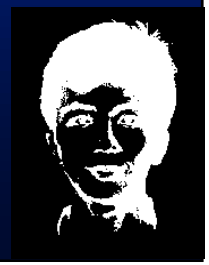
泛談
GP/OO/C++

侯捷

2006/07/24

Yahoo, Beijing



STL/GP 五個層次

- 運用 STL
- C++ Template 語法和語意
- 理解 STL 內部構造與 GP 關鍵技術
- 訂製 STL 相容組件
- 使用其他 GP 產品，運用 GP 技術

STL 優缺點分析

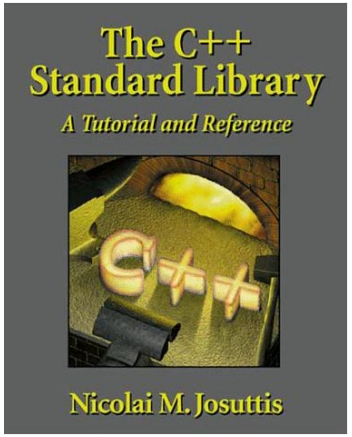
- 速度效能：佳（複雜度有所規範）
- 空間效能：

  1對容器而言，templates 帶來的膨脹是無可避免之惡
  2 對演算法而言，可能因爲過度設計而帶來過大的程式碼
  （尤其應用於 embedded system 時）

- 額外紅利：可對源碼進行客製化（customize）

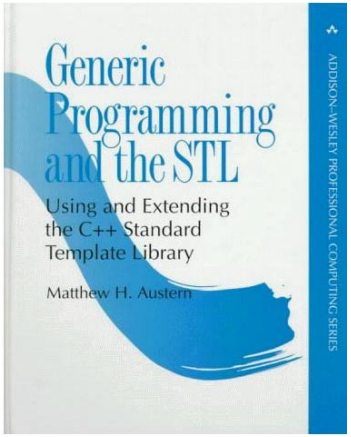
Bibliography

STL百科全書

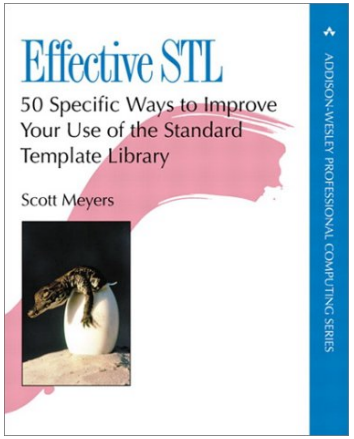


源碼剖析

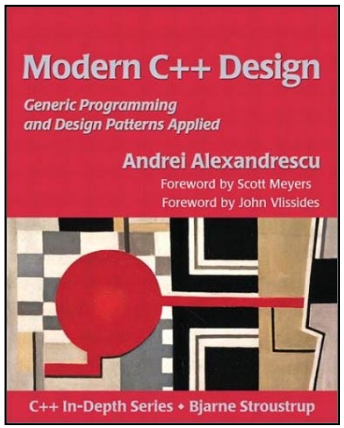
學理與規格

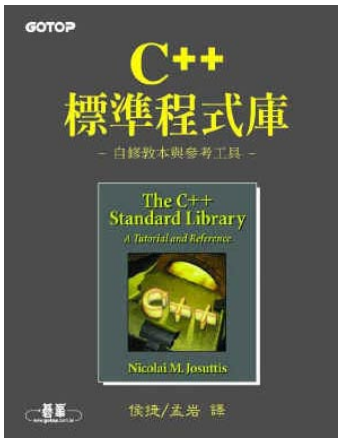


專家運用經驗

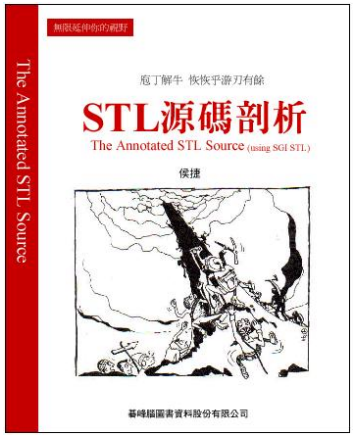


前瞻與極致

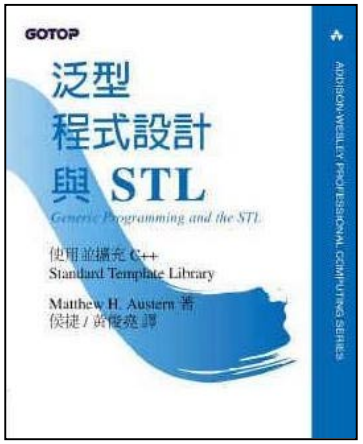




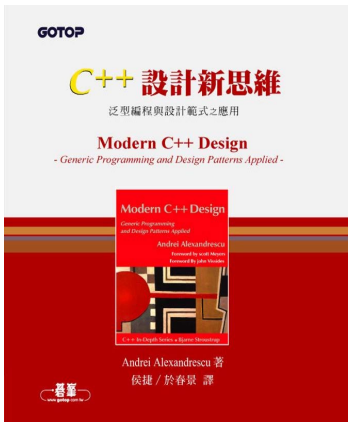
侯捷/孟岩 譯



侯捷 著



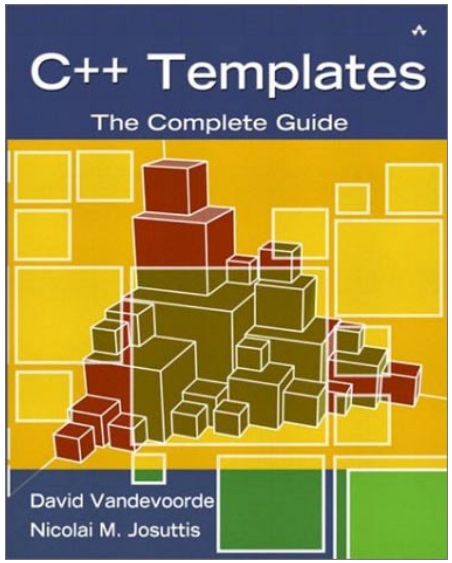
侯捷/黃俊堯 譯



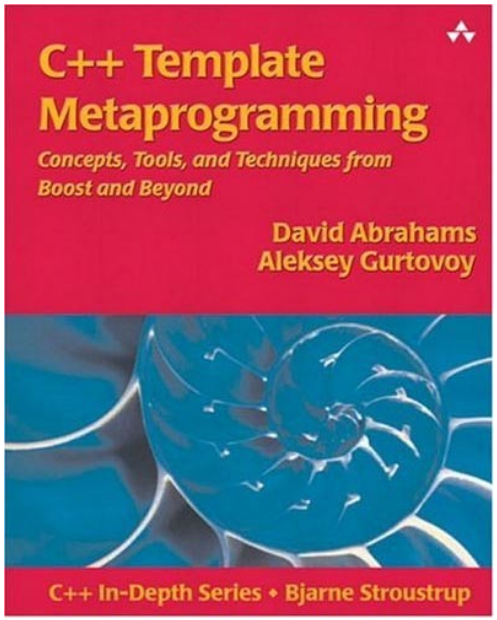
侯捷/於春景 譯

Bibliography

C++ Templates 百科全書



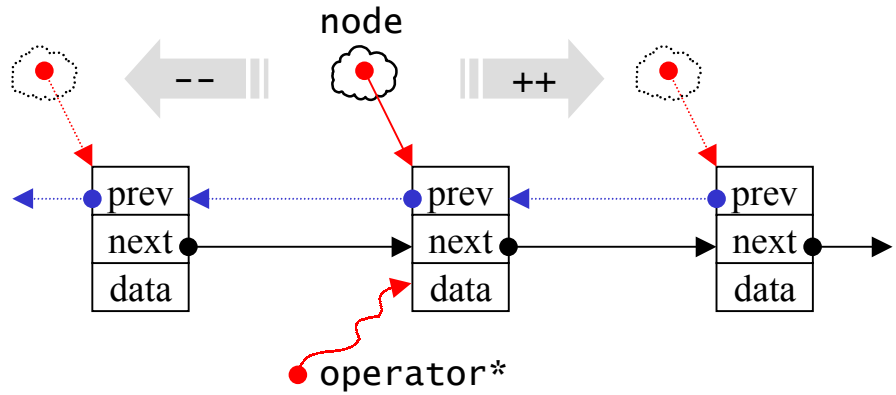
侯捷/榮耀/姜宏 合譯



侯捷/榮耀 合譯

C++ operator overloading, overview

```
__list_iterator<int, int&, int*>
__list_iterator<C, C&, C*>
```




```
template<class T, class Ref, class Ptr>
struct __list_iterator {
  typedef __list_iterator<T, Ref, Ptr> self;
  typedef Ptr pointer;
  typedef Ref reference;
  typedef __list_node<T>* link_type;
  link_type node;  // 原生指標指向節點
  bool operator==(const self& x) const { return node == x.node; }
  bool operator!=(const self& x) const { return node != x.node; }
  reference operator*() const { return (*node).data; }
  pointer operator->() const { return &(operator*()); }
  self& operator++() { node = (link_type)((*node).next); return *this; }
  self operator++(int) { self tmp = *this;  ++*this; return tmp; }
  self& operator--() { node = (link_type)((*node).prev); return *this; }
  self operator--(int) { self tmp = *this; --*this;  return tmp; }
};
```

```
template <class T>
struct __list_node {
  void* prev;
  void* next;
  T data;
};
```

C++ template, overview

所謂 template，是一種實現 parameterized types（generalized types）的 C++ 語法。template 是「倚賴用戶提供之型別資訊、於編譯期才產生」的程式碼。

- class template：「帶有 parameterized type」的 classes
- function template：「帶有 parameterized type」的 functions
- member template：「帶有 parameterized type」的 members

C++ class template

```
template <typename T>
class Complex {
public:
  Complex(const T& re=0, const T& im=0)
      : m_real(re), m_imag(im)  { }
  T real() const { return m_real; }
  T imag() const { return m_imag; }
  T real(const T& re) { m_real = re; return m_real; }
  T imag(const T& im) { m_imag = im; return m_imag; }
  Complex& operator+=(const Complex& x) {
    m_real += x.m_real;
    m_imag += x.m_imag;
    return *this;
  }
  const Complex operator+(const Complex& x) {
    return Complex<T>(*this) += x;
  }
  Complex operator*(const Complex& x) {
    return Complex<T>( m_real * x.m_real - m_imag * x.m_imag,
                       m_real * x.m_imag + m_imag * x.m_real );
  }
private:
  T m_real;  // real part（實部）
  T m_imag;  // imaginary part（虛部）
};
```

template 參數可分爲 type 和 non-type 兩種，個數無限。C++ 不允許「名稱相同但 template 參數個數不同」的 class templates。

運用實例：

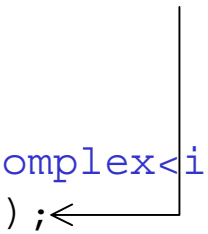
```
Complex<double> c1(1,2), c2(2,3), c3;
c3 = c1 + c2;
```

Complex 並不是個 type，Complex<T> 才是

實例

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

template<typename T>
struct printElem
{
   void operator()(T elem) const { cout << elem << endl; }
};

int main()
{
    std::complex<int> c1(1,2);
    cout << c1 << endl;

    vector< complex<int> > vci;
    vci.push_back(complex<int>(1,2));
    vci.push_back(complex<int>(3,4));
    vci.push_back(complex<int>(5,6));
    for_each(vci.begin(), vci.end(), printElem< complex<int> >());
    printElem< complex<int> >()(complex<int>(1,2));
}
```

這種用法比較少



C++ function template

any type object 都可以運用 min()，
只要該 type 支援 operator<。

關鍵字可改用typename

運用實例：

```
rect r1(2,3), r2(3,3), r3;
r3 = min(r1, r2);
```

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

編譯器會對 function template 進行
引數推導（argument deduction）

```
class rect
{
public:
  rect(int w, int h) : _w(w), _h(h)
      { _area = _w * _h; }
  bool operator< (const rect& rhs) const
      { return _area < rhs._area; }
private:
  int _w, _h, _area;
};
```

引數推導的結果，T 爲 rect，於是
喚起 rect::operator<

function templates 可被重載，例如：

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

partial specialization for class template

partial specialization（偏特化）的意思是：只針對 **template 參數之局部**進行特化。此處之局部可以是**參數的個數**，也可以是**型別參數的更狹窄範圍**。

type param. non-type param

```
// form 1
template <class T, int hi, int wid>
class Screen { … };

// form 2 (partial specialization)
template <class T, int hi>
class Screen <T, hi, 80> { … };

// form 3 (partial specialization)
template <class T, int hi>
class Screen <T*, hi, 25> { … };

// form 4（specified argument 不一定得靠右）
template <class T, int wid>
class Screen <T, 30, wid> { … };
```

```
int main()
{
  Screen<int, 100, 40>   s1;     /
  Screen<int, 100, 80>   s2;     /
  Screen<int, 500, 25>   s3;     /
  Screen<char*, 300, 25> s4;     /
  Screen<double*, 400, 25> s5;   /
  Screen<long, 30, 999> s6;      /
  Screen<long, 30, 80> s7;       /
}
```

```
template <typename T>
class C
{
  ...
};
```

```
template <typename T>
class C<T*>
{
  ...
};
```

重要

traits（特性）

- type traits（SGI-STL, Boost 都提供）
- character traits （用於 string）
- iterator traits（每一 STL 實作版本都提供）

C++ STL 實作版本

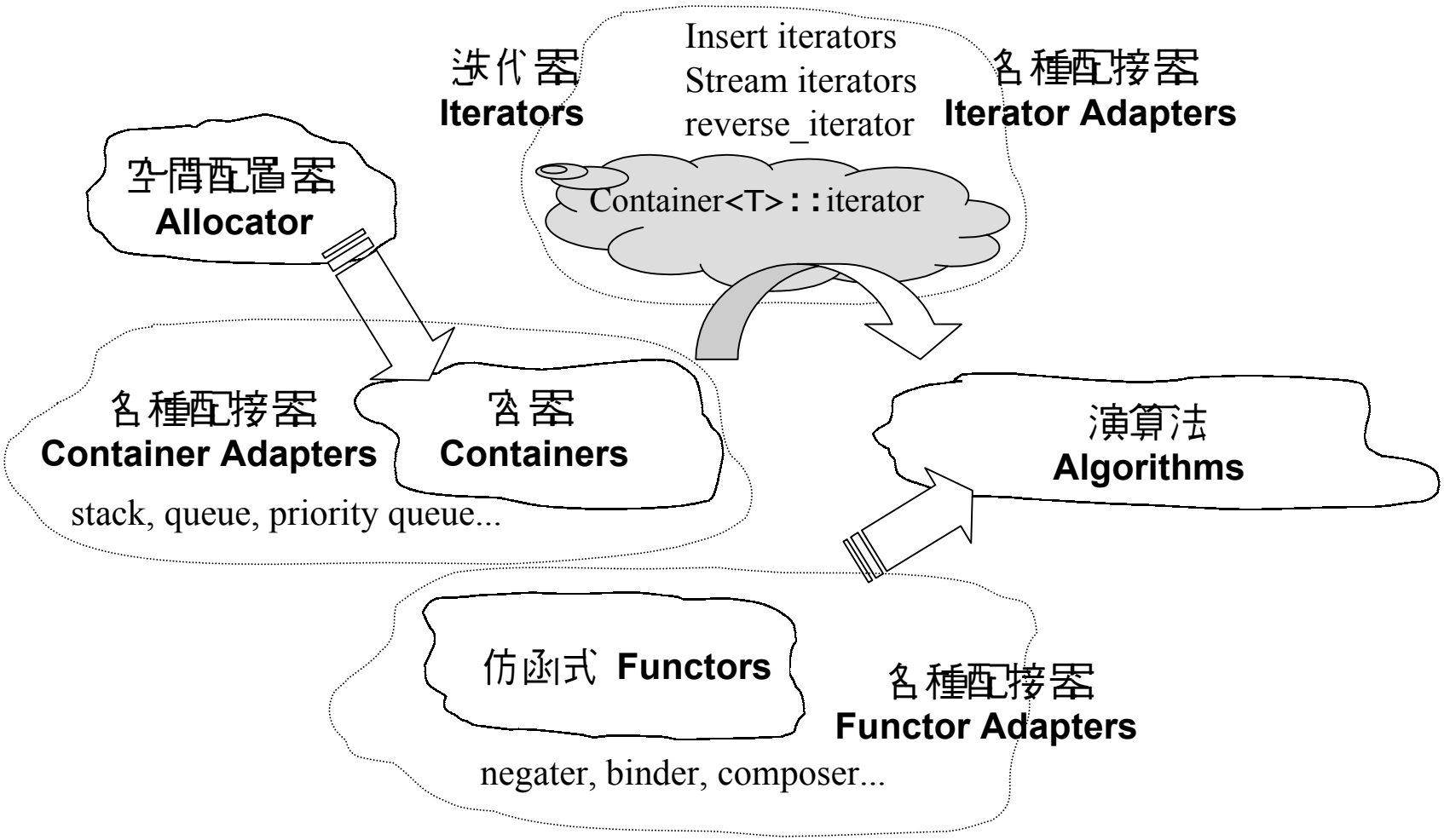
STL 實作版本：
•Hewlett-Packard（**HP** version）
•Microsoft VC6: P.J. Plauger（**PJ** version）
•Inprise C++Builder4: Rogue Wave Software, Inc（**RW** version）
•GNU C++ egcs-2.91.57 : Silicon Graphics Computer Systems, Inc （**SGI** version）

•C++ 標準程式庫的 header files 不帶副檔名（.h），例如 vector
•舊式 C++ header files（帶有副檔名 .h，例如 vector.h）仍然可用
•新式 C header files 不帶副檔名 .h，例如 cstdio
•舊式 C header files（帶有副檔名 .h，例如 stdio.h）仍然可用

•新式 headers內的組件封裝於 namespace "std" 中
•舊式 headers內的組件不封裝於 namespace "std" 中

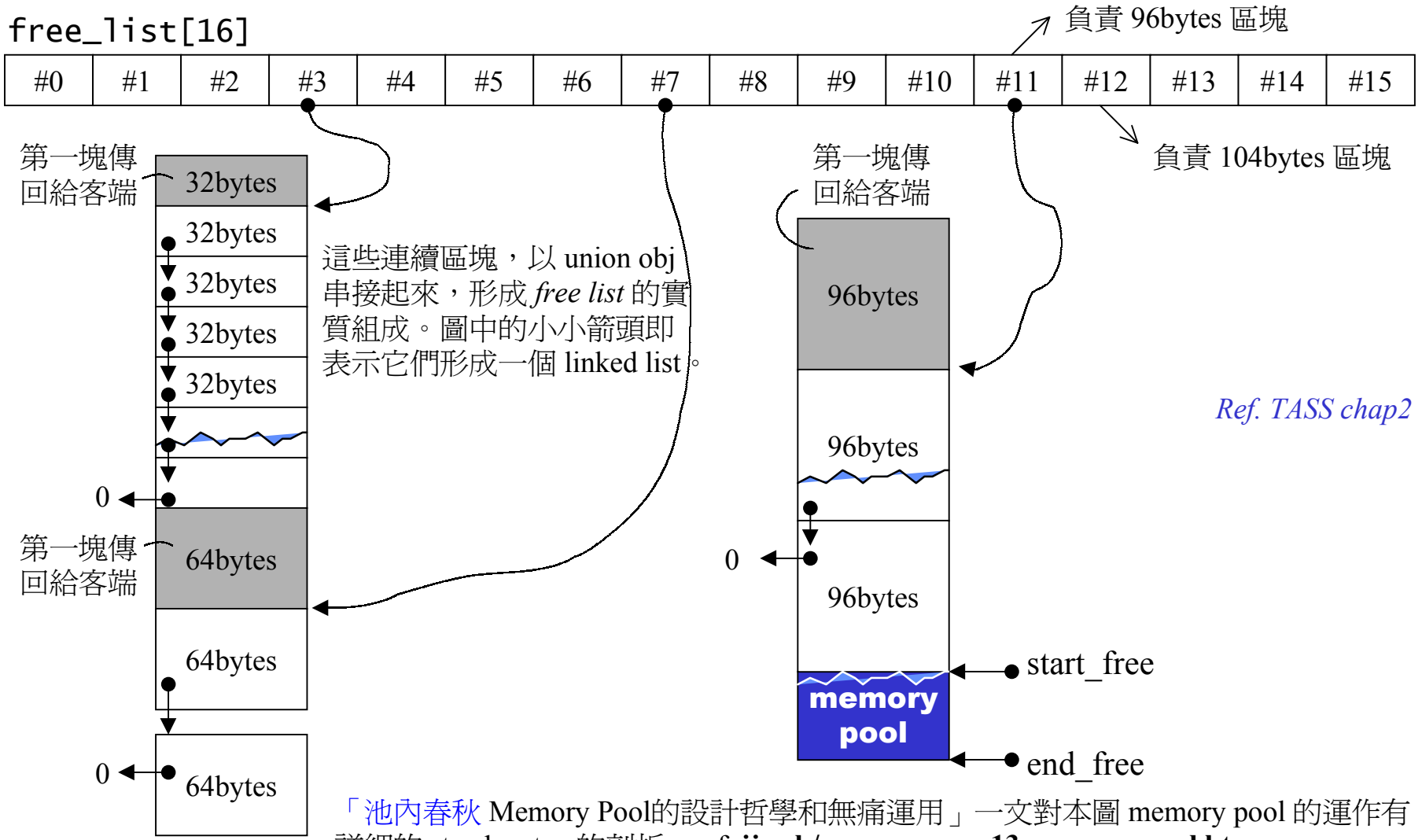
STL 以 header files 型式呈現

STL六大組件


迭代器
Iterators
Insert iterators
Stream iterators
reverse_iterator
各種配接器
Iterator Adapters
空間配置器
Allocator
Container<T>::iterator
各種配接器
Container Adapters
容器
Containers
演算法
Algorithms
stack, queue, priority queue...
仿函式 Functors
各種配接器
Functor Adapters
negater, binder, composer...

STL Allocator（配置器）: pooled memory

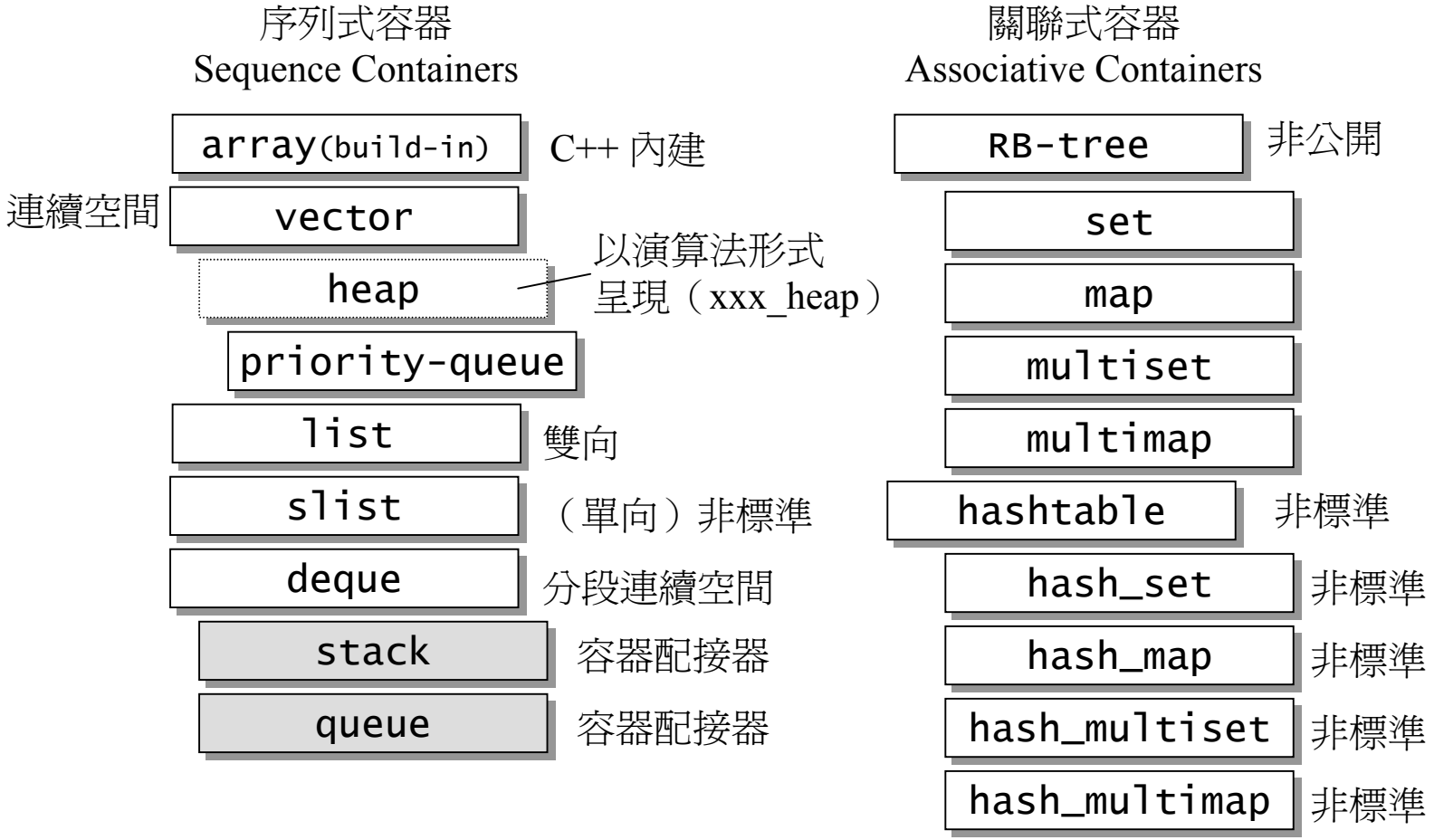
有些 allocator（例如 SGI STL） 設計了 memory pool 機制：



Ref. TASS chap2

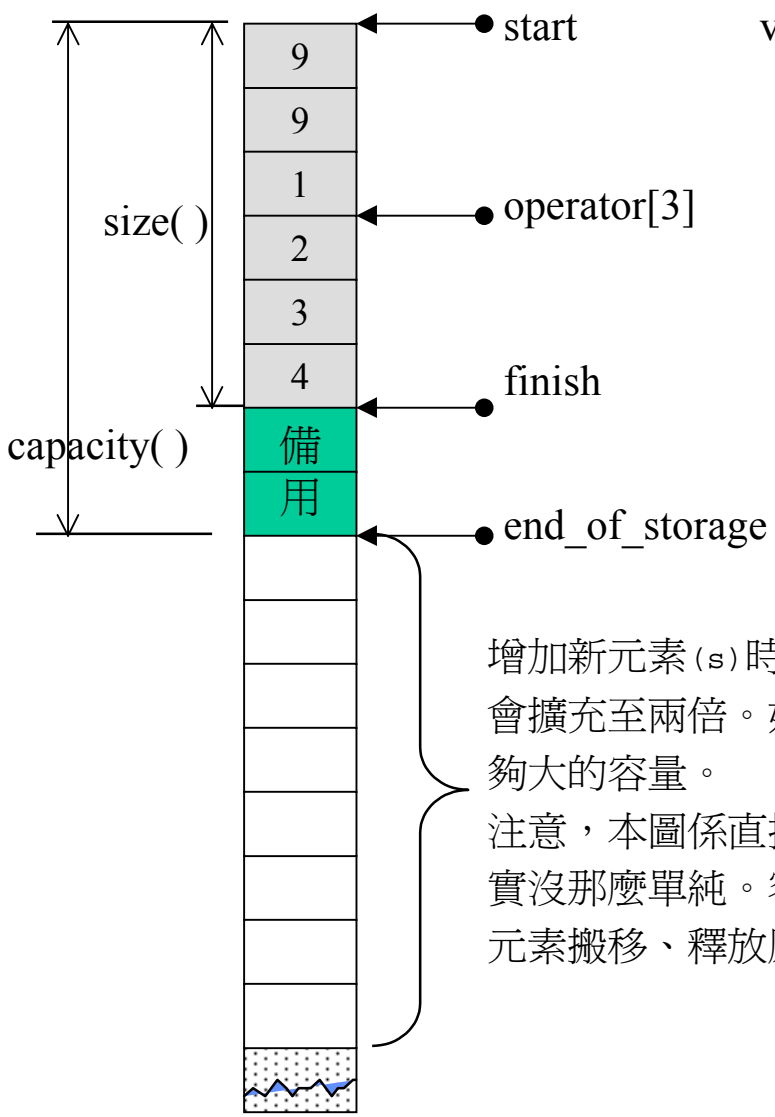
「池內春秋 Memory Pool的設計哲學和無痛運用」一文對本圖 memory pool 的運作有詳細的 step by step 的剖析。ref. **jjweb/programmer-13-memory-pool.htm**

STL Containers（容器）: overview



本圖以內縮方式來表達基層與衍生層的關係。
這裡所謂的衍生，並非繼承（inheritance）關係，而是複合（composition）關係。

vector



vector 是連續空間；提供巡訪操作及iterators。

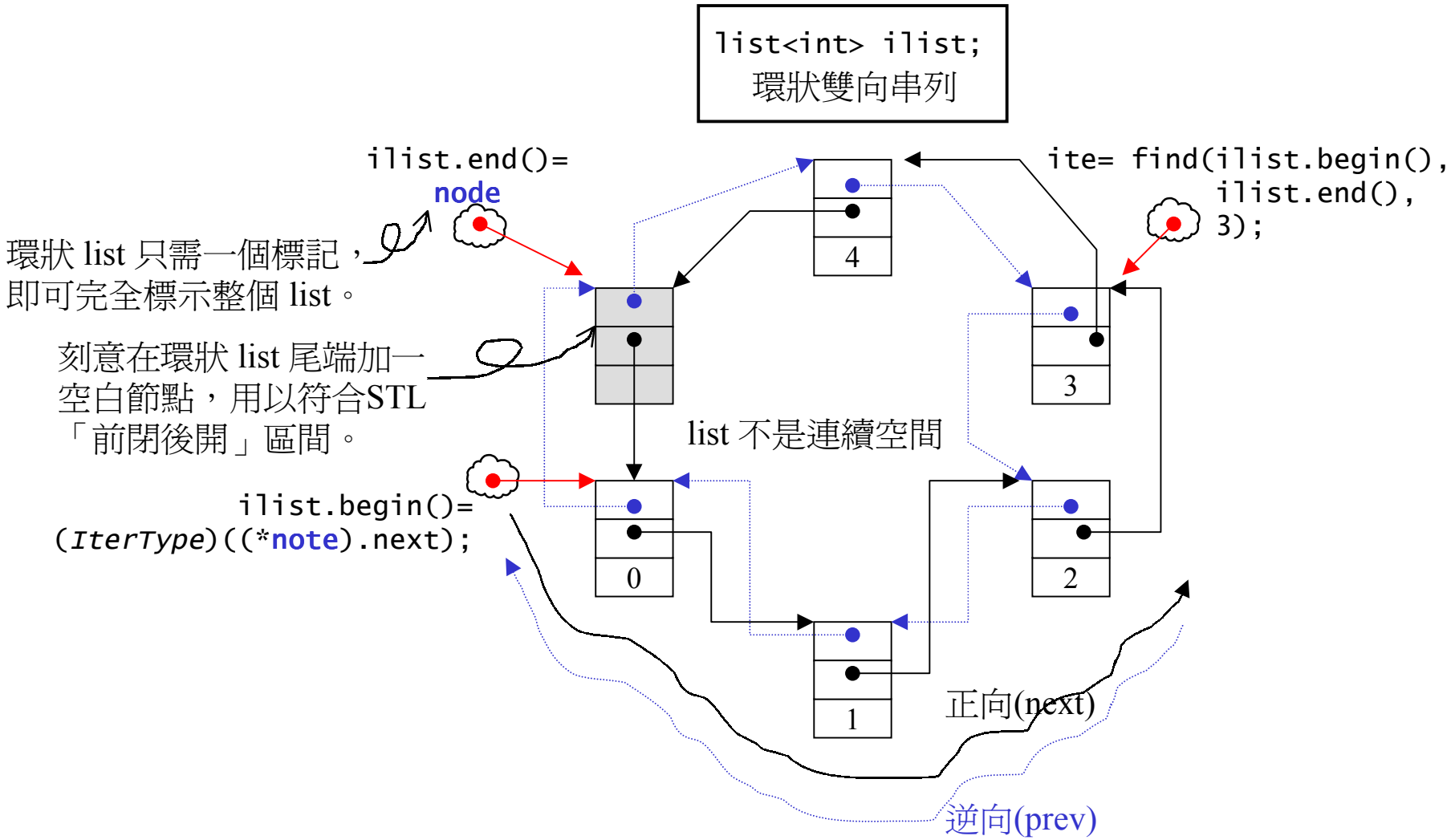
經過以下動作，vector記憶體及各成員呈現左圖狀態

```
vector<int> iv(2, 9);
iv.push_back(1);
iv.push_back(2);
iv.push_back(3);
iv.push_back(4);
```

增加新元素(s)時，如果超過當時的容量，則容量會擴充至兩倍。如果兩倍容量仍不足，就擴張至足夠大的容量。

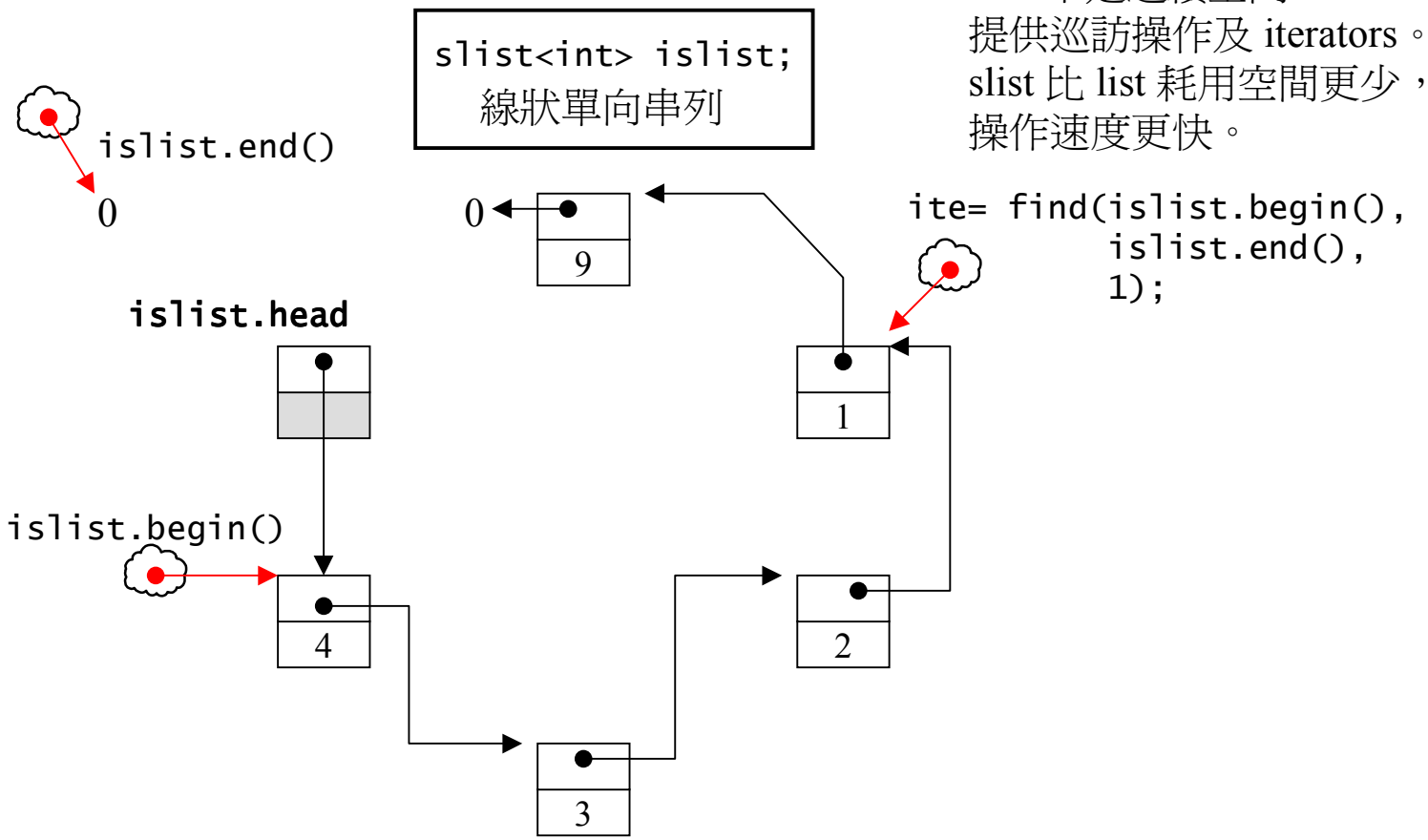
注意，本圖係直接在原空間之後畫上新增空間，其實沒那麼單純。容量的擴張必須經歷「重新配置、元素搬移、釋放原空間」等過程，工程浩大。

list


list<int> ilist;
環狀雙向串列
ilist.end()=
node
環狀 list 只需一個標記，
即可完全標示整個 list。
刻意在環狀 list 尾端加一
空白節點，用以符合STL
「前閉後開」區間。
ilist.begin()=
(IterType)((*note).next);
ite= find(ilist.begin(),
ilist.end(),
3);
list 不是連續空間
4
3
0
2
1
正向(next)
逆向(prev)

slist

slist 不是連續空間；
提供巡訪操作及 iterators。
slist 比 list 耗用空間更少，某些操作速度更快。

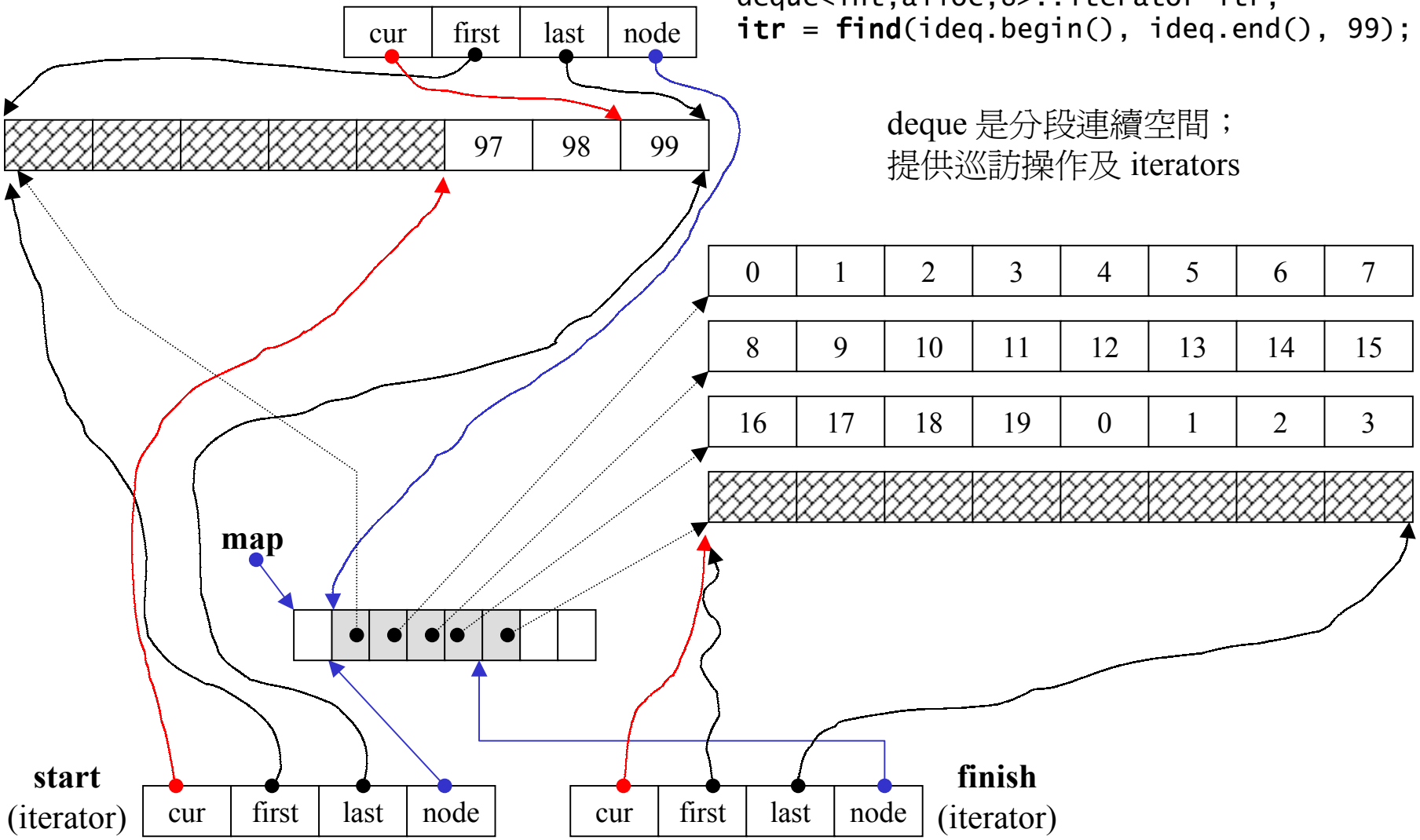


以上是元素 9,1,2,3,4 依序安插到 `slist` 後，形成的結構。
由於 slist 不提供 push_back()，只提供 push_front()，因此 slist 的元素次序會和安插時的次序恰恰相反。

deque

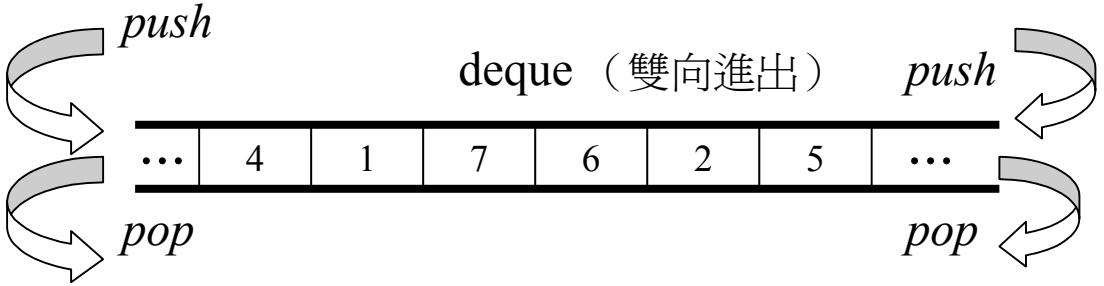
```
deque<int,alloc,8> di;
...//push_back一些元素
deque<int,alloc,8>::iterator itr;
itr = find(ideq.begin(), ideq.end(), 99);
```

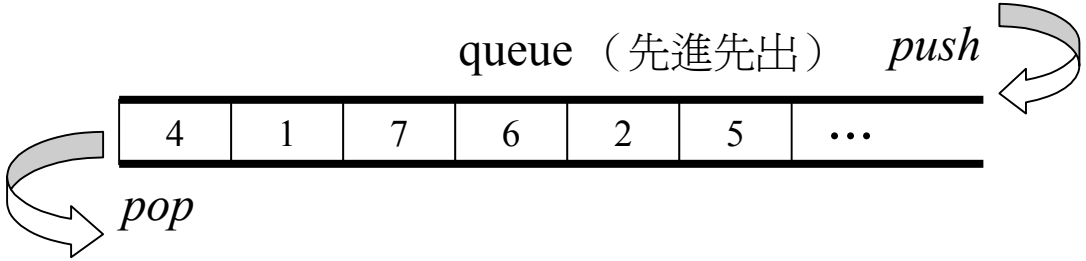
deque 是分段連續空間；
提供巡訪操作及 iterators



stack and queue

stack 或 queue均不提供巡訪操作，也不提供 iterator。
stack 和 queue 均可以 list 和 deque 做爲底層結構。





stack 關鍵實作碼

也可以是 list

```
template <class T, class Sequence = deque<T> >
class stack {
  friend bool operator== <> (const stack&, const stack&);
  friend bool operator< <> (const stack&, const stack&);
public:
  typedef typename Sequence::value_type value_type;
  typedef typename Sequence::size_type size_type;
  typedef typename Sequence::reference reference;
  typedef typename Sequence::const_reference const_reference;
protected:
  Sequence c;       // 底層容器
public:
  // 以下完全利用 Sequence c 的操作，完成 stack 的操作。
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference top() { return c.back(); }
  const_reference top() const { return c.back(); }
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_back(); }
};
```

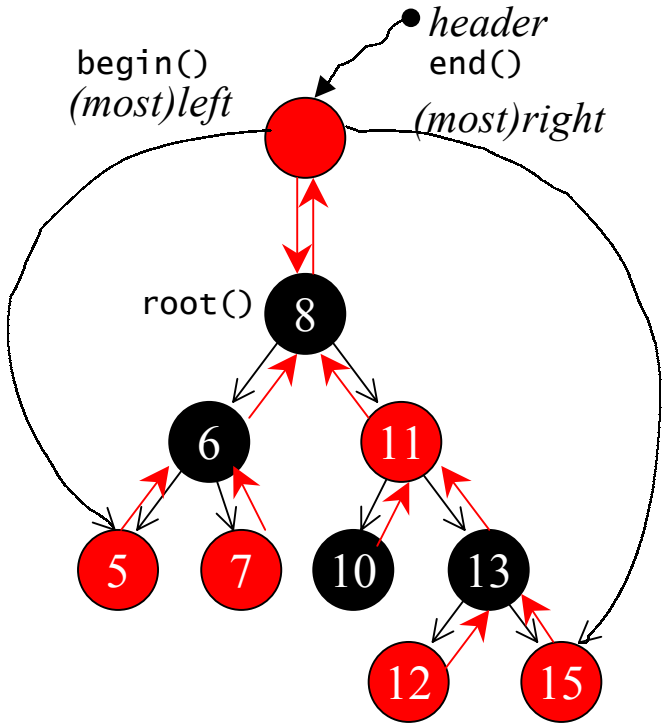
queue 關鍵實作碼

也可以是 list

```
template <class T, class Sequence = deque<T> >
class queue {
  friend bool operator== <> (const queue& x, const queue& y);
  friend bool operator< <> (const queue& x, const queue& y);
public:
  typedef typename Sequence::value_type value_type;
  typedef typename Sequence::size_type size_type;
  typedef typename Sequence::reference reference;
  typedef typename Sequence::const_reference const_reference;
protected:
  Sequence c;       // 底層容器
public:
  // 以下完全利用 Sequence c 的操作，完成 queue 的操作。
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference front() { return c.front(); }
  const_reference front() const { return c.front(); }
  reference back() { return c.back(); }
  const_reference back() const { return c.back(); }
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_front(); }
};
```

rb_tree

Red-Black tree（紅黑樹）是平衡二元搜尋樹（balanced binary search tree）中常被使用的一種。
平衡二元搜尋樹的特徵：排列規則有利搜尋和安插存取，並保持適度平衡（無任何節點過深）。



rb_tree 提供巡訪操作及 iterators。
若按正常規則來走（`++ite`），便形成排序（sorted）狀態。

我們不應使用 rb_tree iterators 改變元素值（因爲元素有其嚴謹排列規則）。程式層級並未阻絕此事。**如此設計是正確的，可允許「元素值」和「鍵值」被分開看待**

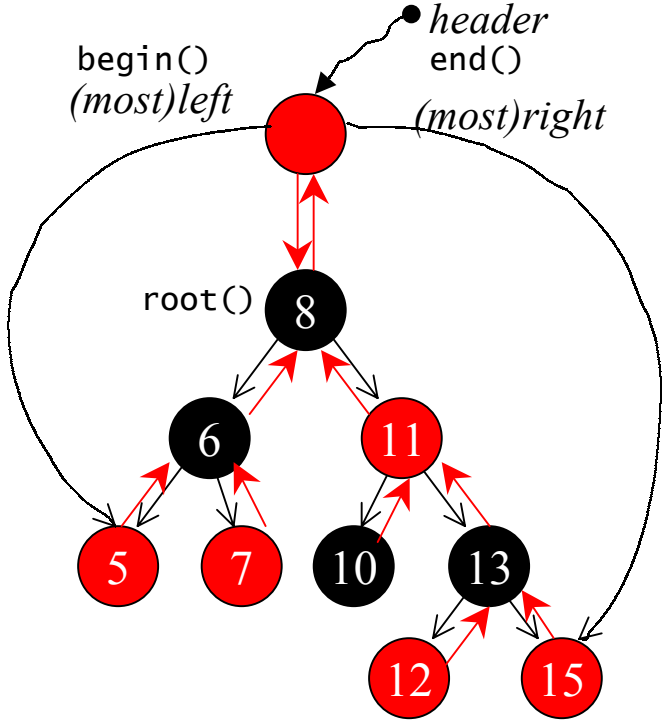
rb_tree提供兩種安插操作：insert_unique() 和 insert_equal()。
前者表示節點的 key 一定在整個 tree 中獨一無二，否則安插失敗；
後者表示節點的 key 可重複。

set

set 以 rb_tree 爲底層結構，因此有「元素自動排序」特性。
排序的依據是 key，而 **set 元素的 value 和 key 合一：value 就是 key**。

set 元素的 key 必須獨一無二，因此使用 rb_tree 的 insert_unique()。

set 提供巡訪操作及 iterators。若按正常規則來走（`++ite`），便形成排序（sorted）狀態。我們無法使用 set iterators 改變 key（因爲 keys 有其嚴謹排列規則）

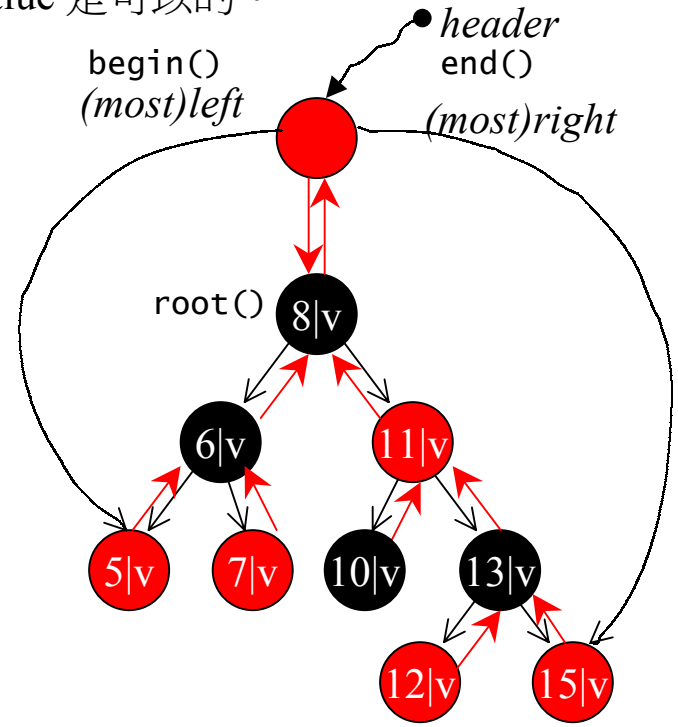


map

map 以 rb_tree 爲底層結構，因此有「元素自動排序」特性。排序的依據是 key。

map的每一個元素都是 pair。pair 的第一元素被視爲 key，第二元素被視爲 value。
map 元素的 key 必須獨一無二，因此使用 rb_tree 的 insert_unique()。

map 提供巡訪操作及 iterators。若按正常規則來走（`++ite`），便形成排序（sorted）狀態。我們不應使用 map iterators 改變 key（因爲 keys 有其嚴謹排列規則），但使用 map iterators 改變 value 是可以的。



multiset, multimap

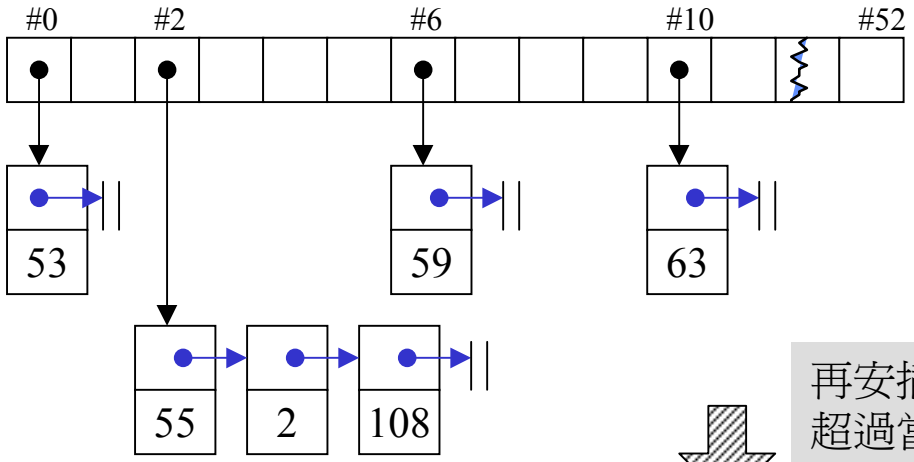
multiset 的特性和用法和 set 完全相同，唯一差別在於它允許 key 重複，因此其安插函式採用底層的 rb_tree::insert_equal()，而非 rb_tree::insert_unique()。

multimap 的特性和用法和 map 完全相同，唯一差別在於它允許 key 重複，因此其安插函式採用底層的 rb_tree::insert_equal()，而非 rb_tree::insert_unique()。

hashtable

TASS, p266, fig5-26

```
insert_unique 次序:59,63,108,2,53,55    num_elements:6
```



buckets vector（大小故意爲質數 53,97,193,389…）

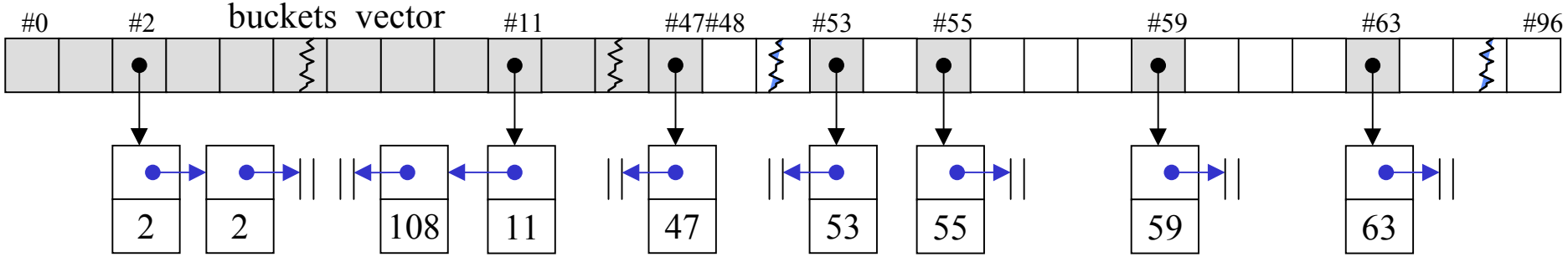
Separate Chaining

雖然 list 是線性搜尋時間，如果 list 夠小，搜尋速度仍然很快

再安插 48 個元素，使總量達到 54 個，超過當時的 buckets vector 大小，於是 **rehashing**

TASS, p267, fig5-27

```
size:97  num_elements:54
```



我們不應使用 hashtable iterators 改變元素值（因爲元素有其嚴謹排列規則）。
程式層級並未阻絕此事。**如此設計是正確的，可允許「元素值」和「鍵值」被分開看待**

HashFcn, ExtractKey, EqualKey for c-style string

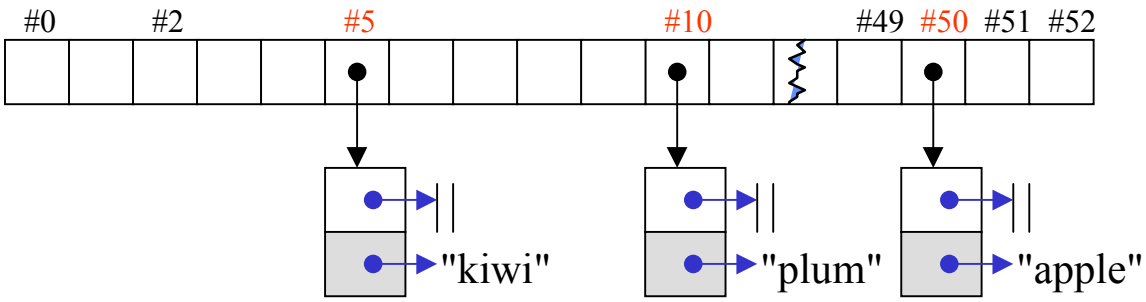
實例 tass\prog\5hashtable-test.cpp

```
hashtable<const char*,
          const char*,
          hash<const char*>,
          identity<const char*>,
          eqstr,
          alloc>
ht(50,hash<const char*>(),eqstr());

ht.insert_unique("kiwi");
ht.insert_unique("plum");
ht.insert_unique("apple");
```

hash code of "kiwi" = 5 * (5 * (5 * 'k' + 'i') + 'w') + 'i'
= 5 * (5 * (5 * 107 + 105) + 119) + 105
= 16700 % 53 = 5

hash code of "plum" = 5 * (5 * (5 * 'p' + 'l') + 'u') + 'm'
= 5 * (5 * (5 * 112 + 108) + 117) + 109
= 17394 % 53 = 10

hash code of "apple" = 5 * (5 * (5 * (5 * 'a' + 'p') + 'p') + 'l')+'e'
= 5 * (5 * (5 * (5 * 97 + 112) + 112) + 108)+101
= 78066 % 53 = 50



TASS, p268
<stl_hash_fun.h>

```
inline size_t
__stl_hash_string(const char* s)
{
  unsigned long h = 0;
  for ( ; *s; ++s)
    h = 5*h + *s;
  return size_t(h);
}
```

hash_set

hash_set 以 hashtable 爲底層結構，無「元素自動排序」特性。

hash_set 元素的 key（亦即 value）必須獨一無二，因此使用 hashtable 的 insert_unique()。

hash_set 提供巡訪操作及 iterators。我們無法使用 hash_set iterators 改變 key（因爲 keys 有其特定排列規則）

hash_map

hash_map 以 hashtable 爲底層結構，無「元素自動排序」特性。

hash_map的每一個元素都是 pair。pair 的第一元素被視爲 key，第二元素被視爲 value。 hash_map 元素的 key 必須獨一無二，因此使用 hashtable 的 insert_unique()。

hash_map 提供巡訪操作及 iterators。我們不應使用 hash_map iterators 改變 key（因爲 keys 有其特定排列規則），但可使用 hash_map iterators 改變 value。

hash_multiset, hash_multimap

hash_multiset 的特性和用法和 hash_set 完全相同，唯一差別在於它允許 key 重複，因此其安插函式採用底層的 hashtable::insert_equal()，而非 hashtable::insert_unique()。

hash_multimap 的特性和用法和 hash_map 完全相同，唯一差別在於它允許 key 重複，因此其安插函式採用底層的 hashtable::insert_equal()，而非 hashtable::insert_unique()。

Algorithms 泛化過程

```
int* find(int* arrayHead, int arraySize, int value)
{
   int i;
   for (i=0; i<arraySize; i++)
      if (arrayHead[i] == value)
          break;
   return &(arrayHead[i]);
}
```

版本一

搜尋 array 中的某個元素：

以上實作太過曝露容器細節（例如容器大小），改以「頭尾區間」的方式來指示操作範圍：

```
int* find(int* begin, int* end, int value)
{
   while (begin != end && *begin != value)
      ++begin;
   return begin;
}
```

版本二

將指標泛化，將元素型別泛化：

```
template <typename Iterator, typename T>
Iterator find(Iterator begin, Iterator end, const T& value)
{
   while (begin != end && *begin != value)
      ++begin;
   return begin;
}
```

這就是 STL source

版本三

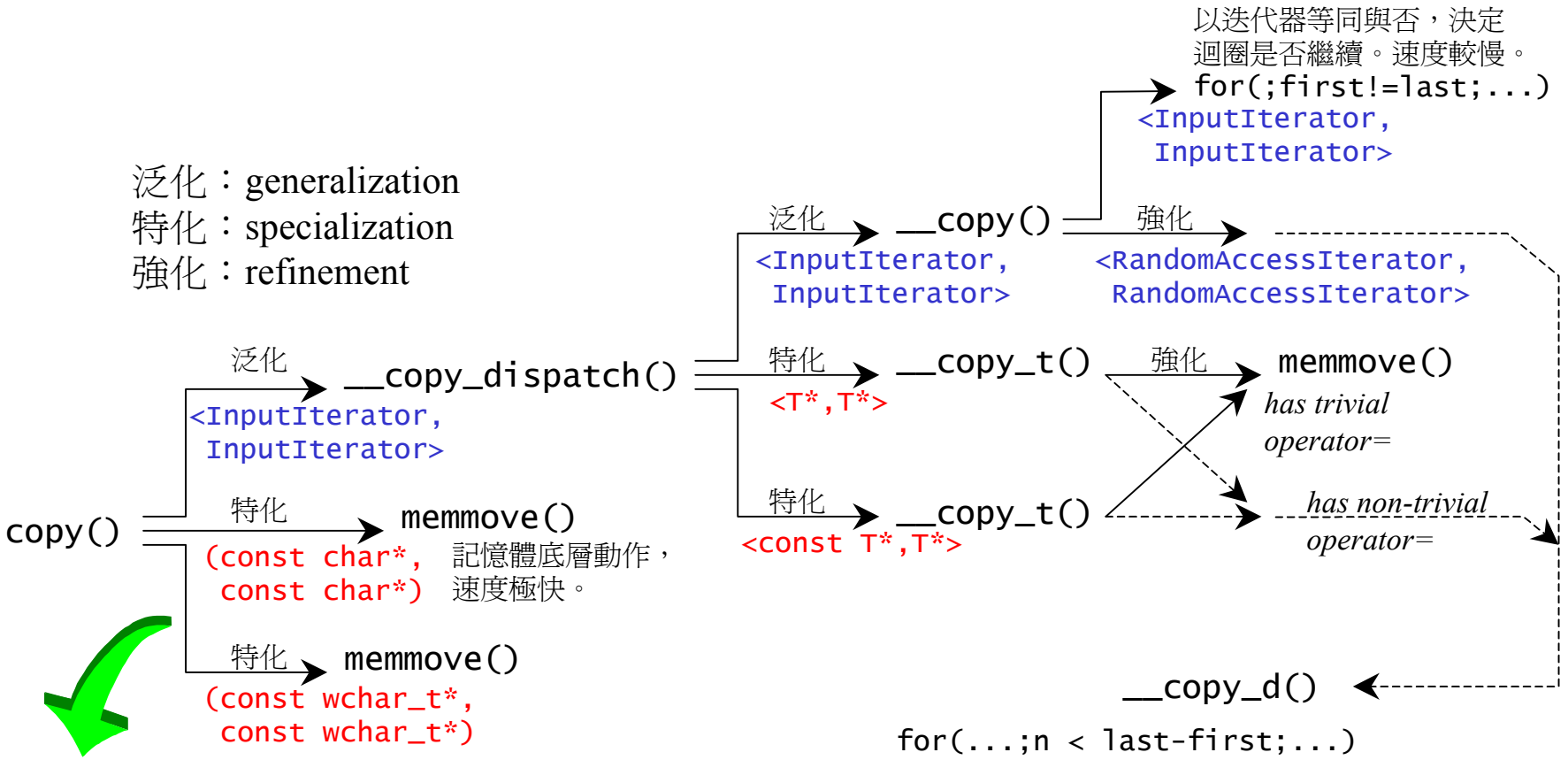
STL Algorithms 實例 : for_each, find, find_if

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f) {
  for ( ; first != last; ++first)
    f(*first);
  return f;
}

template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value) {
  while (first != last && *first != value)
      ++first;
  return first;
}

template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred) {
  while (first != last && !pred(*first))
      ++first;
  return first;
}
```

何時需要這個被傳回的 functor，例見《Effective STL》條款37（最後）

STL Algorithms（演算法）: copy

無所不用其極地強化速度效能

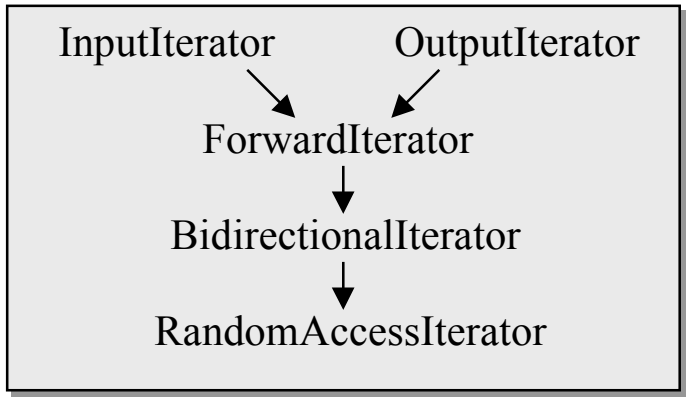


copy() 是 function template，無所謂特化。這兒用的是多載化技巧。爲何要先對上述兩種形式做特殊處理，而不讓它流到後頭的判斷/選擇流程中？我想是因爲這兩種「字串」形式，其 copy 必可使用 memmove()，不必再進入後面的流程。爲的總是效率! 效率!! 效率!!!

STL Iterators（迭代器）: overview

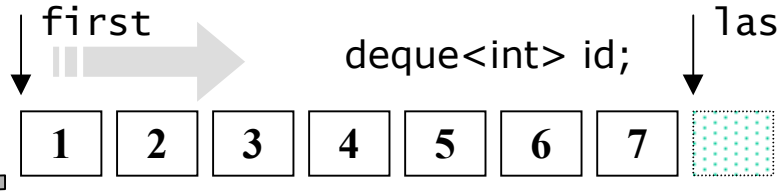
Iterator 是一種「行爲像指標」的東西（something like pointer ），或可說是一種 smart pointer。因此它必須提供指標慣有的 `operator->` 和 `operator*`。iterator 用於 STL 容器身上，主要做爲指位器，也做爲巡訪器，因此它還必須提供前進或後退或跳躍的 `operator++,operator--,operator+(n)` 等操作。

根據 GoF 定義，iterator 是：
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
提供一種巡訪「聚合體內各元素」的通用方法，並且不必曝露聚合物的內部細節
因此iterator必須由容器（聚合體）設計，否則無人能知巡訪的技術細節（前進、後退、提領…）。根據前進、後退、I/O 能力，可區分爲五類：



設計合適的 associated types 是 iterators 的責任。設計合適的 iterators 則是容器的責任。只有容器設計者才知道該設計出怎樣的 iterator 來遍歷 (traversals) 自己。Algorithm 完全獨立於這些之外。

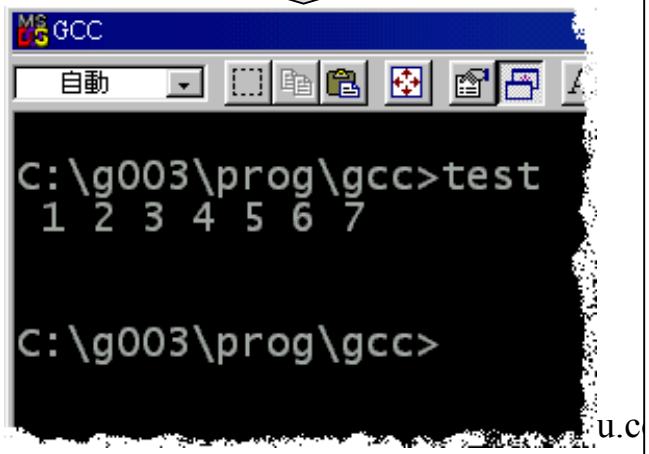
STL Iterators（迭代器）: ostream_iterators




```
ostream_iterator<int> outite(cout, " ");
copy(id.begin(), id.end(), outite);
```

```
template <class RandomAccessIterator, class OutputIterator, class Distance>
inline OutputIterator
__copy_d(RandomAccessIterator first, RandomAccessIterator last,
                 OutputIterator result, Distance*)
{
  for (Distance n = last - first; n > 0; --n, ++result, ++first)
    *result = *first;
  return result;
}
```

result

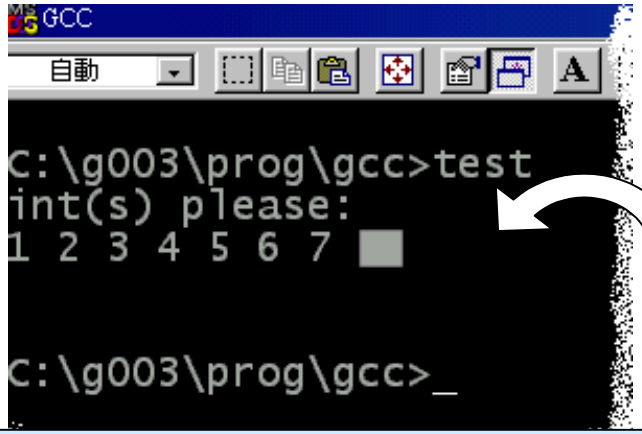


u.c

摘錄

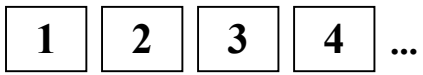
```
template <class T>
class ostream_iterator {
protected:
  ostream* stream;
public:
  ...
  // 對迭代器做賦值（assign）動作，就代表要輸出一筆資料
  ostream_iterator<T>& operator=(const T& value) {
    *stream << value;          // 關鍵：輸出數值
    return *this;
  }
  // 注意以下運算子對上方 __copy_d() 函式內的動作的影響
  ostream_iterator<T>& operator*() { return *this; }
  ostream_iterator<T>& operator++() { return *this; }
};
```

STL Iterators（迭代器）: istream_iterators




```
istream_iterator<int> inite(cin), eos;
copy(inite, eos, inserter(id, id.begin()));
```

result



deque<int> id;

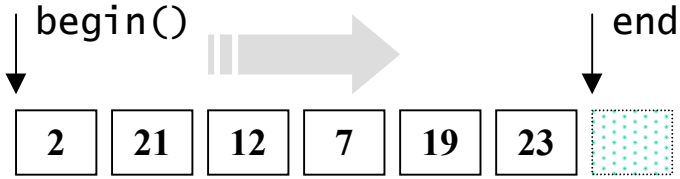
一開始就讀一個元素，合理嗎？還算合理！如此才能一開始就滿足 operator*。ref. TCSL,p280;TASS,p443

摘錄

```
template <class T, class Distance = ptrdiff_t>
class istream_iterator {
protected:
  istream* stream;
  T value;
  bool end_marker;
  void read() { ...
    if (end_marker) *stream >> value;   // 關鍵
  }
public:
  istream_iterator(istream& s) : stream(&s)
    { read();}
  reference operator*() const { return value; }
  istream_iterator<T, Distance>& operator++() {
    read();
    return *this;
  }
};
```

```
template <class InputIterator, class Outp
inline OutputIterator __copy(InputIterator first, InputIterator last,
                             OutputIterator result, input_iterator_tag)
{
  for ( ; first != last; ++result, ++first)
      *result = *first;
  return result;
}
```

STL Adapters（配接器）



bind2nd(...) 會產生一個
binder2nd<Operation>(...); 物件。
此將傳給 count_if() 成爲其**pred** 引數。

```
count_if(iv.begin(), iv.end(), bind2nd(less<int>(), 12),i);
```

```
template <class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last,
              Predicate pred, Size& n) {
  for ( ; first != last; ++first)  // 整個範圍走一篇
    if (pred(*first))  // 如果元素帶入pred 的運算結果爲 true
      ++n;             // 計數器累加1
}
```

註：這是舊版 count_it()

```
template <class Operation>
class binder2nd : public unary_function<...> {
protected:
  Operation op;       // 內部成員
  typename Operation::second_argument_type value;
public:
  binder2nd(const Operation& x,
            const typename Operation::second_argument_type& y)
    : op(x), value(y) {}
  typename Operation::result_type
  operator()(const typename Operation::first_argument_type& x) const {
    return op(x, value);        // 將 value 繫結（binding）爲第二引數
  }
};
```

控制權轉到我們手上，我們當然就可以爲所欲爲了。

修飾functor者（functor adapters），其結果必須像個functor，所以這個class必須準備 op( )。

Metaprogramming, numeric computations

```
template <unsigned long N>
struct binary
{
  static unsigned const value
     = binary<N/10>::value << 1 | N%10;
};

template<>
struct binary<0>   // Compile time Version
{
  static unsigned const value = 0;
};

unsigned const one   = binary<1>::value;
unsigned const two   = binary<10>::value;
unsigned const three = binary<11>::value;
unsigned const four  = binary<100>::value;
unsigned const five  = binary<101>::value;
```

```
unsigned binaryRV1(unsigned long N)  // Runtime Ver1, recursion
{
  return N==0 ? 0 : N%10 + 2*binaryRV1(N/10);
}

unsigned binaryRV2(unsigned long N)   // Runtime Ver2, iteration
{
  unsigned result = 0;
  for (unsigned bit = 0x1; N; N/=10,bit<<=1)
  {
    if (N%10)
        result += bit;
  }
  return result;
}
```

```
int main()
{
  cout << one   << endl;
  cout << two   << endl;
  cout << three << endl;
  cout << four  << endl;
  cout << five  << endl;
  cout << endl;

  cout << binaryRV1(1000) << endl;
  cout << binaryRV1(1001) << endl;
  cout << binaryRV2(1000) << endl;
  cout << binaryRV2(1001) << endl;
}
```

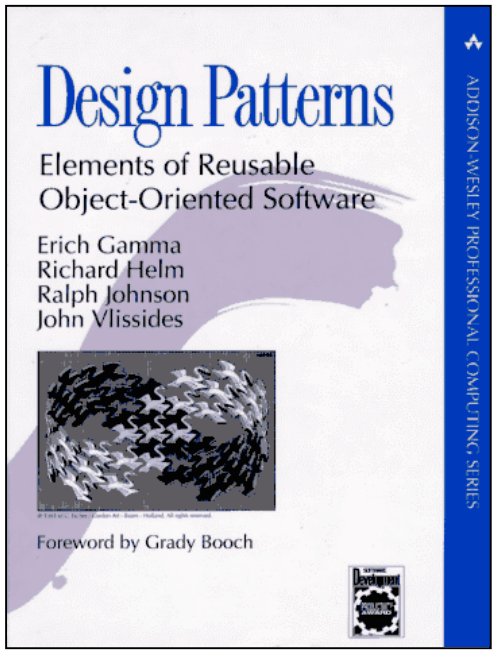
```
binary<101>::value
==>   binary<10>::value << 1 | 1
==>  (binary<1>::value << 1 | 0 ) << 1 | 1
==> ((binary<0>::value << 1 | 1)  << 1 | 0) << 1 | 1
==> ((0 << 1 | 1)  << 1 | 0) << 1 | 1
==> 5
```

Bibliography

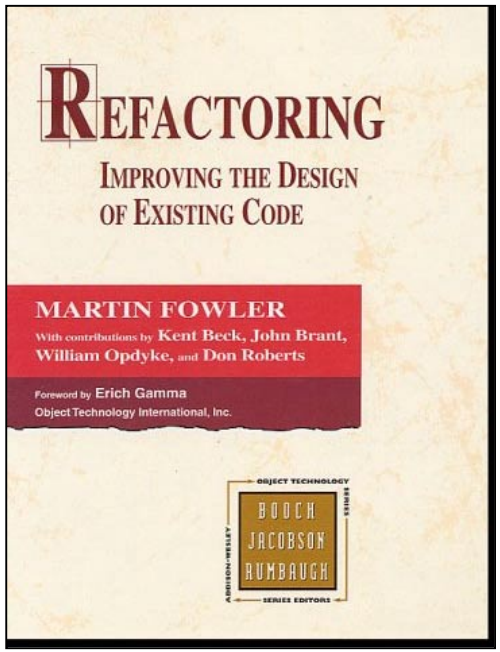
pattern :
*. 花樣,圖案,形態,樣式
*. (服裝裁剪的)紙樣,(澆鑄用的)模具
*. 模範,榜樣,典型

Thinking in Patterns
Problem-Solving Techniques using Java
Bruce Eckel
President, MindView, Inc.
（www.BruceEckel.com）

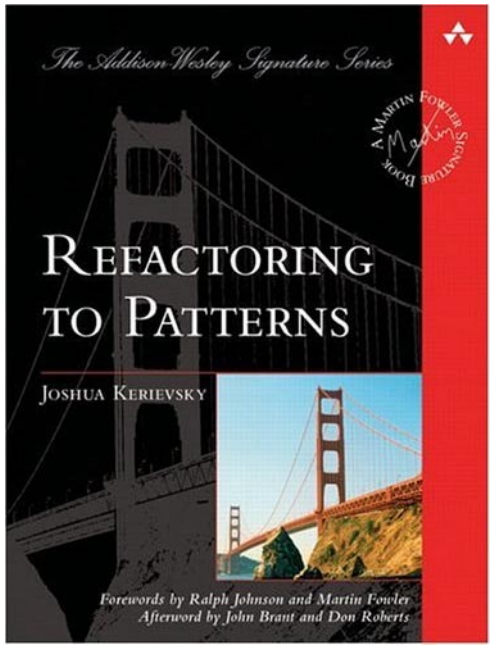
GOF : Gang of Four



（葉秉哲 譯, 2001）



（侯捷 / 熊節 合譯, 2003）



（侯捷 / 陳裕城 譯, 2006）

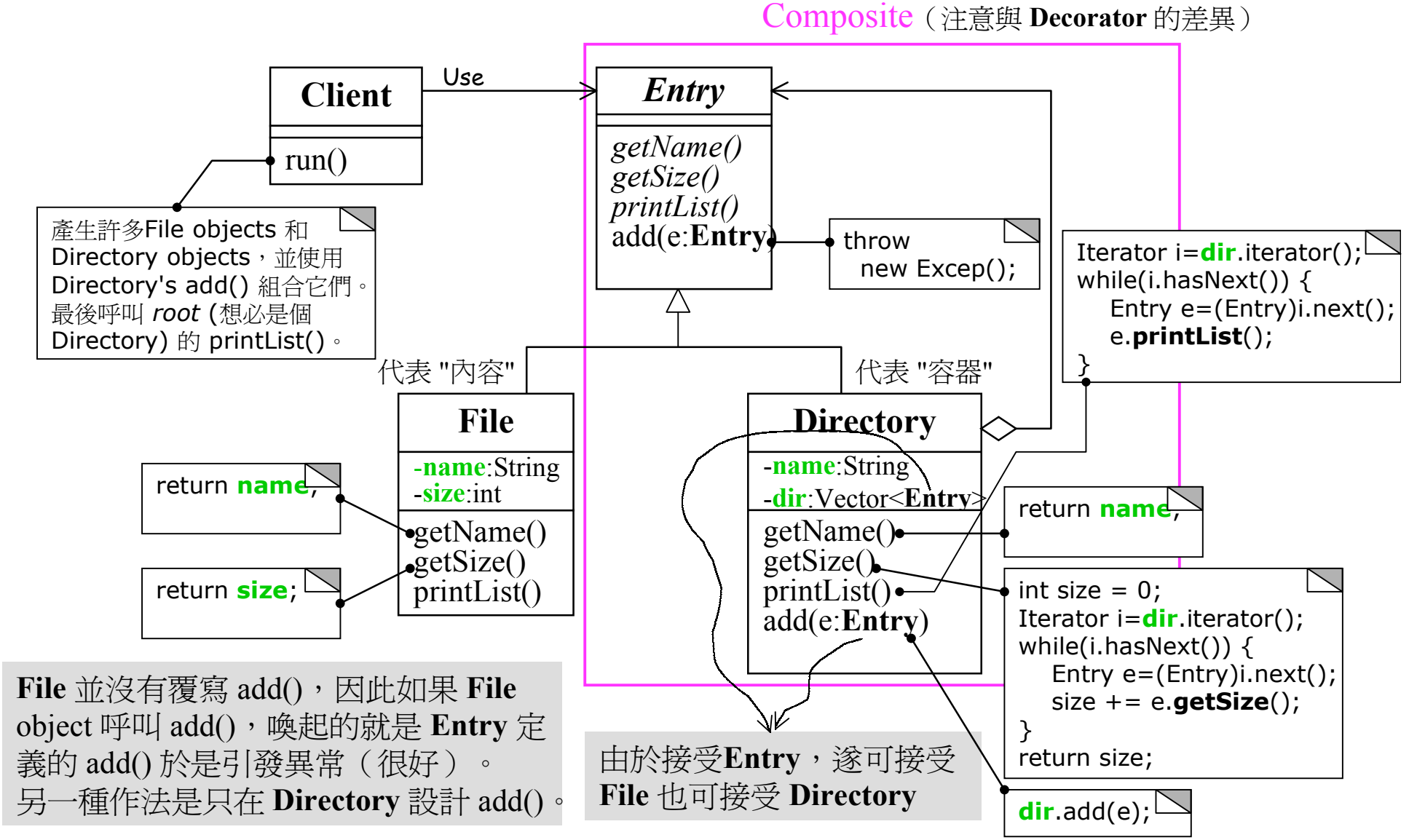
7. Composite

7. Composite (163)
Compose objects into tree structures to represent part-whole hierarchies.
Composite lets clients treat individual objects and compositions of objects uniformly.

將物件(s) 組成/構成 爲樹狀結構，用以表示 "局部-全部" 階層體系。
Composite 可以讓 clients 以一致的方式對待「個別物件」和「合成物件」。

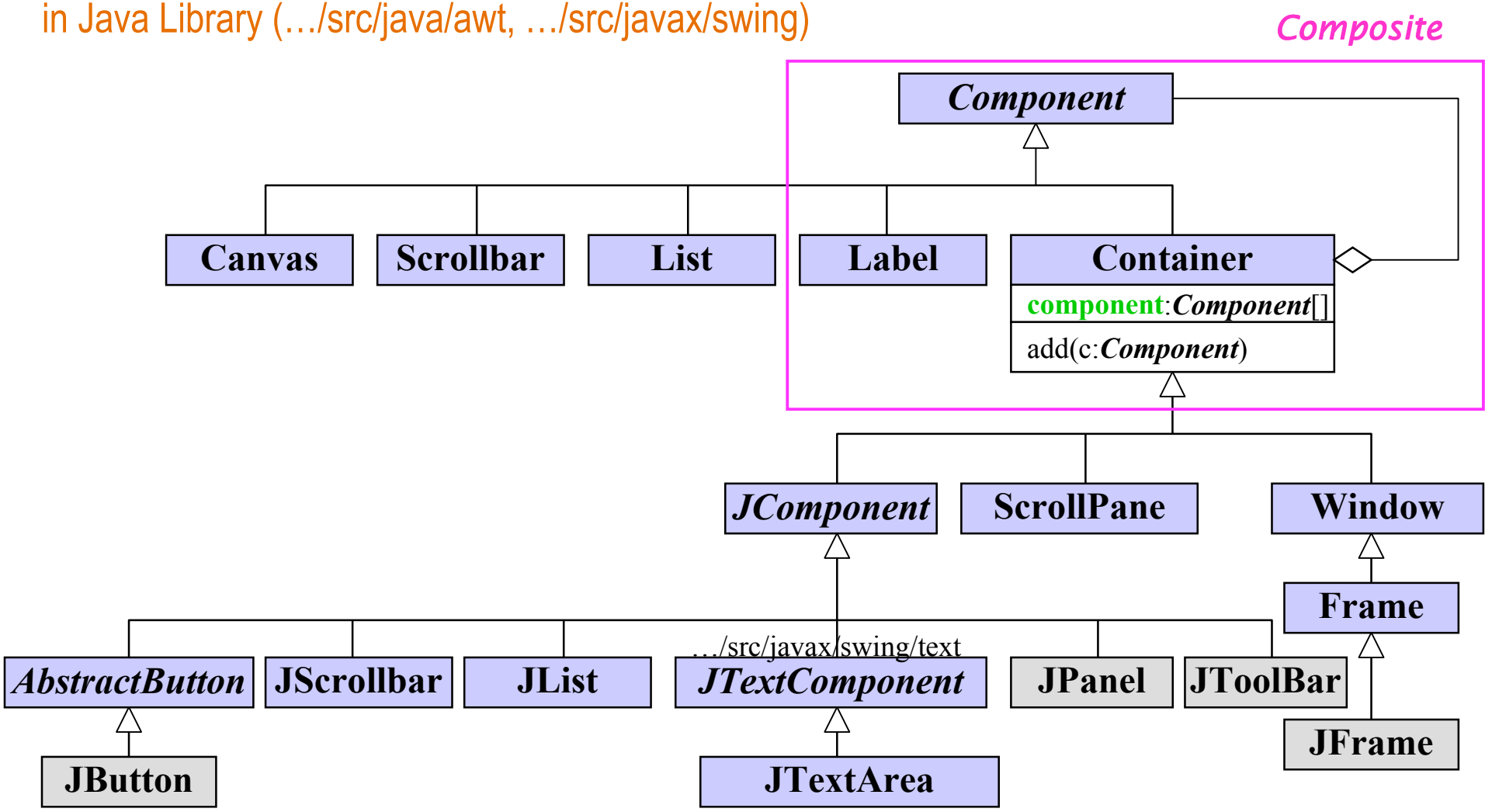
- **Composite**：對內容和容器一視同仁。或說對單複數一視同仁：把單數集中在一起成爲複數時，仍可視之爲一個單數。
- **Decorator**：對內容和裝飾後的內容一視同仁。

7. Composite in DP-in-Java



7. Composite in Java Lib.

in Java Library (…/src/java/awt, …/src/javax/swing)



8. Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

將額外的「權與責」以動態方式附著於物件身上，使不必進行 subclassing 就能擴展功能。

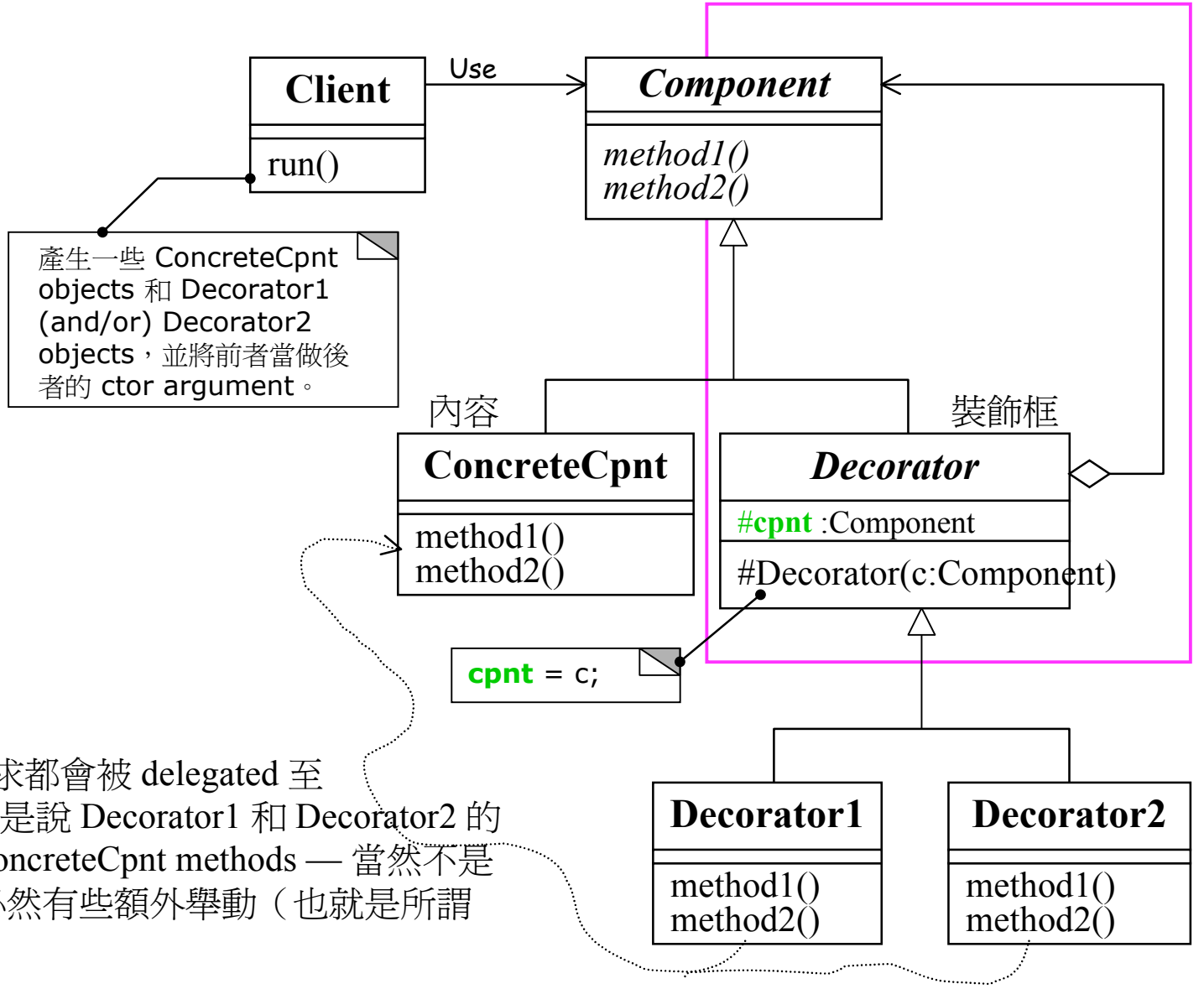
Composite：對內容和容器一視同仁。Composite class 內含一個 "collection of Component" field。

Decorator：對內容和裝飾框（含被裝飾內容）一視同仁。Decorator class 內含一個 Component object field。**Decorator** 和 **Composite** 兩者在「處理遞迴架構」的部分很像，但目的不盡相同。

Decorator 使用多重外框的目的是爲了添加功能。

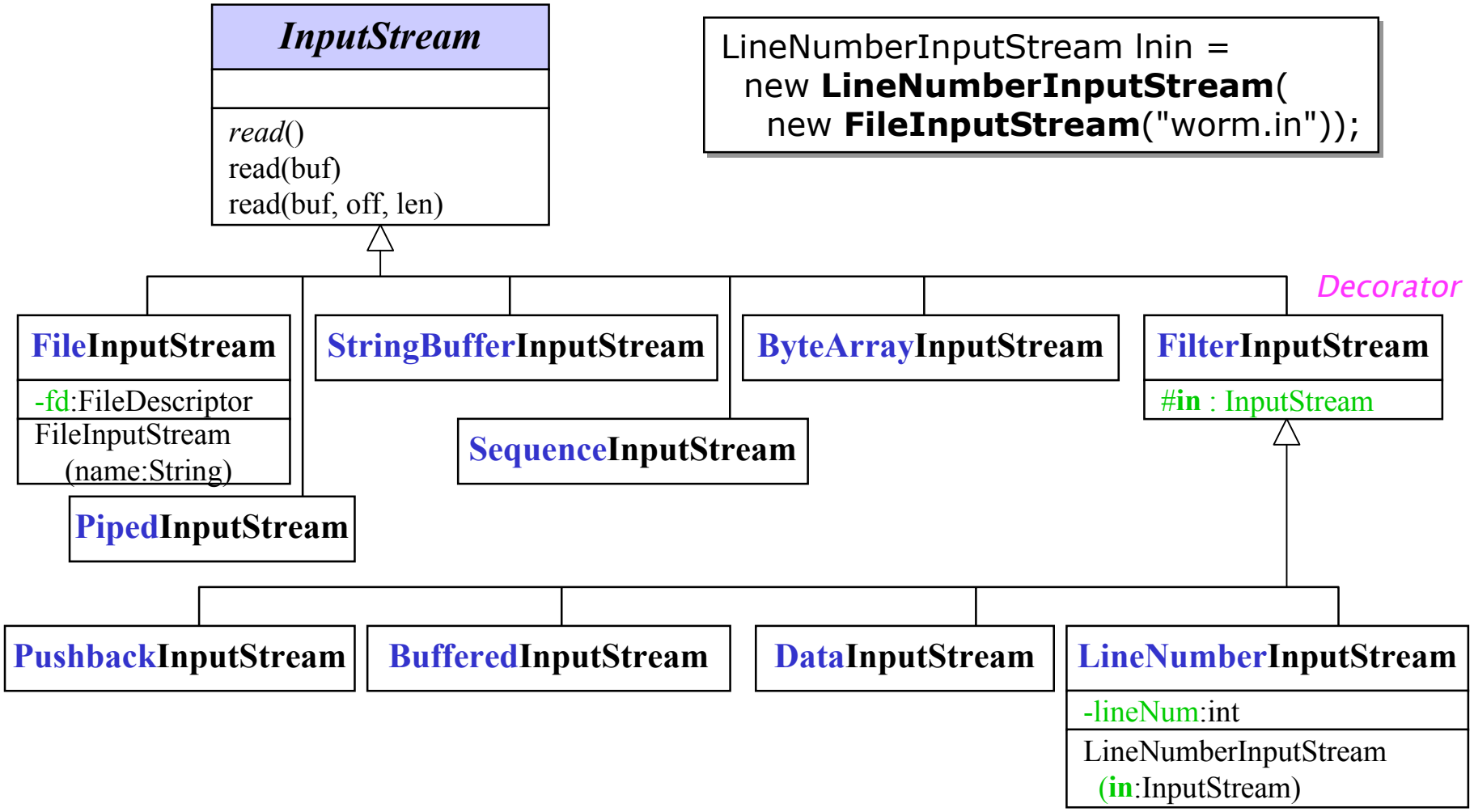
8. Decorator in DP-in-Java

Decorator（注意與 **Composite** 的差異）



所有對 Decorator 的要求都會被 delegated 至 ConcreteCpnt 去。也就是說 Decorator1 和 Decorator2 的 methods 內都呼叫了 ConcreteCpnt methods — 當然不是一成不變地轉呼叫，必然有些額外舉動（也就是所謂的「添加功能」）。

8. Decorator in java.io



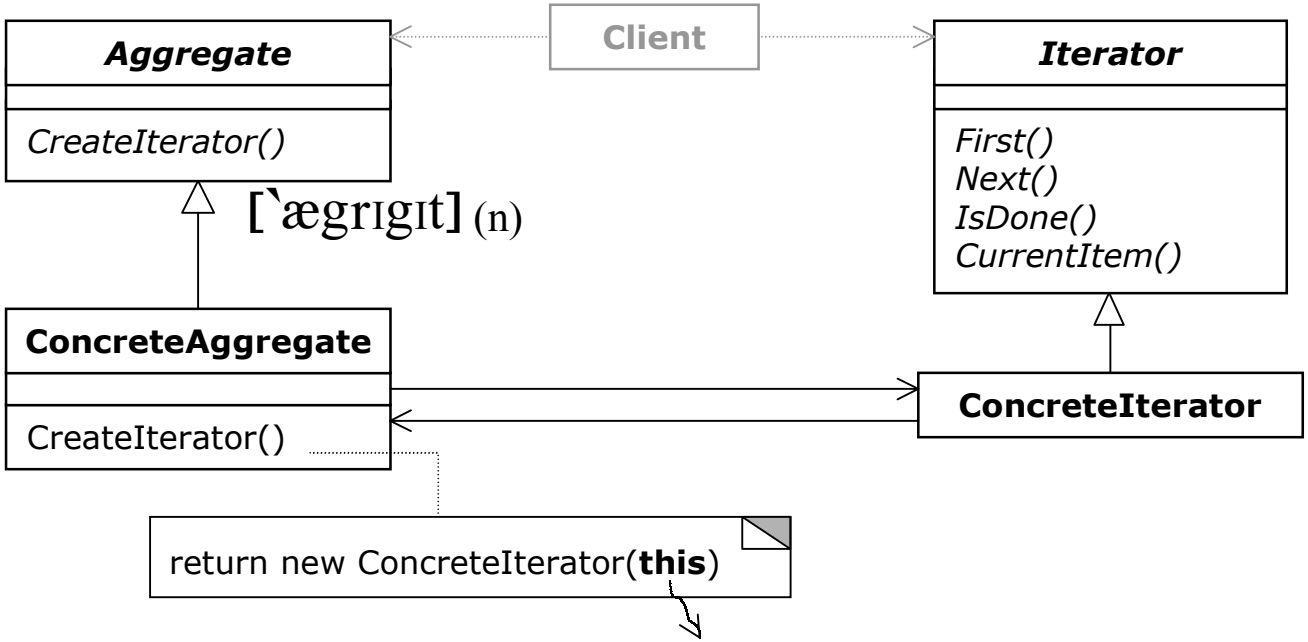
13. Iterator

['ægrɪˌget] (v)

['ægrɪgɪt] (a)

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

提供一種連續（相繼）巡訪「聚合物內各元素」的通用介面，並且不必曝露聚合物的底層表述（內部細節）。



Java Library 中由於 iterator classes 都被設計爲 container class 的 inner class，所以無需傳遞 'this'.
STL 中由於 iterator 直接指涉 container 內部結構所以也無需傳遞 'this'.

13. Iterator in Java Collections

Generics Java 只允許在 collection 內放置 class object，不允許放置 primitive type data. 所以不得寫爲：LinkedList<**int**> il=new...;

用法：

```
LinkedList<Integer> il = new LinkedList<Integer>();
il.add(new Integer(0));
il.add(new Integer(1));
il.add(new Integer(5));
il.add(new Integer(2));

Iterator ite = il.iterator();
while(ite.hasNext()) {
   System.out.println(ite.next());
}
```

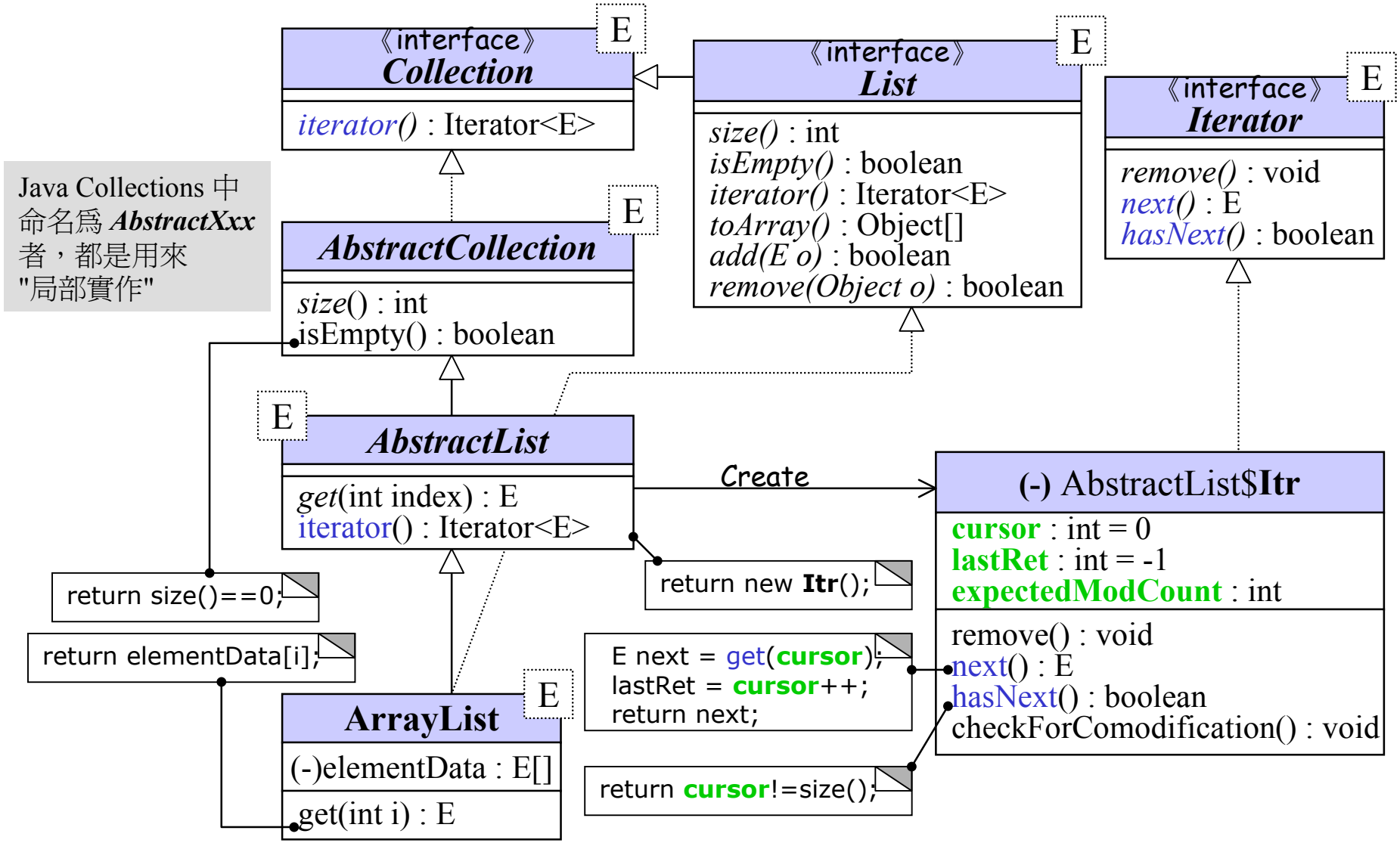
自從JDK5.0提供 box/unbox 之後，可寫爲：

```
il.add(0);
il.add(1);
il.add(5);
il.add(2);
```

每一種 Java collection 都具備 iterator()

每個 iterator 都具備這樣的 methods.

13. Iterator in Java Lib.



16. Observer

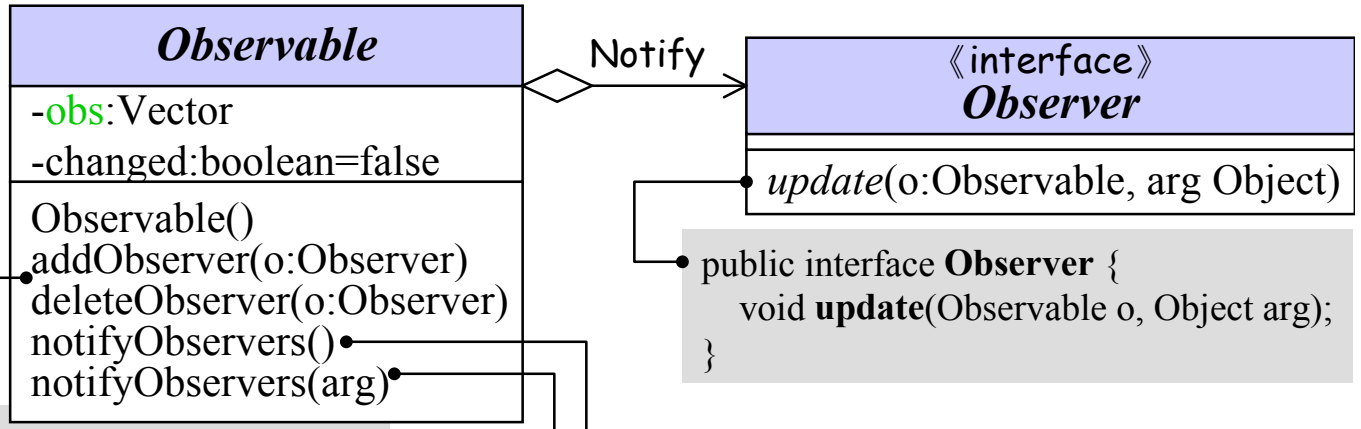
16. Observer (293)
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

在 objects 之間定義 "一對多" 依存性，使得當 object 改變狀態時，它所依存的所有 objects 都會獲得通知並自動更新。

Observer 是被動地被通知，而不是主動觀察，所以這個 pattern 的另一個名稱 **publish-subscribe** 比較更合適些。

16. Observer in Java Library

缺點：app. 的 ConcreteSubject 必須繼承 Java's **Observable**，而 由於Java classes 都是單一繼承，因此ConcreteSubject 將無法再繼承其他 class。再者由於 Observable 不是個 interface，所以我們無法建立自己的 implement.




```
public interface Observer {
    void update(Observable o, Object arg);
}
```

```
public class Observable {
    private boolean changed = false;
    private Vector obs;

    public Observable() {obs = new Vector(); }
    public synchronized void addObserver(Observer o) {
        if (!obs.contains(o))  obs.addElement(o);
    }
    public synchronized void deleteObserver(Observer o) {
        obs.removeElement(o);
    }
    ...
```

```
1 public void notifyObservers() {
            notifyObservers(null);
    }
2 public void notifyObservers(Object arg) {
        Object[] arrLocal;
        synchronized (this) {
            if (!changed)  return;
            arrLocal = obs.toArray();
            clearChanged();
        }
        for (int i = arrLocal.length-1; i>=0; i--)
            ((Observer)arrLocal[i]).update(this, arg);
    }
    //  …另有 setChanged(), clearChanged(),
    //      hasChanged(), countObserver()
```

(1) 法用 pull 法
(2) 法用 push 法

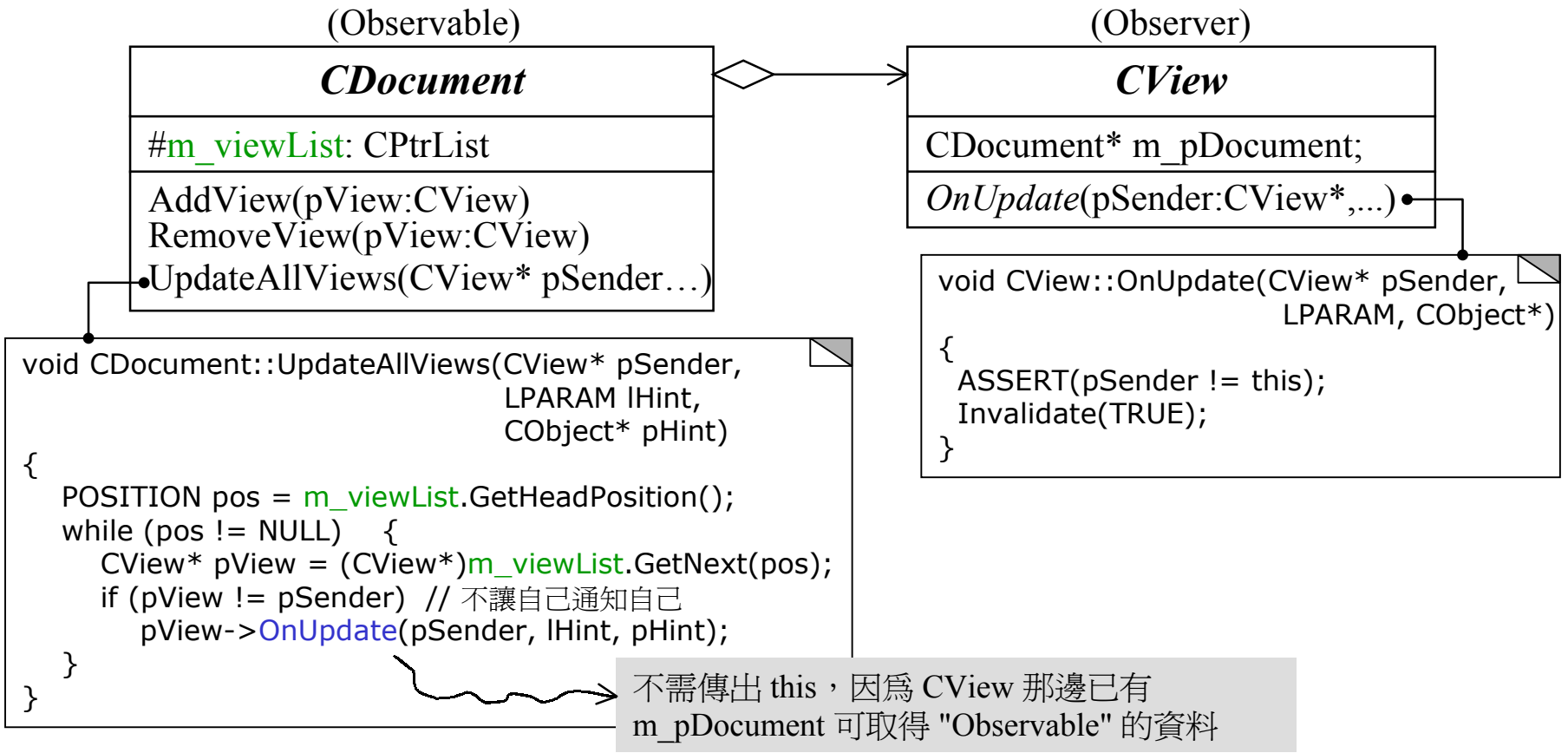
又是直接存取field，
又是透過accessor存取，
不一致。何不改爲：
if(!hasChanged()) return;

16. Observer

MVC中的 Model 和 View（例如 MFC 的 document/view）與「Subject 和 Observer」的關係相呼應。例如應用端這麼寫：

```
void CMyView::OnLButtonUp(...) {
  ...
  pDoc->UpdateAllViews(this);
}
```

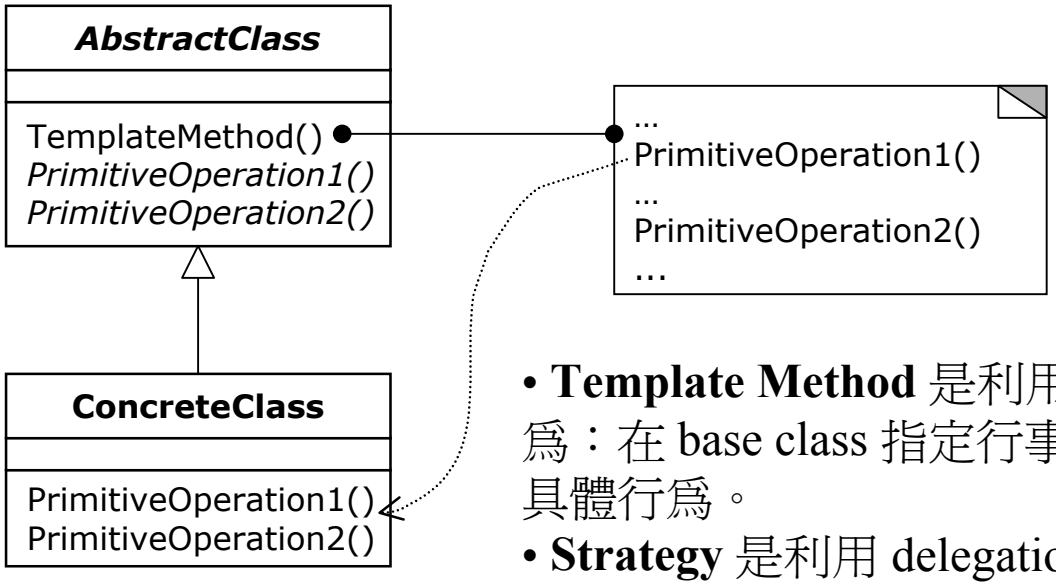
滑鼠左鍵被放開時決定更新所有 views



22. Template Method in GOF

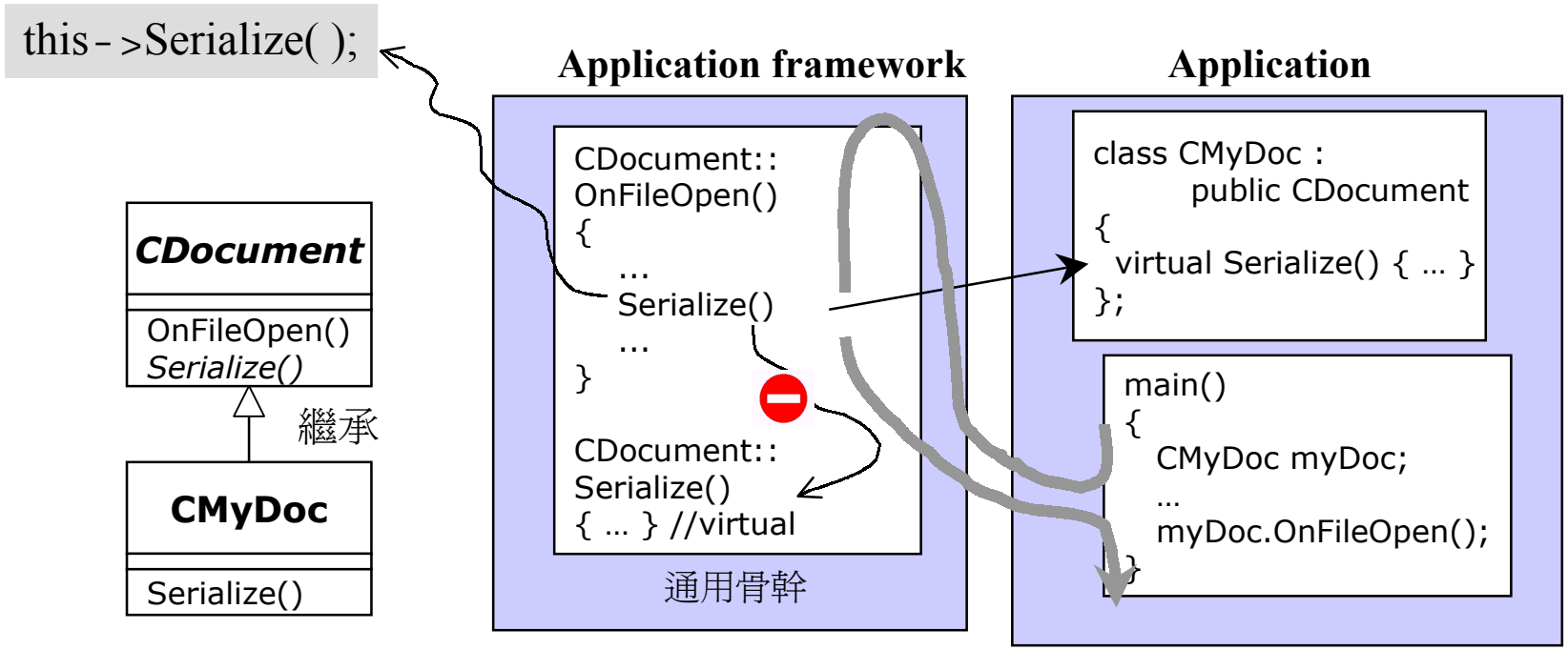
Define the skeleton of an algorithm in an operation, de'ferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

定義演算法骨幹，延緩其中某些步驟，使它們在subclasses 中才獲得真正定義。**Template Method** 使 subclasses 得以重新定義演算法內的某些動作，而不需改變演算法的總體結構。



- **Template Method** 是利用 inheritance 來改變程式行爲：在 base class 指定行事大綱，在 sub class 指定具體行爲。
- **Strategy** 是利用 delegation 來改變程式行爲：改變的不是演算法局部，而是切換整個演算法。

22. Template Method in MFC



《深入淺出MFC》p.84：*CDocument*::*OnFileOpen* 中所呼叫的 *Serialize* 是哪一個 class 的成員函式呢？如果它是一般（non-virtual）函式，毫無問題應該是 *CDocument*::*Serialize*。但因這是個虛擬函式，情況便有不同。既然 derived class 已經改寫了虛擬函式 *Serialize*，那麼理當喚起 derived class 之 *Serialize*函式。這種行爲模式非常頻繁地出現在 application framework 身上。

22. Template Method simulation

```
01 #include <iostream>
02 using namespace std;
03
04 // 這裡扮演application framework 的角色
05 class CDocument
06 {
07 public:
08     void OnFileOpen()  // 此即所謂Template Method
09     {
10       // 這是一個演算法，骨幹已完成。每個cout輸出代表一個實際應有動作
11       cout << "dialog..." << endl;
12       cout << "check file status..." << endl;
13       cout << "open file..." << endl;
14       Serialize();      // 喚起哪一個函式？
15       cout << "close file..." << endl;
16       cout << "update all views..." << endl;
17     }
18
19     virtual void Serialize() { };
20 };
... 接下頁 ...
```

22. Template Method simulation

```
... 接上頁 ...
21 // 以下扮演 application 的角色
22 class CMyDoc : public CDocument
23 {
24 public:
25     virtual void Serialize()
26     {
27       // 只有應用程式員自己才知道如何讀取自己的文件檔
28       cout << "CMyDoc::Serialize()" << endl;
29     }
30 };
31 int main()
32 {
33   CMyDoc myDoc;  // 假設主視窗功能表的[File/Open]被按下後執行至此。
34   myDoc.OnFileOpen();
35 }
```

執行結果：

```
dialog...
check file status...
open file...
CMyDoc::Serialize()
close file...
update all views...
```

Object Factory/Factory Method 是一種典型的 **Template Method**，只不過是特別（專門）應用於 object 創建工作上而已（其 return type 有比較嚴格的限制，需爲各 objects 之 base class）。

MFC Classes Hierarchy