先確認程式碼已經準備好要開始正式審查，設定審查時程，閱讀所有使用者的審查，最後決定是否要接受這個函式庫。
- Boost 軟體授權條款允許複製、修改、使用、發布 Boost 原始碼和執行檔做爲商業和非商業用途。
- 你可以在 www.boost.org/more/getting_started/index.html 找到適用於各種平台和編譯器的安裝指南。

23.4 Boost 函式庫概述

- Boost.Array 提供固定大小的陣列包裝器，支援 STL 容器介面。
- Boost.Bind 擴充標準函式 bind1st 和 bind2nd 的功能。它允許你轉換多達九個引數的函式。也讓你輕鬆地將傳給函式的引數重新排序。
- Boost.Function 讓你在函式包裝器中儲存函式指標、成員函式指標和函式物件。boost::function 可以儲存函式，將它的引數和回傳型別轉換爲符合函式包裝器的簽章。
- Boost.Random 讓你可以建立各種各樣的亂數產生器和亂數分布。
- 虛擬亂數產生器 (pseudo-random number generator) 利用初始狀態來產生虛擬的亂數，也就是說，使用相同的初始狀態會產生同樣的一串數目。
- 正規表示式是用來比對文字中的特定字元樣式。
- 你可以使用 Boost.Regex 搜尋 string 中的特殊表示式，將符合正規表示式的部份取代爲其他字元，或是使用正規表示式將 string 切割成字符。
- Boost.Smart_ptr 定義聰明指標，可以幫助你管理動態配置的資源。
- shared_ptr 處理動態配置物件的生命週期管理。假如沒有其他 shared_ptr 指向記憶體，該記憶體就會被釋放。

- weak_ptr 讓你可以觀察 shared_ptr 中的值，而不用負任何管理責任。
- Boost.Tuple 讓你可以產生一組物件，用在泛型函式中。
- ype_traits 類別讓你判斷某個型別的特色，然後執行型別轉換，讓物件使用通用的程式碼。

23.5　正規表示式與 Boost.Regex 函式庫

- 正規表示式 (Regular expressions) 是格式化的 string，用來搜尋文字中的特定樣式。
- basic_regex 代表正規化表示式，
- regex_match 演算法會在整個字串符合正規表示式時，回傳 true 值。
- regex_search 演算法會在 string 的任何一部分符合正規表示式時，回傳 true 值。
- 使用 Boost.Regex 時，需含括標頭檔 <boost/regex.hpp>。
- 字元類別用來表示一組字元。
- 文字字元 (word character) 是任何字母數字或是底線字元。空白字元 (\s) 爲空白鍵、tab 鍵、換行、新行或表格饋入字元 (form feed)。數字字元 (\d) 是所有可以數學操作的字元。

22.5.1　正規表示式範例

- 在字串中，每個字元類別的反斜線字元都必須使用另一個反斜線來跳脫。
- 除了事先定義的字元類別以外，若要指定某一組字元，可以將這些字元列在方括號 [] 中。指定一範圍的字元可以用連字號 (-) 表示。在 [] 以外的"－"是被當做字面字元來看待的。
- 將 ^ 放在方括號中的第一個位置，會將樣式指定爲方括號之外的字元。
- 符合正規表示式的字串會儲存在 match_results 物件中。typedef smatch 代表一個 match_results，經由 string::const_iterator 提供符合的結果。
- match_not_dot_newline 會防止 "." 字元比對換行字元。
- match_results 的成員函式 suffix 會傳回上一個符合字串尾端到搜尋字串尾端的字串。
- 量化詞 "*" 表示該樣式出現零次以上。
- 量化詞 "+" 表示該樣式出現一次以上。
- 量化詞 "?" 表示該樣式出現零或一次。
- 大括號中間包含一個數字 ({n}) 表示該樣式恰好出現 n 次。
- 在大括號中含有一個數字後面接著一個逗點表示該樣式至少出現 n 次以上。
- {n,m}代表前方樣式出現 n 到 m 次之間(包括兩端點值)。
- 所有的量化詞都是「貪婪的」(greedy)，這表示只要仍符合，他們會對應到出現最多指定樣式的情況。

- 然而，如果在量化詞後面加上問號 (?)，量化詞將變爲懶惰的 (lazy)。表示盡可能的以規定的表示式符合越短的字串。

22.5.2 利用正規表示式驗證使用者輸入

- 字元 "^" 和 "$" 分別代表 string 的開頭和結尾。
- 字元 "|" 表示左右兩邊的表示法都許可，
- 我們可以將量化詞放在括號中，用來建立更複雜的正規表示式。

22.5.3 置換以及切割字串

- regex_replace 演算法只要原來的字串符合正規化表示法，就會將字串中的文字以新的文字取代。
- 在一個正規化表示法的特殊字元前面加上 \，等於告知正規化表示的搜尋引擎我們所欲搜尋的是原始字元"*"。
- format_first_only 告訴 regex_replace，我們只要替換第一個符合正規表示式的子字串就好了。regex_replace 通常會將樣式所出現的每一個地方都替換掉。
- regex_token_iterator 會走訪 string 中符合正規表示式的部份。
- 我們將兩個循環器傳遞給建構子，以建立循環器，它們分別代表要比對的 string 之開頭及結尾。
- 將-1 傳給 regex_token_iterator 表示程式應該走訪每一個不符合正規表示式的子字串。

23.6 聰明指標與 Boost.Smart_ptr

- 聰明指標強化記憶體配置和釋放的流程，以避免錯誤的發生。
- 在拋出例外之後，聰明指標的解構子仍會替你呼叫指標的 delete。

22.6.1 參照計數的 shared_ptr

- shared_ptr 會替某個需要與程式中其他物件一同分享的資源 (例如動態配置的物件) 儲存一個內部指標。
- 當 shared_ptr 所指向的資源改變時，會被所有指向該資源的 shared_ptr「看見」。
- shared_ptr 使用參照計數 (reference counting) 來判斷有幾個 shared_ptr 指向此資源。當參照計數等於零時，內部指標就會被刪除。
- shared_ptr 可以被安全地複製，因此可以使用在 STL 容器中。

- 你可以使用自訂的 deleter 來建立 shared_ptr，指定清除資源的方式。自訂的 deleter 函式必須使用 shared_ptr 的內部指標型別做爲引數。
- 我們必需先含括 <boost/shared_ptr.hpp> 標頭檔，才能使用 shared_ptr。
- shared_ptr 建構子接收一個指向物件的指標做爲引數。建構子也可以接收另一個 shared_ptr，在這種情況下，資源會與另一個 shared_ptr 分享，而且參照計數會遞增 1。
- 第一個指向資源的 shared_ptr 必須使用 new 運算子來建立。
- shared_ptr 提供指標運算子 * 和 ->。
- 我們利用 shared_ptr 的成員函式 use_count 取得參照計數，它會回傳指向資源的 shared_ptr 數量。
- reset 函式會釋放目前的資源，並將 shared_ptr 設定爲 NULL。你也可以傳遞一個指向新資源的指標或是 shared_ptr 給 reset 函式，這種情況下，shared_ptr 會管理這個新的資源。

23.6.2 weak_ptr：shared_ptr 觀察者

- weak_ptr 可以指向 shared_ptr 所管理的資源，但是卻不必負任何管理責任，shared_ptr 的參照計數不會遞增。
- 當最後一個 shared_ptr 被清除時，這個資源就會被刪除，剩餘的 weak_ptr 則會被設爲 NULL。
- weak_ptr 沒辦法直接存取它指向的資源，你必須從 weak_ptr 建立一個 shared_ptr 才能存取資源。你可以將 weak_ptr 傳遞給 shared_ptr 建構子。你也可以呼叫 weak_ptr 的成員函式 lock，它會回傳一個指向 weak_ptr 的資源的 shared_ptr。
- 我們必需先含括 <boost/weak_ptr.hpp> 標頭檔，才能使用 weak_ptr。

23.7 Technical Report 1

- TR1 是對 C++ 標準函式庫的變更及新增建議。新增的函式庫大多包含在 11 個 Boost 函式庫中。
- Visual Studio 2008 SP1 以及最新版的 GNU C++ 支援大部份的 TR1 功能。
- Boost 提供了一個相容層，會在編譯器不支援時自動回到 Boost 函式庫實作。
- Technical Report 2 (TR2) 包含了 TR1 所沒有的其他建議函式庫。
- 無序關聯式容器函式庫定義四種新的容器：unordered_set、unordered_map、unordered_multiset 和 unordered_multimap。這些關聯式容器是實作成雜湊表 (hash tables)，定義在 <unordered_set> 和 <unordered_map> 標頭檔中。
- unordered_set 和 unordered_multiset 使用元素本身作爲鍵值。unordered_map 和 unordered_multimap 則儲存鍵值對。

- unordered_set 和 unordered_map 要求所有的鍵值必須是唯一的，unordered_multiset 和 unordered_multimap 則沒有此限制。
- TR1 提供了三角、雙曲線、指數、對數、乘冪和其他的特殊函式。
- TR1 含括 C99 的標頭，以增加 C++ 和 C99 的相容性。

23.8 C++0x

- 新的 C++標準 － C++0x 中包含了哪些核心語言以及標準函式庫的改變。
- 國際標準化組織 (International Organization for Standardization，ISO) 監督國際程式語言標準的制定。ISO Working Group 21 負責維護 C++ 標準。
- 新標準的主要目標是要讓 C++更容易學習，改善函式庫建構能力，並增加與 C 程式語言的相容性。

23.9 核心語言的改變

- GNU C++ 編譯器有 C++0x 模式可供選擇，讓你能夠試驗許多核心語言的改變。Visual Studio 2010 亦支援某些 C++0x 的功能。
- C++0x 的 rvalue 參照型別允許你將 rvalue (臨時物件) 連結到一個非 const 參照。
- rvalue 參照宣告爲 T&& (T 是被參照的物件型別) 。
- rvalue 參照可以有效地實作搬移語意。
- static_assert 宣告讓你可以在編譯時期測試程式的某些部分。
- static_assert 宣告會接收一個常數整數運算式以及一個字串。假如運算式的值等於 0 (false)，則編譯器會利用宣告中提供的字串發出錯誤訊息。
- C++0x 加入了 1999 C 標準中新增的許多變更，包括了前置處理器、新增 long long 整數型別以及新增整數型別 (例如 128 位元的整數型別) 的擴充原則。讓新的 C 程式能如同 C++ 一般地正確編譯。
- 建構式可以直接呼叫另一個類別的建構子。
- C++ 編譯器會將>>辨識爲樣板的一部分。
- 關鍵字 auto 會依照變數的初始值運算式，自動決定變數型別。auto 可以取代完整的型別名稱。
- variadic 樣板會接受任意數目的引數。
- Variadic 樣版可以簡化此類函式庫 (像是 Boost.Bind、Boost.Tuple 以及 Boost.Function) 的實作。
- 跟 typedef 不同，當我們使用樣板別名時，可以固定某些參數，並讓其他參數保持其可變動性。

- 在 C++0x 中，類別可以讓建構子接收型別爲 `std::initializer_list <T>` 的參數。這樣就可以使用初始值列表來初始化某一類別的物件。
- 基於範圍的 `for` 敘述式 (range-based for statement) 讓你使用相同的語法來走訪陣列和容器。
- Lambda 運算式 (或 lambda 函式) 替函式的定義提供一種簡化的語法，就是在使用函式物件的同時定義它們。
- lambda 函式可以「抓到」(以傳值或傳參考的方式) 區域變數，在 lambda 本體內處理這些數值。
- Lambda 運算式以 `[]`(lambda introducer) 開始，後面接著參數列和函式本體。假如函式本體是 `return` expression；型態的單一敘述式，就可以直接推論出回傳型別，否則回傳型別預設爲 `void`。
- 在 lambda introducer (`[]`) 中標示出區域變數，以抓取它們。利用 & 以傳參考方式抓取區域變數。
- 概念 (Concepts) 提供了一個方法來解決編譯樣板函式庫時會碰到的錯誤訊息問題。每個樣板會附加一個概念定義，指定樣板的需求規格。假如樣板使用的型別與 concept 所指定的需求不符時，編譯器會發出適當的訊息。

術語

`*` 量化詞 (零個以上) [`*` quantifier (0 or more)]
`+` 量化詞 (一個以上) [+quantifier (1 or more)]
`?` 量化詞 (零或一個) [`?` quantifier (0 or more)]
$ 字串結尾 ($ end of a string)
`^` 字串開頭 (`^` beginning of a string)
`\d` 字元類別 (任意十進位數字) [`\d` character class (any decimal digit)]
`\w` 字元類別 (任意文字字元) [`\w` character class (any word character)]
`\s` 字元類別 (任意空白字元) [`\s` character class (any whitespace character)]
`\D` 字元類別 (任意非數字) [`\D` character class (any non-digit)]
`\W` 字元類別 (任意非文字字元) [`\W` character class (any non-word character)]
`\S` 字元類別 (任意非空白字元) [`\S` character class (any non-whitespace character)]
`{n,}`量化詞 (至少出現 n 次) [`{n,}` quantifier (at least n)]
`{n,m}`量化詞 (n 到 m 次之間)[quantifier (between n and m)]
`{n}`量化詞 (正好 n 次)[quantifier (exactly n)]
`auto` 關鍵字 (auto keyword)
`auto_ptr`
`bad_weak_ptr` 例外 (bad_weak_ptr exception)
`basic_regex` 類別 (basic_regex class)
`bind1st` 函式 (bind1st function)
`bind2nd` 函式 (bind2nd function)
Boost C++ 函式庫 (Boost C++ libraries)
`boost::bad_weak_ptr`
`boost::format_first_only`

Boost.Array 函式庫 (Boost.Array library)
Boost.Bind 函式庫 (Boost.Bind library)
Boost.Function 函式庫 (Boost.Function library)
Boost.Random 函式庫 (Boost.Random library)
Boost.Regex 函式庫 (Boost.Regex library)
Boost.Smart_ptr 函式庫 (Boost.Smart_ptr library)
Boost.Tuple 函式庫 (Boost.Tuple library)
Boost.Type_traits 函式庫 (Boost.Type_traits library)
boost 命名空間 (boost namespace)
Boost Sandbox
Boost 軟體授權條款 (Boost Software License)
桶 (雜湊表)[bucket (hash table)]
C++0x
C99
字元類別 (character class)
循環參照資料 (circularly referential data)
概念 (concepts)
委派建構子 (delegating constructors)
deleter 函式 (shared_ptr) (deleter function (shared_ptr))
位元 (digit)
format_first_only
greedy quantifier (貪婪的量化詞)
雜湊函式 (hash function)
雜湊表 (hash table)
initializer_list 類別樣板 (initializer_list class template)
國際標準化組織 (International Organization for Standardization，ISO)
鍵 (雜湊表)key (hash table)]
lambda 運算式 (lambda expression)
lambda 函式 (lambda function)
lazy quantifier (懶惰的量化詞)
weak_ptr 類別的 lock 成員函式 (lock member function of class weak_ptr)
match_not_dot_newline
match_results 類別 (match_results class)
非確定性亂數 (nondeterministic random numbers)
虛擬亂數產生器 (pseudo-random number generator)
量化詞 (quantifier)
亂數分布 (random number distribution)
亂數產生器 (random number generator)
基於範圍的 for 敘述式 (range-based for statement)
參照計數 (reference count)
參照計數 (reference counting)
regex typedef
regex_match 演算法 (regex_match algorithm)
regex_replace 演算法 (regex_replace algorithm)
regex_search 演算法 (regex_search algorithm)
regex_token_iterator 類別樣板 (regex_token_iterator class template)
正規表示式 (regular expression)
shared_ptr 類別的成員函式 reset (reset member function of class shared_ptr)
rvalue 參照 (rvalue reference)
shared_ptr 類別 (shared_ptr class)
聰明指標 (smart pointer)
smatch typedef
static_assert 宣告 (static_assert declaration)
match_results 類別的 suffix 成員函式 (suffix member function of class match_results)
Technical Report 1 (TR1)
Technical Report 2 (TR2)
樣板別名 (template alias)
元組 (tuple)
均勻分佈 (uniform distribution)
unordered_map 容器 (unordered_map container)
unordered_multimap 容器 (unordered_multimap container)

unordered_multiset 容器 (unordered_multiset container)
unordered_set 容器 (unordered_set container)
shared_ptr 類別的 use_count 成員函式 (use_count member function of class shared_ptr)
可變參數樣板 (variadic template)
weak_ptr 類別樣板 (weak_ptr class template)
空白字元 (whitespace character)
word character (文字字元)

自我測驗

23.1 請填入下列敘述中的空白：

a)_____是對 C++ 標準函式庫的變更建議。
b) 使用_____函式庫可以管理動態配置記憶體的釋放以避免記憶體遺漏。
c) Boost.Bind 擴充標準函式庫函式_____和_____的功能。
d) shared_ptr 利用_____幫助你決定何時刪除資源。
e) _____類別代表 Boost.Regex 中的正規表示式。
f) regex_token_iterator 類別位在命名空間_____。
g) Boost.Regex 的演算法_____會將指定樣式所出現的每一個地方都替換成另一個字串。
h) 正規表示式量化詞_____表示該樣式出現零次以上。
i) 將正規表示式運算子_____放在方括號內，表示任何位於括號內的字元都不符合。
j) C++0x 的關鍵字_____會在變數初始化時，自動決定其型別。
k) C++0x 使用_____來實現搬移語意和轉呼叫函式。

23.2 說明下列何者爲對，何者爲錯。如果答案是錯，請解釋爲什麼。

a) auto_ptr 可以用在 STL 容器。
b) 建立指向資源的 weak_ptr 會遞增參照計數。
c) 正規表示式會比對字串與樣式是否相符。
d) 正規表示式中的\d 代表所有的字母。

23.3 試撰寫敘述式完成下列的工作：

a) 建立一個正規表示式，比對具有五個字母的單字，或是五位數的數字。
b) 建立一個正規表示式，比對格式爲 (123) 456-7890 的電話號碼。
c) 建立一個名爲 intPtr 的 shared_ptr，指向 int 5。
d) 建立一個名爲 weakIntPtr 的 weak_ptr，指向 intPtr。
e) 使用 weakIntPtr 存取 int 的值。

自我測驗解答

23.1 a) TR1。b) `Boost.Smart_ptr`。c) `bind1st`，`bind2nd`。d) 參照計數。e) `regex` 或 `__basic_regex`。f) `boost`。g) `regex_replace`。h) `*`。i) `^`。j) `auto`。k) `rvalue` 參照。

23.2 a) 錯。`auto_ptr` 無法被安全地複製，因此無法使用在 STL 容器中。b) 錯。`weak_ptr` 認爲它不具有資源的所有權，因此不會影響參照計數。c) 對。d) 錯。正規表示式中的運算式`\d` 代表所有的十進位數字。

23.3 a)
```
boost::regex( "\\w{5}|\\d{5}" );
```
b)
```
boost::regex( "\\(\\d{3}\\)\\s\\d{3}-\\d{4}" );
```
c)
```
boost::shared_ptr< int > intPtr( new int( 5 ) );
```
d)
```
boost::weak_ptr< int > weakIntPtr( intPtr );
```
e)
```
boost::shared_ptr< int > sharedIntPtr = weakIntPtr.lock();
*sharedIntPtr;
```

習題

23.4 (Pig Latin) 請寫出一個可以將英文轉換成 pig Latin 的程式。Pig Latin 是一種編碼語言，常在娛樂用途出現。有許多不同的方法可產生 Pig Latin 片語。爲簡單起見，請使用以下的演算法：

要把單獨的英文字翻譯成 pig-Latin 文字，請將每個英文字的第一個字母移到字的末端，然後加上字母 `"ay"`。因此，`"jump"` 會變成 `"umpjay"`，`"the"` 變成 `"hetay"`，且 `"computer"` 變成 `"omputercay"`。字與字之間依然保持空白。讓我們假設：英文片語是由空白隔開的字組成，且沒有標點符號，所有的字都有二個或更多的字母。讓使用者能夠輸入句子。使用一個 `regex_token_iterator` 將一個句子切割成多個單字。使用 `getPigLatin` 函式將單字轉換成 `Pig Latin`。

23.5 (使用正規表示式將字母轉換為大寫) 撰寫一個程式，使用正規表示式，將所有單字的第一個字母轉換成大寫。讓程式可以處理使用者輸入的任意字串。

23.6 (使用正規表示式轉換字元型別) 使用正規表示式，計算數字的位數，以及字串中字元和空白的數量。

23.7 (搜尋數字) 寫一個正規表示式，搜尋字串並比對出合法的數字。這個數字可以具有任意位數，但是只能包含數字和一個小數點。小數點是可有可無的，但是假如該數字具有小數，應該只有一個小數點，而且它的左邊和右邊都應該要有數字。合法的數字中間不應該出現空白字元，兩邊不應該出現行頭或行尾字元。負數的前面會加上負號。

23.8 (計算 HTML 標籤) 撰寫一個程式，接受一個 HTML 作爲輸入值，並輸出字串中的 HTML tag 數量。程式應該使用正規表示式來計算每一層的元素數量。例如，下面的 HTML

```
<p><strong>hi</strong></p>
```

有一個 p 元素(第 0 層，也就是說，它的外爲沒有其他 tag)，和一個 strong 元素(第 1 層) 。爲了簡化這個習題，我們使用的 HTML 不具有同型別的巢狀元素，例如，table 元素內部不應該包含另一個 table 元素。

本習題需要使用一個正規表示式的概念，稱爲向後參照 (back reference)，用來決定 HTML 元素中的開始和結束 tag。要找到這些 tag，同樣的單字必須出現在開始和結束 tag 中。向後參照讓你可以將前一個符合的字串用在正規表示式的任何部分。當你把正規表示式的一部分用小括號括起來時，就會替你儲存符合的子運算式。你可以使用文法 \digit 來存取該表示式的結果，其中 digit 是範圍 1-9 的整數。例如，底下的正規表示式

```
^(7*).*\1$
```

會尋找以一個以上的 7 開始或結尾的字串。"777abcd777" 和 "7abcdef7" 都符合這個正規表示式。表示式中的\1 是一個向後參照，表示只要有字串符合子運算式 (7*)，就應該出現在字串的結尾。第一個小括號內的子運算式使用\1 做爲向後參照，第二個子運算式使用\2 做爲向後參照，依此類推。

你會需要一個遞迴函式來處理巢狀的 HTML 元素。將元素的內容以 string 傳給遞迴函式，例如本範例中，p 元素的內容爲

```
<strong>hi</strong>
```

使用小括號來儲存開始和結尾 tag 之間符合正規表示式的內容。將這個值存在 match_results 物件中，在物件上使用[]運算子來存取該值。.利用向後參照，符合的子運算式索引應該標記爲 1-9。

23.9 (移除多餘的空白) 寫一個程式，要求使用者輸入一個句子，使用正規表示式來判斷該句的單字中間是否包含多於一個的空白。假如是，程式應該刪除這些額外的空白。例如，"Hello World" 應該變成 "Hello World"。

23.10 回答下列有關聰明指標的問題：

a) 簡單敘述 shared_ptr 比 auto_ptr 好的優點。

b) 請描述自訂的 deleter 適用於何種狀況。

c) 描述一種你想使用 weak_ptr 卻不負責管理其資源的狀況。