

C++ 標準程式庫

C++ Standard Library

教本與手冊 (A Tutorial and Handbook)

Nicolai M. Josuttis

侯捷 / 孟岩 合譯

— |

| —

— |

| —

目 錄

| | |
|---|-----------|
| 前言 | xvii |
| 致謝 (Acknowledgments) | xix |
| 1 關於本書 | 1 |
| 1.1 緣起 | 1 |
| 1.2 閱讀前的必要基礎 | 2 |
| 1.3 本書風格與結構 | 2 |
| 1.4 如何閱讀本書 | 4 |
| 1.5 目前情勢 | 5 |
| 1.6 範例程式碼及額外資訊 | 5 |
| 1.7 回應 | 5 |
| 2 C++ 及其標準程式庫簡介 | 7 |
| 2.1 沿革 | 7 |
| 2.2 新的語言特性 | 9 |
| 2.2.1 Templates (模板) | 9 |
| 2.2.2 基本型別的顯式初始化 | 14 |
| 2.2.3 異常處理 (Exception Handling) | 15 |
| 2.2.4 命名空間 (Namespaces) | 16 |
| 2.2.5 bool 型別 | 18 |
| 2.2.6 關鍵字 explicit | 18 |
| 2.2.7 新的型別轉換運算子 | 19 |
| 2.2.8 常數靜態成員 (Constant Static Members) 的初始化 | 20 |
| 2.2.9 main() 的定義 | 21 |
| 2.3 複雜度與 Big-O 表示法 | 21 |
| 3 一般概念 | 23 |

| | |
|--|-----------|
| 3.1 命名空間 <code>std</code> | 23 |
| 3.2 表頭檔 (Header Files) | 24 |
| 3.3 錯誤處理和異常處理 (Error and Exception Handling) | 25 |
| 3.3.1 標準異常類別 (Standard Exception Classes) | 25 |
| 3.3.2 異常類別 (Exception Classes) 的成員 | 28 |
| 3.3.3 丟擲標準異常 (Throwing Standard Exceptions) | 29 |
| 3.3.4 從標準異常類別 (Exception Classes) 中衍生新的類別 | 30 |
| 3.4 配置器 (Allocators) | 31 |
| 4 通用工具 (Utilities) | 33 |
| 4.1 Pairs | 33 |
| 4.1.1 便捷函式 <code>make_pair()</code> | 36 |
| 4.1.2 <code>Pair</code> 的使用實例 | 37 |
| 4.2 Class <code>auto_ptr</code> | 38 |
| 4.2.1 <code>auto_ptr</code> 的發展動機 | 38 |
| 4.2.2 <code>auto_ptr</code> 的擁有權 (Ownership) 轉移 | 40 |
| 4.2.3 <code>auto_ptr</code> s 做為成員之一 | 44 |
| 4.2.4 <code>auto_ptr</code> s 的錯誤運用 | 46 |
| 4.2.5 <code>auto_ptr</code> 運用實例 | 47 |
| 4.2.6 <code>auto_ptr</code> 實作細目 | 51 |
| 4.3 數值極限 (Numeric Limits) | 59 |
| 4.4 輔助函式 | 66 |
| 4.4.1 處理最小值和最大值 | 66 |
| 4.4.2 兩值互換 | 67 |
| 4.5 輔助的比較運算子 (Comparison Operators) | 69 |
| 4.6 表頭檔 <code><cstdint></code> 和 <code><cstdintlib></code> | 71 |
| 4.6.1 <code><cstdint></code> 內的各種定義 | 71 |
| 4.6.2 <code><cstdintlib></code> 內的各種定義 | 71 |
| 5 Standard Template Library (標準模板程式庫) | 73 |
| 5.1 STL 組件 | 73 |
| 5.2 容器 (Containers) | 75 |
| 5.2.1 序列式容器 (Sequence Containers) | 76 |
| 5.2.2 關聯式容器 (Associative Containers) | 81 |
| 5.2.3 容器配接器 (Container Adapters) | 82 |

| | |
|---|------------|
| 5.3 迭代器 (Iterators) | 83 |
| 5.3.1 關聯式容器的運用實例 | 86 |
| 5.3.2 迭代器類型 (Iterator Categories) | 93 |
| 5.4 演算法 (Algorithms) | 94 |
| 5.4.1 區間 (Ranges) | 97 |
| 5.4.2 處理多個區間 | 101 |
| 5.5 迭代器 之 配接器 (IteratorAdapters) | 104 |
| 5.5.1 Insert Iterators (安插型迭代器) | 104 |
| 5.5.2 Stream Iterators (串流迭代器) | 107 |
| 5.5.3 Reverse Iterators (逆向迭代器) | 109 |
| 5.6 更易型演算法 (Manipulating Algorithms) | 111 |
| 5.6.1 移除 (Removing) 元素 | 111 |
| 5.6.2 更易型演算法和關聯式容器 | 115 |
| 5.6.3 演算法 v.s. 成員函式 | 116 |
| 5.7 使用者自定之泛型函式 (User-Defined Generic Functions) | 117 |
| 5.8 以函式做為演算法的引數 | 119 |
| 5.8.1 實例示範「以函式做為演算法的引數」 | 119 |
| 5.8.2 條件式 (Predicates) | 121 |
| 5.9 仿函式 (Functors or Function Objects) | 124 |
| 5.9.1 什麼是仿函式 | 124 |
| 5.9.2 預先定義好的仿函式 | 131 |
| 5.10 容器內的元素 (Container Elements) | 134 |
| 5.10.1 容器元素的條件 | 134 |
| 5.10.2 實值語意 或 參照語意 (Value Semantics or Reference Semantics) | 135 |
| 5.11 STL 內的錯誤處理和異常處理 | 136 |
| 5.11.1 錯誤處理 (Error Handling) | 137 |
| 5.11.2 異常處理 (Exception Handling) | 139 |
| 5.12 擴充 STL | 141 |
| 6 STL 容器 (Containers) | 143 |
| 6.1 共通的容器能力與操作 (Abilities and Operations) | 144 |
| 6.1.1 共通的容器能力 | 144 |
| 6.1.2 共通的容器操作 | 144 |
| 6.2 Vectors | 148 |
| 6.2.1 Vectors 的能力 | 148 |
| 6.2.2 Vector 的操作 | 150 |

| | |
|---|-----|
| 6.2.3 把 Vectors 當做一般的 Arrays 使用 | 155 |
| 6.2.4 異常處理 | 155 |
| 6.2.5 Vectors 運用實例 | 156 |
| 6.2.6 Class <code>vector<bool></code> | 158 |
| 6.3 Deques | 160 |
| 6.3.1 Deques 的能力 | 161 |
| 6.3.2 Deque 的操作 | 162 |
| 6.3.3 異常處理 | 164 |
| 6.3.4 Deques 運用實例 | 164 |
| 6.4 Lists | 166 |
| 6.4.1 Lists 的能力 | 166 |
| 6.4.2 List 的操作 | 167 |
| 6.4.3 異常處理 | 172 |
| 6.4.4 Lists 運用實例 | 172 |
| 6.5 Sets 和 Multisets | 175 |
| 6.5.1 Sets 和 Multisets 的能力 | 176 |
| 6.5.2 Set 和 Multiset 的操作 | 177 |
| 6.5.3 異常處理 | 185 |
| 6.5.4 Sets 和 Multisets 運用實例 | 186 |
| 6.5.5 在執行期指定排序準則 (Sorting Criterion) | 191 |
| 6.6 Maps 和 Multimaps | 194 |
| 6.6.1 Maps 和 Multimaps 的能力 | 195 |
| 6.6.2 Map 和 Multimap 的操作 | 196 |
| 6.6.3 把 Maps 視為關聯式 Arrays | 205 |
| 6.6.4 異常處理 | 207 |
| 6.6.5 Maps 和 Multimaps 運用實例 | 207 |
| 6.6.6 綜合實例：運用 Maps, Strings, 並於執行期指定排序準則 | 213 |
| 6.7 其他的 STL 容器 | 217 |
| 6.7.1 Strings 可被視為一種 STL 容器 | 217 |
| 6.7.2 Arrays 可被視為一種 STL 容器 | 218 |
| 6.7.3 Hash Tables | 221 |
| 6.8 動手實現參照語意 (Reference Semantics) | 222 |
| 6.9 不同容器的運用時機 | 226 |
| 6.10 細說容器內的型別和成員 | 230 |
| 6.10.1 容器內的型別 | 230 |

| | |
|---|------------|
| 6.10.2 產生 (Create)、複製 (Copy)、摧毀 (Destroy) | 231 |
| 6.10.3 「不會更改元素內容」的操作行為 | 233 |
| 6.10.4 賦值 (Assignments) | 236 |
| 6.10.5 直接元素存取 | 237 |
| 6.10.6 「會產出迭代器」的操作行為 | 239 |
| 6.10.7 安插 (Inserting) 和移除 (Removing) 元素 | 240 |
| 6.10.8 Lists 內特殊的成員函式 | 244 |
| 6.10.9 配置器 (Allocator) 的支援 | 246 |
| 6.10.10 綜觀 STL 容器內的異常處理 | 248 |
| 7 STL 迭代器 (Iterators) | 251 |
| 7.1 迭代器的表頭檔 | 251 |
| 7.2 迭代器類型 | 251 |
| 7.2.1 Input Iterators (輸入迭代器) | 252 |
| 7.2.2 Output Iterators (輸出迭代器) | 253 |
| 7.2.3 Forward Iterators (前行迭代器) | 254 |
| 7.2.4 Bidirectional Iterators (雙向迭代器) | 255 |
| 7.2.5 Random Access Iterators (隨機存取迭代器) | 255 |
| 7.2.6 Vector 迭代器的前進和後退問題 | 258 |
| 7.3 輔助用的迭代器相關函式 | 259 |
| 7.3.1 advance() 可令迭代器前進 | 259 |
| 7.3.2 distance() 可處理迭代器之間的距離 | 261 |
| 7.3.3 iter_swap() 可令兩個迭代器內容互換 | 263 |
| 7.4 迭代器配接器 (Iterator Adapters) | 264 |
| 7.4.1 Reverse Iterators (逆向迭代器) | 264 |
| 7.4.2 Insert Iterators (安插式迭代器) | 271 |
| 7.4.3 Stream Iterators (串流迭代器) | 277 |
| 7.5 迭代器特性 (Iterator Traits) | 283 |
| 7.5.1 為迭代器撰寫泛型函式 (Generic Functions) | 285 |
| 7.5.2 使用者自定的迭代器 | 288 |
| 8 STL 仿函式 (Functors or Function Objects) | 293 |
| 8.1 仿函式的概念 | 293 |
| 8.1.1 仿函式可當做排序準則 | 294 |
| 8.1.2 仿函式可擁有自己的內部狀態 (Internal State) | 296 |
| 8.1.3 for_each() 的回傳值 | 300 |
| 8.1.4 條件式 (Predicates) 與仿函式 (Functors) | 302 |

| | |
|--|------------|
| 8.2 預先定義好的仿函式 | 305 |
| 8.2.1 函式配接器 (Function Adapters) | 306 |
| 8.2.2 針對成員函式而設計的函式配接器 | 307 |
| 8.2.3 針對一般函式而設計的函式配接器 | 309 |
| 8.2.4 讓你自定的仿函式也可使用函式配接器 | 310 |
| 8.3 輔助用 (組合型) 的仿函式 | 313 |
| 8.3.1 一元組合仿函式配接器 (Unary Compose Function Object Adapters) | 314 |
| 8.3.2 二元組合仿函式配接器 (Binary Compose Function Object Adapters) | 318 |
| 9 STL 演算法 (Algorithms) | 321 |
| 9.1 演算法表頭檔 | 321 |
| 9.2 演算法概觀 | 322 |
| 9.2.1 簡介 | 322 |
| 9.2.2 演算法分門別類 | 323 |
| 9.3 輔助函式 | 332 |
| 9.4 <code>for_each()</code> 演算法 | 334 |
| 9.5 「非修改型」演算法 (Nonmodifying Algorithms) | 338 |
| 9.5.1 計算元素個數 | 338 |
| 9.5.2 求最大值和最小值 | 339 |
| 9.5.3 搜尋元素 | 341 |
| 9.5.4 比較兩個區間的異同 | 356 |
| 9.6 「修改型」演算法 (Modifying Algorithms) | 363 |
| 9.6.1 拷貝元素 | 363 |
| 9.6.2 轉換元素及組合元素 | 366 |
| 9.6.3 互換元素內容 | 370 |
| 9.6.4 賦予新值 | 372 |
| 9.6.5 取代元素 | 375 |
| 9.7 「移除 (Removing)」演算法 | 378 |
| 9.7.1 移除某些元素 | 378 |
| 9.7.2 移除重複元素 | 381 |
| 9.8 「變化型」演算法 (Mutating Algorithms) | 386 |
| 9.8.1 逆轉所有元素的次序 | 386 |
| 9.8.2 旋轉 (Rotating) 元素 | 388 |
| 9.8.3 排列 (Permuting) 元素 | 391 |
| 9.8.4 重排 (Shuffling) 元素 | 393 |
| 9.8.5 將元素往前搬移 | 395 |

| | |
|---|------------|
| 9.9 排序 (Sorting) 演算法 | 397 |
| 9.9.1 將所有元素排序 | 397 |
| 9.9.2 局部排序 (Partial Sorting) | 400 |
| 9.9.3 根據第 <i>n</i> 個元素進行排序動作 | 404 |
| 9.9.4 Heap 演算法 | 406 |
| 9.10 適用於「已序區間 (Sorted Range)」的演算法 | 409 |
| 9.10.1 搜尋元素 | 410 |
| 9.10.2 合併元素 | 416 |
| 9.11 數值演算法 (Numeric Algorithms) | 425 |
| 9.11.1 加工運算後產生結果 | 425 |
| 9.11.2 相對關係與絕對關係之間的轉換 | 429 |
| 10 特殊容器 | 435 |
| 10.1 Stacks | 435 |
| 10.1.1 核心介面 | 436 |
| 10.1.2 Stacks 運用實例 | 437 |
| 10.1.3 Class <code>stack<></code> 細部討論 | 438 |
| 10.1.4 一個使用者自定的 Stack Class | 441 |
| 10.2 Queues | 444 |
| 10.2.1 核心介面 | 445 |
| 10.2.2 Queues 運用實例 | 446 |
| 10.2.3 Class <code>queue<></code> 細部討論 | 447 |
| 10.2.4 一個使用者自定的 Queue Class | 450 |
| 10.3 PriorityQueues | 453 |
| 10.3.1 核心介面 | 455 |
| 10.3.2 PriorityQueues 運用實例 | 455 |
| 10.3.3 Class <code>priority_queue<></code> 細部討論 | 456 |
| 10.4 Bitsets | 460 |
| 10.4.1 Bitsets 運用實例 | 460 |
| 10.4.2 Class <code>bitset</code> 細部討論 | 463 |
| 11 Strings (字串) | 471 |
| 11.1 動機 | 471 |
| 11.1.1 第一個例子：產生一個暫時檔名 | 472 |
| 11.1.2 第二個例子：產生一段文字，並反向列印 | 476 |
| 11.2 String Classes 細部描述 | 479 |
| 11.2.1 String 的各種相關型別 | 479 |

| | |
|---|------------|
| 11.2.2 操作行為綜觀 | 481 |
| 11.2.3 建構式和解構式 (Constructors and Destructors) | 483 |
| 11.2.4 Strings 和 C-Strings | 484 |
| 11.2.5 大小和容量 | 485 |
| 11.2.6 元素的存取 | 487 |
| 11.2.7 比較 | 488 |
| 11.2.8 更改內容 | 489 |
| 11.2.9 子字串及字串接合 | 492 |
| 11.2.10 輸入/輸出 運算子 (Input/Output Operators) | 492 |
| 11.2.11 搜尋 (Searching and Finding) | 493 |
| 11.2.12 數值 npos 的意義 | 495 |
| 11.2.13 迭代器對 Strings 的支援 | 497 |
| 11.2.14 國際化 | 503 |
| 11.2.15 效率 | 506 |
| 11.2.16 Strings 和 Vectors | 506 |
| 11.3 細說 String Class | 507 |
| 11.3.1 內部的型別定義和靜態值 | 507 |
| 11.3.2 產生 (Create)、拷貝 (Copy)、摧毀 (Destroy) | 508 |
| 11.3.3 大小和容量 | 510 |
| 11.3.4 比較 | 511 |
| 11.3.5 字元的存取 (Character Access) | 512 |
| 11.3.6 產出 C-Strings 和 Character Arrays (字元陣列) | 513 |
| 11.3.7 更改內容 | 514 |
| 11.3.8 搜尋 (Searching and Finding) | 520 |
| 11.3.9 子字串及字串接合 | 524 |
| 11.3.10 輸入/輸出函式 | 524 |
| 11.3.11 產出迭代器 | 525 |
| 11.3.12 對配置器的支援 | 526 |
| 12 數值 (Numerics) | 529 |
| 12.1 複數 (Complex Numbers) | 529 |
| 12.1.1 Class Complex 運用實例 | 530 |
| 12.1.2 複數的操作 | 533 |
| 12.1.3 Class complex<> 細部討論 | 541 |
| 12.2 Valarrays | 547 |
| 12.2.1 認識 Valarrays | 547 |

| | |
|--|------------|
| 12.2.2 Valarray 的元素子集 | 553 |
| 12.2.3 Class valarray 細部討論 | 569 |
| 12.2.4 Valarray 子集類別 (Subset Classes) 細部討論 | 575 |
| 12.3 全域性的數值函式 | 581 |
| 13 以 Stream Classes 完成輸入/輸出 | 583 |
| 13.1 I/O Streams 基本概念 | 584 |
| 13.1.1 Stream Objects (串流物件) | 584 |
| 13.1.2 Stream Classes (串流類別) | 584 |
| 13.1.3 Global Stream Objects (全域串流物件) | 585 |
| 13.1.4 Stream Operators (串流運算子) | 586 |
| 13.1.5 Manipulators (操控器) | 586 |
| 13.1.6 一個簡單的例子 | 587 |
| 13.2 基本的 Stream Classes 和 Objects | 588 |
| 13.2.1 Classes 和 Class 階層體系 | 588 |
| 13.2.2 全域性的 Stream Objects | 591 |
| 13.2.3 表頭檔 | 592 |
| 13.3 標準的串流運算子 (Stream Operators) << 和 >> | 593 |
| 13.3.1 輸出運算子 << | 593 |
| 13.3.2 輸入運算子 >> | 594 |
| 13.3.3 特殊型別的輸入/輸出 | 595 |
| 13.4 Streams 的狀態 | 597 |
| 13.4.1 用來表示 Streams 狀態的一些常數 | 597 |
| 13.4.2 用來處理 Streams 狀態的一些成員函式 | 598 |
| 13.4.3 Stream 狀態與布林條件測試 | 600 |
| 13.4.4 Stream 狀態和異常 | 602 |
| 13.5 標準的輸入/輸出函式 | 607 |
| 13.5.1 輸入用的成員函式 | 607 |
| 13.5.2 輸出用的成員函式 | 610 |
| 13.5.3 運用實例 | 611 |
| 13.6 操控器 (Manipulators) | 612 |
| 13.6.1 操控器如何運作 | 612 |
| 13.6.2 使用者自定操控器 | 614 |
| 13.7 格式化 | 615 |
| 13.7.1 格式旗標 (Format Flags) | 615 |
| 13.7.2 布林值 (Boolean Values) 的輸入/輸出格式 | 617 |

| | |
|--|------------|
| 13.7.3 欄位寬度、填充字元、位置調整 | 618 |
| 13.7.4 正符號與大寫字 | 620 |
| 13.7.5 數值進制 (Numeric Base) | 621 |
| 13.7.6 浮點數 (Floating-Point) 表示法 | 623 |
| 13.7.7 一般性的格式定義 | 625 |
| 13.8 國際化 (Internationalization) | 625 |
| 13.9 檔案存取 | 627 |
| 13.9.1 檔案旗標 (File Flags) | 631 |
| 13.9.2 隨機存取 | 634 |
| 13.9.3 使用檔案描述器 (File Descriptors) | 637 |
| 13.10 連接輸入串流和輸出串流 | 637 |
| 13.10.1 以 <code>tie()</code> 解放耦合關係 | 637 |
| 13.10.2 以 <code>Stream Buffers</code> 繃緊耦合關係 | 638 |
| 13.10.3 將標準的 <code>Streams</code> 重新導向 | 641 |
| 13.10.4 用於讀入和寫出的 <code>Streams</code> | 643 |
| 13.11 應用於 <code>Strings</code> 身上的 <code>Stream Classes</code> | 645 |
| 13.11.1 <code>String Stream Classes</code> | 645 |
| 13.11.2 <code>char* Stream Classes</code> | 649 |
| 13.12 「使用者自定型別」之輸入/輸出運算子 | 652 |
| 13.12.1 實作一個輸出運算子 | 652 |
| 13.12.2 實作一個輸入運算子 | 654 |
| 13.12.3 以輔助函式完成輸入/輸出 | 656 |
| 13.12.4 以 <code>Unformatted Functions</code> 完成使用者自定運算子 | 658 |
| 13.12.5 使用者自定的格式旗標 (Format Flags) | 659 |
| 13.12.6 使用者自定之輸入/輸出運算子的數個依循慣例 | 662 |
| 13.13 <code>Stream Buffer Classes</code> | 663 |
| 13.13.1 從使用者的角度看 <code>Stream Buffers</code> | 663 |
| 13.13.2 <code>Stream Buffer</code> 的迭代器 | 665 |
| 13.13.3 使用者自定的 <code>Stream Buffers</code> | 668 |
| 13.14 效率考量 | 681 |
| 13.14.1 與 C 標準串流 (Standard Streams) 同步 | 682 |
| 13.14.2 <code>Stream Buffers</code> 中的緩衝機制 | 682 |
| 13.14.3 直接使用 <code>Stream Buffers</code> | 683 |
| 14 國際化 | 685 |
| 14.1 不同的字元編碼 | 686 |

| | |
|--|------------|
| 14.1.1 寬字元 (Wide-Character) 和多位元組文字 (Multibyte Text) | 686 |
| 14.1.2 字元特性 (Character Traits) | 687 |
| 14.1.3 特殊字元的國際化 | 691 |
| 14.2 Locales (本土化) 的概念 | 692 |
| 14.2.1 使用 Locales | 693 |
| 14.2.2 Locale Facets | 698 |
| 14.3 Locales 細部討論 | 700 |
| 14.4 Facets 細部討論 | 704 |
| 14.4.1 數值格式化 | 705 |
| 14.4.2 時間和日期格式化 | 708 |
| 14.4.3 貨幣符號格式化 | 711 |
| 14.4.4 字元的分類與轉換 | 715 |
| 14.4.5 String 校準 (Collation) | 724 |
| 14.4.6 國際化的訊息 | 725 |
| 15 配置器 (Allocators) | 727 |
| 15.1 應用程式開發人員 (Application Programmer) 如何使用配置器 | 727 |
| 15.2 程式庫開發人員 (Library Programmer) 如何使用配置器 | 728 |
| 15.3 預設的配置器 | 732 |
| 15.4 使用者自定的配置器 | 735 |
| 15.5 配置器細部討論 | 737 |
| 15.5.1 內部定義的型別 | 737 |
| 15.5.2 操作 | 739 |
| 15.6 可處理「未初始化記憶體」之公用工具 (Utilities) 細部討論 | 740 |
| 網際網路上的資源 (Internet Resources) | 743 |
| 參考書目 (Bibliography) | 745 |
| 索引 (Index) | 747 |

前言

Preface

一開始，我只不過想寫一本篇幅不大的有關於 C++標準程式庫的德文書（也就 400 多頁吧）。萌生這個想法是在 1993 年。而在 1999 年的今天，您看到了這個想法的結果：一本英文書，厚達 800 多頁，其中包含大量的文字描述、圖片和範例。我的目標是，詳盡講解 C++ 標準程式庫，使你的所有（或幾乎所有）編程問題都能夠在你遇到之前就先給你解答。然而，請注意，這不是一種完整描述 C++標準程式庫的所有面向的書籍，我透過「在 C++ 中利用其標準程式庫進行學習與編程」的形式，表現最重要的主題。

每一個主題都是以一般性概念為基礎而開展，然後導入日常程式設計工作所必須了解的具體細節。為了幫助你理解這些概念和細節，書中提供詳盡的範例程式。

這就是我的前言 — 言簡意賅！撰寫此書的過程中，我得到了很多樂趣，希望你閱讀本書時，能夠像我一樣快樂。請享用！

致謝

Acknowledgments

這本書表達的觀點、概念、解決方案和範例，來源十分廣泛。從這個意義上講，封面上只有我一個人的名字，未免不公平。所以我願在此向過去幾年來幫助和支持我的人和公司，表示誠摯的謝意。

我第一個要感謝的是 Dietmar Kühl。Dietmar 是一位 C++ 專家，尤其精通 I/O streams（資料串流）和國際化（他曾經僅僅爲了好玩而寫了一個 I/O stream library）。他不僅將本書的大部分從德文譯爲英文，而且還親自動筆，發揮專長，爲本書撰寫了數節內容。除此之外，在過去的數年裡，他向我提供了很多寶貴的回饋意見。

其次，我要感謝所有的檢閱者和那些向我表達過意見的人。他們的努力使本書的品質獲得巨大提昇。以下由於名單太長，所以如有任何疏漏，還請見諒。英文版的檢閱者包括 Chuck Allison, Greg Comeau, James A. Crotinger, Gabriel Dos Reis, Alan Ezust, Nathan Myers, Werner Mossner, Todd Veldhuizen, Chichiang Wan, Judy Ward, Thomas Wike-hult。德文版檢閱者包括 Ralf Boecker, Dirk Herrmann, Dietmar Kühl, Edda Lörke, Herbert Scheubner, Dominik Strasser, Martin Weitzel。其他投入者包括 Matt Austern, Valentin Bonnard, Greg Colvin, Beman Dawes, Bill Gibbons, Lois Goldthwaite, Andrew Koenig, Steve Rumsby, Bjarne Stroustrup, 和 David Vandevoorde。

我要特別感謝 Dave Abrahams, Janet Cocker, Catherine Ohala 和 Maureen Willard，他們對全書進行了非常細緻的檢閱和編輯。他們的回饋意見讓本書的品質獲得了難以置信的提昇。

我也要特別感謝我的「活字典」Herb Sutter，它是著名的 "Guru of the Week" 的創造者，這是一個常態性的 C++ 難題講解專欄，播出於 `comp.std.c++.moderated`。

我還要感謝一些公司和個人，他們的幫助使我有機會在各個不同的平台上，使用各種不同的編譯器，來測試自己的範例程式。非常感謝來自 EDG 的 Steve

Adamczyk, Mike Anderson 和 John Spicer, 他們的編譯器真是太棒了, 在 C++ 標準化過程和本書寫作過程中, 提供了巨大的支援。感謝 P. J. Plauger 和 Dinkumware, Ltd, 他們很早以來就持續進行與 C++ 標準規範相容的 C++ 標準程式庫實作工作。感謝 Andreas Hommel 和 Metrowerks, 他們完成了深具價值的 CodeWarrior 程式開發環境。感謝 GNU 和 egcs 編譯器的所有開發者。感謝 Microsoft, 他們完成了深具價值的 Visual C++。感謝 Siemens Nixdorf Informations Systems AG 的 Roland Hartinger, 他提供了他們的 C++ 編譯器產品的一份測試版本。感謝 Topjects GmbH, 因為他的一份深具價值的 ObjectSpace library 實作品。

感謝 Addison Wesley Longman 公司裡頭與我共同工作過的每一個人。包括 Janet Cocker, Mike Hendrickson, Debbie Lafferty, Marina Lang, Chanda Leary, Catherine Ohala, Marty Rabinowitz, Susanne Spitzer, 和 Maureen Willard 等等。這項工作真是太有趣了。

此外, 我還要感謝 BREDEX GmbH 的人們, 感謝 C++ 社群中的所有人, 特別是那些參與了標準化過程的人, 感謝他們的支持和耐心 (有時候我問的問題確實挺傻的)。

最後, 也是最重要的, 我要將我的感謝 (附上一個親吻) 送給我的家人: Ulli, Lucas, Anica, 和 Frederic。爲了這本書, 我很長時間沒有好好陪他們了。

但願各位能從這本書獲得樂趣, 另外, 請保持寬厚。

1

關於本書

1.1 緣起

C++ 問世後不久，就成為物件導向程式設計領域的實質標準（*de facto standard*），因此正式標準化的呼聲也就浮上了檯面。一旦我們有了一個可以依循的標準規格，我們才可能寫出跨越 PC 乃至大型主機之間各種不同平台的程式。此外，如果能夠建立起一個標準程式庫，程式員便得以使用可移植的、通用的組件（*components*）和更高層次的抽象性質，而不必從頭創造世界。

C++ 的標準化過程始於 1989 年，由國際性的 ANSI/ISO 委員會負責。標準化工作係由 Bjarne Stroustrup 的兩本書 *The C++ Programming Language* 和 *The Annotated C++ Reference Manual* 為根基。這份標準規格於 1997 年通過後，又在數個國家進行了一些正式程序，最後於 1998 年成為國際性的 ISO/ANSI 標準。標準化過程本身包含一個任務：建立 C++ 標準程式庫。作為核心語言的拓展，標準程式庫提供了一些通用組件。藉由大量運用 C++ 新的抽象能力和泛型（*generic types*）特性，標準程式庫提供了一系列共同的類別和介面。程式員藉此獲得了更高層次的抽象能力。這個標準程式庫提供了以下組件：

- String 型別
- 各種資料結構（例如 *dynamic array*、*linked lists*、*binary trees*）
- 各種演算法（例如各種排序演算法）
- 數值類別（*numeric classes*）
- 輸入/輸出（*I/O*）類別
- 國際化支持類別

所有這些組件的介面都十分簡單。這些組件在很多程式中都很重要。如今的資料處理工作，通常就是意味著輸入、計算、處理和輸出大量的資料（通常是字串）。

這個標準程式庫的用法並非不言自明。想要從其強大能力中受惠，你需要一本好書，不能僅僅列出每一個類別和其函式了事，而是詳細解釋各組件的概念和重要細節。本書正是以此為目標。本書首先從概念上介紹標準程式庫及其所有組件，

The C++ Standard Library

然後描述實際編程所需了解的細節。爲了展示組件的確切用法，書中還包括了大量實例。因此，這本書不論對初學者或是編程老手，都是極爲詳盡的 C++ 標準程式庫文件。以本書所提供的資料來武裝自己的頭腦，你就能從 C++ 標準程式庫中獲得最大的利益。

注意，我不擔保本書所有的內容都容易理解。標準程式庫非常靈活，但這種非同尋常的靈活性是有代價的。標準程式庫中有一些陷阱和缺陷，你必須小心應對。碰到它們時我會爲你指出，並提出一些建議，幫助你迴避問題。

1.2 閱讀前的必要基礎

要想讀懂本書的大部分內容，你需要先了解 C++。本書講述 C++ 標準組件，而不是語言本身。你應該熟悉類別（classes）、繼承（inheritance）、模板（templates）和異常處理（exception handling）的概念，但不必熟知語言的每一個細節。某些重要的細節，本書也會講解（至於次要細節對程式庫實作者可能很重要，對程式庫使用者就不那麼重要了）。注意，在標準化過程中，這個語言發生了很大的變化，也許你的知識已經過時了。2.2 節簡單介紹了一些「使用標準程式庫時，你需要了解的最新語言特性」。如果你不確定自己是否了解 C++ 的新特性（例如關鍵字 `typename` 以及 `namespace` 的概念），請先閱讀該節。

1.3 本書風格與結構

標準程式庫內的組件有相當程度的獨立性，但彼此又存在關聯，所以很難在描述某一部分時全然不提其他部分。我爲這本書考慮了幾種不同的組織方式。一是按照 C++ *standard* 的次序來。不過這並非完整介紹 C++ 標準程式庫的最佳選擇。另一種方式是，首先縱覽所有組件，再逐章詳細介紹。第三個方式則是，由我依照「組件之交叉參考」程度高低，由最低者開始介紹，逐次介紹最複雜的組件。最終我綜合了三種方式：首先簡短介紹標準程式庫所涉及的總體概念和工具，然後分章詳述各個組件，每個組件一章或數章。首當其衝的便是 STL（Standard Template Library，標準模板庫）。STL 無疑是標準程式庫中最強大、最複雜、最激動人心的部分，其設計深刻影響了其他組件。然後我再講解較易理解的組件，例如特殊容器、strings 和數值類別。再來是你或許已經使用多時的老朋友：IOStream library。最後是國際化的討論，這部分對於 IOStream library 有些影響。

講述每個組件時，我首先給出該組件的目的、設計和範例，然後藉由各種使用方法和注意事項的描述，講解組件的細節。最後是一個參考章節（reference section），你可以在其中找到組件類別和其函式的確切型式（exact signature）。

以下是本書的內容。最前面的四章總體介紹了本書及 C++ 標準程式庫：

- 第 1 章：關於本書
這一章（也就是你此刻正閱讀的）介紹本書的主題和內容。
- 第 2 章：簡介 C++ 和其標準程式庫
這一章對於 C++ 標準程式庫的歷史和背景進行簡短綜覽。此章也包括了本書及標準程式庫的技術背景的大致介紹，例如新的語言特性和所謂複雜度概念。
- 第 3 章 一般性概念
本章描述標準函式庫的基本概念，這些概念對於你理解和使用函式庫中的所有組件都是必須的。明確地說，本章介紹了 `namespace std`、表頭檔格式、對於錯誤和異常處理（exception handling）的一般性支持。
- 第 4 章 公用設施（Utilities）
本章描述數種提供給程式庫用戶和程式庫本身運用的小工具，更明確地說是 `max()`、`min()`、`swap()` 輔助函式，以及 `pair`、`auto_ptr` 和 `numeric_limits` 型別。後者提供了數值型別實際產品的相關資訊。

第 5 章至第 9 章，從各個面向描述了 STL：

- 第 5 章：STL（Standard Template Library，標準模板庫）
STL 提供了用於處理資料的容器類別和演算法。本章詳細介紹 STL 的概念，並逐步解釋概念、問題、特殊編程技術，以及其中各部份的任務（角色扮演）。
- 第 6 章：STL 容器（Containers）
本章解釋 STL 容器類別的概念和能力。首先，通過詳盡的例子，分別講解 `vectors`、`deque`s、`lists`、`sets` 和 `maps`，然後介紹它們的共通能力。最後，以簡明的形式列出並描述所有容器的函式，做為一份上手的參考資料。
- 第 7 章：STL 迭代器（Iterators）
本章具體介紹了 STL 迭代器。解釋了迭代器的分類、輔助函式、以及配接器（iterator adapter），例如 `stream iterators`、`reverse iterators`、`insert iterators`。
- 第 8 章：STL 仿函式（Functors，又名 Function Objects）
本章詳細講解了 STL 仿函式。
- 第 9 章：STL 演算法（Algorithms）
本章羅列並描述 STL 演算法。在簡單介紹和比較這些演算法後，藉由一個（或多個）範例，對每個演算法進行詳細的描述。

第 10 章至 12 章描述了一些「簡單的」標準類別：

- 第 10 章：特殊容器
本章描述 C++ 標準程式庫的各種特殊容器，並涵蓋容器配接器（`container adapters`）`queues`, `stacks`，以及 `class bitset`，後者可管理任意數量的 bits 或 flags。
- 第 11 章：Strings（字串）
本章描述了 C++ 標準程式庫的 `string` 型別（不止一種哦）。C++ *standard* 將 strings 設計為一種能夠處理不同字元類型之基本資料型別，而且簡明易用。
- 第 12 章：數值（Numerics）
本章描述 C++ 標準程式庫中的數值組件，包括複數（`complex`），以及一些用來處理數值陣列的類別（可用於矩陣 `matrices`、向量 `vectors` 和方程式 `equations`）。

第 13 章和第 14 章的主題是 I/O 和國際化（兩者緊密相關）：

- 第 13 章：以 Stream Classes 完成輸入/輸出
本章涵蓋 C++ 的 I/O 組件。該組件是眾所周知的 `IOStream library` 的標準化形式。本章也講述了一些對程式員而言可能很重要、但又鮮為人知的細節。例如如何定義及整合特殊的 I/O 通道，這是一個在實務過程中經常被搞錯的題目。
- 第 14 章：國際化
本章涵蓋「將程式國際化」的概念和類別，包括對不同字元集（`character sets`）的處理，以及如何使用不同格式的浮點數和日期。

本書的剩餘部分包括：

- 第 15 章：配置器（`Allocators`）
本章描述 C++ 標準程式庫中不同的記憶體模式的概念。
- 附錄，包括：
 - 網際網路上的資源（`Internet Resources`）
 - 參考書目（`Bibliography`）
 - 索引（`Index`）

1.4 如何閱讀本書

本書既是介紹性的使用指南，又是 C++ 標準程式庫的結構化參考手冊。C++ 標準程式庫的各個組件在相當程度上是彼此獨立的，所以讀完第 2 章至第 4 章後，你可以按任意順序閱讀其他各章節。不過切記，第 5 章至第 9 章講述的是同一組東西。要理解 STL，應該從介紹性的第 5 章開始。

如果你是一位想總體認識 C++ 標準程式庫概念和其各個面向的程式員，可以簡略瀏覽這本書，並略過參考章節（reference sections）。當你需要運用某個 C++ 標準程式庫組件時，最好的辦法就是透過索引（index），找出相關資料。我已盡力把索引做得詳盡，希望能夠節省你查找的時間。

以我的經驗來看，學習新東西的最佳方式就是閱讀範例。因此，你會發現本書通篇有大量的範例，可能是幾行代碼，也可能是完整程式。如果是後者，註解第一行列有其檔案名稱。你可以在我的網站 <http://www.josuttis.com/libbook/> 找到這些檔案。

1.5 目前情勢

C++ *standard* 在我撰寫本書期間完成。請記住，有些編譯器可能還無法與之相容。這一點很可能在不久的將來得到很大的改善。但是現在，你可能會發現，本書所談的東西並非一定能夠在你的系統上有相同的表現，或許得作一番修改，才能在你的環境中正常運作。我所使用的 EGCS 編譯器 2.8 版，及其更高版本，能夠編譯書中所有範例程式。該編譯器幾乎適用於所有平台，並可從網際網路（<http://egcs.cygnum.com/>）和某些軟體光碟中獲得。

1.6 範例程式碼及額外資訊

從我的網站 <http://www.josuttis.com/libbook/> 上，你可以獲得所有範例程式碼、本書及 C++ 標準程式庫的其他相關資訊。你也可以在網際網路上找到許多其他相關資訊，詳見 743 頁。

1.7 回饋

歡迎你對本書的任何回應（不論好的或壞的）。我已盡力而為，但我畢竟是凡人，而且總有結稿的時候，所以難免有一些錯誤或前後不一的地方，需要進一步改善。你的回應將使我的新版本有機會進步。與我聯繫的最佳方式是透過電子郵件：

`libbook@josuttis.com`。

也可以透過電話、傳真、或「蝸牛般的」信件與我聯繫：

Nicolai M. Josuttis
Berggarten 9
D-38108 Braunschweig
Germany
Phone: +49 5309 5747
Fax: +49 5309 5774

非常感謝。

2

C++ 及其標準程式庫簡介

2.1 沿革

C++的標準化過程始於 1989 年，於 1997 年底完成。不過，由於某些原因，最終的標準規格遲至 1998 年 9 月才公佈，整個努力的成果就是國際標準化組織（ISO，International Organization for Standardization）發布的一份長達 750 頁的標準規格參考手冊。這份標準被命名為 "Information Technology — Programming Language — C++"，文件編號為 ISO/IEC 14882-1998，由 ISO 的各國成員機構發佈。例如美國的 ANSI 機構¹。

標準規格的建立，是 C++ 的一個重要里程碑，它準確定義了 C++ 的內容和行為，簡化了 C++ 的教學、使用、以及在不同平台之間的移植工作，給予用戶選擇不同 C++ 實作品的自由。它的穩定性和可移植性，對於程式庫、工具庫的供應者和實現者都是一樁佳音。因此，這份標準規格幫助 C++ 應用程式開發人員更快更好地創建應用程式，減輕維護上的精力。

標準程式庫是 C++ 標準規格的一部分，提供一系列核心組件，用以支援 I/O、字串（strings）、容器（資料結構）、演算法（排序、搜索、合併等等）、數值計算組件及國別（例如不同字元集，character sets）等主題。

這個標準化過程竟花費了近 10 年的時間，未免讓人奇怪。如果你了解到一些細節，更會奇怪為什麼這麼久之後此一標準仍然未臻完美。十年其實不夠！儘管從標準化的歷史和內容來看，確實完成了許多東西，也形成了可在實務中應用的成果，但是距離完美尚遠（畢竟世間無完美之物）。

¹ 本書寫作之時，你可以花 18 美元從 ANSI Electronics Standard Store 獲得這份 C++ 標準文件（見 <http://www.ansi.org/>）。

這份標準並非某個公司花費大把鈔票和大量時間後的產物。那些為制定標準而辛勤工作的人們，幾乎沒有從標準化組織那兒獲取任何報酬。對於他們而言，如果他們所處的公司對 C++ 標準化漠不關心，那麼他們就只能以興趣為全部的動力了。謝天謝地，有這麼多勇於奉獻的人，能夠拿出時間和財力參與其中。

這份 C++ 標準規格並非從零開始，它以 C++ 創始人 Bjarne Stroustrup 對於這個語言所作的描述做為基礎。然而標準程式庫並非基于某本書或某一個現成的函式庫，而是將各種不同的類別（classes）整合而成²，因此其結果並非十分地同質同種。你會發現不同組件背後有不同的設計原則，string class 和 STL 之間的差別就是很好的例子，後者是一個資料結構和演算法構架（framework）。

- string classes 被設計為安全易用的組件，其界面幾乎不言自明，並能對許多可能的錯誤作檢驗。
- STL 的設計目標，是將不同的演算法和資料結構相結合，並獲取最佳效率，所以 STL 並不非常便利，也不檢驗許多可能的邏輯錯誤。要運用 STL 強大的架構和優異的效率，你就必須通曉其概念並小心運用。

標準程式庫中有一個組件，在標準化之前就已經作為「準」標準而存在，那就是 IOStream library。IOStream 於 1984 年開發，1989 年重做過一次，部分內容重新設計。由於很多程序員早就已經使用 IOStream，所以 IOStream library 的概念沒有改變，保持回溯相容。

總體來說，整份標準規格（語言和程式庫）是在來自全球各地的數百位人士的大量討論和影響下誕生的，例如日本人就是國際化（internationalization）組件的重要支持者。當然，我們曾經犯下錯誤，曾經改弦更張，曾經意見不一。到了 1994 年，當人們認為這個標準接近完成時，STL 又被加了進來，此舉從根本上改變了整個程式庫。然而事情總要有個結束，終於有那麼一天，人們決定不再考慮任何重大擴張，無論這個擴展多麼有價值。就因為這樣，hash tables 沒有被納入標準——儘管它作為一種常用的資料結構，理應在 STL 之中享有一席之地。

現有的標準並不是終極產品，每五年會有一個新版本，對於錯誤和不一致的地方進行修正。然而，起碼在接下來的數年之中，C++ 程式員終於有了一個標準，有機會編寫功能強大並可移植到各種平台上的 C++ 程式了。

² 你可能會奇怪，為什麼標準化過程中不從頭設計一個新的程式庫。要知道，標準化的主要目的不是為了發明新東西或發展出某些東西，而是為了讓既有的東西調和共處。

2.2 新的語言 特性

C++ 核心語言和 C++ 程式庫是同時被標準化的，這麼一來，程式庫可以從語言的發展中受益，語言的發展也可以從程式庫的實作經驗中得到啟發。事實上，在標準化過程中，程式庫經常用到已規劃但尚未被實現出來的語言特性。

今天的 C++ 可不是五年前的 C++ 了。如果你沒有緊跟其發展，可能會對程式庫所使用的語言新特性大感驚訝。本節對這些新特性進行簡單的概括說明，至於細節，請查閱語言相關書籍。

我寫這本書的時候（1998 年），並非所有編譯器都提供所有的語言新特性，這就限制了程式庫的使用。我希望（並預期）這種情況能很快獲得改善（絕大多數編譯器廠商都參與了標準化過程）。實作一份可移植的程式庫時，通常都要考慮你的電算環境是否支持你所用到的特性 — 通常我們會使用一些測試程式，檢查哪些語言特性被支援，然後根據檢驗結果設置前處理器指令（preprocessor directives）。我會在書中以註腳方式指出所有典型而重要的限制。

接下來數個小節描述與 C++ 標準程式庫有關的最重要的幾個語言新特性。

2.2.1 template（模板）

程式庫中幾乎所有東西都被設計為 template 形式。不支援 template，就不能使用標準程式庫。此外程式庫還需要一些新的、特殊的 template 特性，我將在簡介之後詳細說明。

所謂 templates，是針對一個或多個尚未明確的型別所撰寫的函式或類別。使用 template 時，可以明白地（explicitly）或隱喻地（implicitly）將型別當作引來傳遞。下面是一個典型例子，傳回兩數之中的較大數：

```
template <class T>
inline const T& max (const T& a, const T& b)
{
    // 若 a < b 則取 b，否則取 a
    return a < b ? b : a;
}
```

在這裡，第一行定義 `T` 為任意資料型別，當此函式被呼叫時，由呼叫者指定。任何合法的識別符號都可以拿來作為參數名稱，但通常以 `T` 表示，這差不多成了一個「準」標準。型別由關鍵字 `class` 引導，但此型別不一定非得是 `class` — 任何資料型別只要提供 template 定義式內所用到的操作，都可適用於此 template³。

³這裡使用關鍵字 `class`，原本是為了避免為此而增加新的關鍵字。然而最終還是不得不引入一個新的關鍵字 `typename`，此處亦可用之（詳見 p11）。

遵循同樣原則，你可以將 `class` 參數化，並使用任意型別的參數。這一點對容器類別非常有用。你可以實作出有能力操控任意型別元素的容器。C++ 標準程式庫提供了許多 `template container classes`（詳見第 6 章和第 10 章）。標準函式庫對於 `template classes` 的使用，還有其他原因。例如可以以字元型別（`character type`）和字元集（`character set`）屬性，將 `string class` 參數化（詳見第 11 章）。

`Template` 並非一次編譯便生出適合所有型別的代碼，而是針對被使用的一個（或一組）型別進行編譯。這導致一個重要的問題：實際處理 `template` 時，你必須先提供某個 `template function` 的實作，然後才能呼叫之，如此方可通過編譯。所以，目前唯一讓「`template` 的運用」具有可攜性的方法，就是在表頭檔中以 `inline function` 實現 `template function`⁴。

欲實現 C++ 標準程式庫的完整功能，編譯器不僅需要提供 `template` 一般支持，還需要很多新的 `template` 標準特性，以下分別探討。

Nontype Templates ~~參數~~（非型別模板參數）

型別（`type`）可作為 `template` 的參數，非型別（`nontype`）也可以作為 `template` 參數。非型別參數因而可被看作是型別的一部分。例如可以把標準類別 `bitset<>`（本書 10.4 節 p460 會介紹它）的位元數量以 `template` 參數指定之。

以下述句定義了兩個由 `bits` 構成的容器，分別為 32 個位元空間和 50 個位元空間：

```
bitset<32> flags32;    // 32 位 bitset
bitset<50> flags50;    // 50 位 bitset
```

這些位元空間使用不同的 `template` 參數，所以有不同的型別，因此不能互相賦值或進行比較（除非提供了相應的型別轉換機制）。

Default Template Parameters（預設模板參數）

`Template classes` 可以有預設引數。例如以下宣告，允許你使用一個或兩個 `template` 引數來宣告 `MyClass` 物件⁵：

```
template <class T, class container = vector<T> >
class MyClass;
```

如果只傳給它一個引數，那麼預設參數可作為第二引數使用：

⁴ 目前 `template` 必須定義於表頭檔中。為了消除這個限制，標準規格導入了一個 `template compilation model`（模板編譯模型）和一個關鍵字 `export`。可惜據我所知，目前尚無任何編譯器實現了這一特性。

⁵ 注意，兩個 `>` 之間必須有一個空格，如果你沒寫空格，`>>` 會被解讀為移位運算子，導致語法錯誤。

```
MyClass<int> x1; //相當於： MyClass<int, vector<int> >
```

注意，`template` 預設引數可根據前一個（或前一些）引數而定義。

關鍵字 `typename`

關鍵字 `typename` 被用來做為型別之前的標識符號。考慮下面例子：

```
template <class T>
class MyClass
    typename T::SubType * ptr;
    ...
};
```

這裡，`typename` 指出 `SubType` 是 `class T` 中定義的一個型別，因此 `ptr` 是一個指向 `T::SubType` 型別的指標。如果沒有關鍵字 `typename`，`SubType` 會被當成一個 `static member`，於是：

```
T::SubType * ptr
```

會被解釋為型別 `T` 內的數值 `SubType` 與 `ptr` 的乘積。

`SubType` 成為一個型別的條件是，任何一個用來取代 `T` 的型別，其內都必須提供一個內隱型別（inner type）`SubType` 的定義。例如，將型別 `Q` 當做 `template` 參數：

```
MyClass<Q> x;
```

必要條件是型別 `Q` 有如下的內隱型別定義：

```
class Q
    typedef int SubType;
    ...
};
```

此時，`MyClass<Q>` 的 `ptr` 成員應該變成一個指向 `int` 型別的指標。子型別（subtype）也可以是抽象資料型別（例如 `class`）：

```
class Q
    class SubType;
    ...
};
```

注意，如果要把一個 `template` 中的某個標識符號指定為一種型別，就算意圖顯而易見，關鍵字 `typename` 也不可或缺，因此 C++ 通常規定，除了以 `typename` 修飾之外，`template` 內的任何標識符號都被視為一個實值（value）而非型別。

`typename` 還可在 `template` 宣告式中用來替換關鍵字 `class`：

```
template <typename T> class MyClass;
```

member template （成員模板）

`class member function` 可以是個 `template`，但這樣的 `member template` 既不能是 `virtual`，也不能有預設參數。例如：

```

class MyClass
{
...
    template <class T>
    void f(T);
};

```

這裡，`MyClass::f` 宣告了一個成員函式集，適用任何型別參數。只要某個型別提供有 `f()` 用到的所有操作，它就可以被當做引數傳遞進去。這個特性通常用來為 `template classes` 中的成員提供自動型別轉換。例如以下定義式中，`assign()` 的參數 `x`，其型別必須和呼叫端所提供的物件的型別完全吻合：

```

template <class T>
class MyClass
{
private:
    T value;
public:
    void assign (const MyClass<T>& x) { // x 必須和*this 型別相同
        value = x.value;
    }
    ...
};

```

即使兩個型別之間可以自動轉換，如果我們對 `assign()` 使用不同的 `template` 型別，也會出錯：

```

void f()
{
    MyClass<double> d;
    MyClass<int> i;

    d.assign(d); // OK
    d.assign(i); // 錯誤: i 是 MyClass<int>
                // 但要求的卻是 MyClass<double>
}

```

如果 C++ 允許我們為 `member function` 提供不同（一個以上）的 `template` 型別，就可以放寬「必須精確吻合」這條規則；只要型別可被賦值（*assignable*），就可以被當做上述 `member template function` 的引數。

```

template <class T>
class MyClass
{
private:
    T value;
};

```

```

public:
    template <class X> // member template
    void assign (const MyClass<X>& x) { // allows different template types
        value = x.getValue();
    }
    T getValue () const
        return value;
    }
    ...
};

void f()
{
    MyClass<double> d;
    MyClass<int> i;

    d.assign(d); // OK
    d.assign(i); // OK (int 可指定為 double)
}

```

請注意，現在，`assign()`的參數 `x` 和 `*this` 的型別並不相同，所以你不再能夠直接存取 `MyClass<>` 的 `private` 成員和 `protected` 成員，取而代之的是，此例中你必須使用類似 `getValue()` 之類的東西。

`template constructor` 是 `Member template` 的一種特殊形式。`Template constructor` 通常用於「在複製物件時實現隱式型別轉換」。注意，`template constructor` 並不遮蔽 (*hide*) `implicit copy constructor`。如果型別完全吻合，`implicit copy constructor` 就會被產生出來並被喚起。舉個例子：

```

template <class T>
class MyClass
public:
    //copy constructor with implicit type conversion
    // - does not hide implicit copy constructor
    template <class U>
    MyClass (const MyClass<U>& x);
    ...
};

void f()
{
    MyClass<double> xd;
}

```

```

...
MyClass<double> xd2(xd);    // calls built-in copy constructor
MyClass<int> xi(xd);        // calls template constructor
...
}

```

在這裡，`xd2` 和 `xd` 的型別完全一致，所以它被內建的 `copy constructor` 初始化。`xi` 的型別和 `xd` 不同，所以它使用 `template constructor` 進行初始化。因此，撰寫 `template constructor` 時，如果 `default copy constructor` 不符合需要，別忘了自己提供一個 `copy constructor`。`member template` 的另一個例子詳見 p33, 4.1 節。

Nested Template Classes

巢狀的 (*nested*) classes 本身也可以是個 `template`：

```

template <class T>
class MyClass
{
...
    template <class T2>
    class NestedClass;
...
};

```

2.2.2 基本型別的顯式初始化

如果採用不含參數的、明確的 `constructor` 呼叫語法，基本類型會被初始化為零：

```

int i1;           // undefined value
int i2 = int();   // initialized with zero

```

這個特性可以確保我們在撰寫 `template` 程式碼時，任何型別都有一個確切的初值。例如下面這個函式中，`x` 保證被初始化為零。

```

template <class T>
void f()
{
    T x = T();
    ...
}

```


2.2.3 異常處理 (Exception Handling)

藉由異常處理，C++ 標準程式庫可以在不「污染」函式介面（亦即參數和回返值）的情況下處理異常。如果你遇到一個意外情況，可以藉由「丟出一個異常」來停止一般的（正常的）處理過程：

```
class Error;
void f()
{
    ...
    if (exception-condition)
        throw Error(); // create object of class Error and throw it as exception
    ...
}
```

述句 `throw` 開始了 *stack unwinding*（堆疊輾轉開解）的過程，也就是說，它將導致退離任何函式區段，就像遭遇 `return` 述句一般，然而程式卻不會跳轉到任何地點。對於所有被宣告於某區段 — 而該區段卻因程式異常而退離 — 的區域物件而言，其 `destructor` 會被喚起。*Stack unwinding* 的動作會持續直到退出 `main()` 或直到有某個 `catch` 子句捕捉並處理了該異常為止。如果是第一種情況，程式會結束：

```
int main()
{
    try
    {
        ...
        f();
        ...
    }
    catch (const Error&)
    {
        ... // 處理異常
    }
    ...
}
```

這裡，`try` 區塊中任何型別為 `Error` 的異常都將在 `catch` 子句中獲得處理⁶。

⁶ 異常（exception）會終結某個函式呼叫，而該函式正是異常發生之處。異常處理機制有能力將某個物件當作引數，回傳給函式呼叫者。但這並非反向函式回呼（call back）。所謂反向是指從「問題被發現的地方」到「問題被處理的地方」，也就是從下往上的方向。以此觀之，異常處理和信號處理（signal handling）完全是兩碼事。

異常物件 (exception objects) 就是一般類別或基本型別的物件，可以是 ints、strings，也可以是類別體系中的某個 template classes。通常你會設計一個特殊的 error 類別體系。你可以運用異常物件的狀態 (state)，將任何資訊從錯誤被偵測到的地點帶往錯誤被處理的地點。

注意，這個概念叫作「異常處理 (exception handling)」，而不是「錯誤處理 (error handling)」，兩者未必相同。舉個例子，許多時候無效的用戶輸入 (這經常發生) 並非一種異常；這時候最好是在區域範圍內採用一般的錯誤處理技術來處理。

你可以運用所謂的異常規格 (exception specification) 來指明某個函式可能拋出哪些異常，例如：

```
void f() throw(bad_alloc); // f() 只可能丟出 bad_alloc 異常
```

如果宣告一個空白異常規格，那就表明該函式不會丟擲任何異常：

```
void f() throw(); // f() 不丟擲任何異常
```

違反異常規格將會導致特殊行為的發生，詳見 p26 關於異常類別 *bad_exception* 的描述。

C++ 標準程式庫提供了一些通用的異常處理特性，例如標準異常類別 (standard exception classes) 和 *auto_ptr* 類別 (詳見 p25, 3.3 節和 p38, 4.2 節)。

2.2.4 命名空間 (Namespaces)

越來越多的軟體以程式庫、模組 (modules) 和組件拼湊完成。各種不同事物的結合，可能導致一場名稱大衝突。Namespaces 正是用來解決此一問題。

Namespaces 將不同的標識符號集合在一個具名範圍內。如果你在 namespace 之內定義所有標識符號，則 namespace 本身名稱就成了唯一可能與其他全域符號衝突的標識符號。你必須採用「在標識符號之前加上 namespace 名字」的手法，才能援引 namespace 內的符號，這和 class 處理方式雷同。namespace 的名字和標識符號之間也是以 :: 分隔開來。

```
// defining identifiers in namespace josuttis
namespace josuttis
{
    class File;
    void myGlobalFunc();
    ...
}
...
// using a namespace identifier
josuttis::File obj;
...
josuttis::myGlobalFunc();
```

與 `class` 不同的是，`namespaces` 是開放的，你可以在不同模組（`modules`）中定義和擴展 `namespace`。因此你可以使用 `namespace` 來定義模組、程式庫或組件，甚至在多個檔案之間完成。`Namespace` 定義的是邏輯模組，而非實質模組。請注意，在 UML 及其他建模表示法（`modeling notations`）中，模組也被稱為 *package*。

如果某個函式的一或多個引數型別，乃定義於函式所處的 `namespace` 中，那麼你不必為該函式指定 `namespace`。這個規則稱為 *Koenig lookup*（譯注：Andrew Koenig 是 C++ 社群代表人物之一，對 C++ 的形成和發展有重要的貢獻）。例如：

```
// defining identifiers in namespace josuttis
namespace josuttis
{
    class File;
    void myGlobalFunc(const File&);
    ...
}
...

josuttis::File obj;
...
myGlobalFunc(obj); // OK, lookup finds josuttis::myGlobalFunc()
```

藉由 *using declaration*，我們可以避免一再寫出冗長的 `namespace` 名稱。例如以下宣告式：

```
using josuttis::File;
```

會使 `File` 成為當前範圍（`current scope`）內代表 `josuttis::File` 的一個同義字。

using directive 會使 `namespace` 內的所有名字曝光。*using directive* 等於將這些名字宣告於 `namespace` 之外。但這麼一來，名稱衝突問題就可能死灰復燃。例如：

```
using namespace josuttis;
```

會使 `File` 和 `MyGlobalFunc()` 在當前範圍內都曝光。如果全域範圍內已存在同名的 `File` 或 `MyGlobalFunc()`，而且使用者毫無其他任何限定地使用這兩個名字，編譯器將東西難辨。

注意，如果前後狀態不甚清楚（例如在表頭檔、模組或程式庫中），你不應該使用 *using directive*。這個指令可能會改變 `namespace` 的作用範圍，從而使程式碼被包含或使用於另一模組中，導致意外行為的發生。事實上在表頭檔中使用 *using directive* 相當不智。

C++ 標準程式庫在 `namespace std` 中定義了它所採用的所有標識符號，詳見 p23, 3.1 節。

2.2.5 bool 型別

爲了支持布林值（真假值），C++ 增加了 `bool` 型別。`bool` 可增加程式的可讀性，並允許你得以對布林值實現重載（*overloaded*）動作。兩個字面常數 `true` 和 `false` 同時亦被引入 C++。此外 C++ 還提供布林值與整數值之間的自動轉換。0 值相當於 `false`，非 0 值相當於 `true`。

2.2.6 關鍵字 `explicit`

藉由關鍵字 `explicit` 的作用，我們可禁止「單引數建構式（single argument constructor）」被用於自動型別轉換。典型的例子便是群集類別（collection classes），你可以將初始長度作為引數傳給 constructor，例如你可以宣告一個 constructor，將堆疊的初始大小當做引數：

```
class Stack
    explicit Stack(int size); // create stack with initial size
    ...
};
```

在這裡，`explicit` 的應用非常重要。如果沒有 `explicit`，這個 constructor 有能力將一個 `int` 自動轉型為 `Stack`。如果這種情況發生，你甚至可以給 `Stack` 指派一個整數值：

```
Stack s;
...
s = 40; // Oops, creates a new Stack for 40 elements and assigns it to s
```

「自動型別轉換」動作會把 40 轉換為有 40 個元素的 `stack`，並指派給 `s`，這幾乎肯定不是我們所要的結果。如果我們將 constructor 宣告為 `explicit`，上述指派動作就會導致編譯錯誤（那很好）。

注意，`explicit` 同樣也能阻絕「以賦值動作進行帶有轉型現象之初始化行為」：

```
Stack s1(40);    // OK
Stack s2 = 40;   // ERROR
```

這是因為以下兩組動作：

```
X x;
Y y(x);    // explicit conversion
```

和

```
X x;
Y y = x;    // implicit conversion
```

存在一個小差異。前者透過顯式轉換，根據型別 `x` 產生了一個型別 `Y` 的新物件；後者透過隱式轉換，產生了一個型別 `Y` 的新物件。

2.2.7 新的型別轉換運算子 (Type Conversion Operators)

爲了讓你對引數的顯式型別轉換更透徹，C++ 引入以下四個新的運算子：

1. `static_cast`

將一個值以符合邏輯的方式轉型。這可看做是「利用原值重建一個臨時物件，並在設立初值時使用型別轉換」。唯有當上述的型別轉換有所定義，整個轉換才會成功。所謂的「有所定義」，可以是內建的規則，也可以是自定的轉換動作。例如：

```
float x;
...
cout << static_cast<int>(x);    // print x as int
...
f(static_cast<string>("hello")); // call f() for string instead of char*
```

2. `dynamic_cast`

將多形型別 (polymorphic type) 向下轉型 (downcast) 爲其實際靜態型別 (real static type)。這是唯一在執行期進行檢驗的轉型動作。你可以用它來檢驗某個多形實值 (polymorphic value) 的型別，例如：

```
class Car; // abstract base class (has at least one virtual function)

class Cabriolet : public Car
...
};

class Limousine : public Car
...
};

void f(Car* cp)
{
    Cabriolet* p = dynamic_cast<Cabriolet*>(cp);
    if (p == NULL)
        // did not refer to an object of type Cabriolet
        ...
}
```

在這個例子中，針對具有 `Cabriolet` 靜態型別的物件，`f()` 有一個特殊行爲。當引數是個 `reference`，而且型別轉換失敗時，`dynamic_cast` 丟出一個 *bad_cast* 異常 (*bad_cast* 的描述詳見 p26)。注意，以設計角度而言，你應該在運用多形型別的程式中，避免這種依賴具體型別的寫法。

3. `const_cast`

加上或去除型別的常數性（`constness`），亦可去除 `volatile` 飾詞。除此之外的任何轉換都不允許。

4. `reinterpret_cast`

此運算子的行為由實作品（編譯器）定義。可能重新解釋位元意義，但也不一定如此。使用此一轉型動作通常帶來不可移植性。

這些運算子取代了以往小圓括號舊式轉型，能夠清楚闡明轉型目的。小圓括號轉型可替換 `dynamic_cast` 之外的其他三種轉型，也因此當你運用它時，你無法明確陳述使用它的確切理由。這些新式轉型運算子使編譯器收到更多資訊，讓編譯器清楚知道轉型的理由，並在轉型失敗時釋出一份錯誤報告。

注意，這些運算子都只接受一個引數。試看以下例子：

```
static_cast<Fraction>(15,100) // Oops, creates Fraction(100)
```

從這個例子中你得不到你想要的結果。它只用一個數值 100，將 `Fraction` 暫時物件設定初值，而非設定為分子 15、分母 100。逗號在這裡並不起分隔作用，而是形成一個 `comma`（逗號）運算子，將兩個算式組合為一個算式，並傳回第二個算式的結果。將 15 和 100「轉換」為分數的正確做法是：

```
Fraction(15,100) // fine, creates Fraction(15,100)
```

2.2.8 常數靜態成員（Constant Static Members）的初始化

如今，我們終於能夠在 `class` 宣告式中對整數型（`integral`）常數靜態成員直接賦予初值。初始化後，這個常數便可用於 `class` 之中，例如：

```
class MyClass
{
    static const int num = 100;
    int elems[num];
    ...
};
```

注意，你還必須為 `class` 之中所宣告的常數靜態成員，定義一個空間：

```
const int MyClass::num; // no initialization here
```

2.2.9 main() 的定義

我樂於澄清這個語言中的一個重要而又常被誤解的問題，那就是正確而可移植的 `main()` 的唯一寫法。根據 C++ 標準規格，只有兩種 `main()` 是可移植的：

```
int main()
{
    ...
}
```

和

```
int main (int argc, char* argv[])
{
    ...
}
```

這裡 `argv`（命令行參數陣列）也可定義為 `char**`。請注意，由於不允許隱喻存在的 `int`，所以回返型別明白寫為 `int` 是有必要的。你可以使用 `return` 句來結束 `main()`，但不必一定如此。這一點和 C 不同，換句話說，C++ 在 `main()` 的尾端定義了一個隱喻的

```
return 0;
```

這意味如果你不採用 `return` 句離開 `main()`，實際上就表示成功退出（傳回任何一個非零值都代表某種失敗）。出於這個原因，本書範例在 `main()` 尾端都沒有 `return` 句。有些編譯器可能對此發出警告（[譯註](#)：例如 Visual C++），有的甚至認為這是錯誤的。唔，那正是標準制定前的黑暗日子。

2.3 複雜度與 Big-O 表示法

對於 C++ 標準程式庫的某些部分（特別是 STL），演算法和成員函式的效率需要嚴肅考慮，因此需要動用「複雜度」的概念。計算機科學家運用特定符號，比較演算法的相對複雜度，於是便可以很快依演算法的運行時間分類，進行演算法之間的定性比較。這種衡量方法叫做 Big-O 表示法。

Big-O 表示法將一個演算法的運行時間以輸入量 n 的函數表示。例如，當運行時間隨元素個數成線性增長時（亦即，元素個數呈倍數增長，運行時間亦呈倍數增長），複雜度為 $O(n)$ ；如果運行時間獨立於輸入量，複雜度為 $O(1)$ 。表 2.1 列出典型的複雜度和其 Big-O 表示法。

請注意，Big-O 表示法隱藏了（忽略了）指數較小的因子（例如常數因子），這一點十分重要，更明確地說，它不關心演算法到底耗用多長時間。根據這種測量法，任何兩個線性演算法都被視為具有相同的接受度。甚至可能發生某種情況，使得線性演算法的常數非常巨大，導致帶有小常數的指數演算法反而耗時較短。這是

對 Big-O 表示法的一種合理質疑。記住，這只是一種量度規則。具有最佳複雜度的演算法，不一定就是最好的（最快的）演算法。

| 型別 | 表示法 | 含義 |
|---------|----------------|--------------------------|
| 常數 | $O(1)$ | 運行時間與元素個數無關。 |
| 對數 | $O(\log(n))$ | 運行時間隨元素個數的增加呈對數增長。 |
| 線性 | $O(n)$ | 運行時間隨元素個數的增加呈線性增長。 |
| n-log-n | $O(n*\log(n))$ | 運行時間隨元素個數的增加呈線性和對數的乘積增長。 |
| 二次 | $O(n^2)$ | 運行時間隨元素個數的增加呈平方增長。 |

表 2.1 典型的五種複雜度

表 2.2 列出了所有複雜度分類，並以某些元素個數來說明運行時間隨元素個數增長的程度。一如你所看到，當元素較少時，運行時間的差別很小，此時 Big-O 表示法所隱藏的常數因子可能會帶來很大影響。但是當元素個數越多，運行時間差別越大，常數因子也就變得無關緊要了。當你考慮複雜度時，請記住，輸入量必須夠大才有意義。

| 複雜度 | | 元素數目 | | | | | | |
|---------|----------------|------|---|----|-----|-------|--------|-----------|
| 型別 | 表示法 | 1 | 2 | 5 | 10 | 50 | 100 | 1000 |
| 常數 | $O(1)$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 對數 | $O(\log(n))$ | 1 | 2 | 3 | 4 | 6 | 7 | 10 |
| 線性 | $O(n)$ | 1 | 2 | 5 | 10 | 50 | 100 | 1,000 |
| n-log-n | $O(n*\log(n))$ | 1 | 4 | 15 | 40 | 300 | 700 | 10,000 |
| 二次方 | $O(n^2)$ | 1 | 4 | 25 | 100 | 2,500 | 10,000 | 1,000,000 |

表 2.2 運行時間、複雜度、元素個數 對照表

C++參考手冊中的某些複雜度被稱為 *amortized*（分期攤還），意思是，長期而言，大量操作將如描述那般進行，但單一操作卻可能花費比平均值更長的時間。舉個例子，如果你為一個動態陣列追加元素，運行時間將取決於陣列是否尚有備用記憶體。如果記憶體足夠，這就屬於常數複雜度，因為在尾端加入一個新元素，總是花費相同的時間。如果備用記憶體不足，這就是線性複雜度，因為，你必須配置足夠的記憶體並搬動（複製）它們，耗用時間取決於當時的元素個數。記憶體重新配置動作並不常發生（[譯註](#)：STL 的動態陣列容器會以某種哲學來保持備用記憶體），所以任何長度充份的序列，元素附加動作幾乎可說是常數複雜度。這種複雜度我們便稱為 *amortized*（分期攤還）常數時間。

3

一般概念

本章講述 C++ 標準程式庫中的基本概念。幾乎所有 C++ 標準程式庫組件都需要這些概念。

- 命名空間 std
- 表頭檔的名稱與格式
- 錯誤處理和異常處理的一般概念
- 配置器 (allocator) 的簡單介紹

3.1 命名空間 std

當你採用不同的模組和程式庫時，經常會出現名稱衝突的現象，這是因為不同的模組和程式庫可能針對不同的物件使用了相同的標識符號。namespaces (參見 p16, 2.2.4 節對 namespace 的介紹) 用來解決這個問題。所謂 namespace，是指標識符號的某種可見範圍。和 class 不同，namespace 具有擴展開放性，可以發生於任何源碼檔案上。因此你可以利用一個 namespace 來定義一些組件，而它們可散佈於多個實質模組上。這類組件的典型例子就是 C++ 標準程式庫，因此 C++ 標準程式庫使用了一個 namespace。事實上，C++ 標準程式庫中的所有標識符號都被定義於一個名為 std 的 namespace 中。

由於 namespace 的概念，使用 C++ 標準程式庫的任何標識符號時，你有三種選擇：

1. 直接指定標識符號。例如 `std::ostream` 而不是 `ostream`。完整語句類似這樣：

```
std::cout << std::hex << 3.4 << std::endl
```

2. 使用 *using declaration* (詳見 p17)。例如，以下程式片段使我們不必再寫出範圍修飾符號 `std::`，而可直接使用 `cout` 和 `endl`：

```
using std::cout;  
using std::endl;
```

於是先前的例子可以寫成這樣：

```
cout << std::hex << 3.4 << endl;
```

3 · 使用 *using directive* (詳見 p17)，這是最簡便的選擇。如果對 `namespace std` 採用 *using directive*，便可以讓 `std` 內定義的所有標識符號都有效（曝光），就好像它們被宣告全域標識符號一樣。因此，寫下：

```
using namespace std;
```

之後，就可以直接寫：

```
cout << hex << 3.4 << endl;
```

但請注意，由於某些晦澀的重載（*overloading*）規則，在複雜的程式中，這種方式可能導致意外的命名衝突，更糟的是甚至導致不同的行為。如果使用場合不夠明確（例如在表頭檔、模組或程式庫中），就應避免使用 *using directive*。

本書的例子都很小，所以，爲了我自己方便，書中我通常在範例程式中採用最後一種手法。

3.2 表頭檔

將 C++ 標準程式庫中所有標識符號都定義於 `namespace std` 裡頭，這種做法是標準化過程中引入的。這個作法不具回溯相容性，因爲原先的 C/C++ 表頭檔都將 C++ 標準程式庫的標識符號定義於全域範圍（*global scope*）。此外標準化過程中有些 *classes* 的介面也有了更動（當然啦，儘可能以回溯相容爲目標）。因此，這裡引入了一套新的表頭檔命名風格。這麼一來組件供應商得以藉由「提供舊的表頭檔」來達到回溯相容目的。

既然有必要重新定義標準表頭檔的名字，正好藉此機會把表頭檔副檔名加以規範。以往，表頭檔副檔名五花八門，包括 `.h`, `.hpp`, `.hxx`。相較之下標準表頭檔的副檔名簡潔得令人吃驚：根本就沒有副檔名了。於是，標準表頭檔的 `#include` 句如下：

```
#include <iostream>
#include <string>
```

這種寫法也適用於 C 的標準表頭檔。但必須採用前綴字元 `c`，而不再是副檔名 `.h`：

```
#include <cstdlib>    // was: <stdlib.h>
#include <cstring>    // was: <string.h>
```

在這些表頭檔中，每一個標識符號都被宣告於 `namespace std`。

這種命名方式的優點之一是可以區分舊表頭檔中的 `char*` C 函式，和新表頭檔中的標準的 C++ `string` `class`：

```
#include <string>      // C++ class string
#include <cstring>     // char* functions from C
```

注意，以作業系統角度觀之，新表頭檔命名方式並非意味標準表頭檔沒有副檔名。標準表頭檔的 `#include` 該如何處理，由編譯器決定。C++ 系統可以自動添加一個副檔名，甚至可以使用內建宣告，不讀入任何檔案。不過實際上大多數系統只是簡單含入一個「名稱與 `#include` 句中的檔名完全相同」的檔案。所以，在大部份系統中，C++ 標準表頭檔都沒有副檔名。注意，「無副檔名」這一條件只適用於標準表頭檔。一般而言，為你自己所寫的表頭檔加上一個良好的副檔名，仍然是個好想法，有助於輕易識別出這些檔案的性質。

為了回溯相容於 C，舊式的 C 標準表頭檔仍然有效，如果需要，你還是可以使用它們，例如：

```
#include <stdlib.h>
```

此時，標識符號同時宣告於全域範圍和 `namespace std` 中。事實上這些表頭檔的行為類似先在 `std` 中宣告所有標識符號，再悄悄使用 *using declaration* 把這些標識符號引入全域範圍（參見 p17）。

至於 `<iostream.h>` 這一類 C++ 舊式表頭檔，標準規格中並未加以規範（這一點在標準化過程中曾經多次改變），意味不再支持這些表頭檔。不過目前大多數廠商都會提供它們，以保證回溯相容。

注意，除了引入 `namespace std`，表頭檔還有很多改變。所以，你要麼就採用表頭檔舊名，要麼就應該完全改用新的標準名稱。

3.3 錯誤處理和異常處理

C++ 標準程式庫由不同的成份構成。來源不同，設計與實現風格迥異。而錯誤處理和異常處理正是這種差異的典型表徵。標準程式庫中有一部分，例如 `string classes`，支援具體的錯誤處理，它們檢查所有可能發生的錯誤，並於錯誤發生時丟出異常。至於其它部分如 STL（standard template library）和 `valarrays`，效率甚於安全性，因此幾乎不檢驗邏輯錯誤，並且只在執行期錯誤發生時才丟擲異常。

3.3.1 標準異常類別（Standard Exception Classes）

語言本身或標準程式庫所丟擲的所有異常，都衍生自基礎類別 `exception`。這是其他幾個標準異常類別的基礎類別，共同構成一個類別階層體系。如圖 3.1。這些標準異常類別可分為三組：

1. 用於語言本身所支援的異常
2. 用於 C++ 標準程式庫所發生的異常
3. 用於程式作用域（scope of a program）之外發生的異常

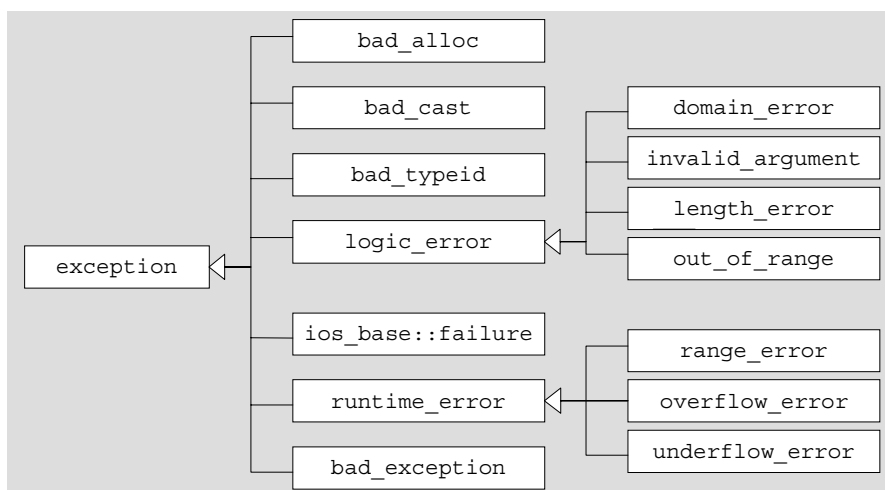


圖 3.1 標準異常（Standard Exceptions）階層體系

語言本身所支援的異常

此類異常是用以支持某些語言特性，所以，從某種角度來說它們不是標準程式庫的一部分，而是核心語言的一部分。當以下操作失敗時，會丟擲這一類異常。

- 如果全域運算子 `new` 操作失敗，就會丟擲 *bad_alloc* 異常（採用 `new` 的 `nothrow` 版本則另當別論）。由於這個異常可能於任何時間在任何較複雜的程式中發生，所以可說是最重要的一個異常。
- 執行期間，當一個加諸於 `reference` 之上的「動態型別轉換動作」失敗時，`dynamic_cast` 會丟擲 *bad_cast* 異常。P19 對於 `dynamic_cast` 運算子有所描述。
- 執行時期型別辨識（RTTI）的過程中，如果交給 `typeid` 的引數為零或空指標，則 `typeid` 運算子會丟擲 *bad_typeid* 異常。
- 如果發生非預期的異常，*bad_exception* 異常會接手處理，方式如下：當函式擲出不在異常規格（exception specification，p16 有介紹）之列的異常，*bad_exception* 就會喚起 `unexpected()`。例如：

```
class E1;
class E2; // not derived from E1
```

```

void f() throw(E1) // throws only exceptions of type E1
{
    ...
    throw E1(); // throws exception of type E1
    ...
    throw E2(); // calls unexpected(), which calls terminate()
}

```

`f()` 之中丟擲出「型別為 *E2*」的異常，這種動作違反了異常規格（exception specification），於是喚起 `unexpected()`，而該函式通常會喚起 `terminate()` 來終止程式。

然而如果你在你的異常規格之中列出 *bad_exception*，那麼 `unexpected()` 總是會重新擲出（rethrows）*bad_exception* 異常。

```

class E1;
class E2;    // not derived from E1

void f() throw(E1, std::bad_exception)
           // throws exception of type E1 or
           // bad_exception for any other exception type
{
    ...
    throw E1(); // throws exception of type E1
    ...
    throw E2(); // calls unexpected(), which throws bad_exception
}

```

因此，如果異常規格羅列了 *bad_exception*，那麼任何未列於規格的異常，都將在函式 `unexpected()` 中被代之以 *bad_exception*¹。

C++ 標準程式庫所發生的異常

C++ 標準程式庫異常總是衍生自 *logic_error*。就理論而言，我們能夠透過一些手段，在程式中避免邏輯錯誤——例如對函式引數進行額外測試等等。所謂邏輯錯誤包括違背邏輯前提或違反 *class* 的不變性。C++ 標準程式庫提供以下邏輯錯誤類別：

- *invalid_argument* 表示無效引數，例如將 *bitset*（陣列，用以存放位元）以 *char* 而非 '0' 或 '1' 進行初始化。
- *length_error* 指出「可能超越最大容忍極限」的行為，例如對著字串附加太多字元。

¹ 你可以修改 `unexpected()` 的具體操作。然而只要宣告有異常規格，函式就絕不會丟擲出規格中未列的異常。

- *out_of_range* 指出「不在預期範圍內」的引數值，例如在諸如 *array* 的群集（collection）或字串（string）中採用了一個錯誤索引。
- *domain_error* 指出專業領域範疇內的錯誤。

此外，標準程式庫的 I/O 部分提供了一個名為 *ios_base::failure* 的特殊異常。當資料串流（data stream）由於錯誤或由於到達檔案尾端而發生狀態的改變時，就可能丟擲出這個異常。此一異常的具體行為詳見 p602, 13.4.4 節。

程式作用域（scope of a program）之可能發生的異常

衍生自 *runtime_error* 的異常，用來指出「不在程式範圍內，或無法簡單避免」的事件。C++ 標準程式庫針對執行期錯誤提供了以下三個 classes：

- *range_error* 指出內部計算時發生區間錯誤（range error）。
- *overflow_error* 指出算術運算發生上溢位（overflow）。
- *underflow_error* 指出算術運算發生下溢位（underflow）。

標準程式庫丟擲的異常

C++ 標準程式庫自身可能丟擲 *range_error*、*out_of_range* 和 *invalid_argument* 異常。然而由於標準程式庫會用到語言特性及客戶所寫的程式碼，所以也可能間接擲出任何異常。尤其是，無論何時配置存儲空間，都有可能擲出 *bad_alloc* 異常。

標準程式庫的任何具體實作品，都可能提供額外的異常類別（或作為兄弟類別，或衍生為子類別）。使用這些非標準類別將會導致程式難以移植，因為如果你想採用其他的標準程式庫版本，就不得不痛苦地修改你的程式。所以最好只使用標準異常。

異常類別的表頭檔

基礎類別 *exception* 和 *bad_exception* 定義於 `<exception>`。 *bad_alloc* 定義於 `<new>`。 *bad_cast* 和 *bad_typeid* 定義於 `<typeinfo>`。 *ios_base::failure* 定義於 `<ios>`。所有其他類別定義於 `<stdexcept>`。

3.3.2 異常類別成員

為了在 *catch* 子句中處理異常，你必須採用異常所提供的介面。所有標準異常的介面只包括一個用以獲取「型別本身以外的附加資訊」的成員函式：*what()*，它傳回一個以 *null* 結束的字串：

```
namespace std {
    class exception {
    public:
```

```

        virtual const char* what() const throw();
        ...
    };
}

```

被傳回的字串，其內容由實作廠商定義。它很大程度（當然並非必然）決定了幫助的級別和資訊的詳細度。注意，該字串有可能是個以 `null` 結尾的 "multibyte" 字串，可被輕鬆轉換為 `wstring`（詳見 p480, 11.2.1 節）並顯示出來。`what()` 傳回的 C-string 在其所屬的異常物件被摧毀後，就不再有效了²。

標準異常中的其他成員，用來處理生成、複製、指派、摧毀等動作。要注意的是，除了 `what()`，再沒有任何異常提供任何其他成員能夠描述異常的種類。例如，沒有一致方法可以找出異常發生的脈絡，或找出區間錯誤（`range error`）發生時的那個錯誤索引值。因此，唯一通用的異常核定手段，大概只有列印一途了：

```

try {
    ...
}
catch (const std::exception& error) {
    // print implementation-defined error message
    std::cerr << error.what() << std::endl;
    ...
}

```

唯一可能實現的另一個異常核定手段是，根據異常的精確型別，自己得出推論。例如如果擲出 `bad_alloc` 異常，可能是因為程式企圖獲得更多記憶體。

3.3.3 丟擲標準異常

你可以在自己的程式庫或程式內部丟擲某些標準異常。允許你這般運用的標準異常，生成時都只需要一個 `string` 參數（第 11 章對 `string` class 有所描述），它將成為被 `what()` 傳回的描述字串。例如 `logic_error` 定義如下：

```

namespace std {
    class logic_error : public exception {
    public:
        explicit logic_error (const string& whatString);
    };
}

```

² C++ 標準規格並未對 `what()` 回返值的壽命加以規範，這裡所講的是被提出的一種建議解決方案。

提供這種功能的標準異常有：*logic_error* 及其衍生類別、*runtime_error* 及其衍生類別，*ios_base::failure*。你不能丟擲 *exception*，也不能丟擲任何用以支援語言核心性質的異常。

想要丟擲一個標準異常，只需生成一個描述該異常的字串，並將它初始化，交給異常物件：

```
std::string s;
...
throw std::out_of_range(s);
```

由於 `char*` 可以隱式轉換為 `string`，所以你可以直接使用字串字面常數：

```
throw std::out_of_range("out_of_range (somewhere, somehow)");
```

3.3.4 從標準異常類別衍生新類別

另一個在程式中採用標準異常類別的可能情況，是定義一個直接或間接衍生自 *exception* 的特定異常類別。要這麼做，首先必須確保 `what()` 機制正常運作。`what()` 是個虛擬函式，所以提供 `what()` 的方法之一就是自己實現 `what()`：

```
namespace MyLib {
    /* user-defined exception class
     * derived from a standard class for exceptions
     */
    class MyProblem : public std::exception {
    public:
        ...
        MyProblem(...) { // special constructor
        }
        virtual const char* what() const throw() { // what() function
            ...
        }
    };
    ...

    void f() {
        ...
        // create an exception object and throw it
    }
}
```



```
        throw MyProblem(...);  
        ...  
    }  
}
```

提供 `what()` 函式的另一種方法是，令你的異常類別衍生自 3.3.3 節所描述的標準異常：

```
namespace MyLib {  
    /* user-defined exception class  
    * - derived from a standard class for exceptions  
    * that has a constructor for the what() argument  
    */  
  
    class MyRangeProblem : public std::out_of_range {  
    public:  
        MyRangeProblem (const string& whatString)  
            : out_of_range(whatString) {  
        }  
    };  
    ...  
  
    void f() {  
        ...  
        // create an exception object by using a string constructor and throw it  
        throw MyRangeProblem("here is my special range problem");  
        ...  
    }  
}
```

完整程式見 p441 的 `Stack` 和 p450 的 `Queue`。

3.4 配置器 (Allocators)

C++ 標準程式庫在許多地方採用特殊物件來處理記憶體的配置和定址，這樣的物件稱為配置器 (allocator)。配置器表現出一種特定的記憶體模型，成為一個抽象表徵，表現出「對記憶體的使用需求」至「對記憶體的底階呼叫」的轉換。同時運用多個不同的配置器物件，便可讓你在一個程式中採用不同的記憶體模型。

配置器最初是作為 STL 的一部份而引進，用於處理諸如 PC 上不同指標型別（例如 `near`, `far`, `huge` 指標）這一類亂七八糟的問題；現在則作為一種技術方案的基礎，

使得諸如共享記憶體、垃圾收集、物件導向資料庫等等特定記憶體模型能夠保持一致的介面。但是這種用法還相當新，尚未獲得廣泛的接受（情況正在改變中）。

C++ 標準程式庫定義了一個預設的配置器（default allocator）如下：

```
namespace std {  
    template <class T>  
        class allocator;  
}
```

預設配置器可在任何「配置器得以被當作引數使用」的地方擔任預設值。預設配置器將會進行記憶體配置和回收的一般手法，也就是呼叫 `new` 和 `delete` 運算子。但是，C++ 並沒有對於「在什麼時候以什麼方式調用這些運算子」加以明確規定。所以，預設配置器甚至可能對於已配置的記憶體施行內部快取（internal cache）作法。

絕大多數程式都採用預設配置器，但有時候其他程式庫也可能提供某些配置器，以滿足特定需求。這種情況下只需簡單地將它們當作引數即可。自行設計並實作配置器的實際意義不大。實際生活中最典型的方式還是直接採用預設配置器，所以我將遲至第 15 章才詳細探討配置器。

4

通用工具

Utilities

本章講解 C++ 標準程式庫中的通用工具。它們由短小精幹的類別和函式構成，執行最一般性的工作。這些工具包括：

- 數種通用型別
- 一些重要的 C 函式
- 數值極值¹

大部分工具在 C++ 標準規格書第 20 款（`clause`）中描述，定義於標準表頭檔 `<utility>` 之中。其餘工具則與標準程式庫中一些比較主要的組件一起描述，這是因為那一類工具主要是與那些組件共同使用，抑或由於歷史因素。例如某些通用輔助型函式被定義於 `<algorithm>` 表頭檔中，但按照 STL 的定義，它們不算是演算法（參見第 5 章）。

這些工具中的一部分在 C++ 標準程式庫中也有應用。特別是型別 `pair`，凡需要將兩個值視為一個單元的場合（例如必須回傳兩個值的某函式），就必須用到它。

4.1 Pairs

`class pair` 可以將兩個值視為一個單元。C++ 標準程式庫內多處使用了這個 `class`。尤其容器類別 `map` 和 `multimap`，就是使用 `pairs` 來管理其鍵值/實值（`value/key`）的成對元素（詳見 6.6 節，p194）。回傳兩個值的函式，也需要 `pair`。

¹ 可能有人會認為數值極值應該屬於第 12 章，也就是專門講解數值的那一章中，但這些數值極值在程式庫的其他部分也會被用到，所以我決定把它放在這裏。

Structure `pair` 定義於 `<utility>` :

```
namespace std {
    template <class T1, class T2>
    struct pair {
        // type names for the values
        typedef T1 first_type;
        typedef T2 second_type;

        // member
        T1 first;
        T2 second;

        /* default constructor
        *-T1() and T2() force initialization for built-in types
        */
        pair()
            : first(T1()), second(T2()) {
        }

        // constructor for two values
        pair(const T1& a, const T2& b)
            : first(a), second(b) {
        }

        // copy constructor with implicit conversions
        template<class U, class V>
        pair(const pair<U,V>& p)
            : first(p.first), second(p.second) {
        }
    };

    // comparisons
    template <class T1, class T2>
    bool operator== (const pair<T1,T2>&, const pair<T1,T2>&);
    template <class T1, class T2>
    bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
    ... // similar: !=, <=, >, >=
```

```

// convenience function to create a pair
template <class T1, class T2>
pair<T1,T2> make_pair (const T1&, const T2&);
}

```

注意，`pair` 被定義為 `struct`，而不是 `class`，這麼一來，所有成員都是 `public`，我們因此可以直接存取 `pair` 中的個別值。

Default constructor 生成一個 `pair` 時，以兩個「被其 default constructor 個別初始化」的值做為初值。根據語言規則，基本型別（如 `int`）的 default constructor 也可以引起適當的初始化動作，所以：

```
std::pair<int, float> p; // initialize p.first and p.second with zero
```

就是用 `int()` 和 `float()` 來初始化 `p`。這兩個 constructor 都傳回零值。p14 曾經討論過基本型別的顯式初始化動作。

這裏之所以使用 `template` 形式的 copy constructor，是因為建構過程中可能需要隱式型別轉換。如果 `pair` 物件被複製，喚起的是由系統隱喻合成的那個 copy constructor²。例如：

```

void f(std::pair<int, const char*>);
void g(std::pair<const int, std::string>);
...
void foo {
    std::pair<int, const char*> p(42, "hello");
    f(p); // OK: calls built-in default copy constructor
    g(p); // OK: calls template constructor
}

```

Pair 之間的比較

爲了比較兩個 `pair` 物件，C++ 標準程式庫提供了爲大家所慣用的運算子。如果兩個 `pair` 物件內的所有元素都相等，這兩個 `pair` 物件就被視為相等：

```

namespace std {
    template <class T1, class T2>
    bool operator== (const pair<T1,T2>& x, const pair<T1,T2>& y) {
        return x.first == y.first && x.second == y.second;
    }
}

```

² `template` 形式的 constructor 並不會遮掩隱喻合成的 default constructor。詳見 p13。

兩個 `pairs` 互相比較時，第一個元素具有較高的優先序。所以如果兩個 `pairs` 的第一元素不相等，其比較結果就是整個比較行為的結果。如果第一元素相等，才繼續比較第二元素，並把比較結果當作整體比較結果。

```
namespace std {
    template <class T1, class T2>
    bool operator< (const pair<T1,T2>& x, const pair<T1,T2>& y) {
        return x.first < y.first ||
            (!(y.first < x.first) && x.second < y.second);
    }
}
```

其他比較運算子也如法炮製。

4.1.1 便捷函式 `make_pair()`

`template` 函式 `make_pair()` 使你無需寫出型別，就可以生成一個 `pair` 物件³：

```
namespace std {
    // create value pair only by providing the values
    template <class T1, class T2>
    pair<T1,T2> make_pair (const T1& x, const T2& y) {
        return pair<T1,T2>(x, y);
    }
}
```

例如，你可以這樣使用 `make_pair()`：

```
std::make_pair(42, '@')
```

而不必費力地這麼寫：

```
std::pair<int, char>(42, '@')
```

當我們有必要對一個接受 `pair` 引數的函式傳遞兩個值時，`make_pair()` 顯得尤其方便，請看下例：

```
void f(std::pair<int, const char*>);
void g(std::pair<const int, std::string>);
...
void foo {
    f(std::make_pair(42, "hello")); // pass two values as pair
    g(std::make_pair(42, "hello")); // pass two values as pair
    // with type conversions
}
```

³ 使用 `make_pair` 並不會多花你任何執行時間，編譯器應該會將此一動作最佳化。

從例子中可以看出，`make_pair()`使得「將兩值當作一個 `pair` 引數來傳遞」的動作更容易。即使兩個值的型別並不準確符合要求，也能在 `template constructor` 提供的支援下順利工作。當你使用 `map` 和 `multimap`，你會經常用到這個特點（詳見 p203）。

注意，一個算式如果明白指出型別，便帶有一個優勢：產生出來的 `pair` 將有絕對明確的型別。例如：

```
std::pair<int, float>(42, 7.77)
```

其結果與：

```
std::make_pair(42, 7.77)
```

不同。後者所生成的 `pair`，第二元素的型別是 `double`（因為無任何飾詞的浮點字面常數的型別被視為 `double`）。當我們使用 `overloaded function`（重載函式）或 `template`，確切的型別非常重要。例如，爲了提高效率，程式員可能同時提供分別針對 `float` 和 `double` 的 `function` 或 `template`，這時候確切型別就非常重要了。

4.1.2 `Pair` 的運用實例

C++ 標準程式庫大量運用了 `pair`。例如 `map` 和 `multimap` 容器的元素型別便是 `pair`，也就是一組鍵值/實值（`key/value`）。關於 `maps` 和 `multimaps` 的一般性描述，詳見 p194, 6.6 節。p91 有一個 `pair` 型別的運用實例。C++ 標準程式庫中凡是「必須傳回兩個值」的函式，也都會利用 `pair` 物件（實例請見 p183）。

4.2 Class `auto_ptr`

本節描述 `auto_ptr` 型別。C++ 標準程式庫提供了 `auto_ptr` 是一種智慧型指標（smart pointer），幫助程式員防止「異常被擲出時發生資源洩漏」。注意我說的是「一種」智慧型指標，現實生活中有許多有用的智慧型指標，`auto_ptr` 只是針對某個特定問題而設計，對於其他問題，`auto_ptr` 無能為力。所以，請謹慎閱讀以下內容。

4.2.1 `auto_ptr` 的發長動機

函式的操作經常依以下模式進行⁴：

1. 獲取一些資源。
2. 執行一些動作。
3. 釋放所獲取的資源。

如果一開始獲取的資源，被繫結於區域物件（local objects）身上，當函式退出時，它們的解構式被喚起，從而自動釋放這些資源。然而事情並不總是如此順利，如果資源是以顯式方式（explicitly）獲得，而且未被繫結於任何物件身上，那就必須以顯式方式釋放。這種情形常常發生在指標身上。

一個典型的例子就是運用 `new` 和 `delete` 來產生和銷毀物件：

```
void f()
{
    ClassA* ptr = new ClassA; // create an object explicitly
    ...                      // perform some operations
    delete ptr;               // clean up (destroy the object explicitly)
}
```

也許你尚未意識到，這個函式其實是一系列麻煩的根源。一個顯而易見的問題是，我們經常忘掉 `delete`，特別是當函式中間存在 `return` 述句時更是如此。然而真正的麻煩發生於更隱晦之處，那就是當異常發生時我們所要面對的災難。異常一旦出現，函式將立刻退離，根本不會呼叫函式尾端的 `delete` 述句。結果可能是記憶體遺失，或更一般地說是資源的遺失。防止這種資源遺失的常見辦法就是捕捉所有異常，例如：

⁴ `class auto_ptr` 的推動，是以 Scott Meyers 所著《*More Effective C++*》一書中的相關資料為基礎（並獲得他的允許）。這個問題的一般性技術最早描述於 Bjarne Stroustrup 的《*The C++ Programming Language*》2nd Edition 和《*The Design and Evolution of C++*》，當時的主題是 "resource allocation is initialization"。`auto_ptr` 被加入 C++ 標準之中，正是為了支援此一技術。


```

void f()
{
    ClassA* ptr = new ClassA;    // create an object explicitly

    try {
        ...                    // perform some operations
    }
    catch (...) {               // for any exception
        delete ptr;             // - clean up
        throw;                  // - rethrow the exception
    }

    delete ptr;                // clean up on normal end
}

```

你看，爲了在異常發生時處理物件的刪除工作，源碼變得多麼複雜和累贅！如果還有第二個物件，如果還要照此辦理，如果還需要更多的 `catch` 子句，那簡直是一場惡夢。這不是優良的編程風格，複雜而且容易出錯，必須盡力避免。

如果使用智慧型指標，情形就會大不相同。這個智慧型指標應該保證，無論在何種情形下，只要自己被摧毀，就一定連帶釋放其所指的資源。而由於指標本身就是區域變數，所以無論是正常退出，還是異常退出，只要函式退出，它就一定會被銷毀。`auto_ptr` 正是這種智慧型指標。

`auto_ptr` 是這樣一種指標：它是它所指向的物件的擁有者（owner）。所以，當身為物件擁有者的 `auto_ptr` 被摧毀時，該物件也將遭到摧毀。`auto_ptr` 要求，一個物件只能有一個擁有者，嚴禁一物二主。

下面是上例改寫後的版本：

```

// header file for auto_ptr
#include <memory>

void f()
{
    // create and initialize an auto_ptr
    std::auto_ptr<ClassA> ptr(new ClassA);

    ...                    // perform some operations
}

```

不再需要 `delete`，也不再需要 `catch` 了。`auto_ptr` 的介面與一般指標非常相似，`operator*` 用來提領其所指物件，`operator->` 用來指向物件中的成員。然而，所有指標算術（包括 `++`）都沒有定義（這可能是件好事，因為指標算術是一大麻煩根源）。

注意，`auto_ptr<>` 不允許你使用一般指標慣用的賦值（`assign`）初始化方式。你必須直接使用數值來完成初始化⁵：

```
std::auto_ptr<ClassA> ptr1(new ClassA);           // OK
std::auto_ptr<ClassA> ptr2 = new ClassA;          // ERROR
```

4.2.2 `auto_ptr` 擁有權的轉移

`auto_ptr` 所界定的乃是一種嚴格的擁有權觀念。也就是說，由於一個 `auto_ptr` 會刪除其指向物件，所以這個物件絕對不能同時被其他物件「擁有」。絕對不應該出現多個 `auto_ptr`s 同時擁有一個物件的情況。不幸的是，這種事情可能會發生（如果你以同一個物件為初值，將兩個 `auto_ptr`s 初始化，就會出現這種事）。程式員必須負責防範這種錯誤。

這個條件導致了一個問題：`auto_ptr` 的 `copy constructor` 和 `assignment operator` 應當如何運作？此類操作往往是將此處資料拷貝到彼處。然而這種操作恰恰會導致上面所提的情形。解決辦法很簡單，但意義深遠：令 `auto_ptr` 的 `copy constructor` 和 `assignment operator` 將物件擁有權交出去。試看下例 `copy constructor` 的使用：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);

// copy the auto_ptr
// - transfers ownership from ptr1 to ptr2
std::auto_ptr<ClassA> ptr2(ptr1);
```

第一個述句中，`ptr1` 擁有了那個 `new` 出來的物件。第二個述句中，擁有權由 `ptr1` 轉交給 `ptr2`。此後 `ptr2` 就擁有了那個 `new` 出來的物件，而 `ptr1` 不再擁有它。這樣，物件就只會被 `delete` 一次 — 當 `ptr2` 被銷毀的時候。

⁵ 下面兩種情況實際上是有分別的：

```
X x;
Y y(x); // 顯式轉換
```

和：

```
X x;
Y y = x; // 隱式轉換
```

前者使用顯式轉換，以型別 `x` 建構型別 `Y` 的一個新物件，後者使用隱式轉換。

指派（賦值）動作也差不多：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
std::auto_ptr<ClassA> ptr2;    // create another auto_ptr
ptr2 = ptr1; // assign the auto_ptr
           // - transfers ownership from ptr1 to ptr2
```

在這裡，賦值（*assign*）動作將擁有權從 `ptr1` 轉移至 `ptr2`。於是，`ptr2` 擁有了先前被 `ptr1` 所擁有的那個物件。

如果 `ptr2` 被賦值之前正擁有另一個物件，賦值動作發生時會呼叫 `delete`，將該物件刪除：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
// initialize another auto_ptr with a new object
std::auto_ptr<ClassA> ptr2(new ClassA);

ptr2 = ptr1;          // assign the auto_ptr
                     // - delete object owned by ptr2
                     // - transfers ownership from ptr1 to ptr2
```

注意，擁有權的轉移，意味實值並非只是被簡單拷貝而已。只要發生了擁有權轉移，先前的擁有者（本例為 `ptr1`）就失去了擁有權，結果是，擁有者一旦交出擁有權，就兩手空空，只剩一個 `null` 指標在手了。在這裡，`copy constructor` 更動了「用以初始化新物件」的原物件，而賦值操作也修改了右手邊物件，這與程式語言中慣常的初始化動作和賦值（指派）動作可說是大相逕庭。那麼誰來保證那個「失去了所有權、只剩一個 `null` 指標」的原 `auto_ptr` 不會再次進行提領動作呢？是的，還是程式員的責任。

只有 `auto_ptr` 可以拿來當做另一個 `auto_ptr` 的初值，普通指標是不行的：

```
std::auto_ptr<ClassA> ptr;          // create an auto_ptr

ptr = new ClassA;                   // ERROR
ptr = std::auto_ptr<ClassA>(new ClassA); // OK, delete old object
                                     // and own new
```

起點和終站（source and sink）

擁有權的移轉，使得 `auto_ptr`s 產生一種特殊用法：某個函式可以利用 `auto_ptr` 將擁有權轉交給另一個函式。這種事情可能在兩種情形下出現：

1. 某函式是資料的終站。如果 `auto_ptr` 以 **by value** (傳值) 方式被當做一個引數傳遞給該函式，就屬於這種情況。此時被呼叫函式的參數獲得了這個 `auto_ptr` 的擁有權，如果函式不再將它傳遞出去，它所指的物件就會在函式退出時被刪除：

```
void sink(std::auto_ptr<ClassA>);    // sink() gets ownership
```

2. 某函式是資料的起點。當一個 `auto_ptr` 被傳回，其擁有權便被轉交給呼叫端了。見下例：

```
std::auto_ptr<ClassA> f()
{
    std::auto_ptr<ClassA> ptr(new ClassA); // ptr owns the new object
    ...
    return ptr; // transfer ownership to calling function
}

void g()
{
    std::auto_ptr<ClassA> p;

    for (int i=0; i<10; ++i) {
        p = f(); // p gets ownership of the returned object
                // (previously returned object of f() gets deleted)
        ...
    }
    // last-owned object of p gets deleted
```

每當 `f()` 被呼叫，它都 `new` 一個新物件，然後把該物件連同其擁有權一起傳回給呼叫端。將返回值被指派給 `p`，同時也完成了擁有權的移轉。一旦迴圈再次執行這個指派動作，`p` 原先擁有的物件將被刪除。離開 `g()` 時，`p` 也會被銷毀，這樣就刪除了 `p` 所擁有的最後一個物件。無論如何都不會有資源遺失之虞。即使有異常被擲出，擁有資料的 `auto_ptr` 也會盡職地將自己的資料刪除。

缺陷

`auto_ptr` 的語義本身就涵蓋擁有權，所以如果你無意轉交你的擁有權，就不要在參數列中使用 `auto_ptr`，也不要它作為返回值。下面例子是一個幼稚的作法，原本是想將 `auto_ptr` 所指物件列印出來，實際上卻引發一場災難：

```
// this is a bad example
template <class T>
void bad_print(std::auto_ptr<T> p) // p gets ownership of passed argument
{
```

```

    // does p own an object ?
    if (p.get() == NULL) {
        std::cout << "NULL";
    }
    else {
        std::cout << *p;
    }
} // Oops, exiting deletes the object to which p refers

```

只要有一個 `auto_ptr` 被當做引數，放進這個 `bad_print()` 函式，它所擁有的物件（如果有的話）就一定會被刪除。因為作為引數的 `auto_ptr` 會將擁有權轉交給參數 `p`，而當函式退出時，會刪除 `p` 所擁有的物件。這恐怕不是程式員所希望的，最終必然會引起致命的執行期錯誤：

```

std::auto_ptr<int> p(new int);
*p = 42;           // change value to which p refers
bad_print(p);      // Oops, deletes the memory to which p refers
*p = 18;           // RUNTIME ERROR

```

你可能會認為，將 `auto_ptrs` 以 **pass by reference** 的方式傳遞就萬事大吉。然而這種行為卻會使「擁有權」的概念變得難以捉摸，因為面對一個「透過 **reference** 而獲得 `auto_ptr`」的函式，你根本無法預知擁有權是否被轉交。所以以 **by reference** 方式傳遞 `auto_ptr` 是非常糟糕的設計，應該全力避免。

考慮到 `auto_ptr` 的概念，我們倒是可以運用 **constant reference**，向函式傳遞擁有權。然而這十分危險，因為當你傳遞一個 **constant reference** 時，通常預期該物件不會被更動。幸好 `auto_ptrs` 的一個晚期設計降低了此一危險性。藉由某些實作技巧，我們可以令 **constant reference** 無法交出擁有權。事實上，你無法變更任何 **constant reference** 的擁有權：

```

const std::auto_ptr<int> p(new int);
*p = 42;           // change value to which p refers
bad_print(p);      // COMPILE-TIME ERROR
*p = 18;           // OK

```

這一方案使得 `auto_ptrs` 比以前顯得更安全一些。很多介面在需要內部拷貝時，都藉由 **constant reference** 獲得原值。事實上，C++ 標準程式庫的所有容器（例見第 6 章或第 10 章）都如此，大致像這樣：

```

template <class T>
void container::insert (const T& value)
{
    ...
    x = value; // assign or copy value internally
    ...
}

```

如果這一類賦值動作對 `auto_ptr` 有效，那麼擁有權就會被轉交給容器。然而正由於 `auto_ptr` 的實際設計，這種行為必然會導致編譯錯誤：

```
container<std::auto_ptr<int> > c;
const std::auto_ptr<int> p(new int);
...
c.insert(p); // ERROR
...
```

總而言之，常數型 `auto_ptr` 減小了「不經意轉移擁有權」所帶來的危險。只要一個物件藉由 `auto_ptr` 傳遞，就可以使用常數型 `auto_ptr` 來終結擁有權移轉鏈，此後擁有權將不能再進行移轉。

在這裡，關鍵字 `const` 並非意味你不能更改 `auto_ptr` 所擁有的物件，而是意味你不能更改 `auto_ptr` 的擁有權。例如：

```
std::auto_ptr<int> f()
{
    const std::auto_ptr<int> p(new int); // no ownership transfer possible
    std::auto_ptr<int> q(new int); // ownership transfer possible

    *p = 42; // OK, change value to which p refers
    bad_print(p); // COMPILE-TIME ERROR
    *p = *q; // OK, change value to which p refers
    p = q; // COMPILE-TIME ERROR
    return p; // COMPILE-TIME ERROR
}
```

如果使用 `const auto_ptr` 作為引數，對新物件的任何賦值（assign）動作都將導致編譯期錯誤。就常數特性而言，`const auto_ptr` 比較類似常數指標（`T* const p`），而非指向常數的指標（`const T* p`）——儘管其語法看上去比較像後者。

4.2.3 `auto_ptr` 作為成員變元

在 `class` 中使用 `auto_ptr`，你可以因而避免資源的遺失。如果你以 `auto_ptr` 而非一般指標作為成員，當物件被刪除時，`auto_ptr` 會自動刪除其所指的成員物件，於是你也就不再需要 `destructor` 了。此外，即使在初始化期間丟擲異常，`auto_ptr` 也可以幫助避免資源遺失。注意，只有當物件被完整建構成功，才有可能於將來呼叫其解構式。這造成了資源遺失的隱憂：如果第一個 `new` 成功了，第二個 `new` 卻失敗了，就會造成資源遺失。例如：

```
class ClassB {
private:
    ClassA* ptr1; // pointer members
    ClassA* ptr2;
public:
    // constructor that initializes the pointers
    // - will cause resource leak if second new throws
    ClassB (ClassA val1, ClassA val2)
        : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }

    // copy constructor
    // - might cause resource leak if second new throws
    ClassB (const ClassB& x)
        : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
    }

    // assignment operator
    const ClassB& operator= (const ClassB& x) {
        *ptr1 = *x.ptr1;
        *ptr2 = *x.ptr2;
        return *this;
    }

    ~ClassB () {
        delete ptr1;
        delete ptr2;
    }
    ...
};
```

使用 auto_ptr，你就可以輕鬆避免這場悲劇，：

```
class ClassB {
private:
    const std::auto_ptr<ClassA> ptr1; // auto_ptr members
    const std::auto_ptr<ClassA> ptr2;
public:
    // constructor that initializes the auto_ptr
    // - no resource leak possible
```

```

ClassB (ClassA val1, ClassA val2)
    : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {

// copy constructor
// - no resource leak possible
ClassB (const ClassB& x)
    : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {

// assignment operator
const ClassB& operator= (const ClassB& x) {
    *ptr1 = *x.ptr1;
    *ptr2 = *x.ptr2;
    return *this;
}

// no destructor necessary
// (default destructor lets ptr1 and ptr2 delete their objects)
...
};

```

然而請注意，儘管你可以略過 `destructor`，卻還是不得不親自撰寫 `copy construct` 和 `assignment operator`。預設狀況下，這兩個操作都會轉交擁有權，這恐怕並非你所願。正如 p42 所說，爲了避免擁有權的意外轉交，如果你的 `auto_ptr` 在整個生命期內都不必改變其所指物件的擁有權，你可以使用 `const auto_ptr`。

4.2.4 `auto_ptr` 的錯誤運用

`auto_ptr` 確實解決了一個特定問題，那就是在異常處理過程中資源遺失的問題。不幸的是由於 `auto_ptr` 的具體行爲方式曾經三番五次地改動，而且 C++ 標準程式庫中只此一個智慧型指標（`smart pointer`），別無分號，所以人們總是會誤用 `auto_ptr`。爲了幫助你正確使用它，這裏給出一些要點：

1. `auto_ptr` 之間不能共享擁有權

一個 `auto_ptr` 千萬不能指向另一個 `auto_ptr`（或其他物件）所擁有的物件。否則，當第一個指標刪除該物件後，另一個指標突然間指向了一個已被銷毀的物件，那麼，如果再透過那個指標進行讀寫操作，就會引發一場災難。

2. 並不存在針對 `array` 而設計的 `auto_ptr`s

`auto_ptr` 不可以指向 `array`，因為 `auto_ptr` 是透過 `delete` 而非 `delete[]` 來釋放其所擁有的物件。注意，C++ 標準程式庫並未提供針對 `array` 而設計的 `auto_ptr`。標準程式庫提供了數個容器類別，用來管理資料群集（參見第 5 章）。

3. `auto_ptr`s 決非一個「四海通用」的智慧型指標

並非任何適用智慧型指標的地方，都適用 `auto_ptr`。特別請注意的是，它不是參用計數（*reference counting*）型指標 — 這種指標保證，如果有一組智慧型指標指向同一個物件，那麼唯有（if and only if）當最後一個智慧型指標被銷毀時，該物件才會被銷毀。

4. `auto_ptr`s 不滿足 STL 容器對其元素的要求

`auto_ptr` 並不滿足 STL 標準容器對於元素的最基本要求，因為在拷貝（copy）和賦值（assign）動作之後，原本的 `auto_ptr` 和新產生的 `auto_ptr` 並不相等。是的，拷貝和賦值之後，原本的 `auto_ptr` 會交出擁有權，而不是拷貝給新的 `auto_ptr`。因此請絕對不要將 `auto_ptr` 作為標準容器的元素。幸好語言和程式庫的設計，本身就可以防止這種誤用，如果你的工作環境符合標準，這類誤用應該無法通過編譯。

不幸的是，某些時候，即使誤用 `auto_ptr`，程式仍然能夠順利運作。就此點而言，使用一個非常數（nonconstant）`auto_ptr`，並不比使用一個一般指標更安全。如果你的誤用行為沒有導致全盤崩潰，你或許會暗自慶幸，而這其實是真正的不幸，因為你或許根本就沒有意識到你已經犯了錯誤。關於「參用計數型（reference counting）智慧指標」的討論請見 p135, 5.10.2 節，p222, 6.8 節有一份實作碼。當我們有必要在不同容器之間共用元素時，這種指標非常有用。

4.2.5 `auto_ptr` 運用實例

下面第一個例子展示出 `auto_ptr`s 移轉擁有權的行為：

```
// util/autoptr1.cpp
#include <iostream>
#include <memory>
using namespace std;

/* define output operator for auto_ptr
 * - print object value or NULL
 */
template <class T>
ostream& operator<< (ostream& strm, const auto_ptr<T>& p)
{
    // does p own an object ?
    if (p.get() == NULL) {
```

```
        strm << "NULL";           // NO: print NULL
    }
    else {
        strm << *p;               // YES: print the object
    }
    return strm;
}

int main()
{
    auto_ptr<int> p(new int(42));
    auto_ptr<int> q;

    cout << "after initialization:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;

    q = p;
    cout << "after assigning auto pointers:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;

    *q += 13;           // change value of the object q owns
    p = q;
    cout << "after change and reassignment:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
}
```

程式輸出如下：

```
after initialization:
p: 42
q: NULL
after assigning auto pointers:
p: NULL
q: 42
after change and reassignment:
p: 55
q: NULL
```

注意，`output` 運算子的第二個參數是一個 `const reference`，所以並沒有發生擁有權的移轉。

正如我在 p40 所說，請時刻銘記於心，你不能以一般指標的賦值手法來初始化一個 `auto_ptr`：

```
std::auto_ptr<int> p(new int(42));           // OK
std::auto_ptr<int> p = new int(42);         // ERROR

p = std::auto_ptr<int>(new int(42));        // OK
p = new int(42);                           // ERROR
```

這是因為，「根據一般指標生成一個 `auto_ptr`」的那個 `constructor`，被宣告為 `explicit`（關於 `explicit`，詳見 p18, 2.2.6 節）。

下面這個例子展示 `const auto_ptr` 的特性：

```
// util/autoptr2.cpp

#include <iostream>
#include <memory>
using namespace std;

/* define output operator for auto_ptr
 * - print object value or NULL
 */
template <class T>
ostream& operator<< (ostream& strm, const auto_ptr<T>& p)
{
    // does p own an object ?
    if (p.get() == NULL) {
        strm << "NULL";    // NO: print NULL
    }
    else {
        strm << *p;        // YES: print the object
    }
    return strm;
}

int main()
{
    const auto_ptr<int> p(new int(42));
    const auto_ptr<int> q(new int(0));
    const auto_ptr<int> r;
```

```

    cout << "after initialization:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
    cout << " r: " << r << endl;

    *q = *p;
    // *r = *p;    // ERROR: undefined behavior
    *p = -77;
    cout << "after assigning values:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
    cout << " r: " << r << endl;

    // q = p;      // ERROR at compile time
    // r = p;      // ERROR at compile time
}

```

程式輸出如下：

```

after initialization:
p: 42
q: 0
r: NULL
after assigning values:
p: -77
q: 42
r: NULL

```

這個例子為 `auto_ptr` 定義了一個 `output` 運算子，其中將 `auto_ptr` 以 `const reference` 的方式傳遞。根據 p43 的討論，你不應該以任何形式傳遞 `auto_ptr`，但此處是個例外。

注意下列指派（賦值）動作是錯誤的：

```
*r = *p;
```

這個句子對於一個「未指向任何物件」的 `auto_ptr` 進行提領（*dereference*）動作。C++ 標準規格書宣稱，這會導致未定義行為，比如說導致程式的崩潰。從這個例子可以看出，你可以操作 `const auto_ptr` 所指物件本身，但「它所擁有的究竟是哪個物件」這一事實無法改變。就算 `r` 不具常數性，最後一個述句也會失敗，因為 `p` 具有常數性，其擁有權不得被更改。

4.2.6 auto_ptr 實作細目

class auto_ptr 宣告於 <memory> :

```
#include <memory>
```

auto_ptr 定義於 namespace std 中，是「可用於任何型別身上」的一個 template class。下面是 auto_ptr 的確切宣告⁶：

```
namespace std {  
    // auxiliary type to enable copies and assignments  
    template <class Y> struct auto_ptr_ref {};  
  
    template<class T>  
    class auto_ptr {  
    public:  
        // type names for the value  
        typedef T element_type;  
  
        // constructor  
        explicit auto_ptr(T* ptr = 0) throw();  
  
        // copy constructors (with implicit conversion)  
        // - note: nonconstant parameter  
        auto_ptr(auto_ptr&) throw();  
        template<class U> auto_ptr(auto_ptr<U>&) throw();  
  
        // assignments (with implicit conversion)  
        // - note: nonconstant parameter  
        auto_ptr& operator= (auto_ptr&) throw();  
        template<class U>  
        auto_ptr& operator= (auto_ptr<U>&) throw();  
  
        // destructor  
        ~auto_ptr() throw();  
  
        // value access  
        T* get() const throw();  
        T& operator*() const throw();  
        T* operator->() const throw();  
    };  
};
```

⁶ 這裡所給的版本較之 C++ 標準程式庫的版本作了一些小小改進，修正了幾個小問題（這裡的 auto_ptr_ref 是全域的，這裡並且設定了從 auto_ptr_ref 到 auto_ptr 的賦值運算子。請參見 p55）。

```

        // release ownership
        T* release() throw();

        // reset value
        void reset(T* ptr = 0) throw();

        // special conversions to enable copies and assignments
    public:
        auto_ptr(auto_ptr_ref<T>) throw();
        auto_ptr& operator= (auto_ptr_ref<T> rhs) throw();
        template<class U> operator auto_ptr_ref<U>() throw();
        template<class U> operator auto_ptr<U>() throw();
    };
}

```

個別成員的詳細描述將在後續數節中進行。討論過程中我把 `auto_ptr<T>` 簡寫為 `auto_ptr`。p56 有一份完整的 `auto_ptr` 實作範例。

型別定義

`auto_ptr::element_type`

- `auto_ptr` 所擁有之物件的型別

建構式 (constructor)、賦值運算子 (assign operator)、解構式 (destructor)

`auto_ptr::auto_ptr()` throw()

- 預設建構式 (default constructor)。
- 生成一個不擁有任何物件的 `auto_ptr`。
- 將 `auto_ptr` 的值初始化為零。

`explicit auto_ptr::auto_ptr(T* ptr) throw()`

- 生成一個 `auto_ptr`，並擁有 `ptr` 所指物件。
- 此一動作完成後，`*this` 成為 `ptr` 所指物件的唯一擁有者。不允許再有其他擁有者。

- 如果 `ptr` 本身不是 `null` 指標，那就必須是個 `new` 回返回值，因為 `auto_ptr` 解構式會對其所擁有的物件自動呼叫 `delete`。
- 不能用 `new[]` 所生成的 `array` 作為初值。當你需要 `array`，請考慮使用 STL 容器，p75, 5.2 節對此有些介紹。

```
auto_ptr::auto_ptr(auto_ptr& ap) throw()
template<class U> auto_ptr::auto_ptr(auto_ptr<U>& ap) throw()
```

- 針對 `non-const values` 而設計的一個 `copy constructor`。
- 生成一個 `auto_ptr`，在入口處將 `ap` 所擁有的物件（如果有的話）的擁有權奪取過來。
- 此操作完畢之後，`ap` 不再擁有任何物件，其值變為 `null` 指標。所以，和一般 `copy constructor` 不同，這個操作改變了原物件。
- 注意，此函式有一個多載化的 `member template`（關於 `member template`，請參考 p11），使得 `ap` 可藉由型別自動轉換，構造出合適的 `auto_ptr`。例如，根據一個「衍生類別的物件」，建構出一個基礎類別物件的 `auto_ptr`。
- 擁有權移轉問題，請參考 p40, 4.2.2 節。

```
auto_ptr& auto_ptr::operator=(auto_ptr& ap) throw()
template<class U> auto_ptr& auto_ptr::operator=(auto_ptr<U>& ap)
                                                    throw()
```

- 針對 `non-const value` 而設計的一個賦值（指派）運算子
- 如果自身原本擁有物件，進入本動作時將被刪除，然後獲得 `ap` 所擁有的物件。於是，原本 `ap` 所擁有的物件，其擁有權就移轉給了 `*this`。
- 此一動作完成後，`ap` 不再擁有任何物件。其值變為 `null` 指標。與一般賦值動作不同，此處這個動作改變了原物件。
- 左手邊的 `auto_ptr` 原本所指物件將被刪除（`deleted`）。
- 注意，此函式有一個多載化的 `member template`（關於 `member template`，詳見 p11）。這使得 `ap` 可藉由「型別自動轉換」指派給合適的 `auto_ptr`。例如，將一個「衍生類別的物件」，指派給一個基礎類別物件的 `auto_ptr`。
- 擁有權移轉問題，請參考 p40, 4.2.2 節。

```
auto_ptr::~~auto_ptr() throw()
```

- 解構式
- 如果 `auto_ptr` 擁有某個物件，此處將呼叫 `delete` 刪除之。

數值存取 (value access)

`T* auto_ptr::get() const throw()`

- 傳回 `auto_ptr` 所指物件的位址。
- 如果 `auto_ptr` 未指向任何物件，傳回 `null` 指標。
- 這個動作並不改變擁有權。退出此函式時，`auto_ptr` 仍然保有對物件（如果有的話）的擁有權。

`T& auto_ptr::operator*() const throw()`

- 提領運算子（dereference operator）
- 傳回 `auto_ptr` 所擁有的物件。
- 如果 `auto_ptr` 並未擁有任何物件，此呼叫導致未定義行為（可能導致崩潰）。

`T* auto_ptr::operator->() const throw()`

- 成員存取運算子（member access operator）
- 返回 `auto_ptr` 所擁有的物件中的一個成員。
- 如果 `auto_ptr` 並未擁有任何物件，此呼叫導致未定義行為（可能導致崩潰）。

數值操作

`T* auto_ptr::release() throw()`

- 放棄 `auto_ptr` 原先所擁有之物件的擁有權。
- 傳回 `auto_ptr` 原先擁有物件（如果有的話）的位址。
- 如果 `auto_ptr` 原先並未擁有任何物件，傳回 `null` 指標。

`void auto_ptr::reset(T* ptr = 0) throw()`

- 以 `ptr` 重新初始化 `auto_ptr`。
- 如果 `auto_ptr` 原本擁有物件，則此動作開始時先刪除之。
- 呼叫結束後，`*this` 成為 `ptr` 所指物件的擁有者。注意，不應該有任何其他擁有者。
- 如果 `ptr` 不是 `null` 指標，應當是一個由 `new` 傳回的值，因為 `auto_ptr` 的解構式（`dtor`）會呼叫 `delete` 來刪除其所擁有的物件。
- 注意，不得將透過 `new[]` 生成的 `array` 當作引數傳進來。如果需要使用 `array`，請考慮使用 STL 容器類別，詳見 p75, 5.2 節。

Conversions 轉型操作

`auto_ptr` 中剩餘的內容（輔助型別 `auto_ptr_ref`，及其相關函式）涉及非常精緻的技巧，使我們得以拷貝和指派 `non-const auto_ptr`s，卻不能拷貝和指派 `const auto_ptr`s（詳見 p44）。下面是一份扼要解釋⁷。我們有兩個需求：

1. 我們需要將 `auto_ptr` 作為右值（*rvalue*）傳遞到函式去，或由函式中傳回⁸。
由於 `auto_ptr` 是個類別，所以這些工作應當由建構式完成。
2. 拷貝 `auto_ptr` 時，原指標務必放棄擁有權。這就要求拷貝動作必須修改原本的那個 `auto_ptr`。

一般的 `copy constructor` 當然可以拷貝右值，但為了做到這點，它必須將其參數型別宣告為一個 *reference to const object*。如果在 `auto_ptr` 中使用一般的 `copy constructor`，我們恐怕不得不將 `auto_ptr` 內含的實際指標宣告為 `mutable`，只有這樣，才能在 `copy constructor` 中更改它。你以為萬事大吉了嗎？錯，這種做法將允許用戶拷貝那些宣告為 `const` 的物件，將其擁有權轉交他人，這與其原本的常數性背道而馳。

變通作法是找出一種機制，能夠將右值轉化為左值。「直接轉型為 *reference*」的那種簡單的轉型運算子派不上用場，因為當你實際上是把一個物件轉化為自己原本型別，不會有任何轉型操作被喚起（切記，*reference* 屬性並非型別特性的一部份）。為此才有了 `auto_ptr_ref` 類別的引進，協助我們將右值轉化為左值。這一機制的理論基礎，是「多載化」與「*template* 引數推導規則」之間一個細微的不同處。這個差別實在太細微了，不大可能成為一般程式設計技巧，用於別處，但卻足以在這裡讓 `auto_ptr` 正確運作。這就夠了。

如果你的編譯器對於 `non-const` 和 `const auto_ptr`s 之間的區別尚不能做出很好的闡釋，請不必驚訝。但是請你保持清醒的頭腦，如果你的編譯器尚未達到這一水平，那麼 `auto_ptr` 的使用就會變得更加危險。因為這種情況下很容易意外地將所有權旁落他人之手。

⁷ 感謝 Bill Gibbons 指出這一點。

⁸ *rvalue*（右值）和 *lvalue*（左值）的名稱由來，是從賦值運算 `expr1 = expr2` 得來。在這種運算式中，左運算元 `expr1` 必須是一個（可更改的）*lvalue*。不過或許更貼切的描述是：*lvalue* 代表 *locator value*。也就是說，這個算式藉由名字和位址（*pointer* 或 *reference*）來指定一個物件。*lvalue* 並非一定「可被更改」。例如常數物件的名字就是一個不可被改動的 *lvalue*。所有 *non-lvalues* 物件，都是 *rvalues*。尤其顯式生成（`T()`）的暫時物件和函式返回值，都是 *rvalue*。

類別 `auto_ptr` 的實作範例

下面的源碼展示了一個符合標準的 `auto_ptr` 類別的實作範例⁹：

```
// util/autoptr.hpp

/* class auto_ptr
 * - improved standard conforming implementation
 */
namespace std {
    // auxiliary type to enable copies and assignments (now global)
    template<class Y>
    struct auto_ptr_ref {
        Y* yp;
        auto_ptr_ref (Y* rhs)
            : yp(rhs) {
        }
    };

    template<class T>
    class auto_ptr {
    private:
        T* ap; // refers to the actual owned object (if any)
    public:
        typedef T element_type;

        // constructor
        explicit auto_ptr (T* ptr = 0) throw()
            : ap(ptr) {
        }

        // copy constructors (with implicit conversion)
        // - note: nonconstant parameter
        auto_ptr (auto_ptr& rhs) throw()
            : ap(rhs.release()) {
        }
    };
}
```

⁹ 感謝 Greg Colvin 提供的這個 `auto_ptr` 實作內容。注意，這個實作版本並不完全契合 C++ 標準。事實證明，C++ 標準所規定的形式中，當利用 `auto_ptr_ref` 進行轉型時，在某種特殊情況下仍會出現小小瑕疵。這裡所給的方案很有可能徹底解決所有問題。不過，撰寫本書的時候，仍有一些相關討論正在進行。

```
template<class Y>
auto_ptr (auto_ptr<Y>& rhs) throw()
    : ap(rhs.release()) {
}

// assignments (with implicit conversion)
// - note: nonconstant parameter
auto_ptr& operator= (auto_ptr& rhs) throw() {
    reset(rhs.release());
    return *this;
}
template<class Y>
auto_ptr& operator= (auto_ptr<Y>& rhs) throw() {
    reset(rhs.release());
    return *this;
}

// destructor
~auto_ptr() throw() {
    delete ap;
}

// value access
T* get() const throw() {
    return ap;
}
T& operator*() const throw() {
    return *ap;
}
T* operator->() const throw() {
    return ap;
}

// release ownership
T* release() throw() {
    T* tmp(ap);
    ap = 0;
    return tmp;
}
```

```
// reset value
void reset (T* ptr=0) throw() {
    if (ap != ptr) {
        delete ap;
        ap = ptr;
    }
}

/* special conversions with auxiliary type to enable copies and
assignments
*/
auto_ptr(auto_ptr_ref<T> rhs) throw()
    : ap(rhs.yp) {
}
auto_ptr& operator= (auto_ptr_ref<T> rhs) throw() { // new
    reset(rhs.yp);
    return *this;
}
template<class Y>
operator auto_ptr_ref<Y>() throw() {
    return auto_ptr_ref<Y>(release());
}
template<class Y>
operator auto_ptr<Y>() throw() {
    return auto_ptr<Y>(release());
}
};
}
```

4.3 數值極限 (Numeric Limits)

一般說來，數值型別 (Numeric types) 的極值是一種與平台相依的特性。C++ 標準程式庫藉由 `template numeric_limits` 提供這些極值，取代傳統 C 語言所採用的預處理器常數 (preprocessor constants)。你仍然可以使用後者，其中整數常數定義於 `<climits>` 和 `<limits.h>`，浮點常數定義於 `<cfloat>` 和 `<float.h>`。新的極值概念有兩個優點，第一是提供了更好的型別安全性，第二是程式員可藉此寫出一些 templates 以核定 (evaluate) 這些極值。

本節的剩餘部份專門討論極值問題。注意，C++ 標準規格書中規定了各種型別必須保證的最小精度，如果你能夠注意並運用這些極值，就比較容易寫出與平台無關的程式。這些最小值列於表 4.1。

| 型別 | 最小長度 |
|-------------|-----------------|
| char | 1 byte (8 bits) |
| short int | 2 bytes |
| int | 2 bytes |
| long int | 4 bytes |
| float | 4 bytes |
| double | 8 bytes |
| long double | 8 bytes |

表 4.1 內建型別的最小長度

Class `numeric_limits<>`

使用 `template`，通常是為了對所有型別一次性地撰寫出一個通用解決方案。除此之外，你還可以在必要時候以 `template` 為每個型別提供共同介面。方法是：不但提供通用性的 `template`，還提供其特化 (specialization) 版本。`numeric_limits` 就是這項技術的一個典型例子，作法如下：

- 通用性的 `template`，為所有型別提供預設極值：

```
namespace std {
    /* general numeric limits as default for any type
    */
    template <class T>
    class numeric_limits {
    public:
        // no specialization for numeric limits exist
        static const bool is_specialized = false;
        ... // other members that are meaningless for the general numeric limits
    };
}
```

這個通用性 `template` 將成員 `is_specialized` 設為 `false`，意思是，對型別 `T` 而言，無所謂極值的存在。

- 各具體型別的極值，由特化版本（specialization）提供：

```
namespace std {
    /* numeric limits for int
     * - implementation defined
     */
    template<> class numeric_limits<int> {
    public:
        // yes, a specialization for numeric limits of int does exist
        static const bool is_specialized = true;

        static T min() throw() {
            return -2147483648;
        }
        static T max() throw() {
            return 2147483647;
        }
        static const int digits = 31;
        ...
    };
}
```

這裡把 `is_specialized` 設為 `true`，所有其他成員都根據特定型別的具體極值加以設定。

通用性的 `numeric_limits` `template`，及其特化版本都被放在 `<limits>` 表頭檔中。C++ 標準規格所囊括的特化版本，涵蓋了所有數值基本型別，包括：`bool`, `char`, `signed char`, `unsigned char`, `wchar_t`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`。你可以輕易為你自定的數值型別加上補充。

表 4.2 和 4.3 列出 `class numeric_limits<>` 的所有成員及其意義。在最右面的一列顯示對應的 C 常數，它們分別定義於 `<climits>`, `<limits.h>`, `<cfloat>`, `<float.h>` 之中。

| 成員 | 意義 | 對應 C 常數 |
|--------------------------------|---|--|
| <code>is_specialized</code> | 型別是否有極值 | |
| <code>is_signed</code> | 型別帶有正負號 | |
| <code>is_integer</code> | 整數型別 | |
| <code>is_exact</code> | 計算結果不產生捨入誤差（此成員對所有整數型別而言均為 <code>true</code> ） | |
| <code>is_bounded</code> | 數值集的個數有限（對所有內建型別而言，此成員均為 <code>true</code> ） | |
| <code>is_modulo</code> | 兩正值相加，其結果可能因溢位而回繞為較小的值。 | |
| <code>is_iec559</code> | 遵從 IEC 559 及 IEEE 754 標準 | |
| <code>min()</code> | 最小值（對浮點數而言，是標準化後的值；只有當 <code>is_bounded</code> 成立時才有意義） | <code>INT_MIN, FLT_MIN, CHAR_MIN, ...</code> |
| <code>max()</code> | 最大值（只有當 <code>is_bounded</code> 成立時才有意義） | <code>INT_MAX, FLT_MAX, ...</code> |
| <code>digits</code> | 字元和整數：不帶正負號之位元的個數。 | <code>CHAR_BIT</code> |
| | 浮點數：尾數中之 radix（見下）位元的個數。 | <code>FLT_MANT_DIG, ...</code> |
| <code>digits10</code> | 十進位數的個數（只有當 <code>is_bounded</code> 成立時才有意義） | <code>FLT_DIG, ...</code> |
| <code>radix</code> | 整數：表示式的基底（base），幾乎總是 2。 | |
| | 浮點數：指數表示式的基底（base） | <code>FLT_RADIX</code> |
| <code>min_exponent</code> | Minimum negative integer exponent for base radix | <code>FLT_MIN_EXP, ...</code> |
| <code>max_exponent</code> | Maximum positive integer exponent for base radix | <code>FLT_MAX_EXP, ...</code> |
| <code>min_exponent10</code> | Minimum negative integer exponent for base 10 | <code>FLT_MIN_10_EXP, ...</code> |
| <code>max_exponent10</code> | Maximum positive integer exponent for base 10 | <code>FLT_MAX_10_EXP, ...</code> |
| <code>epsilon()</code> | Difference of one and least value greater than one | <code>FLT_EPSILON, ...</code> |
| <code>round_style</code> | 捨入（rounding）風格（見 p63） | |
| <code>round_error()</code> | 最大捨入誤差量測（根據 ISO/IEC 10967-1 標準） | |
| <code>has_infinity</code> | 有「正無窮大」表示式 | |
| <code>infinity()</code> | 表現出「正無窮大」（如有的話） | |
| <code>has_quiet_NaN</code> | Type has representation for nonsignaling "Not a Number" | |
| <code>quiet_NaN</code> | Representation of quiet "Not a Number" if available | |
| <code>has_signaling_NaN</code> | Type has representation for signaling "Not a Number" | |
| <code>signaling_NaN()</code> | Representation of signaling "Not a Number" if available | |

| | |
|-----------------|--|
| has_denorm | Whether type allows denormalized values (variable members of exponent bits, see page 63) |
| has_denorm_loss | Loss of accuracy is detected as a denormalization rather than as an inexact result |
| denorm_min | Minimum positive denormalized value |
| traps | Trapping is implemented |
| tinyness_before | Tinyness is detected before rounding |

表 4.2. class numeric_limits<> 的所有成員

譯註 1：本表格「意義」欄中的諸多解釋，涉及專業之數位表示法專業術語。譯者這方面的能力有限，沒能把握正確譯出。為免誤導，部份欄位保留原文，望諒。

譯註 2：本表格在英文版中因分頁而分為表 4.2 與表 4.3。中文版合而為一。後續表格之編號從表 4.4 開始。

下面是對於 float 型別的數值限定模板特殊化的一個完全實作版，當然是平臺相關的。這裏同時還給出了各成員的確切簽名(signatures):

```
namespace std {
    template<> class numeric_limits<float> {
    public:
        // yes, a specialization for numeric limits of float does exist
        static const bool is_specialized = true;

        inline static float min() throw() {
            return 1.17549435E-38F;
        }
        inline static float max() throw() {
            return 3.40282347E+38F;
        }

        static const int digits = 24;
        static const int digits10 = 6;

        static const bool is_signed = true;
        static const bool is_integer = false;
        static const bool is_exact = false;
        static const bool is_bounded = true;
        static const bool is_modulo = false;
        static const bool is_iec559 = true;

        static const int radix = 2;

        inline static float epsilon() throw() {
```



```

        return 1.19209290E-07F;
    }

    static const float_round_style round_style
        = round_to_nearest;
    inline static float round_error() throw() {
        return 0.5F;
    }

    static const int min_exponent = -125;
    static const int max_exponent = +128;
    static const int min_exponent10 = -37;
    static const int max_exponent10 = +38;

    static const bool has_infinity = true;
    inline static float infinity() throw() { return ...; }
    static const bool has_quiet_NaN = true;
    inline static float quiet_NaN() throw() { return ...; }
    static const bool has_signaling_NaN = true;
    inline static float signaling_NaN() throw() { return ...; }
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;
    inline static float denorm_min() throw() { return min(); }

    static const bool traps = true;
    static const bool tinyness_before = true;
};
}

```

注意，所有 `data members` 如果不是 `const`，便是 `static`，這麼一來其值便可在編譯期間確定。至於由函式所定義的 `member`，在某些編譯器中恐怕無法在編譯期間確定其值。因此，同一份目的碼（`object code`）雖然可以在不同的處理器上執行，但可能會有不同的浮點值。

`round_style` 的值列於表 4.4，`has_denorm` 的值列於表 4.5。`has_denorm` 其實也許應該稱為 `denorm_style` 更貼切，可惜並非如此。這是因為 C++ 標準化後期才決定將這裡原本的 `bool` 型別改變為列舉值之故。不過你還是可以把 `has_denorm` 當成 `bool` 值來用，C++ 標準規格保證，如果 `denorm_absent` 為 0，就等於 `false`，如果 `denorm_present` 為 1 而且 `denorm_indeterminate` 為 -1，那麼兩者都等於 `true`。因此你可以把 `has_denorm` 視為一個 `bool` 值，用以判定某個型別是否允許所謂的 "denormalized values"。

| 捨入風格 | 意義 |
|---------------------------|-------------|
| round_toward_zero | 向零捨入 |
| round_to_nearest | 向最接近的可表示值捨入 |
| round_toward_infinity | 向正無限捨入 |
| round_toward_neg_infinity | 向負無限捨入 |
| round_indeterminate | 不確定 |

表 4.4. numeric_limits<> 的捨入 (round) 風格

| 捨入風格 | 意義 |
|----------------------|-------------------------------------|
| denorm_absent | 此型別不允許 denormalized values |
| denorm_present | 此型別允許向最接近的可表示值做 denormalized values |
| denorm_indeterminate | 不確定 |

表 4.5. numeric_limits<> 的 "denormalization style"

numeric_limits<> 使用範例

下面的例子展示了某些型別極值的可能運用，例如用來瞭解某個型別的最大值，或確定 char 是否帶正負號：

```
// util/limits1.cpp

#include <iostream>
#include <limits>
#include <string>
using namespace std;
int main()
{
    // use textual representation for bool
    cout << boolalpha;

    // print maximum of integral types
    cout << "max(short): " << numeric_limits<short>::max() << endl;
    cout << "max(int): " << numeric_limits<int>::max() << endl;
    cout << "max(long): " << numeric_limits<long>::max() << endl;
    cout << endl;
```

```
// print maximum of floating-point types
cout << "max(float): "
    << numeric_limits<float>::max() << endl;
cout << "max(double): "
    << numeric_limits<double>::max() << endl;
cout << "max(long double): "
    << numeric_limits<long double>::max() << endl;
cout << endl;

// print whether char is signed
cout << "is_signed(char): "
    << numeric_limits<char>::is_signed << endl;
cout << endl;

// print whether numeric limits for type string exist
cout << "is_specialized(string): "
    << numeric_limits<string>::is_specialized << endl;
}
```

程式的輸出結果與執行平台有關，下面是其中一種可能：

```
max(short): 32767
max(int): 2147483647
max(long): 2147483647

max(float): 3.40282e+38
max(double): 1.79769e+308
max(long double): 1.79769e+308

is_signed(char): false

is_specialized(string): false
```

最後一行表示，型別 `string` 並沒有定義數值極限。這是理所當然的，因為 `strings` 並非數值型別。正如本例所示，你可以對任何型別進行詢問，無論它是否定義了極限值。

4.4 輔助函式

演算法程式庫（定義於表頭檔 `<algorithm>`）中包含了三個輔助函式，一個用來在兩值之中挑選較大者，另一個用來在兩值之中挑選較小者，第三個用來交換兩值。

4.4.1 挑選較小值和較大值

「在兩物之間選擇較大值和較小值」的函式定義於 `<algorithm>`，如下所示：

```
namespace std {
    template <class T>
    inline const T& min (const T& a, const T& b) {
        return b < a ? b : a;
    }

    template <class T>
    inline const T& max (const T& a, const T& b) {
        return a < b ? b : a;
    }
}
```

如果兩值相等，通常會傳回第一值。不過你的程式最好不要依賴這一點。

上述兩個函式還有另一個版本，接受一個額外的 `template` 引數作為「比較準則」：

```
namespace std {
    template <class T, class Compare>
    inline const T& min (const T& a, const T& b, Compare comp) {
        return comp(b,a) ? b : a;
    }

    template <class T, class Compare>
    inline const T& max (const T& a, const T& b, Compare comp) {
        return comp(a,b) ? b : a;
    }
}
```

作為「比較準則」的那個引數應該是個函式或仿函式（仿函式曾於 5.9 節，p124 中介紹），接受兩個引數並進行比較：在某種特定規則下，判斷第一引數是否小於第二引數，並傳回判斷結果。

下面這個例子示範如何傳入特定的比較函式作為引數，以此方式來使用 `max()`：

```
// util/minmax1.cpp

#include <algorithm>
using namespace std;

/* function that compares two pointers by comparing the values to which
they point
*/
bool int_ptr_less (int* a, int* b)
{
    return *a < *b;
}

int main()
{
    int x = 17;
    int y = 42;
    int* px = &x;
    int* py = &y;
    int* pmax;

    // call max() with special comparison function
    pmax = max (px, py, int_ptr_less);
    ...
}
```

注意，`min()`和`max()`都要求它們所接受的兩個引數的型別必須一致。如果不一致，你將無法正確呼叫之：

```
int i;
long l;
...
l = std::max(i,l); // ERROR: argument types don't match
```

不過你倒是可以明白地宣告引數型別（這樣也就確定了回返值的型別）：

```
l = std::max<long>(i,l); // OK
```

4.4.2 值交換

函式 `swap()` 用以交換兩物件的值。其泛用型實作碼定義於 `<algorithm>`：

```

namespace std {
    template<class T>
    inline void swap(T& a, T& b) {
        T tmp(a);
        a = b;
        b = tmp;
    }
}

```

運用這個函式，你可以如此交換任意兩個變數 *x* 和 *y* 的值：

```
std::swap(x, y);
```

當然啦，只有當 `swap()` 所依賴的拷貝建構操作（`copy constructions`）和賦值操作（`assignments`）存在時，這個呼叫才可能有效。

`swap()` 的最大優勢在於，透過 `template specialization`（模板特化）或 `function overloading`（函式多載化），我們可以為更複雜的型別提供特殊的實作版本；我們可以交換物件內部成員，不必勞師動眾地反復指派（賦值），這無疑將大大地節約時間。標準程式庫中的所有容器（6.1.2 節，p147）以及 `strings`（11.2.8 節，p490），都運用了這項技術。舉個例子，有個簡單容器，僅僅內含一個 `array` 和一個成員（用來指示 `array` 元素個數量），那麼為它特別實作的 `swap()` 可以是這樣：

```

class MyContainer {
private:
    int* elems;        // dynamic array of elements
    int numElems;      // number of elements
public:
    ...
    // implementation of swap()
    void swap(MyContainer& x) {
        std::swap(elems, x.elems);
        std::swap(numElems, x.numElems);
    }
    ...
};

// overloaded global swap() for this type
inline void swap (MyContainer& c1, MyContainer& c2)
{
    c1.swap(c2); // calls implementation of swap()
}

```

你瞧，呼叫 `swap()` 而非直接透過反覆指派（賦值）動作來交換兩容器的值，會帶來效率上的提升。對於你自己定義的型別，如果確實能夠帶來效率上的改善，你就應該義不容辭地為它提供 `swap()` 特化版本。

4.5 輔助性「比較運算子」 (Comparison Operators)

有四個 `template functions`，分別定義了 `!=`, `>`, `<=`, `>=` 四個比較運算子。它們都是利用 `==` 運算子和 `<` 運算子完成的。這四個函式定義於 `<utility>`：

```
namespace std {
    namespace rel_ops {
        template <class T>
        inline bool operator!= (const T& x, const T& y) {
            return !(x == y);
        }

        template <class T>
        inline bool operator> (const T& x, const T& y) {
            return y < x;
        }

        template <class T>
        inline bool operator<= (const T& x, const T& y) {
            return !(y < x);
        }

        template <class T>
        inline bool operator>= (const T& x, const T& y) {
            return !(x < y);
        }
    }
}
```

只需定義 `<` 和 `==` 運算子，你就可以使用它們。只要加上 `using namespace std::rel_ops`，上述四個比較運算子就自動獲得了定義。例如：

```
#include <utility>

class X {
    ...
public:
```

```

        bool operator== (const X& x) const;
        bool operator< (const X& x) const;
        ...
};

void foo()
{
    using namespace std::rel_ops; // make !=, >, etc., available
    X x1, x2;
    ...

    if (x1 != x2) {
        ...
    }
    ...

    if (x1 > x2) {
        ...
    }
    ...
}

```

注意，這些運算子都定義於 `std` 的次命名空間（sub-namespace）`rel_ops` 中。之所以如此安排，是爲了防止和用戶所（可能）定義的全域命名空間中的同類形運算子發生衝突。於是，就算你這樣使用 `using directive`：

```
using namespace std; // operators are not in global scope
```

因而把 `std` 的全部識別字引入全域命名空間，也沒問題。

另一方面，那些想向 `rel_ops` 借一臂之力的用戶可以這麼做：

```
using namespace std::rel_ops; // operators are in global scope
```

於是四個新的運算子就輕鬆到手了，無需使用複雜的搜尋規則來引用它們。

某些實作版本採用兩個不同的引數型別來定義上述 `template`：

```

namespace std {
    template <class T1, class T2>
    inline bool operator!=(const T1& x, const T2& y) {
        return !(x == y);
    }
    ...
}

```


這麼做的好處是，兩個運算元的型別可以不同（只要它們之間「可以比較」就行）。但這並非 C++ 標準程式庫所支援的作法。所以，如果想占這個便宜，就得付出可攜性方面的代價。

4.6 表頭檔 <cstdlib> 和 <stdlib>

表頭檔 <cstdlib> 和 <stdlib> 與 C 的對應版本相容，在 C++ 程式中經常用到。它們是 C 表頭檔 <stdlib.h> 和 <stdlib.h> 的較新版本，定義了一些常用的常數、巨集、型別和函式。

4.6.1 <cstdlib> 的各種定義

表 4.6 列出表頭檔 <cstdlib> 的各個定義項。NULL 通常用來表明一個不指向任何物件的指標，其實就是 0（其型別可以是 int，也可以是 long）。注意，C 語言中的 NULL 通常定義為 (void*)0。在 C++ 中這並不正確，NULL 的型別必須是個整數型別，否則你無法將 NULL 指派給一個指標。這是因為 C++ 並沒有定義從 void* 到任何其他型別的自動轉型操作¹⁰。NULL 同時也定義於表頭檔 <stdio>, <stdlib>, <string>, <ctime>, <wchar>, <locale>。

| 識別字 | 意義 |
|-----------|-------------------------------|
| NULL | 指標值，用來表示「未定義」或「無值」。 |
| size_t | 一種無正負號的型別，用來表示大小（例如元素個數）。 |
| ptrdiff_t | 一種帶有正負號的型別，用來表示指標之間的距離。 |
| offsetof | 表示一個成員在 struct 或 union 中的偏移量。 |

表 4.6 <cstdlib> 中的定義項

4.6.2 <stdlib> 的各種定義

表 4.7 列出表頭檔 <stdlib> 之中最重要的一些定義項。常數 EXIT_SUCCESS 和 EXIT_FAILURE 用來當做 exit() 的引數，也可以當做 main() 的返回值。

經由 atexit() 登錄的函式，在程式正常退出時會依登錄的相反次序被一一呼叫起來。無論是透過 exit() 退出或從 main() 尾部退出，都會如此，不傳遞任何引數。

¹⁰ 鑒於 NULL 型別有這些晦澀的問題，有人建議 C++ 中最好不要使用 NULL，最好直接使用 0 或用用戶自定的（例如）NIL 常數。不過我還是使用 NULL，所以本書範例程式中你還是可以看到它的蹤跡。

| 定義 | 意義 |
|--|--|
| <code>exit(int status)</code> | 退出（離開， <code>exit</code> ）程式（並清理 <code>static</code> 物件） |
| <code>EXIT_SUCCESS</code> | 程式正常結束。 |
| <code>EXIT_FAILURE</code> | 程式不正常結束。 |
| <code>abort()</code> | 退出程式（在某些系統上可能導致崩潰）。 |
| <code>atexit (void (*function)())</code> | 退出（ <code>exit</code> ）程式時呼叫某些函式。 |

表 4.7 `<cstdlib>` 中的定義項

函式 `exit()` 和 `abort()` 可用來在任意地點終止程式運行，無需返回 `main()`：

- `exit()` 會銷毀所有靜態（`static`）物件，將所有緩衝區（`buffer`）清空（`flushes`），關閉所有 I/O 通道（`channels`），然後終止程式（之前會先呼叫經由 `atexit()` 登錄的函式）。如果 `atexit()` 登錄的函式擲出異常，就會喚起 `terminate()`。
- `abort()` 會立刻終止函式，不做任何清理工作。

這兩個函式都不會銷毀區域物件（`local objects`），因為堆疊輾轉開展動作（`stack unwinding`）不會被執行起來。為確保所有區域物件的 `destructor`（解構式）獲得呼叫，你應該運用異常（`exceptions`）或正常回返機制，然後再由 `main()` 離開。

5

Standard Template Library

標準模板庫

STL (標準模板庫) 是 C++ 標準程式庫的核心，它深刻影響了標準程式庫的整體結構。STL 是一個泛型 (generic) 程式庫，提供一系列軟體方案，利用先進、高效的演算法來管理資料。程式員無需了解 STL 的原理，便可享受資料結構和演算法領域中的這一革新成果。從程式員的角度看來，STL 是由一些可適應不同需求的群集類別 (collection classes)，和一些能夠在這些資料群集上運作的演算法構成。STL 內的所有組件都由 templates (模板) 構成，所以其元素可以是任意型別。更妙的是，STL 建立了一個架構，在此架構下，你可以提供其他的群集類別或演算法，與現有的組件搭配，共同運作。總之，STL 賦予 C++ 新的抽象化層次。把 dynamic arrays (動態陣列)、linked list (串列)、binary trees (二元樹) 之類的東西拋開吧，也不用再操心不同的搜尋演算法了。你只需使用恰當的群集類別，然後呼叫其成員函式和 (或) 演算法來處理資料，就萬事大吉。當然，如此的靈活性並非免費午餐，代價總是有的。首要的一點是，STL 並不好懂。也正因為如此，本書傾注了好幾章篇幅的努力，為你講解 STL 的內容。這一章介紹 STL 的總體概念，探討其使用技術。第一個範例展示如何使用 STL，運用過程中有何考量。第 6 章至第 9 章詳細討論 STL 的各個組件 (包括容器 containers、迭代器 iterators、仿函式 functors, function objects、演算法 algorithms)，並提供更多範例。

5.1 STL 組件 (components)

若干精心勾畫的組件共同合作，構築了 STL 的基礎。這些組件中最關鍵的是容器、迭代器和演算法。

- 容器 **Containers**，用來管理某類物件的集合。每一種容器都有其優點和缺點，所以，為了應付程式中的不同需求，STL 準備了不同的容器類型。容器可以是 arrays 或是 linked lists，或者每個元素有一個特別的鍵值 (key)。
- 迭代器 **Iterators**，用來在一個物件群集 (collection of objects) 的元素上進行

走訪 (step through) 動作。這個物件群集或許是個容器，或許是容器的一部分。迭代器的主要好處是，為所有容器提供了一組很小的公共介面。利用這個介面，某個操作 (operations) 就可以行進至群集內的下一個元素。至於如何做到，當然取決於該群集的內部結構。不論這個群集是 array 或是 tree，此一行進動作都能成功。為什麼？因為每一種容器都提供了自己的迭代器，而這些迭代器了解該種容器的內部結構，所以能夠知道如何正確行進。

迭代器的介面和一般指標差不多，以 `operator++` 累進，以 `operator*` 提領所指之值。所以，你可以把迭代器視為一種 *smart pointer*，能夠把「前進至下一個元素」的意圖轉換成合適的操作。

- 演算法 **Algorithms**，用來處理群集內的元素。它們可以出於不同的目的而搜尋、排序、修改、使用那些元素。透過迭代器的協助，我們只需撰寫一次演算法，就可以將它應用於任意容器之上，這是因為所有容器的迭代器都提供一致的介面。你還可以提供一些特殊的輔助性函式供演算法呼叫，從而獲取最佳的靈活性。這樣你就可以運用標準演算法並適應自己特殊或複雜的需求。例如，你可以提供自己的搜尋準則，或者合併元素時的特殊操作。

STL 的基本觀念就是將資料和操作分離。資料由容器類別加以管理，操作則由可定制 (configurable) 之演算法定義。迭代器在這兩者之間充當粘合劑，使任何演算法都可以和任何容器交互運作 (圖 5-1)。

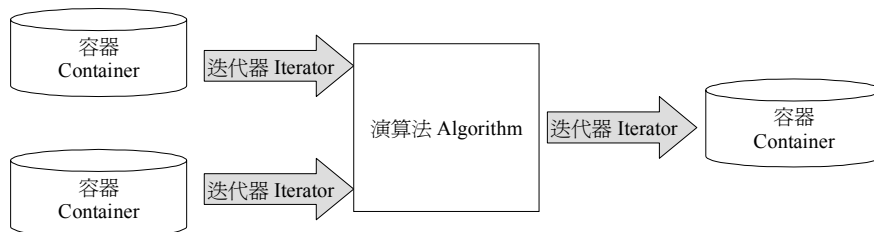


圖 5.1 STL 組件之間的合作

STL 將資料和演算法分開對待，而不是合併考慮。因此從某種意義上說，STL 的概念與物件導向程式設計 (OOP) 的最初思想是矛盾的。然而這麼做有著很重要的原因。首先，你可以將各種容器與各種演算法結合起來，在很小的框架 (framework) 內達成非常大的彈性。

STL 的一個根本特性是，所有組件都可以針對任意型別 (types) 運作。顧名思義，*standard template library* 意味其內的所有組件都是「可接受任意型別」的 *templates*，前提是這些型別必須能夠執行必要的操作。因此，STL 成了泛型程式設計 (*generic*

programming) 概念下的一個出色範例。容器和演算法對任意型別 (types) 和類別 (classes) 而言，都已經被一般化了。

STL 甚至提供更泛型化的組件。藉由特定的配接器 (*adapters*) 和仿函式 (*functors*，或稱 *function objects*)，你可以補充、約束或定制演算法，以滿足特別需求。目前說這些似乎為時太早，眼下我還是先透過實例，循序漸進地講解概念，這才是理解並熟悉 STL 的最佳方法。

5.2 容器 (Containers)

容器類別，或簡稱容器，用來管理一組元素。為了適應不同需要，STL 提供了不同的容器，如圖 5.2。

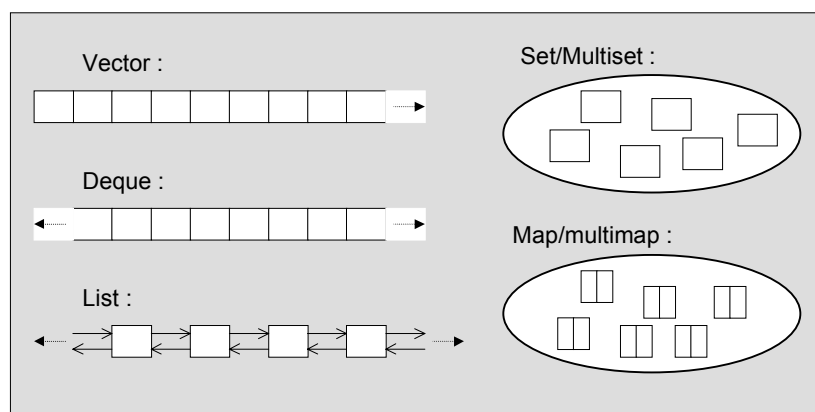


圖 5.2 STL 的容器種類

總的來說，容器可分為兩類：

1. 序列式容器 **Sequence containers**，此乃有序集合，其中每個元素均有固定位置，取決於插入的時機和地點，和元素值無關。如果你以附加方式對一個群集置入六個元素，它們的排列次序將和置入次序一致。STL 提供三個定義好的序列式容器：`vector`，`deque`，`list`。
2. 關聯式容器 **Associative containers**，這也是有序集合，元素位置取決於特定的排序準則。如果你將六個元素置入這樣的群集中，它們的位置取決於其值，和插入次序無關。STL 提供了四個關聯式容器：`set`，`multiset`，`map`，`multimap`。

關聯式容器也可被視為特殊的序列式容器，因為已序群集 (*sorted collections*) 正是根據某個排序準則排列 (*ordered*) 而成。如果你曾經用過其他的群集庫，例如

Smalltalk 和 NIHCL¹ 所提供者，你可能已經估計到這一點。在那些程式庫中，sorted collections 由 ordered collections 衍生而來。不過請注意，STL 所提供的群集型別 (collection types) 彼此獨立，各自實現，毫無關聯。

關聯式容器自動對其元素排序，這並不意味它們就是用來排序的。你也可以手動對序列式容器的元素加以排序。自動排序帶來的主要優點是，當你查找元素時，可獲取更佳效率。更明確地說你可以放心使用二分搜尋法 (binary search)。該演算法具有對數 (logarithmic) 複雜度，而非線性複雜度。這什麼意思呢？如果你想 在 1000 個元素中搜尋某一個元素，平均而言只需 10 次比較，而非 500 次比較（參見 2.3 節，21 頁）。因此，自動排序只是關聯式容器的一個（有用的）「副作用」。

下面各個小節詳細討論各容器類別。其中特別講解容器的典型實作方法。嚴格說來，C++ Standard 並未定義某一種容器的具體實作方式。然而，C++ Standard 所規定的行為和其對複雜度的要求，讓實作者沒有太多變化餘地。所以，實際上，各個實作品之間只在細節上有所差異。第 6 章會談到容器類別的確切行為、描述它們共有和特有的能力、並詳細分析其成員函式。

5.2.1 序列式容器 (Sequence Containers)

STL 內部預先定義了以下三個序列式容器：

- Vectors
- Deques
- Lists

此外，你也可以將 strings 和 array 當作一種序列式容器。

Vectors

Vector 將其元素置於一個 dynamic array 中加以管理。它允許隨機存取，也就是說你可以利用對應的索引直接存取每一個元素。在 array 尾部附加或移除元素均非常快速²，但是在 array 的中部或頭部插入元素就比較費時，因為，為了保持原來的相對次序，安插點之後的所有元素都必須移動，挪出位子來。

¹ The National Institute of Health's Class Library 是最早的 C++ class libraries 之一。

² 嚴格地說，元素附加動作是一種「分攤後的 (amortized)」高速。單一附加動作可能是緩慢的，因為 vector 可能需要重新分配記憶體，並將現有元素拷貝到新位置。不過這種事情不常發生，所以總體看來這個操作十分迅速。見 22 頁的複雜度討論。

以下例子針對整數型別定義了一個 `vector`，插入六個元素，然後列印所有元素：

```
// stl/vector1.cpp

#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> coll;    // vector container for integer elements

    // append elements with values 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_back(i);
    }

    // print all elements followed by a space
    for (int i=0; i<coll.size(); ++i) {
        cout << coll[i] << ' ';
    }
    cout << endl;
}
```

其中的：

```
#include <vector>
```

含入的表頭檔中有 `vector` 的宣告。

以下宣告式：

```
vector<int> coll;
```

生成一個「元素型別為 `int`」的 `vector`。由於沒有任何初始化參數，`default constructor` 就將它建構為空群集（empty collection）。

`push_back()` 函式為容器附加元素：

```
coll.push_back(i);
```

所有序列式容器都提供有這個成員函式。

`size()` 成員函式回傳容器中的元素個數：

```
for (int i=0; i<coll.size(); ++i) {
    ...
}
```

所有容器類別都提供有這個函式。

你可以通過下標運算子 (subscript operator) `[]`，訪問 `vector` 內的某個元素：

```
cout << coll[i] << ' ';
```

在這裡，元素被寫至標準輸出裝置，所以整個程式的輸出是：

```
1 2 3 4 5 6
```

Deque

所謂 `deque` (發音類似 "check"³)，是 "double-ended queue" 的縮寫。它是一個 `dynamic array`，可以向兩端發展，因此不論在尾部或頭部安插元素，都十分迅速。在中間部份安插元素則比較費時，因為必須移動其他元素。

以下例子宣告了一個浮點數 (floating-point) 型別的 `deque`，並在容器頭部安插 1.1 至 6.6 共六個元素，最後列印出所有元素。

```
// stl/deque1.cpp

#include <iostream>
#include <deque>
using namespace std;

int main()
{
    deque<float> coll; // deque container for floating-point elements

    // insert elements from 1.1 to 6.6 each at the front
    for (int i=1; i<=6; ++i) {
        coll.push_front(i*1.1); // insert at the front
    }

    // print all elements followed by a space
    for (int i=0; i<coll.size(); ++i) {
        cout << coll[i] << ' ';
    }
    cout << endl;
}
```

³ 有時候 "deque" 聽起來頗為類似 "hack"，不過這純屬巧合 ☺

本例之中，

```
#include <deque>
```

含入一個表頭，內含 `deque` 的宣告。

下面這一句：

```
deque<float> coll;
```

會產生一個空的浮點值群集。

`push_front()` 函式可以用來安插元素：

```
coll.push_front(i*1.1);
```

`push_front()` 將元素安插於群集前端。注意，這種安插方式造成的結果是，元素排放次序與插入次序恰好相反，因為每個元素都安插於上一個元素的前面。因此，程式輸出如下：

```
6.6 5.5 4.4 3.3 2.2 1.1
```

你也可以使用成員函式 `push_back()` 在 `deque` 尾端附加元素。`vector` 並未提供 `push_front()`，因為其時間效能很差（在 `vector` 前端安插一個元素，需要移動全部元素）。一般而言，STL 容器只提供通常具備良好時間效能的成員函式（所謂「良好」的時間效能，通常意味具有常數複雜度或對數複雜度），這可以防止程式員呼叫性能很差的函式。

Lists

List 由雙向串列（doubly linked list）實作而成。這意味 list 內的每個元素都以一部分記憶體指示其前導元素和後繼元素。List 不提供隨機存取。因此，如果你要存取第 10 個元素，你必須沿著串鏈，依次走過前 9 個元素。不過，移動至下一個元素或前一個元素的行為，可以在常數時間內完成。因此一般的元素存取動作將花費線性時間（平均距離與元素數量成比例）。這比 `vector` 和 `deque` 提供的「分攤性（*amortized*）」常數時間，性能相差很多。

List 的優勢是：任何位置上執行安插或刪除動作都非常迅速，因為所需改變的只是連結（links）而已。這表示在 list 中段移動元素比在 `vector` 和 `deque` 中快得多。

以下個例子產生一個準備放置字元的空的 list，然後將 'a' 至 'z' 的所有字元插入其中，利用一個迴圈，每次列印並移除群集的第一個元素，從而印出所有元素：

```
// stl/list1.cpp
```

```
#include <iostream>
#include <list>
```

```
using namespace std;

int main()
{
    list<char> coll; // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    /* print all elements
    * - while there are elements
    * - print and remove the first element
    */
    while (! coll.empty()) {
        cout << coll.front() << ' ';
        coll.pop_front();
    }
    cout << endl;
}
```

就像先前一樣，<list> 這個表頭包含了 lists 的宣告。以下定義一個「元素型別為字元」的 list：

```
list<char> coll;
```

成員函式 `empty()` 的回傳值告訴我們容器中是否還有元素。只要這個函式回傳 `false`（也就是說容器內還有元素），迴圈就繼續進行：

```
while (! coll.empty()) {
    ...
}
```

迴圈之內，成員函式 `front()` 會回傳第一個元素：

```
cout << coll.front() << ' ';
```

`pop_front()` 函式會刪除第一個元素：

```
coll.pop_front();
```

注意，`pop_front()` 並不會回傳被刪除的元素，所以你無法將上述兩個語句合而為一。

程式的輸出結果取決於所用字集。如果是 ASCII 字集，輸出如下⁴：

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

當然，爲了列印 `list` 的所有元素，「採用迴圈輸出並刪除第一個元素」的做法實在是很奇怪。通常你只需走訪所有元素即可。`lists` 並沒有提供以 `operator[]` 直接存取元素的能力，因爲 `lists` 不支持隨機存取，如果採用 `operator[]` 將會導致不良效能。運用迭代器也可以走訪並列印出所有元素。當我介紹過迭代器後，我會給一個例子。如果你等不及，請跳到 84 頁。

Strings

你也可以將 `string` 當作 STL 容器來使用。這裡的 `strings` 是指 C++ `string` 類別族系（`basic_string<>`，`string`，`wstring`）的物件，第 11 章對此有所介紹。`Strings` 跟 `vectors` 很相似，只不過其元素爲字元。11.2.13 節（497 頁）對此有詳細解說。

Arrays

另一種容器並非是個類別（`class`），而是 C/C++ 語言核心所支持的一個型別（`type`）：具有靜態大小或動態大小的 `array`。但 `array` 並非 STL 容器，它們並沒有類似 `size()` 和 `empty()` 的成員函式。儘管如此，STL 的設計允許你爲 `array` 呼叫 STL 演算法。當我們把 `static arrays` 作爲初值列（`initializer list`）時，這一點特別有用。

`Array` 的運用並無新意，針對 `arrays` 使用演算法，才是新的議題。這個議題將在 6.7.2 節，218 頁討論。

值得注意的是，我們沒有必要再直接編寫 `dynamic array` 了。`Vectors` 已經具備了 `dynamic array` 的全部屬性，並提供更安全更便捷的介面。詳見 6.2.3 節，155 頁。

5.2.2 關聯式容器 (Associative Containers)

關聯式容器依據特定的排序準則，自動爲其元素排序。排序準則以函式形式呈現，用來比較元素的值，或是元素的鍵值（`key`）。預設情況下使用 `operator<` 來進行比較動作，不過你也可以提供自己的比較函式，定義出不同的排序準則。

通常關聯式容器由二元樹（`binary tree`）實作出來。在二元樹中，每個元素（節點）都有一個父節點和兩個子節點；左子樹中的所有元素值都比自己小，右子樹中的所有元素值都比自己大。關聯式容器的差別主要在於元素的類型以及處理重複元

⁴ 如果是 ASCII 以外的字集，輸出結果可能包含非字母的字元，甚至可能什麼都沒有（如果 'z' 不大於 'a' 的話）。

素時的方式（態度）。

下面是 STL 中預定義的關聯容器。由於訪問其中的元素需要用到 `iterator`，所以我推遲到 87 頁，討論過 `iterators` 之後再舉例子。

- **Sets**

`Set` 的內部元素依據其值自動排序，每個元素只能出現一次，不允許重複。

- **Multisets**

`Multiset` 和 `set` 相同，只不過它允許重複元素，也就是說 `multiset` 可包括多個其值相同的元素。

- **Maps**

`Map` 的元素都是「鍵值/實值」所形成的一個對組（`key/value pairs`）。每個元素有一個鍵值，是排序準則的基礎。每一個鍵只能出現一次，不允許重複。`Map` 可被視為關聯式陣列（`associative array`），也就是具有任意索引型別（`index type`）的陣列（詳見 91 頁）。

- **Multimaps**

`Multimap` 和 `map` 相同，只不過允許重複元素，也就是說 `multimap` 可包含多個鍵值（`key`）相同的元素。`Multimap` 可被當作字典使用。209 頁有一個範例。

所有關聯式容器都有一個可供選擇的 `template argument`（模板引數），指明排序準則。預設採用 `operator<`。排序準則同時也用來測試互等性（`equality`）：如果兩個元素都不小於對方，則兩者必然相等。

你可以將 `set` 視為一種特殊的 `map`，令其元素值就是鍵值。實際產品中，所有這些關聯式容器通常都由二元樹實作而成。

5.2.3 容器配接器 (Container Adapters)

除了以上幾個根本的容器類別，為了滿足特殊需求，C++ 標準程式庫還提供了一些特別的（預先定義好的）容器配接器，根據基本容器類別實作而成，包括：

- **Stacks**

名字說明了一切。`Stack` 容器對元素採取 LIFO（後進先出）管理策略。

- **Queues**

`Queue` 容器對元素採取 FIFO（先進先出）管理策略。也就是說，它是一個普通的緩衝區（`buffer`）。

- **Priority Queues**

`Priority Queue` 容器中的元素可以擁有不同的優先權。所謂優先權，乃是基於程

式員提供的排序準則（預設使用 `operator<`）而定義。Priority queue 的效果相當於這樣一個 buffer：「下一元素永遠是 queue 之中優先權最高的元素」。如果同時有多個元素具備最高優先權，則其次序不定。

5.3 迭代器 (Iterators)

迭代器是一個「可巡訪 STL 容器中的全部或部分元素」的物件。一個迭代器用來指出容器中的一個特定位置。基本操作如下：

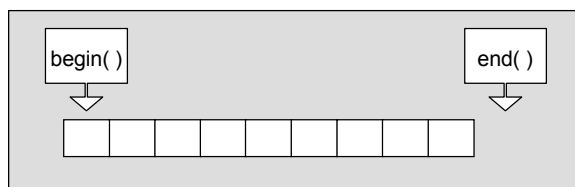
- **Operator ***
回傳當前位置上的元素值。如果該元素擁有成員，你可以透過迭代器，直接以 `operator->` 取用它們⁵。
- **Operator ++**
將迭代器前進至下一元素。大多數迭代器還可以使用 `operator--` 退回到前一個元素。
- **Operators == 和 Operator !=**
判斷兩個迭代器是否指向同一位置。
- **Operator =**
為迭代器賦值（將其所指元素的位置指派過去）。

這些操作和 C/C++ 「操作 array 元素時」的指標介面一致。不同之處在於，迭代器是個所謂的 *smart pointers*，具有走訪複雜資料結構的能力。其下層運作機制取決於其所走訪的資料結構。因此，每一種容器型別都必須提供自己的迭代器。事實上每一種容器都將其迭代器以巢狀方式（nested）定義於內部。因此，各種迭代器的介面相同，型別卻不同。這直接導出了泛型程式設計的概念：所有操作行為都使用相同介面，雖然它們的型別不同。因此，你可以使用 *templates* 將泛型操作公式化，使之得以順利運作那些「能夠滿足介面需求」的任何型別。

所有容器類別都提供有一些成員函式，使我們得以獲得迭代器並以之遍訪所有元素。這些函式中最重要的如下：

- **begin()**
回傳一個迭代器，指向容器起始點，也就是第一個元素（如果有的話）的位置。
- **end()**
回傳一個迭代器，指向容器結束點。結束點在最後一個元素之後，這樣的迭代器又稱作「逾尾（*past-the-end*）」迭代器。

⁵ 某些老舊的 STL 環境並不支援迭代器的 `operator->`。

圖 5.3 容器的 `begin()` 和 `end()` 成員函式

於是，`begin()` 和 `end()` 形成了一個半開區間（half-open range），從第一個元素開始，到最後一個元素的下一位置結束（圖 5.3）。半開區間有兩個優點：

1. 為「走訪元素時的迴圈結束時機」提供了一個簡單的判斷依據。只要尚未到達 `end()`，迴圈就可以繼續進行下去。
2. 不必對空區間採取特殊處理手法。空區間的 `begin()` 就等於 `end()`。

下面這個例子展現了迭代器的用法，將 `list` 容器內的所有元素列印出來（這就是 79 頁那個 `list` 實例的改進版）。

```
// stl/list2.cpp

#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> coll; // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    /* print all elements
     * - iterate over all elements
     */
    list<char>::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

首先產生一個 `list`，然後填入 'a'-'z' 字元，然後在 `for` 迴圈中列印出所有元素：

```
list<char>::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```

迭代器 `pos` 就宣告於迴圈之前，其型別是「指向容器中的不可變元素」的迭代器：

```
list<char>::const_iterator pos;
```

任何一種容器都定義有兩種迭代器型別：

1. `container::iterator`
這種迭代器以「讀/寫」模式走訪元素。
2. `container::const_iterator`
這種迭代器以「唯讀」模式走訪元素。

例如，在 `class list` 之中，它們的定義可能是這樣：

```
namespace std {
    template <class T>
    class list {
    public:
        typedef ... iterator;
        typedef ... const_iterator;
        ...
    };
}
```

至於其中 `iterator` 和 `const_iterator` 的確切型別，則於實作中定義。

在迴圈中，迭代器 `pos` 以容器的第一個元素位置為初值：

```
pos = coll.begin()
```

迴圈不斷進行，直到 `pos` 到達容器的結束點：

```
pos != coll.end()
```

在這裡，`pos` 是在和「逾尾 (*past-the-end*)」迭代器作比較。當迴圈內部執行 `++pos` 述句，迭代器 `pos` 就會前進到下一個元素。

總而言之，`pos` 從第一個元素開始，逐一訪問了每一個元素，直到抵達結束點為止（圖 5.4）。如果容器內沒有任何元素，`coll.begin()` 等於 `coll.end()`，迴圈根本不會執行。

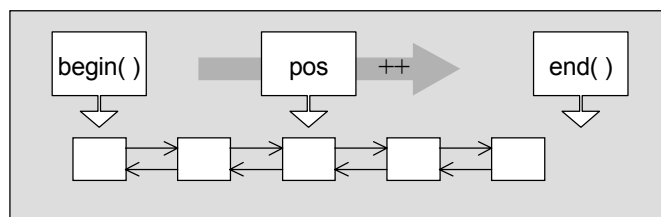


圖 5.4 迭代器 `pos` 走訪 List 的每一個元素

在迴圈內部，述句 `*pos` 代表當前元素。本例在將它輸出之後，又接著輸出了一個空格。你不能改變元素內容，因為 `pos` 是個 `const_iterator`，從迭代器的觀點看去，元素是常量，不能更改。不過如果你採用非常量（`nonconstant`）迭代器，而且元素本身的型別也是非常量（`nonconstant`），那麼就可以透過迭代器來改變元素值。例如：

```
// make all characters in the list uppercase
list<char>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    *pos = toupper(*pos);
}
```

注意，這裏使用前置式累進 `++pos`，因為它比後置式累進 `pos++` 效率高。後置式累進運算子需要一個額外的臨時物件，因為它必須存放迭代器的原本位置並將它回傳，所以一般情況下最好使用 `++pos`，不要用 `pos++`。也就是說，不要這麼寫：

```
for (pos = coll.begin(); pos != coll.end(); pos++) {
    ***** // OK, but slower
    ...
}
```

爲了這個理由，我建議優先採用前置式遞增（`pre-increment`）或前置式遞減（`pre-decrement`）運算子。

5.3.1 關聯式容器的運用實例

上個例子中的迭代器迴圈可應用於任何容器，只需調整迭代器型別即可。現在你知道如何列印關聯式容器內的元素了吧。下面是使用關聯式容器的一些例子。

Sets 和 Multisets 使用實例

第一個例子展示如何在 `set` 之中安插元素，並使用迭代器來列印它們。

```
// stl/set1.cpp

#include <iostream>
#include <set>

int main()
{
    // type of the collection
    typedef std::set<int> IntSet;

    IntSet coll; // set container for int values

    /* insert elements from 1 to 6 in arbitrary order
    *-value1 gets inserted twice
    */
    coll.insert(3);
    coll.insert(1);
    coll.insert(5);
    coll.insert(4);
    coll.insert(1);
    coll.insert(6);
    coll.insert(2);

    /* print all elements
    * - iterate over all elements
    */
    IntSet::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

一如以往，`include` 指令：

```
#include <set>
```

定義了 `sets` 的所有必要型別和操作。

既然容器的型別要用到好幾次，不妨先定義一個短一點的名字：

```
typedef set<int> IntSet;
```

這個述句定義了型別 `IntSet`，實際上就是元素型別為 `int` 的 `set`。這種型別有預設的排序準則，以 `operator<` 為依據，對元素進行排序。這意味元素將以遞增方式排列。如果希望以遞減方式排列，或是希望使用一個完全不同的排序準則，你可以將它傳入做為第二個 `template parameter`。下面的例子即是將元素以遞減方式排列⁶：

```
typedef set<int,greater<int> > IntSet;
```

以上所用的 `greater<>` 是一個預先定義的仿函式，我將在 5.9.2 節 131 頁討論它。8.1.1 節 294 頁有一個例子，僅使用物件元素的部分資料（例如 ID）進行排序。

所有關聯式容器都提供有一個 `insert()` 成員函式，用以安插新元素：

```
coll.insert(3);  
coll.insert(1);  
...
```

新元素會根據排序原則自動安插到正確位置。你不能使用序列式容器的 `push_back()` 和 `push_front()` 函式，它們在這裏毫無意義，因為你沒有權力指定新元素的位置。

所有元素（不論以任何次序）安插完畢後，容器的狀態如圖 5.5。元素被有序地存放在內部的 `tree` 結構中。任何一個元素（節點）的左子樹內的元素，永遠小於其右子樹內的元素（這裏的「小於」是就當前排序原則而言）。Sets 不允許存在重複的元素，所以容器裡頭只有一個 "1"。

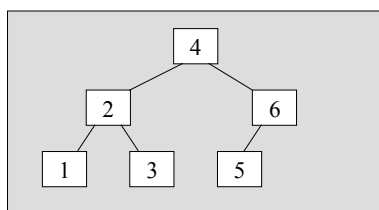


圖 5.5 一個 Set，擁有 6 個元素

⁶ 注意，兩個 ">" 符號之間一定要有一個空格。">>" 會被編譯器視為一個右移運算子（right-shift operator），從而導致語法錯誤。

現在，我們可以運用和先前 `list` 例子中相同的迴圈，來列印容器內的元素。以一個迭代器走訪全部元素，並逐一列印出來：

```
IntSet::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```

再一次我要提醒你，由於迭代器是容器定義的，所以無論容器內部結構是如何複雜，它都知道如何行事。舉個例子，如果迭代器指向第三個元素，運算子 `++` 便會將它移動到上端的第四個元素，再一次 `++` 則會將它移動到下方的第五個元素（圖 5.6）。

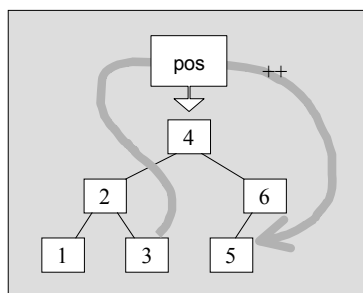


圖 5.6 迭代器 `pos` 走訪 `Set` 內的元素

以下是輸出結果：

```
1 2 3 4 5 6
```

如果你想使用 `multiset`，而不是 `set`，唯一需要改變的就是容器的型別（`set` 和 `multiset` 所需要的表頭檔是一樣的）：

```
typedef multiset<int> IntSet;
```

由於 `multiset` 允許元素重複存在，因此其中可包含兩個數值皆為 1 的元素。輸出結果如下：

```
1 1 2 3 4 5 6
```

Maps 和 Multimaps 的簡單實例

`Map` 的元素是成對的鍵值/實值（`key/value`）。因此其宣告、元素安插、元素存取皆和 `set` 有不同。下面是一個 `multimap` 運用實例：

```
// stl/mmap1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    // type of the collection
    typedef multimap<int,string> IntStringMMap;

    IntStringMMap coll; // container for int/string values

    // insert some elements in arbitrary order
    // - a value with key 1 gets inserted twice
    coll.insert(make_pair(5,"tagged"));
    coll.insert(make_pair(2,"a"));
    coll.insert(make_pair(1,"this"));
    coll.insert(make_pair(4,"of"));
    coll.insert(make_pair(6,"strings"));
    coll.insert(make_pair(1,"is"));
    coll.insert(make_pair(3,"multimap"));

    /* print all element values
    * - iterate over all elements
    * - element member second is the value
    */
    IntStringMMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << pos->second << ' ';
    }
    cout << endl;
}
```

程式的輸出結果可能是這樣：

```
this is a multimap of tagged strings
```

不過，由於 "this" 和 "is" 的鍵值相同，兩者的出現順序也可能反過來。

拿這個例子和 87 頁上的 `set` 實例作比較，你會發現以下兩點不同：

1. 這裡的元素是成對的鍵值/實值 (key/value pair)，所以你必須首先生成這個 pair，然後再插入群集中。輔助函式 `make_pair()` 正是爲了這個目的而打造。這個問題的細節，以及其他安插方法，請見 203 頁，
2. 迭代器所指的是「鍵值/實值」對組 (key/value pair)，因此你無法一口氣列印它們，你必須取出 pair 的成員，亦即所謂的 `first` 和 `second` (pair 型別在 4.1 節 33 頁作過介紹)。因此，以下述句：

```
pos->second
```

便取得了「鍵值/實值」對組中的第二部分，也就是 `multimap` 元素的實值 (value)。和一般指標的情形一樣，上述述句就是以下述句的簡寫方案⁷：

```
(*pos).second
```

同樣道理，以下述句：

```
pos->first
```

取得「鍵值/實值」對組中的第一部分，也就是 `multimap` 元素的鍵值 (key)。

`Multimaps` 也可以用來作爲 *dictionaries*，詳見 209 頁實例。

將 **Maps** 當作關聯式陣列 (associative arrays)

如果上述例子中以 `map` 取代 `multimap`，輸出結果就不會有重複的鍵值 (keys)，實值 (values) 則和上述結果一樣。一個「鍵值/實值」對組所形成的群集中，如果所有的鍵值都是唯一的，我們可將它視爲一個關聯式陣列 (associative array)。考慮以下例子：

```
// stl/map1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    /* type of the container:
    *-map: elements key/value pairs
    *-string: keys have type string
    *-float: values have type float
    */
```

⁷ 某些老舊環境並沒有實現出 `iterator->`，這時候你就只能使用第二個表述式了。

```

typedef map<string,float> StringFloatMap;
StringFloatMap coll;

// insert some elements into the collection
coll["VAT"] = 0.15;
coll["Pi"] = 3.1415;
coll["an arbitrary number"] = 4983.223;
coll["Null"] = 0;

/* print all elements
 * - iterate over all elements
 * - element member first is the key
 * - element member second is the value
 */
StringFloatMap::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << "key: \" << pos->first << \" \"
        << "value: \" << pos->second << endl;
}
}

```

當我們宣告容器型別的時候，必須同時指定鍵值（key）和實值（value）的型別：

```
typedef map<string,float> StringFloatMap;
```

Maps 允許你使用 operator[] 安插元素：

```

coll["VAT"] = 0.15;
coll["Pi"] = 3.1415;
coll["an arbitrary number"] = 4983.223;
coll["Null"] = 0;

```

在這裡，以鍵值做為索引，可為任意型別。這正是關聯式陣列的介面。所謂關聯式陣列就是：索引可以採用任何型別。

注意這裏的下標運算子（subscript operator）和一般陣列所用的下標運算子的行為有些不同：在這裡，索引可以不對應於任何元素。如果你指定了一個新的索引（新的鍵值），會導致產生一個對應的新元素，並被安插於 map。也就是說，沒有任何索引是「錯誤」的。因此，以下述句：

```
coll["Null"] = 0;
```

其中的式子：

```
coll["Null"]
```

會產生一個新元素，其鍵值為 "Null"。賦值運算子（assignment operator）將元素的實值設為 0（並轉化為 float）。6.6.3 節 205 頁更詳細地討論了如何將 maps 當作關聯式陣列。

Multimaps 不允許我們使用下標運算子，因為 multimaps 允許單一索引對應到多個不同元素，而下標運算子卻只能處理單一實值。你必須先產生一個「鍵值/實值」對組，然後再插入 multimap，見 90 頁。當然對於 maps 也可以這麼做，細節請參考 202 頁。

存取 multimaps 或 maps 的元素時，你必須透過 pair 結構的 first 和 second 成員，才能取得鍵值 (key) 和實值 (value)。上述程式的輸出如下：

```
key: "Null" value: 0
key: "Pi" value: 3.1415
key: "VAT" value: 0.15
key: "an arbitrary number" value: 4983.22
```

5.3.2 迭代器類型 (Iterator Categories)

除了基本操作之外，迭代器還有其他能力。這些能力取決於容器的內部結構。STL 總是只提供效率上比較出色的操作，因此，如果容器允許隨機存取（例如 vectors 或 deque），那麼它們的迭代器也能進行隨機操作（例如直接讓迭代器指向第五元素）。

根據能力的不同，迭代器被劃分為五種不同類型。STL 預先定義好的所有容器，其迭代器均屬於以下兩種類型：

1. 雙向迭代器 (**Bidirectional iterator**)

顧名思義，雙向迭代器可以雙向行進：以遞增 (increment) 運算前進，以遞減 (decrement) 運算後退。List、set、multiset、map 和 multimap 這些容器所提供的迭代器都屬此類。

2. 隨機存取迭代器 (**Random access iterator**)

隨機存取迭代器不但具備雙向迭代器的所有屬性，還具備隨機訪問能力。更明確地說，它們提供了「迭代器算術運算」必備的運算子（和「一般指標的算術運算」完全對應）。你可以對迭代器增加或減少一個偏移量、處理迭代器之間的距離、或是使用 < 和 > 之類的相對關係運算子 (relational operators) 來比較兩個迭代器。Vector, deque 和 strings 所提供的迭代器都屬此類。

其他的迭代器類型在 7.2 節 251 頁介紹。

為了撰寫盡可能與容器型別無關的泛型程式碼，你最好不要使用 random access iterators 的特有操作。例如下面這個例子，可以在任何容器上運作：

```
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    ...
}
```

而下面這樣的程式碼就不是所有容器都適用：

```
for (pos = coll.begin(); pos < coll.end(); ++pos) {  
    ...  
}
```

兩者的唯一區別在於測試迴圈條件時，第二例使用 `operator<`，第一例使用 `operator!=`。要知道，只有 `random access iterators` 才支援 `operator<`，所以第二例中的迴圈對於 `lists`、`sets` 和 `maps` 無法運作。爲了寫出適用於任意容器的泛型程式碼，你應該使用 `operator!=` 而非 `operator<`。不過如此一來，程式碼的安全性可能有損，因爲如果 `pos` 的位置在 `end()` 的後面，你未必便能發現（關於 STL 使用上的可能錯誤，請見 5.11 節 136 頁）。究竟使用哪種方式，取決於當時情況，取決於個人經驗，取決於你。

爲了避免誤解，我得再強調一句。注意，我說的是「類型（categories）」，不是「類別（classes）」。所謂類型，只是定義迭代器的能力，無關乎迭代器的型別（type）。STL 的泛型概念（generic concept）可以以純抽象形式工作，也就是說，任何東西，只要行爲「像」一個雙向迭代器，那麼它就是一個雙向迭代器。

5.4 演算法 (Algorithms)

爲了處理容器內的元素，STL 提供了一些標準演算法，包括搜尋、排序、拷貝、重新排序、修改、數值運算等基礎而普遍的演算法。

演算法並非容器類別的成員函式（member functions），而是一種搭配迭代器使用的全域函式（global functions）。這麼做（[譯註](#)：意指搭配迭代器來使用）有一個重要優勢：所有演算法只需實作出一份，就可以對所有容器運作，不必爲每一種容器量身訂製。演算法甚至可以操作不同型別（types）之容器內的元素，也可以與用戶定義的容器搭配。這個概念大幅降低了程式碼的體積，提高了程式庫的能力和彈性。

注意，這裏所闡述的並非物件導向編程思維模式（OOP paradigm），而是泛型函式編程思維模式（generic functional programming paradigm）。在物件導向編程（OOP）概念裡，資料與操作合爲一體，在這裏則被明確劃分，透過特定的介面彼此互動。當然這需要付出代價：首先，其用法有失直觀，其次，某些資料結構與演算法並不相容，更有甚者，某些容器和演算法雖然勉強相容，卻毫無用處（也許導致很糟的效能）。因此，深入學習 STL 的概念並了解其缺陷，顯得十分重要，惟其如此，方能取其利而避其害。我將在本章剩餘篇幅中，透過實例詳細介紹它們。讓我們從一個簡單的 STL 演算法使用範例入手。以下實例展現了某些演算法的使用方式：


```
// stl/alg01.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll;
    vector<int>::iterator pos;

    // insert elements from 1 to 6 in arbitrary order
    coll.push_back(2);
    coll.push_back(5);
    coll.push_back(4);
    coll.push_back(1);
    coll.push_back(6);
    coll.push_back(3);

    // find and print minimum and maximum elements
    pos = min_element (coll.begin(), coll.end());
    cout << "min: " << *pos << endl;
    pos = max_element (coll.begin(), coll.end());
    cout << "max: " << *pos << endl;

    // sort all elements
    sort (coll.begin(), coll.end());

    // find the first element with value 3
    pos = find (coll.begin(), coll.end(),    // range
               3);                          // value
    // reverse the order of the found element with value 3 and
    // all following elements
    reverse (pos, coll.end());

    // print all elements
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

爲了呼叫演算法，首先你必須含入表頭檔 `<algorithm>`：

```
#include <algorithm>
```

最先出現的是演算法 `min_element()` 和 `max_element()`。呼叫它們時，你必須傳入兩個引數，定義出欲處理的元素範圍。如果想要處理容器內的所有元素，可以簡單地使用 `begin()` 和 `end()`。兩個演算法都傳回一個迭代器，分別指向最小或最大元素。因此，以下述句：

```
pos = min_element (coll.begin(), coll.end());
```

演算法 `min_element()` 回傳最小元素的位置（如果最小元素不只一個，則回傳第一個最小元素的位置）。以下述句列印出該元素：

```
cout << "min: " << *pos << endl;
```

當然，你也可以在單一述句中合併上述兩個動作：

```
cout << *max_element(coll.begin(),coll.end()) << endl;
```

接下來的演算法是 `sort()`。顧名思義，它將「由兩個引數設定出來」的區間內的所有元素排序。你可以（選擇性地）傳入一個排序準則。預設的準則是 `operator <`。因此，本例容器內的所有元素以遞增方式排列。

```
sort (coll.begin(), coll.end());
```

排序後的容器元素如下排列：

```
1 2 3 4 5 6
```

再來便是演算法 `find()`。它在給定範圍中搜尋某個值。本例在整個容器內尋找第一個其值爲 3 的元素。

```
pos = find (coll.begin(), coll.end(),           // range
           3);                                  // value
```

如果 `find()` 成功了，便回傳一個迭代器，指向目標元素。如果失敗，回傳一個「逾尾（past-the-end）」迭代器，亦即 `find()` 所接受的第二個引數。本例在第三個元素位置上發現數值 3，因此完成後 `pos` 指向 `coll` 的第三個位置。

本例所展示的最後一個演算法是 `reverse()`，將區間內的元素反轉放置：

```
reverse (pos, coll.end());
```

於是第三個至最後一個元素之間的所有元素都被反轉置放。整個程式輸出如下：

```
min: 1
max: 6
1 2 6 5 4 3
```

5.4.1 區間 (Ranges)

所有演算法都是用來處理一個或多個區間內的元素。這樣的區間可以（但非強行要求）涵蓋容器內的全部元素。因此，爲了得以操作容器元素的某個子集，我們必須將區間首尾當做兩個引數（arguments）傳給演算法，而不是一口氣把整個容器傳遞進去。

這樣的介面靈活而又危險。呼叫者必須確保經由兩引數定義出來的區間是有效的（valid）。所謂有效就是，從起點出發，逐一前進，能夠到達終點。也就是說，程式員自己必須確保兩個迭代器屬於同一個容器，而且前後放置正確。否則結果難料，可能會引起無限迴圈，也可能會存取到記憶體禁區。就此點而言，迭代器就像一般指標一樣地危險。不過請注意，所謂「結果難料」（或說行爲未有定義，undefined behavior）意味任何 STL 實作品均可自由選擇合適的方式來處理此類錯誤。稍後你會發現，確保區間的有效性並不像聽起來那麼簡單。與此相關的一些細節請參見 5.11 節，136 頁。所有演算法處理的都是半開區間（half-open ranges）——包括起始元素的位置，但不包括結尾元素的位置。傳統的數學表示方式爲：

[begin, end)

或

[begin, end[

本書採用第一種表示法。

半開區間的優點已於 84 頁介紹過（主要是單純，可避免對空的群集另外做特殊處理）。當然，金無足赤，世上亦沒有完美的設計。請看下面的例子：

```
// stl/find1.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;
    list<int>::iterator pos;

    // insert elements from 20 to 40
    for (int i=20; i<=40; ++i)
        coll.push_back(i);
}
```

```

/* find position of element with value 3
 * - there is none, so pos gets coll.end()
 */
pos = find (coll.begin(), coll.end(),    // range
           3);                          // value

/* reverse the order of elements between found element and the
 * end - because pos is coll.end() it reverses an empty range
 */
reverse (pos, coll.end());

// find positions of values 25 and 35
list<int>::iterator pos25, pos35;
pos25 = find (coll.begin(), coll.end(),    // range
             25);                          // value
pos35 = find (coll.begin(), coll.end(),    // range
             35);                          // value

/* print the maximum of the corresponding range
 * - note: including pos25 but excluding pos35
 */
cout << "max: " << *max_element (pos25, pos35) << endl;

// process the elements including the last position
cout << "max: " << *max_element (pos25, ++pos35) << endl;
}

```

本例首先以 20 至 40 的整數做為容器初值。當搜尋元素值 3 的任務失敗後，`find()` 傳回區間的結束位置（本例為 `coll.end()`）並且指派給 `pos`。以此 `pos` 作為稍後呼叫 `reverse()` 時的區間起點，純粹是空擺架子，因為其結果相當於：

```
reverse (coll.end(), coll.end());
```

這其實就是在逆轉一個空區間，當然毫無效果了（亦即所謂的 "no-op"）。

如果使用 `find()` 來獲取某個子集的第一個和最後一個元素，你必須考慮一點：半開區間並不包含最後一個元素。所以，上述例子第一次呼叫 `max_element()`：

```
max_element (pos25, pos35)
```

傳回的是 34，而不是 35：

```
max: 34
```

爲了能夠處理最後一個元素，你必須把該元素的下一個位置傳遞給演算法：

```
max_element (pos25, ++pos35)
```

這樣才能得到正確的結果：

```
max: 35
```

注意，本例使用的是 `list` 容器，所以你只能以 `++` 取得 `pos35` 的下一個位置。如果面對的是 `vectors` 或 `deque`s 的隨機存取迭代器 (*random access iterators*)，你可以寫 `pos35 + 1`。這是因爲隨機存取迭代器允許「迭代器算術運算」（參見 93 頁 5.3.2 節，255 頁 7.2.5 節）。

當然，你可以使用 `pos25` 和 `pos35` 來搜尋其間的任何東西。記住，爲了讓搜尋動作及於 `pos35`，必須把元素 35 的下一個位置傳入，例如：

```
// increment pos35 to search with its value included
++pos35;
pos30 = find(pos25, pos35,    // range
             30);            // value
if (pos30 == pos35) {
    cout << "30 is NOT in the subrange" << endl;
}
else {
    cout << "30 is in the subrange" << endl;
}
```

本節中的所有例子都可以正常運作，但那完全是因爲你很清楚 `pos25` 一定在 `pos35` 之前。否則，`[pos25; pos35)` 就不是個有效區間。如果你對於「哪個元素在前，哪個元素在後」心中沒個譜兒，事情可就麻煩了，說不定會導致未定義的行爲。

現在假設你並不知道元素 25 和元素 35 的前後關係，甚至連它們是否存在也心存疑慮。如果你手上用的是隨機存取迭代器 (*random access iterators*)，你可以使用 `operator<` 進行檢查：

```
if (pos25 < pos35) {
    // only [pos25; pos35) is valid
    ...
}
else if (pos35 < pos25) {
    // only [pos35; pos25) is valid
    ...
}
else {
    // both are equal, so both must be end()
    ...
}
```

然而如果你手上的並非隨機存取迭代器，那還真的沒什麼直截了當的辦法可確定哪個迭代器在前。你只能在「起點和某個迭代器」之間，以及「該迭代器和終點」之間中，尋找另外那個迭代器。此時你的解決方法需要一些變化：不是一口氣在整個區間中搜尋兩個值，而是去了解哪個值先找到，哪個值後找到。例如：

```
pos25 = find (coll.begin(), coll.end(),      // range
             25);                          // value
pos35 = find (coll.begin(), pos25,          // range
             35);                          // value
if (pos35 != pos25) {
    /* pos35 is in front of pos25
     * so, only [pos35; pos25) is valid
     */
    ...
}
else {
    pos35 = find (pos25, coll.end(),        // range
                 35);                      // value
    if (pos35 != pos25) {
        /* pos25 is in front of pos35
         * so, only [pos25; pos35) is valid
         */
        ...
    }
    else {
        // both are equal, so both must be end()
        ...
    }
}
```

和前例不同的是，本例並非在 `coll` 的整個區間內搜尋 35，而是先在起點和 `pos25` 之間尋找，如果一無所獲，再在 `pos25` 之後的區間尋找。其結果當然就使你得以完全掌握哪個位置在前面、哪個子區間有效。

這麼做並不是很有效率。當然還有其他高招，可以直接找到 25 或 35 第一次出現的位置，不過那就需要用到目前還未介紹的一些 STL 技術了：

```
pos = find_if (coll.begin(), coll.end(),    // range
               compose_f_gx_hx(logical_or<bool>(), // criterion
                               bind2nd(equal_to<int>(), 25),
                               bind2nd(equal_to<int>(), 35)));
```

```

switch (*pos) {
    case 25:
        // element with value 25 comes first
        pos25 = pos;
        pos35 = find (++pos, coll.end(),    // range
                     35);                  // value
        ...
        break;
    case 35:
        // element with value 35 comes first
        pos35 = pos;
        pos25 = find (++pos, coll.end(),    // range
                     25);                  // value
        ...
        break;
    default:
        // no element with value 25 or 35 found
        ...
        break;
}

```

這裡使用了一個特別的運算式作為搜尋規則，其目的是找到數值 25 或數值 35 第一次出現的位置。這個運算式由好幾個預先定義的仿函式（functors，或名 function objects）組成，我將在 5.9.2 節 131 頁和 8.2 節 305 頁介紹所有預先定義的仿函式。`compose_f_gx_hx` 是個靈巧的輔助型仿函式，我將在 8.3.1 節 316 頁介紹它。

5.4.2 處理多個區間

有數個演算法可以（或說需要）同時處理多個區間。通常你必須設定第一個區間的起點和終點，至於其他區間，你只需設定起點即可，終點通常可由第一區間的元素數量推導出來。下面的例子中，`equal()` 從頭開始逐一比較 `coll1` 和 `coll2` 的所有元素：

```

if (equal (coll1.begin(), coll1.end(),
          coll2.begin())) {
    ...
}

```

因此，`coll2` 之中參與比較的元素數量，間接取決於 `coll1` 內的元素數量。

這使我們導出一個重要心得：如果某個演算法用來處理多個區間，則你呼叫它時，務必確保第二（以及其他）區間所擁有的元素個數，至少和第一區間內的元素個數相同。特別是，執行寫入動作時，務必確保目標區間（destination ranges）夠大。

考慮下面這個程式：

```
// stl/copy1.cpp

#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll1;
    vector<int> coll2;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
    }

    // RUNTIME ERROR:
    // - overwrites nonexisting elements in the destination
    copy (coll1.begin(), coll1.end(), // source
          coll2.begin());           // destination
    ...
}
```

這裏呼叫了 `copy()` 演算法，將第一區間內的全部元素拷貝至目標區間。如上所述，第一區間的起點和終點都已指定，第二區間只指出起點。然而，由於該演算法執行的是覆寫動作（overwrites）而非安插動作（inserts），所以目標區間必須擁有足夠的元素來被覆寫，否則就會像這個例子一樣，導致未定義的行為。如果目標區間內沒有足夠的元素供覆寫，通常意味你會覆寫 `coll2.end()` 之後的任何東西，幸運的話你的程式立即崩潰 — 這起碼還能讓你知道出錯了。你可以強制自己獲得這種幸運：使用 STL 安全版本。在這個安全版本中，所有未定義的行為都會被導向一個錯誤處理程序（error handling procedure），見 5.11.1 節，138 頁。

想要避免上述錯誤，你可以 (1) 確認目標區間內有足夠的元素空間，或是 (2) 採用 *insert iterators*。Insert iterators 將在 5.5.1 節 104 頁介紹。我首先解釋如何修改目標區間，俾使它有足夠的空間。

要想讓目標區間足夠大，你要不一開始就給它一個正確的大小，要不就明白地改變其大小。這兩個辦法都只適用於序列式容器 (vectors, deque, lists)。關聯式容器根本不會有此問題，因為關聯式容器不可能被當作覆寫式演算法的操作目標（原因見 5.6.2 節，115 頁）。以下例子展示如何增加容器的大小：

```
// stl/copy2.cpp

#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll1;
    vector<int> coll2;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
    }

    // resize destination to have enough room for the
    // overwriting algorithm
    coll2.resize (coll1.size());

    /* copy elements from first into second collection
     * - overwrites existing elements in destination
     */
    copy (coll1.begin(), coll1.end(), // source
          coll2.begin());           // destination

    /* create third collection with enough room
     * - initial size is passed as parameter
     */
    deque<int> coll3(coll1.size());
```

```
// copy elements from first into third collection
copy (coll1.begin(), coll1.end(),          // source
      coll3.begin());                     // destination
}
```

在這裡，`resize()`的作用是改變 `coll2` 的元素個數：

```
coll2.resize (coll1.size());
```

`coll3` 則是在初始化時就指明要有足夠的空間，以容納 `coll1` 中的全部元素：

```
deque<int> coll3(coll1.size());
```

注意，這兩種方法都會產出新元素並賦予初值。這些元素由 `default constructor` 初始化，沒有任何引數。你可以傳遞額外的引數給 `constructor` 和 `resize()`，這樣就可以按你的意願將新元素初始化。

5.5 迭代器配接器 (Iterator Adapters)

迭代器 (Iterators) 是一個純粹抽象概念：任何東西，只要其行為類似迭代器，它就是一個迭代器。因此，你可以撰寫一些類別 (classes)，具備迭代器的介面，但有著各不相同的行為。C++ 標準程式庫提供了數個預先定義的特殊迭代器：所謂的迭代器配接器 (iterator adapters)。它們不只是輔助性質而已，它們賦予整個迭代器抽象概念更強大的能力。

下面數個小節簡介三種迭代器配接器 (iterator adapters)：

1. *Insert iterators* (安插型迭代器)
2. *Stream iterators* (串流迭代器)
3. *Reverse iterators* (逆向迭代器)

第 7.4 節 264 頁將對它們做更詳實的講解。

5.5.1 Insert Iterators (安插型迭代器)

迭代器配接器的第一個例子是 *insert iterators*，或稱為 *inserters*。*Inserters* 可以使演算法以安插 (insert) 方式而非覆寫 (overwrite) 方式運作。使用它，可以解決演算法之目標空間不足的問題。是的，它會促使目標區間的大小按需要成長。

Insert iterators 內部將介面做了新的定義：

- 如果你對某個元素設值，會引發「對其所屬群集的安插動作」。至於插入位置是在容器的最前或最後，或是於某個特定位置上，端視三種不同的 *insert iterators* 而定。
- 「單步前進 (step forward)」不會造成任何動靜。

現在請看下面這個例子：

```
// stl/copy3.cpp

#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll1;

    // insert elements from 1 to 9 into the first collection
    for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
    }

    // copy the elements of coll1 into coll2 by appending them
    vector<int> coll2;
    copy (coll1.begin(), coll1.end(),          // source
          back_inserter(coll2));               // destination

    // copy the elements of coll1 into coll3 by inserting them at the
    // front - reverses the order of the elements
    deque<int> coll3;
    copy (coll1.begin(), coll1.end(),          // source
          front_inserter(coll3));              // destination

    // copy elements of coll1 into coll4
    // - only inserter that works for associative collections
    set<int> coll4;
    copy (coll1.begin(), coll1.end(),          // source
          inserter(coll4, coll4.begin()));     // destination
}
```

此例運用了三種預先定義的 *insert iterators*：

1. Back inserters (安插於容器最尾端)

Back inserters 的內部呼叫 `push_back()`，在容器尾端插入元素（此即「附加」動作）。例如以下述句完成之後，`coll1` 的所有元素都會被附加到 `coll2` 中：

```
copy (coll1.begin(), coll1.end(), // source
      back_inserter(coll2));      // destination
```

當然，只有在提供有 `push_back()` 成員函式的容器中，*back inserters* 才能派上用場。在 C++ 標準程式庫中，這樣的容器有三：`vector`, `deque`, `list`。

2. Front inserters (安插於容器最前端)

Front inserters 的內部呼叫 `push_front()`，將元素安插在容器的最前端。以下述句將 `coll1` 的所有元素插入 `coll3`：

```
copy (coll1.begin(), coll1.end(), // source
      front_inserter(coll3));      // destination
```

注意，這種安插動作逆轉了被插入元素的次序。如果你先安插 1，再向前安插 2，那麼 1 會排列在 2 的後面。

Front inserters 只能用於提供有 `push_front()` 成員函式的容器，在標準程式庫中，這樣的容器是 `deque` 和 `list`。

3. General inserters (一般性安插器)

這種一般性的 *inserter*，簡稱就叫 *inserter*，它的作用是在「初始化時所接受的第二個引數」所指示的位置之前方，插入元素。*Inserter* 內部呼叫成員函式 `insert()`，並以新值和新位置做為引數。所有 STL 容器都提供有 `insert()` 成員函式，因此，這是唯一可用於關聯式容器身上的一種預先定義好的 *inserter*。

等等，我不是說過，在關聯式容器身上安插新元素時，不能指定其位置嗎？它們的位置是由它們的值決定的啊！好了，我解釋一下，很簡單：在關聯式容器中，你所給的位置只是一個提示，幫助它確定從什麼地方開始搜尋正確的插入位置。如果提示不正確，效率上的表現會比「沒有提示」更糟糕。7.5.2 節（288 頁）介紹了一個用戶自定的 *inserter*，對關聯式容器特別有用。

表 5.1 列出了 *insert iterators* 的功能。7.4.2 節 271 頁將介紹更多細節。

| 算式 (expression) | Inserter 種類 |
|--|---|
| <code>back_inserter(container)</code> | 使用 <code>push_back()</code> 在容器尾端安插元素，元素之排列次序與安插次序相同。 |
| <code>front_inserter(container)</code> | 使用 <code>push_front()</code> 在容器前端安插元素，元素之排列次序與安插次序相反。 |
| <code>inserter(container, pos)</code> | 使用 <code>insert()</code> 在 <code>pos</code> 位置上安插元素，元素之排列次序與安插次序相同。 |

表 5.1 預先定義的三種 *Insert Iterators*

5.5.2 Stream Iterators (串流迭代器)

另一種非常有用的迭代器配接器 (iterator adapter) 是 *stream iterator*，這是一種用來讀寫 *stream*⁸ 的迭代器。它們提供了必要的抽象性，使得來自鍵盤的輸入像是一個群集 (collection)，而你能夠從中讀取內容。同樣道理，你也可以把一個演算法的輸出結果，重新導向到某個檔案或螢幕上。

下面是展示 STL 威力的一個典型例子。與一般的 C 和 C++ 相比，本例僅用數條述句，就完成了大量複雜工作：

```
// stl/ioiter1.cpp

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    vector<string> coll;

    /* read all words from the standard input
     * - source: all strings until end-of-file (or error)
     * - destination: coll (inserting)
     */
    copy (istream_iterator<string>(cin),          // start of source
          istream_iterator<string>(),              // end of source
          back_inserter(coll));                    // destination

    // sort elements
    sort (coll.begin(), coll.end());

    /* print all elements without duplicates
     * - source: coll
     * - destination: standard output (with newline between elements)
     */
    unique_copy (coll.begin(), coll.end(),          // source
                 ostream_iterator<string>(cout, "\n")); // destination
}
```

⁸ Stream 是一個物件，用來表現 I/O 通道（詳見第 13 章）。

這個程式只用了三個述句，就完成一系列工作：從標準輸入裝置讀取所有文字，排序，然後將它們列印於螢幕。讓我們逐一思考這三個述句。下面這個述句：

```
copy (istream_iterator<string>(cin),
      istream_iterator<string>(),
      back_inserter(coll));
```

用到兩個 *input stream iterators*：

1. `istream_iterator<string>(cin)`

這會產生一個「可從標準輸入串流 (standard input stream) `cin`」讀取資料的 *stream iterator*⁹。其中的 *template argument* (模板引數) `string` 表示，這個 *stream iterator* 專司讀取該種型別的元素 (`string` 型別將在第 11 章介紹)。這些元素透過一般的 `operator>>` 被讀取進來。因此，每當演算法企圖處理下一個元素時，*istream iterator* 就會將這種企圖轉化為以下行動：

```
cin >> string
```

針對 `string` 而執行的輸入運算子 (*input operator*) 通常讀取以空白分隔的文字 (參見 492 頁)，因此上述演算法的行為將是「逐詞讀取」。

2. `istream_iterator<string>()`

呼叫 *istream iterators* 的 *default constructor*，產生一個代表「串流結束符號」(end-of-stream) 的迭代器，它所代表的意義是：你不能再從中讀取任何東西。

只要「不斷遞增前進之第一引數」不同於第二引數，演算法 `copy()` 就持續動作。這裏的 *end-of-stream* 迭代器正是作為區間終點之用，因此這個演算法便從 `cin` 中讀取所有的 `strings`，直到讀無可讀為止 (可能是因為到達了 *end-of-stream*，或可能是因為讀入過程中發生錯誤)。總而言之，演算法的資料源是「來自 `cin` 的所有文字」。在 *back inserter* 的協助下，這些文字被拷貝並插入 `coll` 中。

接下來的 `sort()` 演算法對所有元素進行排序：

```
sort (coll.begin(), coll.end());
```

最後，下面這個述句：

```
unique_copy (coll.begin(), coll.end(),
             ostream_iterator<string>(cout, "\n"));
```

將其中所有元素拷貝到目的地 `cout`。處理過程中，演算法 `unique_copy()` 會消除毗鄰的重複值。其中的運算式：

```
ostream_iterator<string>(cout, "\n")
```

⁹ 在某些老舊系統中，你必須使用 `ptrdiff_t` 作為第二個模板引數，才能產生出一個 *istream iterator* (參見 7.4.3 節，280 頁)。

會產生一個 *output stream iterator*，透過 `operator<<` 向 `cout` 寫入 `strings`。 `cout` 之後的第二個引數被用來作為元素之間的分隔符號，可有可無。本例指定為一個換行符號，因此每個元素都被列印於獨立的一行。

這個程式中的所有組件都是 `templates`（模板），所以你可以輕易改變程式，對其他型別進行排序，例如整數，或更複雜的物件。7.4.3 節 277 頁對於 *iostream iterators* 進行了更詳細的介紹，並附帶更多實例。

本例使用一個宣告和三個述句，對來自標準輸入裝置的所有文字（單詞）進行排序。你還可以更進一步，只用一個宣告和一個述句就搞定一切。詳見 228 頁。

5.5.3 Reverse Iterators (逆向迭代器)

第三種預先定義的迭代器配接器 (iterator adapters) 就是 *reverse iterators*，此物像是倒轉筋脈般地，以逆向方式進行所有操作。它將遞增 (increment) 運算子轉換為遞減 (decrement) 運算子，反之亦然。所有容器都可以透過成員函式 `rbegin()` 和 `rend()` 產生出 *reverse iterators*。例如：

```
// stl/riter1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // print all element in reverse order
    copy (coll.rbegin(), coll.rend(),           // source
          ostream_iterator<int>(cout, " "),     // destination
          cout << endl;
    )
}
```

其中的運算式：

```
coll.rbegin()
```

傳回一個由 `coll` 定義的 *reverse iterator*。這個迭代器可作為「對群集 `coll` 的元素逆向走訪」的起點。它指向群集的結尾位置（也就是最後一個元素的下一位置）。因此，運算式：

```
*coll.rbegin()
```

傳回的是最後一個元素的值。

對應地，運算式：

```
coll.rend()
```

所傳回的 *reverse iterator*，可作為「對群集 `coll` 的元素逆向走訪」的終點。它也是指向「逾尾」（*past-the-end*）位置，只不過方向相反，指的是容器內第一個元素的前一個位置。

以下運算式沒有定義：

```
*coll.rend()
```

同樣情況，以下運算式也沒有定義：

```
*coll.end()
```

注意，當某個位置上並無合法元素時，永遠不要使用 `operator*` 或 `operator->`。

如果採用 *reverse iterators*，所有演算法便可以不需特殊處理就以相反方向操作容器，這自然是美事一樁。使用 `operator++` 前進至下一個元素，被轉化為使用 `operator--` 後退至前一個元素。本例中的 `copy()`，「從尾到頭」地走訪所有 `coll` 元素。程式輸出如下：

```
9 8 7 6 5 4 3 2 1
```

你可以將一般的迭代器轉換為 *reverse iterators*，反之亦可。然而，對於具體某個迭代器而言，這樣的轉換會改變其所指的對象。這些細節在第 7.4.1 節 264 頁介紹。

5.6 更易型演算法 (Manipulating Algorithms)

譯註：根據實質意義，我不把 `manipulating algorithms` 譯為「操縱型」演算法。`manipulating algorithms` 是指「會刪除、重排、修改元素」的演算法，見 115 頁。該頁亦出現另一個相同意義的術語：`modifying algorithms`。有些書籍（例如 *Generic Programming and the STL*）採用 `mutating algorithms` 一詞。為此，我將這些相同意義的術語都譯為「更易型」演算法。我亦曾在某些書中採用「質變演算法」一詞。

某些演算法會變更目標區間的內容。更明確地說可能會刪除元素。一旦這種情況出現，請務必注意幾個特殊問題。本節將對此做出解釋。它們確實令人訝異，並體現了 STL「爲了將容器和演算法分離，以獲取彈性」而付出的代價。

5.6.1 移除 (Removing) 元素

演算法 `remove()` 自某個區間刪除元素。然而如果你用它來刪除容器中的所有元素，其行為肯定會讓你吃驚。例如：

```
// stl/remove1.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    // print all elements of the collection
    cout << "pre: ";
    copy (coll.begin(), coll.end(),           // source
          ostream_iterator<int>(cout, " "),   // destination
          cout << endl;

    // remove all elements with value 3
    remove (coll.begin(), coll.end(),         // range
            3);                               // value

    // print all elements of the collection
    cout << "post: ";
```

```

    copy (coll.begin(), coll.end(),           // source
          ostream_iterator<int>(cout, " "); // destination
    cout << endl;
}

```

缺乏 STL 深度歷練的人，看了這程式，必然認為所有數值為 3 的元素都會從群集中被移除。然而，程式的輸出卻是這樣：

```

pre: 6 5 4 3 2 1 1 2 3 4 5 6
post: 6 5 4 2 1 1 2 4 5 6 5 6

```

啊呀，`remove()` 並沒有改變群集中的元素數量。`end()` 傳回的還是當初那個終點，`size()` 傳回的還是當初那個大小。不過某些事情還是有了變化：元素的次序改變了，有些元素被刪除了。數值為 3 的元素被其後的元素覆蓋了（圖 5.7）。至於群集尾部那些沒有被覆蓋的元素，原封未動。從邏輯的角度來說，這些元素已經不屬於這個群集了。

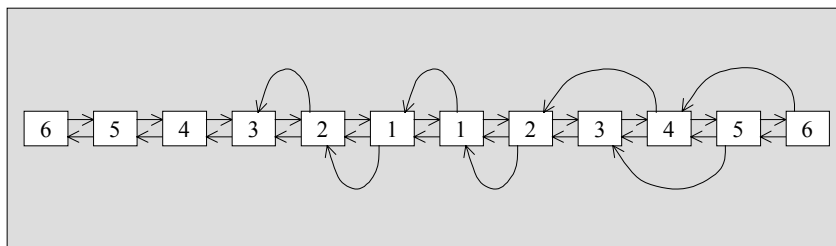


圖 5.7 `remove()` 如何運作

事實上，這個演算法傳回了一個新的終點。你可以利用這個值獲得新區間、縮減容器的大小、或者獲得被刪除元素的個數。看看下面這個改進版本：

```

// stl/remove2.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6

```

```
for (int i=1; i<=6; ++i) {
    coll.push_front(i);
    coll.push_back(i);
}

// print all elements of the collection
copy (coll.begin(), coll.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;

// remove all elements with value 3
// - retain new end
list<int>::iterator end = remove (coll.begin(), coll.end(),
                                 3);

// print resulting elements of the collection
copy (coll.begin(), end,
      ostream_iterator<int>(cout, " "));
cout << endl;

// print number of resulting elements
cout << "number of removed elements: "
      << distance(end, coll.end()) << endl;

// remove "removed" elements
coll.erase (end, coll.end());

// print all elements of the modified collection
copy (coll.begin(), coll.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;
}
```

在這個版本中，`remove()`的回傳值被設定給 `end` 迭代器：

```
list<int>::iterator end = remove (coll.begin(), coll.end(),
                                 3);
```

這個 `end` 正是「被修改之群集」經過元素移除動作後，邏輯上的新終點。接下來你便可以拿它當作新的終點使用：

```
copy (coll.begin(), end,
      ostream_iterator<int>(cout, " "));
```

另一種可能用法是，藉由測定群集之「邏輯」終點和實際終點間的距離，獲得「被刪除元素」的數量：

```
cout << "number of removed elements: "
      << distance(end, coll.end()) << endl;
```

在這裡，一個針對迭代器而設計的輔助函式 `distance()` 發揮了作用。它的功用是傳回兩個迭代器之間的距離。如果這兩個迭代器都是隨機存取迭代器（*random access iterators*），你可以使用 `operator-` 直接計算其距離。不過本例所用的容器是 `list`，只提供雙向迭代器（*bidirectional iterators*）。關於 `distance()` 的細節，詳見 7.3.2 節 261 頁¹⁰。

如果真想把那些被刪除的元素斬草除根，你必須呼叫該容器的相應成員函式。容器所提供的成員函式 `erase()`，正適用於此目的。`erase()` 可以刪除「其引數所指示之區間」內的全部元素：

```
coll.erase (end, coll.end());
```

下面是整個程式的完整輸出：

```
6 5 4 3 2 1 1 2 3 4 5 6
6 5 4 2 1 1 2 4 5 6
number of removed elements: 2
6 5 4 2 1 1 2 4 5 6
```

如果你需要以單一述句來刪除元素，可以如此這般：

```
coll.erase (remove(coll.begin(), coll.end(),
                    3),
            coll.end());
```

為何演算法不自己呼叫 `erase()` 呢？哎，這個問題正好點出 STL 爲了獲取彈性而付出的代價。透過「以迭代器爲介面」，STL 將資料結構和演算法分離開來。然而，迭代器只不過是「容器中某一位置」的抽象概念而已。一般來說，迭代器對自己所屬的容器一無所知。任何「以迭代器訪問容器元素」的演算法，都不得（無法）透過迭代器呼叫容器類別所提供的任何成員函式。

這個設計導致一個重要結果：演算法的操作對象不一定得是「容器內的全部元素」所形成的區間，而可以是那些元素的一個子集。甚至演算法可運作於一個「並未提供成員函式 `erase()`」的容器上（`array` 就是個例子）。所以，爲了達成演算法的最大彈性，不要求「迭代器必須了解其容器細節」還是很有道理的。

¹⁰ `distance()` 的定義已有所變化。在 STL 舊式版本中，爲了使用它，你必須含入 `distance.hpp`，見 263 頁。

注意，通常並無必要刪除那些「已被移除」的元素。通常，以邏輯終點來取代容器的實際終點，就足以應對。你可以以此邏輯終點搭配任何演算法。

5.6.2 更易型演算法和關聯式容器

更易型演算法（指那些會移除 *remove*、重排 *resort*、修改 *modify* 元素的演算法）用於關聯式容器身上會出問題。關聯式容器不能被當作目標，原因很簡單：如果更易型演算法用於關聯式容器身上，會改變某位置上的值，進而破壞其已序（*sorted*）特性，那就推翻了關聯式容器的基本原則：容器內的元素總是根據其排序準則自動排序。因此，為了保證這個原則，關聯式容器的所有迭代器均被宣告為指向常量（不變量）。如果你更易關聯式容器中的元素，會導致編譯錯誤¹¹。

注意，這使你無法在關聯式容器身上運用移除性（*removing*）演算法，因為這類演算法實際上悄悄更易了元素：「被移除元素」被其後的「未被移除元素」覆蓋。

現在問題來了，如何從關聯容器中刪除元素？唔，很簡單：呼叫它們的成員函式！每一種關聯式容器都提供用以移除元素的成員函式。例如你可以呼叫 `erase()` 來移除元素：

```
// stl/remove3.cpp

#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
    set<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.insert(i);
    }

    // print all elements of the collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout, " "));
}
```

¹¹ 糟糕的是，有些系統提供的錯誤處理能力令人不敢恭維。面對錯誤，你無法找出原因。有些編譯器甚至連出錯的源碼都不列出來。希望這種狀況在不久的將來獲得改善。

```

cout << endl;

/* Remove all elements with value 3
 * - algorithm remove() does not work
 * - instead member function erase() works
 */
int num = coll.erase(3);

// print number of removed elements
cout << "number of removed elements: " << num << endl;

// print all elements of the modified collection
copy (coll.begin(), coll.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;
}

```

注意，容器類別提供了多個不同的 `erase()` 成員函式。其中一種形式是以「待刪除之元素值」為唯一引數，它會傳回被刪除的元素個數（第 242 頁）——當然，在禁止元素重複的容器中（例如 `sets` 和 `maps`），其傳回值永遠只能是 0 或 1。

本節範例程式輸出如下：

```

1 2 3 4 5 6 7 8 9
number of removed elements: 1
1 2 4 5 6 7 8 9

```

5.6.3 演算法 vs. 成員函式

就算我們符合種種條件，得以使用某個演算法，那也未必就一定好。容器本身可能提供功能相似而性能更佳的成員函式。

一個極佳例子便是，對 `list` 的元素呼叫 `remove()`。演算法本身並不知道它工作於 `list` 身上，因此，它在任何容器中都一樣，做些四平八穩的工作：改變元素值，從而重新排列元素。如果它移除第一個元素，後面所有元素就會分別被設給各自的前一個元素。這就違反了 `lists` 的主要優點——藉由修改鏈結（`links`）而非實值（`values`）來安插、移動、移除元素。

爲了避免這麼糟糕的表現，`list` 針對所有「更易型」演算法提供了一些對應的成員函式。是的，如果你使用 `list`，你就應該使用這些成員函式。此外請注意，這些成員函式真的移除了「被移除」的元素，如下例所示：

```
// stl/remove4.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    // remove all elements with value 3
    // - poor performance
    coll.erase (remove(coll.begin(), coll.end(),
                       3),
               coll.end());

    // remove all elements with value 4
    // - good performance
    coll.remove (4);
}
```

如果高效率是你的最高目標，你應該永遠優先選用成員函式。問題是，你必須先知道，某個容器確實存在有效率上明顯突出的成員函式。面對 `list` 而卻使用 `remove()` 演算法，你決不會收到任何警告訊息或錯誤通告。然而如果你決定使用成員函式，一旦換用另一種容器，就不得不更動程式碼。在第 9 章的演算法參考小節中，如果有某一個成員函式的性能優於某個演算法，我會明白指出。

5.7 使用者自定之泛型函式 (User-Defined Generic Functions)

STL 乃是一個可擴展的框架 (framework)。這意味你可以撰寫自己的函式和演算法，處理群集中的元素。當然，這些操作函式本身也可以是泛型的 (generic)。

爲了在其中宣告有效的迭代器，你必須使用容器型別，因爲每一種容器都有自己的迭代器。爲了讓我們方便寫出真正的泛型函式，每一種容器都提供了一些內部的型別定義。請看下面的例子：

```
// stl/print.hpp

#include <iostream>

/* PRINT_ELEMENTS()
 * - prints optional C-string optcstr followed by
 * - all elements of the collection coll
 * - separated by spaces
 */
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

本例定義出一個泛型函式，可列印一個（可有可無的）字串，然後列印容器的全部元素。以下宣告式：

```
typename T::const_iterator pos;
```

其中的 `pos` 被宣告爲「被傳入之容器型別」內的迭代器型別，關鍵字 `typename` 在此不可或缺，用以表明 `const_iterator` 是型別 `T` 所定義的一個型別，而不是一個型別爲 `T` 的值（請見第 11 頁對 `typename` 的介紹）。

除了 `iterator` 和 `const_iterator` 之外，容器還提供了其他（內部定義的）型別，幫助你寫出泛型函式。例如它提供了元素型別，以便在元素臨時拷貝時派上用場。詳見 7.5.1 節，285 頁。

`PRINT_ELEMENTS` 的第二個引數是個可有可無的前綴字，列印時放在所有元素之前。你可以這樣使用 `PRINT_ELEMENTS()`：

```
PRINT_ELEMENTS (coll, "all elements: ");
```

我之所以介紹這個函式，因爲本書剩餘部份會大量運用它來列印容器的所有元素。

5.8 以函式做為演算法的引數

一些演算法可以接受用戶定義的輔助性函式，由此提高其彈性和能力。這些函式將在演算法內部被呼叫。

5.8.1 實例示範「以函式做為演算法的引數」

最簡單的例子莫過於 `for_each()` 演算法了。它針對區間內的每一個元素，呼叫一個由用戶指定的函式。下面是個例子：

```
// stl/foreach1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// function that prints the passed argument
void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    vector<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // print all elements
    for_each (coll.begin(), coll.end(),      // range
              print);                        // operation
    cout << endl;
}
```

這裏的 `for_each()` 函式針對 `[coll.begin(), coll.end())` 區間內的每個元素呼叫 `print()` 函式。輸出如下：

```
1 2 3 4 5 6 7 8 9
```

演算法以數種態度來面對這些輔助函式：有的視之為可有可無，有的視之為必要。你可以利用它們來指定搜尋準則、排序準則、或者定義某種操作，以便將某個容器內的元素轉換至另一個容器。

下面是個運用實例：

```
// stl/transform1.cpp

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include "print.hpp"

int square (int value)
{
    return value*value;
}

int main()
{
    std::set<int> coll1;
    std::vector<int> coll2;

    // insert elements from 1 to 9 into coll1
    for (int i=1; i<=9; ++i) {
        coll1.insert(i);
    }
    PRINT_ELEMENTS(coll1,"initialized: ");

    // transform each element from coll1 to coll2
    // - square transformed values
    std::transform (coll1.begin(),coll1.end(),    // source
                    std::back_inserter(coll2),    // destination
                    square);                      // operation

    PRINT_ELEMENTS(coll2,"squared: ");
}
```

此例之中，`square()` 的作用是將 `coll1` 內的每一個元素予以平方運算，然後轉移到 `coll2`（圖 5.8）。輸出如下：

```
initialized: 1 2 3 4 5 6 7 8 9
squared:    1 4 9 16 25 36 49 64 81
```

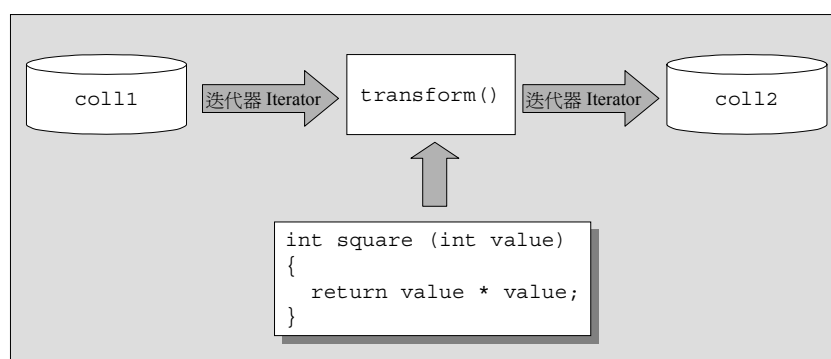


圖 5.8 `transform()` 的運作方式

5.8.2 條件式 (Predicates)

演算法的一類特殊輔助函式叫做 `predicates`（條件式）。所謂 `predicates`，就是回傳值為布林值（`boolean value`）的函式。它們通常被用來指定排序準則和搜尋準則。`Predicates` 可能有一個運算元或兩個運算元，視具體情形而定。注意，並非任何傳回布林值的一元函式或二元函式，就是個合法的 `predicate`。`STL` 要求，面對相同的值，`predicates` 必須得出相同的結果。這條戒律將那些「被呼叫時，會改變自己內部狀態」的函式清除出場。細節請見 8.1.4 節 302 頁。

Unary Predicates（一元條件式）

`Unary predicates` 會檢查唯一引數的某項特性。典型的例子是這樣一個用來搜尋第一個質數的函式：

```
// stl/prime1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include <cstdlib>      // for abs()
using namespace std;
```

```
// predicate, which returns whether an integer is a prime number
bool isPrime (int number)
{
    // ignore negative sign
    number = abs(number);

    // 0 and 1 are prime numbers
    if (number == 0 || number == 1) {
        return true;
    }

    // find divisor that divides without a remainder
    int divisor;
    for (divisor = number/2; number%divisor != 0; --divisor) {
        ;
    }

    // if no divisor greater than 1 is found, it is a prime number
    return divisor == 1;
}

int main()
{
    list<int> coll;

    // insert elements from 24 to 30
    for (int i=24; i<=30; ++i) {
        coll.push_back(i);
    }

    // search for prime number
    list<int>::iterator pos;
    pos = find_if (coll.begin(), coll.end(), // range
                  isPrime); // predicate
    if (pos != coll.end()) {
        // found
        cout << *pos << " is first prime number found" << endl;
    }
    else {
```

```
        // not found
        cout << "no prime number found" << endl;
    }
}
```

在這個例子中，`find_if()` 演算法在給定區間內，尋找第一個使「被傳入之 unary predicate」運算結果為 `true` 的元素。本例中的 predicate 是 `isPrime()` 函式，它會檢查某數是否為質數。透過它，這個演算法可以傳回給定區間內的第一個質數。如果沒有任何元素能夠匹配這個（質數）條件，`find_if()` 演算法就傳回區間終點（也就是第二個引數）。本例中，24 到 30 之間確實存在一個質數，所以程式輸出為：

```
29 is first prime number found
```

Binary Predicates（二元條件式）

Binary predicates 的典型用途是，用來比較兩個引數的特定屬性。例如，為了依照你自己的原則對元素排序，你必須以一個簡單的 predicate 形式提供這項原則。如果元素本身不支持 `operator<`，或如果你想使用不同的排序原則，這就派上用場了。

下面這個例子，根據每個人的姓名，對一組元素進行排序：

```
// stl/sort1.cpp

#include <iostream>
#include <string>
#include <deque>
#include <algorithm>
using namespace std;

class Person {
public:
    string firstname() const;
    string lastname() const;
    ...
};

/* binary function predicate:
 * - returns whether a person is less than another person
 */
```

```

bool personSortCriterion (const Person& p1, const Person& p2)
{
    /* a person is less than another person
    * - if the last name is less
    * - if the last name is equal and the first name is less
    */
    return p1.lastname() < p2.lastname() ||
        (! (p2.lastname() < p1.lastname()) &&
         p1.firstname() < p2.firstname());
}

int main()
{
    deque<Person> coll;
    ...
    sort(coll.begin(), coll.end(), // range
         personSortCriterion);    // sort criterion
    ...
}

```

注意，你也可以使用仿函式（**functor**，或名 **function object**）來實作一個排序準則。這種作法的優點是，製作出來的準則將是一個型別（**type**），可用來做為諸如「宣告一個 **set**，以上述型別為排序準則」之類的事情。詳見 8.1.1 節，294 頁。

5.9 仿函式 (Functors, Function Objects)

譯註：本書英文版通篇採用的術語是 **function object**，對應之譯名為「函式物件」。此物在 STL 發展初期曾經名為 **functor**，取其音義，我譯為「仿函式」。考量 STL 六大組件之譯名整體性，以及「術語最好具備獨特性，且不與其他名詞混淆」的原則，再考慮上下文閱讀的順暢性，我認為「仿函式」較「函式物件」為佳。為此，本中文版將 **function object** 全以 **functor** 取代之，並譯為「仿函式」。

傳遞給演算法的「函式型引數」（**functional arguments**），並不一定得是函式，可以是行為類似函式的物件。這種物件稱為 **function object**（函式物件），或稱 **functor**（仿函式）。當一般函式使不上勁的時候，你可以使用仿函式。STL 大量運用仿函式，也提供了一些很有用的仿函式。

5.9.1 什麼是仿函式

仿函式是泛型程式設計強大威力與純粹抽象概念的又一個例証。你可以說，任何東西，只要它的行為像函式，它就是個函式。所以，如果你定義了一個物件，行為像函式，它就可以被當作函式來用。

好，那麼，什麼才算是函式行爲？所謂函式行爲，是指可以「使用小括號傳遞引數，藉以呼叫某件東西」。例如：

```
function(arg1,arg2); // a function call
```

如果你指望物件也可以如此這般，就必須讓它們也有可能被「呼叫」——透過小括號的運用和引數的傳遞。沒錯，這是可能的（在 C++ 中，很少有什麼是不可能的）。你只需定義 `operator()`，並給予合適的參數型別：

```
class X {
public:
    // define 'function call' operator
    return-value operator() (arguments) const;
    ...
};
```

現在，你可以把這個類別的物件當作函式來呼叫了。

```
X fo;
...
fo(arg1,arg2); // call operator () for function object fo
```

上述呼叫等同於：

```
fo.operator() (arg1,arg2); // call operator () for function object fo
```

下面是個完整例子，是先前 119 頁範例的一個仿函式版本，其行爲與使用一般函式（而非仿函式）完全相同：

```
// stl/foreach2.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// simple function object that prints the passed argument
class PrintInt {
public:
    void operator() (int elem) const {
        cout << elem << ' ';
    }
};

int main()
{
    vector<int> coll;
```

```

// insert elements from 1 to 9
for (int i=1; i<=9; ++i) {
    coll.push_back(i);
}

// print all elements
for_each (coll.begin(), coll.end(), // range
          PrintInt());              // operation
cout << endl;
}

```

PrintInt 所做的定義顯示，你可以對它的物件呼叫 operator()，並傳入一個 int 引數。至於述句：

```

for_each (coll.begin(), coll.end(),
          PrintInt());

```

其中的運算式：

```
PrintInt()
```

產生出此類別的一個臨時物件，當作 for_each() 演算法的一個引數。for_each() 演算法大致如下：

```

namespace std {
    template <class Iterator, class Operation>
    Operation for_each (Iterator act, Iterator end, Operation op)
    {
        while (act != end) {           // as long as not reached the end
            op(*act);                  // - call op() for actual element
            ++act;                     // - move iterator to the next element
        }
        return op;
    }
}

```

for_each() 使用暫時物件 op（一個仿函式），針對每個元素呼叫 op(*act)。如果第三參數是個一般函式，就以*act 為引數呼叫之。如果第三參數是個仿函式，則以*act 為引數，呼叫仿函式 op 的 operator()。因此，本例之中，for_each() 呼叫：

```
PrintInt::operator() (*act)
```

你也許不以爲然，你也許認爲仿函式看起來怪異、令人討厭、甚或毫無意義。的確，它們帶來更複雜的程式碼，然而仿函式有其過人之處，比起一般函式，它們有以下優點：

1. 仿函式是 "smart functions" (精靈函式)

「行為類似指標」的物件，我們稱為 "smart pointers"。「行為類似函式」的物件呢？同樣道理，我們可以稱之為 "smart functions"，因為它們的能力可以超越 `operator()`。仿函式可擁有 `data members` 和 `member functions`。這意味著仿函式擁有狀態(state)。事實上，在同一時間裡，由某個仿函式所代表的單一函式，可能有不同的狀態。這在一般函式中是不可能的。另一個好處是，你可以在執行期(runtime)初始化它們 — 當然要在使用（也就是呼叫）它們之前。

2. 每一個仿函式都有自己的型別

一般函式，唯有在它們的標記式(signatures)不同時，才算型別不同。而仿函式即使標記式相同，也可以有不同的型別。事實上，由仿函式定義的每一個函式行為(functional behavior)都有其自己的型別。這對於「利用 `template` 實現泛型程式設計」乃是一個卓越貢獻，因為如此一來，我們便可以將函式行為當做 `template` 參數來運用。這使得不同型別的容器可以使用同類型的仿函式作為排序準則。這可以確保你不會在排序準則不同的群集(collections)之間賦值、合併或比較。你甚至可以設計仿函式繼承體系(functors hierarchies)，以此完成一些特別事情，例如在一個總體原則下確立某些特殊情況。

3. 仿函式通常比一般函式速度快

就 `template` 概念而言，由於更多細節在編譯期就已確定，所以通常可能進行更好的最佳化。所以，傳入一個仿函式（而非一般函式），可能獲得更好的性能。

這一小節的剩餘部分，我會給出幾個例子，展示仿函式較之一般函式的優勢所在。第 8 章專攻仿函式，有更多的例子和細節。特別是那一章為你展示「以函式行為(functional behavior)作為 `template` 參數」這項技術帶給我們的利益。

假設你需要對群集(collection)中的每個元素加上一個固定值。如果你在編譯時期便確切知道這個固定數，你可以使用一般函式：

```
void add10 (int& elem)
{
    elem += 10;
}

void f1()
{
    vector<int> coll;
    ...
    for_each (coll.begin(), coll.end(),      // range
              add10);                       // operation
}
```

如果你需要數個不同的固定值，而它們在編譯期都已確切，你可以使用 `template`:

```
template <int theValue>
void add (int& elem)
{
    elem += theValue;
}

void f1()
{
    vector<int> coll;
    ...
    for_each (coll.begin(), coll.end(),      // range
              add<10>);                      // operation
}
```

如果你必須在執行時期才處理這個數值，那可就麻煩了。你必須在函式被呼叫之前先將這個數值傳給該函式。這通常會導致產生一些全域變數，「演算法的呼叫者」和「演算法所呼叫的函式」都會用到它們。真是一團糟。

如果你兩次用到該函式，每次加數不同，而都是在執行時期才處理，那麼一般函式根本就無能為力。你要嘛傳入一個標記（`tag`），要嘛乾脆寫兩個函式。你是否有過這樣的經歷：你已握有一個函式，它有個 `static` 變數用以記錄狀態（`state`），而你需要這個函式在同一時間裡有另一個不同的狀態（`state`）？於是你只好拷貝整份函式定義，化為兩個不同的函式。這正是先前所說的問題。

如果使用仿函式，你就可以寫出「更機靈」的函式，遂你所願。物件可以有自己的狀態，可以被正確初始化。下面是一個完整例子¹²：

```
// stl/add1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

// function object that adds the value with which it is initialized
class AddValue {
```

¹² 輔助函式 `PRINT_ELEMENTS()` 已於 p118 介紹過。

```
private:
    int theValue; // the value to add
public:
    // constructor initializes the value to add
    AddValue(int v) : theValue(v) {
    }

    // the 'function call' for the element adds the value
    void operator() (int& elem) const {
        elem += theValue;
    }
};

int main()
{
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    PRINT_ELEMENTS(coll, "initialized: ");

    // add value 10 to each element
    for_each (coll.begin(), coll.end(), // range
              AddValue(10));           // operation

    PRINT_ELEMENTS(coll, "after adding 10: ");

    // add value of first element to each element
    for_each (coll.begin(), coll.end(), // range
              AddValue(*coll.begin())); // operation

    PRINT_ELEMENTS(coll, "after adding first element: ");
}
```

初始化之後，群集內含數值 1 至 9：

```
initialized: 1 2 3 4 5 6 7 8 9
```

第一次呼叫 `for_each()`，將每個數值加 10：

```
for_each (coll.begin(), coll.end(),    // range
          AddValue(10));               // operation
```

這裡，運算式 `AddValue(10)` 生出一個 `AddValue` 物件，並以 10 為初值。`AddValue` constructor 將這個值保存在成員 `theValue` 之中。而在 `for_each()` 之內，針對 `coll` 的每一個元素都呼叫 `"()`”，實際上就是對於傳入的那個暫時的 `AddValue` 物件呼叫 `operator()`，並以容器元素作為引數。仿函式（`AddValue` 物件）將每個元素加 10。結果如下：

```
after adding 10: 11 12 13 14 15 16 17 18 19
```

第二次呼叫 `for_each()` 亦採用相同機能，將第一元素值加到每個元素身上。首先使用第一元素值做為仿函式暫時物件的初值：

```
AddValue(*coll.begin())
```

最後結果如下：

```
after adding first element: 22 23 24 25 26 27 28 29 30
```

p335 有這個例子的改進版，其中 `AddValue` 仿函式的型別被改為一個 `template`，可接納不同的加數。

運用此項技術，先前所說的「一個函式、兩個狀態」的問題就可以用「兩個不同的仿函式」加以解決。例如，你可以宣告兩個仿函式，然後各自運用：

```
AddValue addx(x); // function object that adds value x
AddValue addy(y); // function object that adds value y

for_each (coll.begin(), coll.end(), // add value x to each element
          addx);
...
for_each (coll.begin(), coll.end(), // add value y to each element
          addy);
...
for_each (coll.begin(), coll.end(), // add value x to each element
          addx);
```

同樣道理，你也可以提供一些 `member functions`，在仿函式的生命期間查詢或改變物件狀態。

注意，C++ 標準程式庫並未限制演算法「對著一個容器元素」呼叫仿函式的次數，因此可能導致同一個仿函式有若干副本被傳給元素。如果把仿函式當做判斷式（`predicates`）使用，這個問題會惹來一身麻煩。8.1.4 節, p302 討論了這個問題。

5.9.2 預先定義好的仿函式

C++標準程式庫包含了一些預先定義的仿函式，涵蓋許多基礎運算。有了它們，很多時候你就不必費心自己去寫仿函式了。一個典型的例子是作為排序準則的仿函式。operator< 之預設排序準則乃是 less<>，所以，如果你宣告：

```
set<int> coll;
```

會被擴展為¹³：

```
set<int,less<int> > coll;    // sort elements with <
```

既然如此，想必你能猜到，反向排列這些元素將不是什麼難事¹⁴：

```
set<int,greater<int> > coll; // sort elements with >
```

類似情況，還有許多仿函式用於數值處理。下例是將群集中的全部元素都設為反相（負值）：

```
transform (coll.begin(), coll.end(),    // source
           coll.begin(),                // destination
           negate<int>());              // operation
```

其中運算式：

```
negate<int>()
```

根據預先定義好的 `template class negate` 生成一個仿函式，將傳進來的 `int` 值設定為負。transform() 演算法使用此一運算，將第一群集的所有元素處理之後轉移到第二群集。如果轉移目的地就是自己，那麼這段程式碼就是「對群集中的每一個元素取負值」。

同樣道理，你也可以對群集中的所有元素求平方（二次方）。

```
// process the square of all elements
transform (coll.begin(), coll.end(),    // first source
           coll.begin(),                // second source
           coll.begin(),                // destination
           multiplies<int>());          // operation
```

這裡運用了 transform() 演算法的另一種形式，以某種特定運算，將兩群集中的元素處理之後的結果，寫入第三群集。由於本例中的三個群集實際上是同一個，

¹³ 有些系統並不支持 default template arguments，那麼你只能使用後一種形式。

¹⁴ 注意，兩個 ">" 之間必須保留一個空格，否則 ">>" 會被解析為右移運算子(shift operator)，因而發生語法錯誤。

所以其內的每個元素都被計算了平方值，並寫進群集之中，改寫原有的值¹⁵。

透過一些特殊的函式配接器（function adaptors），你還可以將預先定義的仿函式和其他數值組合在一起，或使用各種特殊狀況。下面是一個完整範例：

```
// stl/fo1.cpp

#include <iostream>
#include <set>
#include <deque>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    set<int,greater<int> > coll1;
    deque<int> coll2;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll1.insert(i);
    }

    PRINT_ELEMENTS(coll1,"initialized: ");

    // transform all elements into coll2 by multiplying 10
    transform (coll1.begin(),coll1.end(),           // source
               back_inserter(coll2),                // destination
               bind2nd(multiplies<int>(),10));       // operation

    PRINT_ELEMENTS(coll2,"transformed: ");

    // replace value equal to 70 with 42
    replace_if (coll2.begin(),coll2.end(),          // range
               bind2nd(equal_to<int>(),70),         // replace criterion
               42);                                  // new value

    PRINT_ELEMENTS(coll2,"replaced: ");
```

¹⁵ STL 早期版本中，乘法運算仿函式名為 `times`。但這與某些作業系統(POSIX, X/Open)中用以計算時間的函式名稱衝突了，所以後來改為更清楚的名稱：`multiplies`。

```

// remove all elements with values less than 50
coll2.erase(remove_if(coll2.begin(), coll2.end(), // range
                     bind2nd(less<int>(), 50)), // remove criterion
            coll2.end());

PRINT_ELEMENTS(coll2, "removed: ");
}

```

其中的述句：

```

transform (coll1.begin(), coll1.end(), // source
          back_inserter(coll2), // destination
          bind2nd(multiplies<int>(), 10)); // operation

```

將 coll1 內的所有元素乘以 10 後轉移（安插）到 coll2 中。這裡使用配接器 bind2nd，使得進行 multiplies<int> 運算時，以源群集的元素作為第一引數，10 作為第二引數。

配接器 bind2nd 的工作方式如下：transform() 期望它自己的第四引數是一個能接納單一引數（也就是容器實際元素）的運算式，然而我們卻希望先把該元素乘以 10，再傳給 transform()。所以我們必須構造出一個運算式，接受兩個引數，並以數值 10 作為第二引數，以此產生一個「只需單一引數」的運算式。bind2nd() 正好勝任這項工作。它會把運算式保存起來，把第二引數當作內部數值也保存起來。當演算法以實際群集元素為引數，呼叫 bind2nd 時，bind2nd 把該元素當作第一引數，把原先保存下來的那個內部數值作為第二個引數，呼叫保留下來的那個運算式，並傳回結果。（譯註：這種繁複的文字說明不具實際效用，實際情況（源碼運作）請看《STL 源碼剖析》by 侯捷，碁峰 2002）

類似情況，以下的：

```

replace_if (coll2.begin(), coll2.end(), // range
           bind2nd(equal_to<int>(), 70), // replace criterion
           42);

```

其中的運算式：

```
bind2nd(equal_to<int>(), 70)
```

被用來當作一項準則，判斷哪些元素將被 42 替代。bind2nd 以 70 作為第二引數，呼叫雙參數判斷式（binary predicate）equal_to，從而定義出一個單參數判斷式（unary predicate），處理群集中的每一個元素。

最後一個述句也是一樣：

```
bind2nd(less<int>(), 50)
```

它被用來判斷群集中的哪些元素應當被掃地出門。所有小於 50 的元素都被移除。程式輸出如下：

```

initialized: 9 8 7 6 5 4 3 2 1
transformed: 90 80 70 60 50 40 30 20 10
replaced: 90 80 42 60 50 40 30 20 10
removed: 90 80 60 50

```

此種類型的程式設計，導致了函式的組合。有趣的是，所有這些仿函式通常都宣告為 `inline`。如此一來，你一方面使用類似函式的表示法或抽象性，一方面又能獲得極出色的效能。

另外還有一些仿函式。某些仿函式可以用來呼叫群集中每個元素的 `member function`：

```

for_each (coll.begin(), coll.end(),           // range
          mem_fun_ref(&Person::save));       // operation

```

仿函式 `mem_fun_ref` 用來呼叫它所作用的元素的某個 `member function`。因此，上例，針對 `coll` 的每個元素呼叫 `Person::save()`。當然啦，唯有當這些元素的型別是 `Person`，或是 `Person` 的衍生類別，以上程式碼才能工作。

8.2 節，p305 對於預先定義的仿函式、函式配接器、以及函式組合，有更詳盡的討論，並告訴你如何撰寫你自己的仿函式。

5.10 容器的元素

容器中的元素必須符合特定條件，因為容器乃是以一種特別的方式來操作它們。本節討論這些條件。此外，容器會在內部對其元素進行複製，我也會討論這種行為的後果。

5.10.1 容器元素的條件

STL 的容器、迭代器、演算法，都是 `templates`，因此可以操作任何型別 — 不論 STL 預先定義好的，或是用戶自行定義的，都可以。然而，由於某些加諸於元素身上的操作行為，一些需求條件也就相應出現了。STL 容器元素必須滿足以下三個基本要求：

1. 必須可透過 `copy constructor` 進行複製。副本與原本必須相等 (`equivalent`)，亦即所有相等測試 (`equality test`) 的結果都必須顯示，原本與副本行為一致。所有容器都會在內部生成一個元素副本，並回傳其暫時性副本，因此 `copy constructor` 會被頻繁呼叫。所以，`copy constructor` 的性能應該儘可能最佳化（這雖然不是條件之一，但可視為獲得良好效能的訣竅）。如果物件的拷貝必須耗費大量時間，你可以選用「`reference` 語義」來使用容器，因而避免拷貝任何物件。詳見 6.8 節，p222。

2. 必須可以透過 `assignment operator` 完成賦值動作。容器和算法都是使用 `assignment operator`，才能以新元素改寫（取代）舊元素。
3. 必須可以透過 `destructor` 完成銷毀動作。當容器元素被移除（*removed*），它在容器內的副本將被銷毀。因此 `destructor` 絕不能被設計為 `private`。此外，依 C++ 慣例，`destructor` 絕不能丟擲異常（`throw exceptions`），否則沒戲唱了。

這三個條件對任何類別（`class`）而言其實都是隱喻成立的。如果某個類別既沒有為上述動作定義特殊版本，也沒有定義任何「可能破壞這些動作之健全性」的特殊成員，那麼它自然而然也就滿足了那些條件。

下面幾個條件，也應當獲得滿足¹⁶：

- 對序列式容器而言，元素的 `default constructor`（預設建構式）必須可用。
我們可以在沒有給予任何初值的情況下，創建一個非空容器，或增加容器的元素個數。這些元素都將以 `default constructor` 建構完成。
- 對於某些動作，必須定義 `operator==` 以執行相等性測試。如果你有搜尋需求，這一點特別重要。
- 在關聯式容器中，元素必須定義出排序準則（`sorting criterion`）。預設情況下是 `operator<`，透過仿函式 `less<>` 被叫用。

5.10.2 實值（value）語義 vs. 參用（reference）語義

所有容器都會建立元素副本，並回傳該副本。這意味容器內的元素與你放進去的物件「相等（`equal`）」但並非「同一（`identical`）」。如果你修改容器中的元素，實際上改變的是副本而不是原先物件。這意味 STL 容器所提供的是實值語義（`value semantics`）。它們所容納的是你所安插的物件值，而不是物件本身。然而在實用上，你也許需要用到參用語義（`reference semantics`），讓容器容納元素的 `reference`。

STL 只支援實值語義，不支援參用語義。這當然是利弊參半。好處是：

- 「拷貝元素」很簡單。
- 使用 `references` 容易導致錯誤。你必須確保 `reference` 所指向的物件仍然健在，並需小心對付偶爾出現的循環參用（`circular references`）狀態。

缺點是：

¹⁶ 在某些老式系統中，即使你未用到這些額外條件，也必須滿足它們。例如某些 `vector` 實作品無論如何用到元素的 `default ctor`（預設建構式）。另一些實作品則要求 `comparison`（比較）運算子必須存在。然而，根據標準，這些要求是錯誤的，所以它們會逐漸被取消。

- 「拷貝元素」可能導致不好的效能；有時甚至無法拷貝。
- 無法在數個不同的容器中管理同一份物件。

實用上你同時需要兩種作法。你不但需要一份獨立（於原先物件）的拷貝（此乃「實值語義」），也需要一份代表原資料、俾能相應改變原值的拷貝（此乃「參用語義」）。不幸的是，C++標準程式庫不支援「參用語義」。不過我們可以利用「實值語義」來實現「參用語義」。

一個顯而易見的方法是以指標作為元素¹⁷。然而一般指標有些常見問題。例如它們指向的物件也許不復存在，「比較」行為也未必如你所預期，因為實際比較的是指標，而非指標所指物件。所以使用一般指標作為容器元素，必須非常謹慎。

好一點的辦法是使用某種智慧型指標（smart pointers），所謂智慧型指標，是一種物件，有著類似指標的介面，但內部作一些額外的檢查和處理工作。這裡有一個重要的問題：它們需要多麼智慧？C++標準程式庫確實提供了一個智慧型指標，名為 `auto_ptr`（詳見 4.2 節，p38），乍見之下用於此處似乎頗合適。然而，你可千萬別使用 `auto_ptr`s，因為它們不符合作為容器元素所需具備的基本要求。當 `auto_ptr` 執行了拷貝（copy）或賦值（assign）動作後，標的物與原物並不相等：原來的那個 `auto_ptr` 發生了變化，其值並不是被拷貝了，而是被轉移了（見 p43 和 p47）。這意味即使是對容器中的元素進行排序和列印，也會摧毀它們！所以，千萬別在容器中放置 `auto_ptr`s（如果你的 C++ 系統符合標準規範，當你企圖將 `auto_ptr` 當作一個容器元素，你應該會收到錯誤訊息）。詳見 p43。

想要獲得適用於 STL 容器的「參用語義」，你必須自己寫個合適的智慧型指標。但請注意：就算你使用帶有參用計數(reference counting)功能的智慧型指標（譯註：請參考《More Effective C++》條款 28），也就是那種「當最後一個指向物件的 reference 不復存在後，能夠自動摧毀該物件」的智慧型指標，仍然很麻煩。舉個例子，如果你擁有直接存取元素的能力，你就可以更改元素的值，而這在關聯式容器中卻會打破元素順序關係。你肯定不想這樣，是吧！6.8 節 p222 更細緻地探討了容器參用語義這個主題。尤其是那裡展示了一種作法，通過「參用計數」智慧型指標，實現 STL 容器的參考語義。

5.11 STL 內部的錯誤和異常

錯誤是無可避免的，可能是程式（程式員）引起的邏輯性錯誤（logical error），也可能是程式運行時的環境或背景（例如記憶體不足）所引起的執行期錯誤（runtime error）。這兩種錯誤都能夠被異常機制（exceptions）所處理（p15 有一個關於異常的簡短介紹）。本節討論 STL 內部如何處理錯誤（error）和異常（exceptions）。

¹⁷ C 程式員或許很能認可「以指標實現參用語義」的手法。因為在 C 語言中，函式引數只能 passed by value（傳值），因此需要藉由指標，才能實現所謂的 call by reference。

5.11.1 錯誤處理

STL 的設計原則是效率優先，安全次之。錯誤檢查相當花費時間，所以幾乎沒有。如果你能正確無誤地編寫程式，自然很好。如果你不行，那就大難臨頭了。C++ 標準程式庫接納 STL 之前，對於是否應該加入更多的錯誤檢驗，曾有過一些討論。大部分人決定不加入，原因有二：

1. 錯誤檢驗會降低效率，而速度始終是程式的總體目標。剛剛才提過，良好的效率是 STL 的設計目標之一。
2. 如果你認為安全重於效率，你還是可以如願：或是增加一層包裝（wrapper），或是使用 STL 特殊版本。但是，一旦錯誤檢驗被放進所有基本動作之中，再想消除它們以獲得高效率，可就沒門了。舉個例子，如果每一個下標運算子（subscript operator）都對索引範圍進行合法性檢驗，你就無法撰寫不作檢驗的下標運算子版本。反過來就可以。

所以，錯誤檢驗是可行的，但並不是 STL 的內在條件。

C++ 標準程式庫指出，對於 STL 的任何運用，如果違反規則，將會導致未定義的行為。因此，如果索引（indexes）、迭代器（iterators）、或區間（ranges）不合法（invalid），結果將未有定義。如果你使用的 STL 並非安全版本，就會導致未定義的記憶體存取，這可能導致難纏的副作用，甚至導致全盤崩潰。從這個意義上說，STL 和 C 指標一樣容易引發錯誤。尋找這樣的錯誤是非常困難的，尤其當你缺乏一個 STL 安全版本時，更是如此。

具體地說，使用 STL，必須滿足以下要求：

- 迭代器務必合法而有效（valid）。例如你必須在使用它們之前先將它們初始化。注意，迭代器可能會因為其他動作的副效應而變得無效。例如當 vectors 和 deque 發生元素的安插、刪除或重新配置時，迭代器可能因此失效。
- 一個迭代器如果指向 past-the-end 位置，它並不指向任何物件。因此，不能對它們呼叫 operator* 或 operator->。這一點適用於任何容器的 end() 和 rend() 所傳回的迭代器。
- 區間（range）必須是合法的：
 - 用以指出某個區間的前後兩迭代器，必須指向同一個容器。
 - 從第一個迭代器出發，必須可以到達第二個迭代器所指位置。
- 如果涉及的區間不只一個，第二區間及後繼各區間必須擁有「至少與第一區間一樣多」的元素。
- 覆蓋（overwritten）動作中的「標的區間」（destination ranges）必須擁有足夠的元素，否則就必須採用 insert iterators（插入型迭代器）。

下面的例子展示了一些可能的錯誤：

```
// stl/iterbug1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll1;           // empty collection
    vector<int> coll2;           // empty collection

    /* RUNTIME ERROR:
    * - beginning is behind the end of the range
    */
    vector<int>::iterator pos = coll1.begin();
    reverse (++pos, coll1.end());

    // insert elements from 1 to 9 into coll2
    for (int i=1; i<=9; ++i) {
        coll2.push_back (i);
    }

    /* RUNTIME ERROR:
    * - overwriting nonexisting elements
    */
    copy (coll2.begin(), coll2.end(), // source
          coll1.begin());             // destination

    /* RUNTIME ERROR:
    * - collections mistaken
    * -begin() and end() mistaken
    */
    copy (coll1.begin(), coll2.end(), // source
          coll1.end());               // destination
}
```

注意，這些錯誤發生在執行期間，而不是編譯期間，因而導致未定義的行為。

誤用 STL 的方法百百種，STL 沒有義務防止你自己打擊自己。因此，在軟體開發階段使用「安全版本」的 STL，是個好主意。第一個 STL 安全版本由 Cay Horstmann 開發¹⁸。不幸的是大部分 STL 開發廠商所供應的 STL，都是植基於 STL 最原始版本，其中並未包含錯誤處理。但是情況正在好轉，有一個帶有警戒能力的 STL 版本，名為 "STLport"，幾乎適用於任何平台，可自 <http://www.stlport.org/> 免費下載。

¹⁸ 你可以從 www.horstmann.com/safestl.html 獲得一份由 Cay Horstmann 開發的 "safe STL"。

5.11.2 異常處理 (Exception Handling)

STL 幾乎不檢驗邏輯錯誤。所以邏輯問題幾乎不會引發 STL 產生異常。事實上 C++ 標準規格只要求一個函式呼叫動作必要時直接引發異常：`vector` 和 `deque` 的成員函式 `at()` (它是下標運算子的受驗版本)。此外，C++ 標準規格要求，只有一般的 (標準的) 異常可以發生，像是因記憶體不足而引發的 `bad_alloc`，或是因客戶自定之操作行為而引發的異常。

異常何時發生？異常發生時，對 STL 組件有何影響？在標準化過程中，很長一段時間裡，並未對此問題定義出相關的行為規範。事實上，每一個異常都會引發未定義的行為。如果執行某項動作的過程中丟擲出異常，那麼即使容器馬上解構，也會導致未定義行為，例如整體崩潰。因此，如果你需要的是有擔保的、確定的行為，STL 無能為力，它甚至不可能正確地將堆疊輾轉開解 (所謂 `stack unwinding`)。

如何處理異常，這是標準化過程中最後的幾個討論議題之一。找到好的解決方法可不容易，而且花了很長時間，因為：

1. 很難確定 C++ 標準程式庫究竟應該提供怎樣的安全程度。你大概認為應該儘可能提供最佳安全性。例如你可能覺得，對著 `vector` 中的任何位置插入一個新元素，要麼成功，要麼應該沒有任何效果。然而，把後繼元素向後移動以空出位置容納新元素，這種行為通常會導致異常，而且無法復原。如果想要達成上面提出的目標，安插動作就必須把 `vector` 的每一個元素拷貝到新位置去，這對效率是莫大的折損！如果優異效能是設計目標之一 (就像 STL)，你絕對無法完美處理所有異常狀況，必須在效率和安全之間尋求某種妥協。
2. 還有一種考量：異常處理代碼本身，也會對效能帶來負面影響。這與「儘可能獲得最佳效能」的設計目標抵觸。然而編譯器撰寫者指出，原則上，異常處理的實作方案應該可以免除任何明顯的效能負荷 (許多編譯器也確實做到了)。毫無疑問，如果效能沒有明顯損耗，又能在異常發生時擁有確定、有保障的行為 (而非當機了事)，那當然比較好。

經過了各種討論，C++ 標準程式庫就「異常處理問題」提供了以下基本保證¹⁹：C++ 標準程式庫在面對異常時，保證不會洩漏資源 (`leak resources`)，也不會與容器的不變性 (`container invariants`) 發生牴觸。

遺憾的是很多時候這還不夠，你需要更強的保證，保證當異常被丟出時，進行中的操作不產生任何影響。以異常的觀點來看，這種操作可被視為「不可切割的」 (`atomic`)。借用資料庫領域的一個術語，這些操作支援所謂「交付或回復，二擇一」 (`commit-or-rollback`) 行為，又稱為「安全交易行為」 (`transaction safe`)。

¹⁹ 特別感謝 Dave Abrahams 和 Greg Colvin 對於 C++ 標準程式庫的異常安全問題所作的貢獻，以及在這個主題上對我的幫助。

考慮到這種強烈的需求，C++標準程式庫如今做出以下保證：

- 對於所有「以節點為構造基礎」(node-based)的容器如 `lists`, `sets`, `multisets`, `maps` 和 `multimaps`，如果節點建構失敗，容器保持不變。移除節點的動作保證不會失敗（當然你得保證解構式不得丟出異常）。然而，如果是對關聯式容器插入多個元素，為保證已序性 (*sorted*)，失敗時無法完全恢復原狀。所有對著關聯式容器「插入單一元素」的操作，則支持 **commit-or-rollback** 行為。也就是說，要不成功，要不沒有任何影響。此外，所有擦拭 (`erase`) 操作，無論是針對單一元素或針對多重元素，肯定會成功。

面對 `lists`，就算同時插入多個元素，這個操作也是屬於「安全交易行為」(**transaction-safe**)。事實上 `list` 的所有操作，除了 `remove()`, `remove_if()`, `merge()`, `sort()` 和 `unique()` 之外，要不成功，要不沒有任何影響（也就是 **commit-or-rollback**）。至於上述各函式，C++標準程式庫也提供了有條件的保證（詳見 p172）。所以，如果你需要一個 **transaction-safe** 容器，就用 `list` 吧。

- 所有「以 `array` 為構造基礎」(array-based)的容器如 `vectors` 和 `deque`s，安插元素時如果失敗，都不可能做到完全回復。要達到完全回復，就必須在安插動作之前拷貝所有（安插點之後的）後繼元素。而且，為了實現拷貝動作的完全回復性，需要耗費大量時間。不過，由於 `push` 和 `pop` 這兩個動作在容器的尾端執行，不需拷貝任何既有元素，所以萬一發生異常，這兩個動作可以保證容器將會回復原狀。此外，如果元素的型別能夠保證拷貝動作（也就是 `copy constructor` 和 `assignment operator`）不丟出異常，則所有加諸於該種元素身上的操作，就都能夠保證「要不成功，要不毫無影響」的行為。

6.10.10 節，p248 有一份詳細整理，讓你對「異常發生時，擁有較強烈的保證」的各種容器操作，有一份了解。

注意，所以這些保證都有一個前提：解構式不得丟擲異常（C++ 中通常如此）。C++ 標準程式庫做了這個的承諾，身為應用程式員的你，也得做出相同的承諾。

如果你需要具備「完全 **commit-or-rollback** 能力」的容器，你應當使用 `list`（但不要呼叫它的 `sort` 和 `unique`），或使用任何關聯式容器（但不要對它安插多個元素）。當你使用它們，可以確保資料不會損失，也確保不會在任何「修改性動作」之前先拷貝元素 — 要知道，對一個容器而言，拷貝動作極可能代價高昂。

如果你不能使用「以節點為構造基礎」(node-based)的容器，而又希望獲得「完全的 **commit-or-rollback** 能力」，只好自己動手為每一個關鍵操作提供一份包裝 (`wrapper`) 了。舉個例子，以下函式對任何容器而言，幾乎都可以安全地將元素安插於某個特定位置上：

```
template <class T, class Cont, class Iter>
void insert (Cont& coll, const Iter& pos, const T& value)
{
```



```
Cont tmp(coll);           // copy container and all elements
tmp.insert(pos,value);    // modify the copy
coll.swap(tmp);           // use copy (in case no exception was thrown)
}
```

注意我的用詞，我說「幾乎」，因為這個函式仍然未臻完美。這是因為，對關聯性容器而言，`swap()` 在複製「比較規則（comparison criterion）」時，會丟擲異常。這下你明白了吧，想完美地處理異常是多麼不容易！

5.12 擴展 STL

STL 被設計成一個框架（framework），可以向任何方向擴展。你可以提供自己的容器、迭代器子、演算法、仿函式，只要你滿足條件，即可。事實上，很多有用的擴展都沒有出現在 C++ 標準程式庫中。不能非難他們，C++ 標準委員會必須在某個時刻停止加入新特性，將精力集中於現有特性的完善上，否則標準化工作永無完結之日。STL 中遺漏的最重要組件是 hash table（屬於容器類）。這完全是因為它太晚被提出之故。新的標準程式庫很可能包含數種不同形式的 hash table。大部分 C++ 標準程式庫的實作品已經提供了 hash table，只可惜彼此之間有些差異。詳見 6.7.3 節，p221。

另一些有用的擴展品是一些額外的仿函式（8.3 節，p313）、迭代器（7.5.2 節，p288）、容器（6.7 節，p217）和演算法（7.5.1 節，p285）。

