



**博览网，高端IT在线学习平台**

[www.boolan.com](http://www.boolan.com)

# 更多精彩课程推荐



## C++设计模式

课程从设计之道和设计之术两方面，通过大量的代码实践与演练，深入剖析经典GOF 23种设计模式。



## C++面向对象开发

助你正确理解面向对象(OO)的精神和实现手法，课程涵盖对象模型、关键机制、编程风格、动态分配。



## C++内存管理机制

上穷碧落下黄泉，带你由语言基本构件至高级分配器的设计与实作，并探及最低阶之 malloc 的内部实现。



## C++新标准C++11-C++14

使你在极短时间内深刻理解C++2.0的诸多新特性，涵盖语言和标准库两层面，只谈新东西。

# C++ 標準庫

## 體系結構與內核分析

(C++ Standard Library — architecture & sources)

第一講



侯捷

— 侯捷 —

1

使用一個東西，  
卻不明白它的道理，  
不高明！



## ■■■■ 你應具備的基礎

- C++ 基本語法  
(包括如何正確使用模板, templates)

## ■■■■ 我們的目標

- level 0: 淺嘗 C++ 標準庫
- level 1: 深入認識 C++ 標準庫 (胸中自有丘壑)
- level 2: 良好使用 C++ 標準庫
- level 3: 擴充 C++ 標準庫

## ■■■■ C++ Standard Library vs. Standard Template Library

### ➤ C++ Standard Library

C++ 標準庫

### ➤ Standard Template Library

STL, 標準模板庫

標準庫以 header files 形式呈現

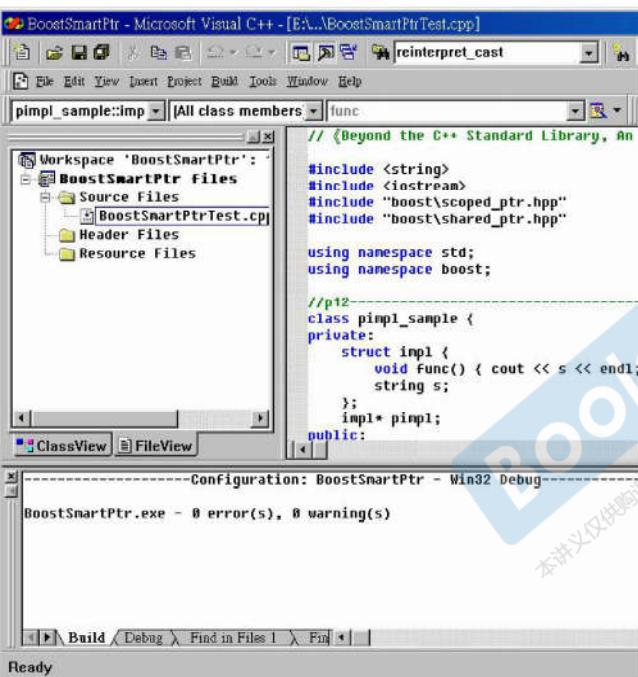
- C++ 標準庫的 header files 不帶副檔名 (.h)，例如 #include <vector>
- 新式 C header files 不帶副檔名 .h，例如 #include <cstdio>
- 舊式 C header files (帶有副檔名 .h) 仍然可用，例如 #include <stdio.h>
- 新式 headers 內的組件封裝於 namespace "std"
  - ➔ using namespace std; or
  - ➔ using std::cout; (for example)
- 舊式 headers 內的組件不封裝於 namespace "std"

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

—侯

## ■■■■ C++ 標準庫, 版本

(Visual C++ 6.0 畫面)



(Dev-C++ 5.11 畫面; with GNU 4.9.2)

```
// 計算性上試用各式各樣的 allocators
cout << sizeof(_gnu_cxx::malloc_allocator<int>) << endl;
cout << sizeof(_gnu_cxx::_pool_allocator<int>) << endl;
cout << sizeof(_gnu_cxx::_mt_allocator<int>) << endl;
cout << sizeof(_gnu_cxx::bitmap_allocator<int>) << endl;
cout << sizeof(_gnu_cxx::array_allocator<int>) << endl;
cout << sizeof(_gnu_cxx::debug_allocator<std::all
//cout << sizeof(_gnu_cxx::throw_allocator<int>) << endl;

// 測試 cookie
list<int, _gnu_cxx::malloc_allocator<int>> list_
deque<int, _gnu_cxx::debug_allocator<std::allocat
vector<int, _gnu_cxx::_pool_allocator<int>> vector_p
2210
2211
2212
2213
2214
2215
2216
2217
cookie_test(std::allocator<int>(), 1);
cookie_test(_gnu_cxx::malloc_allocator<int>(), 1)

-----  
- Errors: 0  
- Warnings: 0  
- Output Filename: D:\handout\C++11-test-DevC++\Test-C+
- Output Size: 2.10485744476318 MiB  
- Compilation Time: 7.05s
```



## 重要網頁, CPlusPlus.com

www.cplusplus.com

Search:  Go Not logged in register log in

Welcome to cplusplus.com © The C++ Resources Network, 2013

**C++**

Information  
Tutorials  
Reference  
Articles  
Forum

**Information**  
General information about the C++ programming language, including non-technical documents and descriptions:  

- Description of the C++ language
- History of the C++ language
- F.A.Q., Frequently Asked Questions

**Tutorials**  
Learn the C++ language from its basics up to its most advanced features.  

- C++ Language: Collection of tutorials covering all the features of this versatile and powerful language. Including detailed explanations of pointers, functions, classes and templates, among others...
- more...

**Reference**  
Description of the most important classes, functions and objects of the Standard Language Library, with descriptive fully-functional short programs as examples:  

- C library: The popular C library, is also part of the of C++ language library.
- iostream library. The standard C++ library for Input/Output operations.
- String library. Library defining the string class.
- Standard containers. Vectors, lists, maps, sets...
- more...

**Articles**  
User-contributed articles, organized into different categories:  

- Algorithms
- Standard library
- C++11
- Windows API
- Other...

You can contribute your own articles!

**Forum**  
Message boards where members can exchange knowledge and comments. Ordered by topics:

**C++ Search**  
Search this website:  Search



## 重要網頁, CppReference.com

The screenshot shows the homepage of CppReference.com. At the top, there's a banner stating "C++ is 30 years old" with the date "October 14, 1985". Below the banner, the main navigation menu includes "Page", "Discussion", "View", "View source", and "History". The left sidebar contains links for "Language", "Headers", "Library concepts", "Utilities library", and "Technical specifications". The main content area is divided into several sections: "Strings library", "Containers library", "Algorithms library", "Iterators library", "Numerics library", "Input/output library", and "Localizations library". Each section lists specific components or classes. At the bottom of the page, there are sections for "Standard library extensions" and "Standard library extensions v2".

C++ is 30 years old  
October 14, 1985 was the release date of the first commercial compiler and the first edition of The C++ Programming Language

## C++ reference

C++98, C++03, C++11, C++14

Language

- ASCII chart
- Compiler support
- Preprocessor
- Keywords
- Operator precedence
- Escape sequences
- Fundamental types

Headers

Library concepts

Utilities library

- Type support
- Dynamic memory management
- Error handling
- Program Utilities
- Date and time
- bitset
- Function objects
- pair - tuple (C++11)
- integer\_sequence (C++14)

Technical specifications

Standard library extensions (library fundamentals TS)  
optional - any - basic\_string\_view  
polymorphic allocators - search

Standard library extensions v2 (library fundamentals TS v2)  
propagate\_const - not\_fn - observer\_ptr  
source\_location - ostream\_joiner  
detection idiom - uniform\_container erasure

Strings library

- basic\_string
- Null-terminated byte strings
- Null-terminated multibyte strings
- Null-terminated wide strings

Containers library

- array (C++11)
- vector - deque
- list - forward\_list (C++11)
- set - multiset
- map - multimap
- unordered\_set (C++11)
- unordered\_multiset (C++11)
- unordered\_map (C++11)
- unordered\_multimap (C++11)
- stack - queue - priority\_queue

Algorithms library

Iterators library

Numerics library

- Common mathematical functions
- Complex numbers
- Pseudo-random number generation

Input/output library

- basic\_streambuf
- basic\_filebuf
- basic\_stringbuf
- ios\_base
- basic\_ios
- basic\_istream
- basic\_ostream
- basic\_iostream
- basic\_ifstream
- basic\_ofstream
- basic\_fstream
- basic\_istringstream
- basic\_ostringstream
- basic\_stringstream

Localizations library

Regular expressions library (C++11)

Atomic operations library (C++11)

Thread support library (C++11)



## 重要網頁, [gcc.gnu.org](https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a00464.html)

libstdc++

https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a00464.html

**std::deque Class Reference**  
Sequences

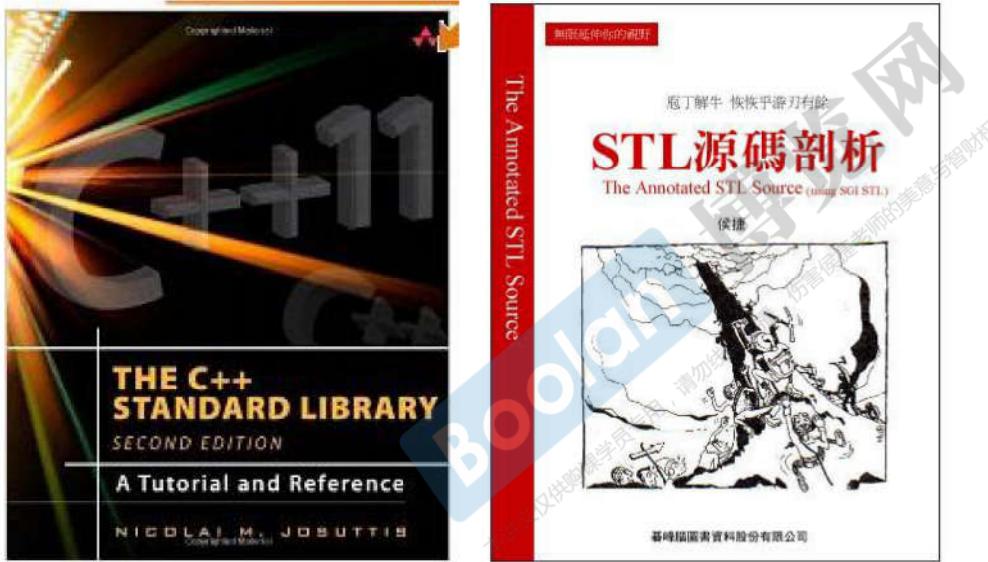
Inheritance diagram for std::deque:

```
graph TD; std::Deque_base["std::Deque_base< _Tp, _Alloc >"] --> std::Deque_base["std::Deque_base< _Tp, _Alloc >"]; std::Deque_base --> std::deque["std::deque"]
```

Public Types

- typedef \_Alloc allocator\_type
- typedef \_Base::const\_iterator const\_iterator
- typedef \_Tp\_alloc\_type::const\_pointer const\_pointer
- typedef \_Tp\_alloc\_type::const\_reference const\_reference
- typedef std::reverse\_iterator< const\_iterator > const\_reverse\_iterator
- typedef ptrdiff\_t difference\_type
- typedef \_Base::iterator iterator
- typedef \_Tp\_alloc\_type::pointer pointer
- typedef \_Tp\_alloc\_type::reference reference

## ■■■■ Bibliography (書目誌)



— 侯捷 —

10

■■■ 你將獲得的代碼

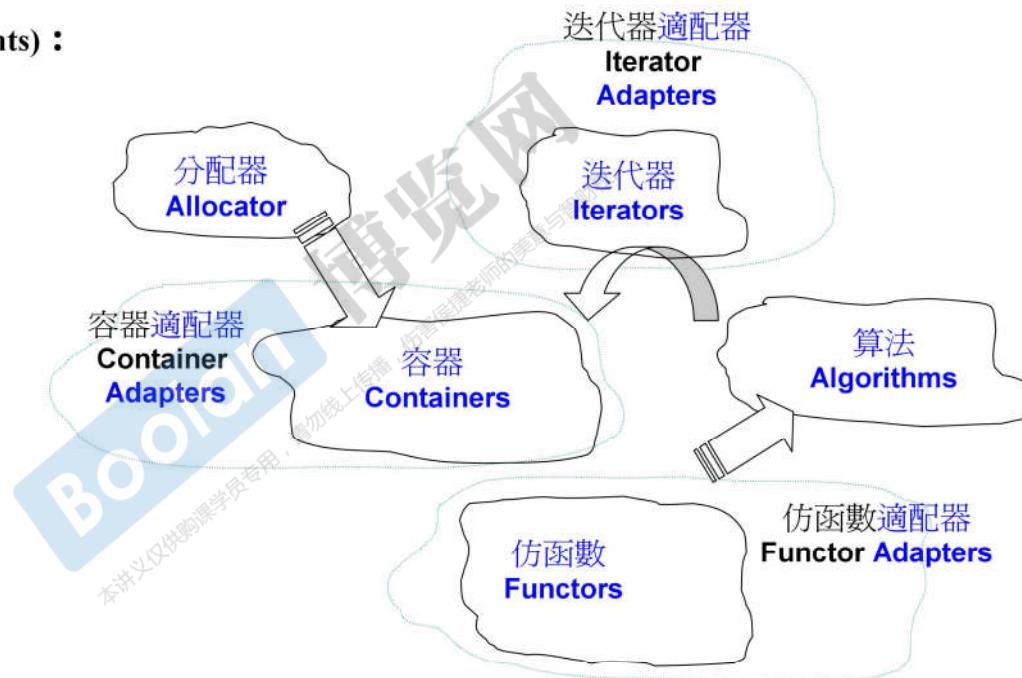
### Test-STL.cpp

內有對 C++ 標準庫的各種測試

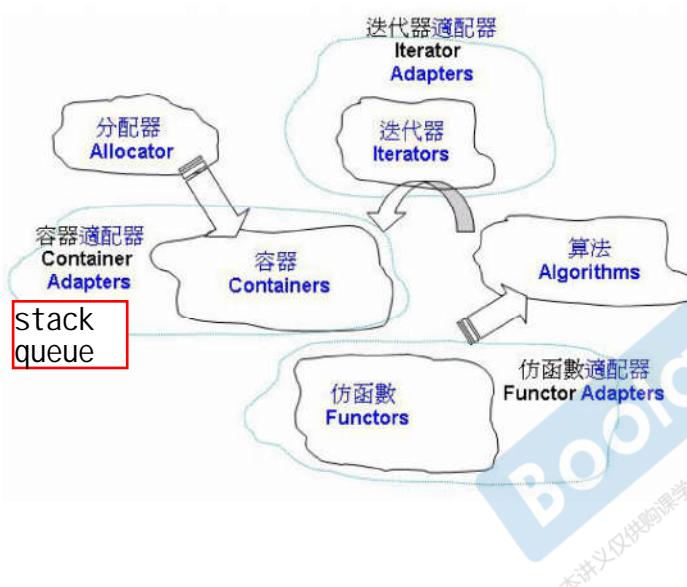
## ■■■■ STL 六大部件

STL 六大部件 (Components) :

- 容器 (Containers)
- 分配器 (Allocators)
- 算法 (Algorithms)
- 迭代器 (Iterators)
- 適配器 (Adapters)
- 仿函式 (Functors)



## STL 六大部件關係



```
01 #include <vector>
02 #include <algorithm>
03 #include <functional>
04 #include <iostream>
05
06 using namespace std;
07
08 int main()
09 {
10     int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
11     vector<int, allocator<int>> vi(ia,ia+6);
12
13     cout << count_if(vi.begin(), vi.end(),
14                      not1(bind2nd(less<int>(), 40)));
15
16     return 0;
17 }
```

Annotations for the code:

- allocator**: Points to the `allocator<int>` in the `vector` constructor.
- container**: Points to the `vector` type.
- iterator**: Points to the `begin()` and `end()` member functions.
- algorithm**: Points to the `count_if` function.
- function adapter (negator)**: Points to the `not1` function.
- function adapter (binder)**: Points to the `bind2nd` function.
- predicate**: Points to the `less<int>()` function.
- function object**: Points to the `40` argument.

— 侯捷 —

13

## ■■■ 複雜度, Complexity, Big-oh

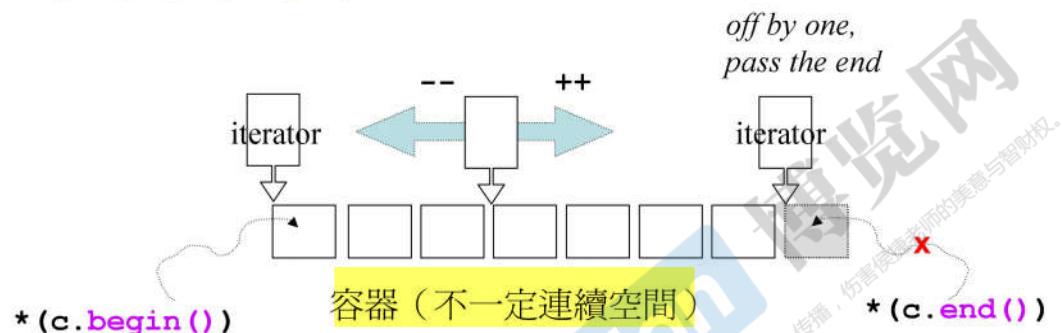
目前常見的 Big-oh 有下列幾種情形：

1.  $O(1)$ 或 $O(c)$ ：稱為常數時間(constant time)
2.  $O(n)$ ：稱為線性時間(linear time)
3.  $O(\log_2 n)$ ：稱為次線性時間(sub-linear time)
4.  $O(n^2)$ ：稱為平方時間(quadratic time)
5.  $O(n^3)$ ：稱為立方時間(cubic time)
6.  $O(2^n)$ ：稱為指數時間(exponential time)
7.  $O(n \log_2 n)$ ：介於線性及二次方成長的中間之行為模式。

## ■■■■ “前閉後開” 區間

[ ] ( ) [ ]

迭代器就是一个泛化的  
指针



```
Container<T> c;  
...  
Container<T>::iterator ite = c.begin();  
for (; ite != c.end(); ++ite)  
    ...
```

— 侯捷 —

15

## |||| range-based for statement (since C++11)

```
for ( decl : coll ) {  
    statement  
}
```

```
for ( int i : { 2, 3, 5, 7, 9, 13, 17, 19 } ) {  
    std::cout << i << std::endl;  
}
```

```
std::vector<double> vec;  
...  
for ( auto elem : vec ) {  
    std::cout << elem << std::endl;  
}  
  
for ( auto& elem : vec ) {  
    elem *= 3;  
}
```

— 侯捷 —

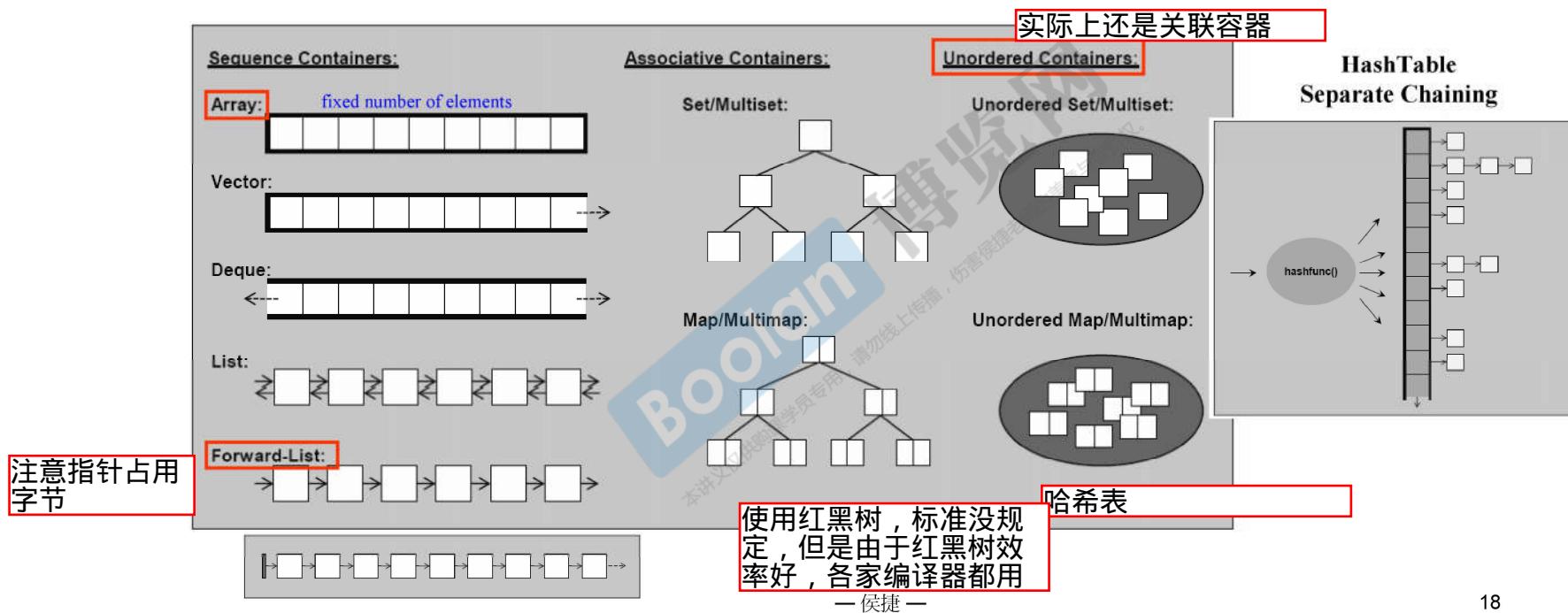
## |||| auto keyword (since C++11)

```
list<string> c;  
...  
list<string>::iterator ite;  
ite = ::find(c.begin(), c.end(), target);
```



```
list<string> c;  
...  
auto ite = ::find(c.begin(), c.end(), target);
```

## 容器 – 結構與分類



## 以下測試程序之 輔助函數

```
11  using std::cin;
12  using std::cout;
13  using std::string;
14
15  long get_a_target_long()
16  {
17      long target=0;
18
19      cout << "target (0~" << RAND_MAX << "): ";
20      cin >> target;
21      return target;
22  }
23
24  string get_a_target_string()
25  {
26      long target=0;
27      char buf[10];
28
29      cout << "target (0~" << RAND_MAX << "): ";
30      cin >> target;
31      sprintf(buf, 10, "%d", target);
32      return string(buf);
33  }
```

```
35  int compareLongs(const void* a, const void* b)
36  {
37      return ( *(long*)a - *(long*)b );
38  }
39
40  int compareStrings(const void* a, const void* b)
41  {
42      if ( *(string*)a > *(string*)b )
43          return 1;
44      else if ( *(string*)a < *(string*)b )
45          return -1;
46      else
47          return 0;
48  }
```

## 使用容器 array

Array:

```
51 #include <array>
52 #include <iostream>
53 #include <ctime>
54 #include <cstdlib> //qsort, bsearch, NULL
55 namespace jj01
56 {
57     void test_array()
58     {
59         cout << "\ntest_array()..... \n";
60
61         array<long,ASIZE> c;
62
63         clock_t timeStart = clock();
64         for(long i=0; i< ASIZE; ++i) {
65             c[i] = rand();
66         }
67         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
68         cout << "array.size()= " << c.size() << endl;
69         cout << "array.front()= " << c.front() << endl;
70         cout << "array.back()= " << c.back() << endl;
71         cout << "array.data()= " << c.data() << endl;
72
73         long target = get_a_target_long();
74
75         timeStart = clock();
76         qsort(c.data(), ASIZE, sizeof(long), compareLongs);
77         long* pItem = (long*)bsearch(&target, (c.data()), ASIZE, sizeof(long), compareLongs);
78         cout << "qsort()+bsearch(), milli-seconds : " << (clock()-timeStart) << endl; //
79         if (pItem != NULL)
80             cout << "found, " << *pItem << endl;
81         else
82             cout << "not found! " << endl;
83     }
84 }
```

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 1

test_array()..... .
milli-seconds : 47
array.size()= 500000
array.front()= 3557
array.back()= 23084
array.data()= 0x47a20
target <0x32767>: 20000
qsort()>+bsearch(), milli-seconds : 187
found, 20000
```

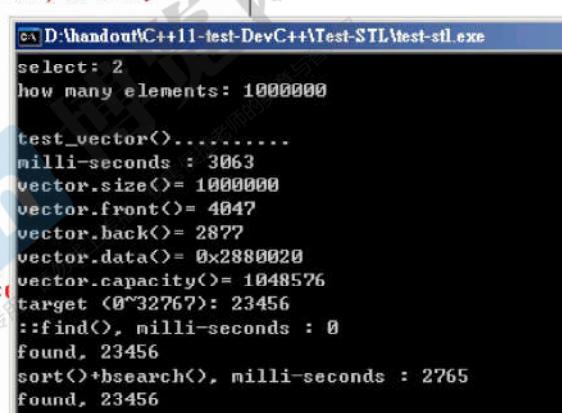
## 使用容器 vector

```
Vector:   
D:\handout\c++11-test-DevC++\Test-STL\test-stl.exe  
select: 2  
how many elements: 1000000  
  
test_vector().....  
milli-seconds : 3063  
vector.size()= 1000000  
vector.front()= 4047  
vector.back()= 2877  
vector.data()= 0x2880020  
vector.capacity()= 1048576  
target (0"32767): 23456  
::find(), milli-seconds : 0  
found, 23456  
sort()>bsearch(), milli-seconds : 2765  
found, 23456
```

```
86 #include <vector>  
87 #include <stdexcept>  
88 #include <string>  
89 #include <cstdlib> //abort()  
90 #include <cstdio> //snprintf()  
91 #include <iostream>  
92 #include <ctime>  
93 #include <algorithm> //sort()  
94 namespace jj02  
95 {  
96     void test_vector(long& value)  
97     {  
98         cout << "\ntest_vector()..... \n";  
99         vector<string> c;  
100        char buf[10];  
101        clock_t timeStart = clock();  
102        for(long i=0; i< value; ++i)  
103        {  
104            try {  
105                snprintf(buf, 10, "%d", rand());  
106                c.push_back(string(buf));  
107            }  
108            catch(exception& p) {  
109                cout << "i=" << i << " " << p.what() << endl;  
110                //曾經最高 i=58389486 then std::bad_alloc  
111                abort();  
112            }  
113        }  
114    }  
115    cout << "milli-seconds : " << (clock()-timeStart) << endl;  
116    cout << "vector.size()= " << c.size() << endl;  
117    cout << "vector.front()= " << c.front() << endl;  
118    cout << "vector.back()= " << c.back() << endl;  
119    cout << "vector.data()= " << c.data() << endl;  
120    cout << "vector.capacity()= " << c.capacity() << endl;  
121  
122  
123 }
```

## 使用容器 vector

```
124 string target = get_a_target_string();
125 {
126     timeStart = clock();
127     auto pItem = ::find(c.begin(), c.end(), target);
128     cout << "::find()", milli-seconds : " << (clock()-timeStart) << endl;
129
130     if (pItem != c.end())
131         cout << "found, " << *pItem << endl;
132     else
133         cout << "not found! " << endl;
134 }
135
136 {
137     timeStart = clock();
138     sort(c.begin(), c.end());
139     string* pItem = (string*)bsearch(&target, (c.data()),
140                                     c.size(), sizeof(string));
141     cout << "sort()>bsearch()", milli-seconds : " << (clock()
142
143     if (pItem != NULL)
144         cout << "found, " << *pItem << endl;
145     else
146         cout << "not found! " << endl;
147 }
148
149 }
```



## 使用容器 list



```
ex D:\handout\C++11-test-DevC++\Test-STL\test
select: 3
how many elements: 1000000

test_list().....
milli-seconds : 3265
list.size()= 1000000
list.max_size()= 357913941
list.front()= 4710
list.back()= 16410
target <0~32767>: 23456
::find(), milli-seconds : 16
found, 23456
c.sort(), milli-seconds : 2312
```

```
159 namespace jj03
160 {
161     void test_list(long& value)
162     {
163         cout << "\ntest_list()..... \n";
164
165         list<string> c;
166         char buf[10];
167
168         clock_t timeStart = clock();
169         for(long i=0; i< value; ++i)
170         {
171             try {
172                 sprintf(buf, 10, "%d", rand());
173                 c.push_back(string(buf));
174             }
175             catch(exception& p) {
176                 cout << "i=" << i << " " << p.what() << endl; // abort();
177             }
178         }
179         cout << "milli-seconds : " << (clock()-timeStart) << endl;
180         cout << "list.size()= " << c.size() << endl;
181         cout << "list.max_size()= " << c.max_size() << endl;
182         cout << "list.front()= " << c.front() << endl;
183         cout << "list.back()= " << c.back() << endl;
184
185         string target = get_a_target_string();
186         timeStart = clock();
187         auto pItem = ::find(c.begin(), c.end(), target);
188         cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
189
190         if (pItem != c.end())
191             cout << "found, " << *pItem << endl;
192         else
193             cout << "not found! " << endl;
194
195         timeStart = clock();
196         c.sort();
197         cout << "c.sort(), milli-seconds : " << (clock()-timeStart) << endl;
198
199     } //又一陣子才離開函數; 它在 destroy...
200 }
```

/\* Returns the number of elements in the %list. \*/
size\_type  
size() const \_GLIBCXX\_NOEXCEPT
{ return std::distance(begin(), end()); }

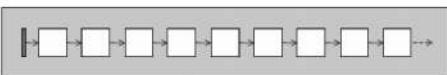
/\* Returns the size() of the largest possible %list. \*/
size\_type  
max\_size() const \_GLIBCXX\_NOEXCEPT
{ return \_M\_get\_Node\_allocator().max\_size(); }

size\_type  
max\_size() const \_GLIBCXX\_USE\_NOEXCEPT
{ return size\_t(-1) / sizeof(\_Tp); }

std::allocator<\_Tp>, 亦即  
new\_allocator, 內有:



## 使用容器 `forward_list`



```
c:\D\handout\c++-test\DevC++\Test-SIL\test-  
select: 4  
how many elements: 10000000  
  
test_forward_list().....  
milli-seconds : 3204  
forward_list.max_size()= 536870911  
forward_list.front()= 8180  
target (0~32767): 23456  
::find(), milli-seconds : 15  
found, 23456  
c.sort(), milli-seconds : 2656
```

```
210 namespace jj04
211 {
212     void test_forward_list(long& value)
213     {
214         cout << "\ntest_forward_list()..... \n";
215
216         forward_list<string> c;
217         char buf[10];
218
219         clock_t timeStart = clock();
220         for(long i=0; i< value; ++i)
221         {
222             try {
223                 sprintf(buf, 10, "%d", rand());
224                 c.push_front(string(buf));
225             }
226             catch(exception& p) {
227                 cout << "i=" << i << " " << p.what() << endl; // abort();
228             }
229         }
230         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
231         cout << "forward_list.max_size()=" << c.max_size() << endl;
232         cout << "forward_list.front()=" << c.front() << endl;
233         // cout << "forward_list.back()=" << c.back() << endl; //no such member function
234         // cout << "forward_list.size()=" << c.size() << endl; //no such member function
235
236
237
238         string target = get_a_target_string();
239         timestamp = clock();
240         auto pItem = ::find(c.begin(), c.end(), target);
241         cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl; //
242
243         if (pItem != c.end())
244             cout << "found, " << *pItem << endl;
245         else
246             cout << "not found! " << endl;
247
248         timeStart = clock();
249         c.sort();
250         cout << "c.sort(), milli-seconds : " << (clock()-timeStart) << endl; //
251     }
252 }
```

g++独有的，和forward\_list的用法相同，单向链表



## 使用容器 slist



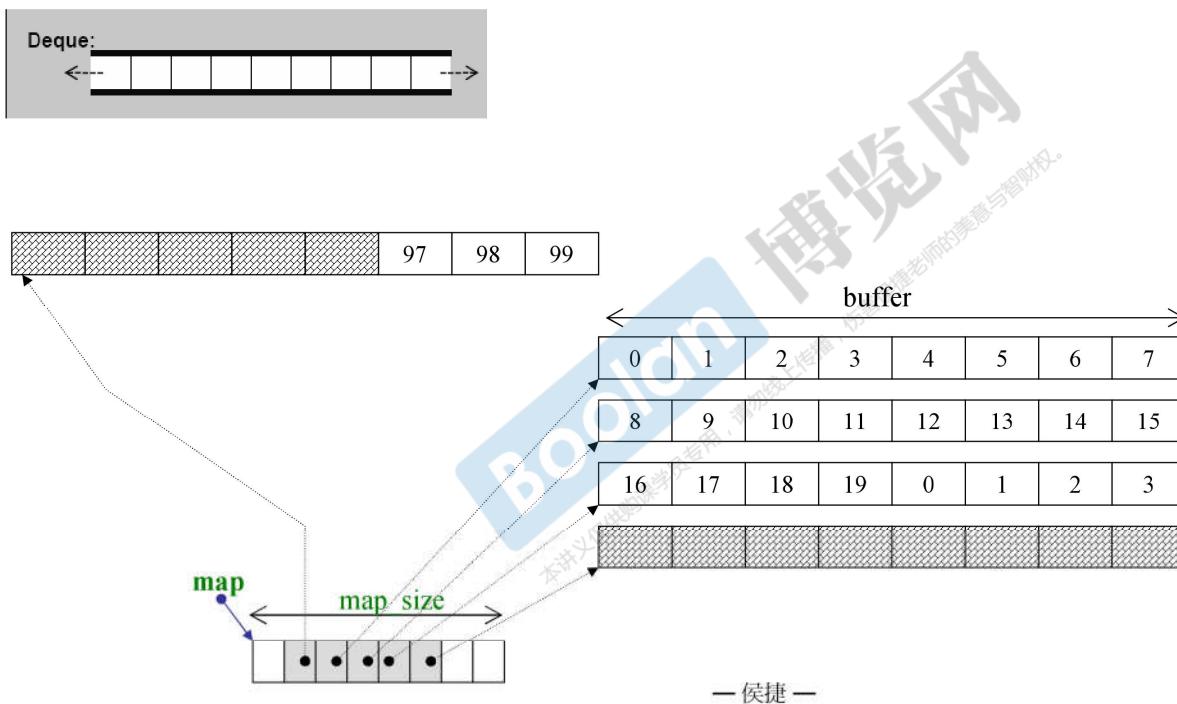
```
cd D:\handout\C++11-test-DevC++\Test-Slist
select: 10
how many elements: 1000000
test_slist().....milli-seconds : 3046
```

```
508 #include <ext\list>
509 #include <stdexcept>
510 #include <string>
511 #include <cstdlib> //abort()
512 #include <cstdio> //snprintf()
513 #include <iostream>
514 #include <ctime>
515 namespace jj10
516 {
517     void test_slist(long& value)
518     {
519         cout << "\ntest_slist().....\n";
520
521         __gnu_cxx::slist<string> c;
522         char buf[10];
523
524         clock_t timeStart = clock();
525         for(long i=0; i < value; ++i)
526         {
527             try
528             {
529                 snprintf(buf, 10, "%d", rand());
530                 c.push_front(string(buf));
531             }
532             catch(exception& p)
533             {
534                 cout << "i=" << i << " " << p.what() << endl; // abort();
535             }
536         }
537         cout << "milli-seconds : " << (clock()-timeStart) << endl;
538     }
539 }
```

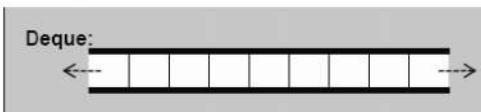


## 使用容器 deque

deque其实不是连续的，只是让使用者使用起来是连续的。++的重载



## 使用容器 deque

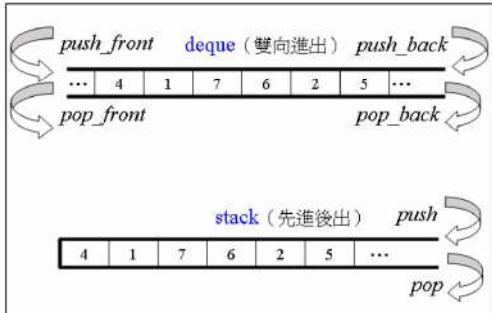


```
D:\handout\c++11-test\DevC++\Test-STL\test>
select: 5
how many elements: 1000000

test_deque().....
milli-seconds : 2704
deque.size()= 1000000
deque.front()= 5249
deque.back()= 1098
deque.max_size()= 1073741823
target <0^32767>: 23456
::find(), milli-seconds : 15
found, 23456
::sort(), milli-seconds : 3110
```

```
261 namespace jj05
262 {
263     void test_deque(long& value)
264     {
265         cout << "\ntest_deque()..... \n";
266
267         deque<string> c;
268         char buf[10];
269
270         clock_t timeStart = clock();
271         for(long i=0; i< value; ++i)
272         {
273             try {
274                 sprintf(buf, 10, "%d", rand());
275                 c.push_back(string(buf));
276             }
277             catch(exception& p) {
278                 cout << "i=" << i << " " << p.what() << endl; // abort();
279             }
280         }
281         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
282         cout << "deque.size()= " << c.size() << endl;
283         cout << "deque.front()= " << c.front() << endl;
284         cout << "deque.back()= " << c.back() << endl;
285         cout << "deque.max_size()= " << c.max_size() << endl;
286
287         string target = get_a_target_string();
288         timeStart = clock();
289         auto pItem = ::find(c.begin(), c.end(), target);
290         cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
291
292         if (pItem != c.end())
293             cout << "found: " << *pItem << endl;
294         else
295             cout << "not found! " << endl;
296
297         timeStart = clock();
298         ::sort(c.begin(), c.end());
299         cout << "::sort(), milli-seconds : " << (clock()-timeStart) << endl;
300
301     }
302 }
```

## 使用容器 stack



```
c:\D\handout\C++11-test>DevC++\Test  
select: 17  
how many elements: 300000  
  
test_stack().....  
milli-seconds : 812  
stack.size()= 300000  
stack.top()= 23929  
stack.size()= 299999  
stack.top()= 12911
```

```
847 namespace jj17  
848 {  
849     void test_stack(long& value)  
850     {  
851         cout << "\ntest_stack()..... \n";  
852         stack<string> c;  
853         char buf[10];  
854  
855         clock_t timeStart = clock();  
856         for(long i=0; i< value; ++i)  
857         {  
858             try {  
859                 snprintf(buf, 10, "%d", rand());  
860                 c.push(string(buf));  
861             }  
862             catch(exception& p){  
863                 cout << "i=" << i << " " << p.what() << endl;  
864                 abort();  
865             }  
866         }  
867         cout << "milli-seconds : " << (clock()-timeStart) << endl;  
868         cout << "stack.size()= " << c.size() << endl;  
869         cout << "stack.top()= " << c.top() << endl;  
870         c.pop();  
871         cout << "stack.size()= " << c.size() << endl;  
872         cout << "stack.top()= " << c.top() << endl;  
873     }  
874 }  
875 }
```

都是利用了dequeue

不能支持迭代器，会破坏语义



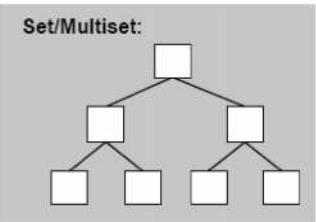
## 使用容器 queue



```
ex D:\handoutC++11-test-DevC++\Test-  
select: 18  
how many elements: 300000  
  
test_queue().....  
milli-seconds : 890  
queue.size()= 300000  
queue.front()= 12074  
queue.back()= 14585  
queue.size()= 299999  
queue.front()= 6888  
queue.back()= 14585
```

```
884 namespace jj18  
885 {  
886 void test_queue(long& value)  
887 {  
888     cout << "\ntest_queue()..... \n";  
889  
890     queue<string> c;  
891     char buf[10];  
892  
893     clock_t timeStart = clock();  
894     for(long i=0; i< value; ++i)  
895     {  
896         try {  
897             snprintf(buf, 10, "%d", rand());  
898             c.push(string(buf));  
899         }  
900         catch(exception& p) {  
901             cout << "i=" << i << " " << p.what() << endl;  
902             abort();  
903         }  
904     }  
905     cout << "milli-seconds : " << (clock()-timeStart) << endl;  
906     cout << "queue.size()= " << c.size() << endl;  
907     cout << "queue.front()= " << c.front() << endl;  
908     cout << "queue.back()= " << c.back() << endl;  
909     c.pop();  
910     cout << "queue.size()= " << c.size() << endl;  
911     cout << "queue.front()= " << c.front() << endl;  
912     cout << "queue.back()= " << c.back() << endl;  
913 }  
914 }
```

## 使用容器 multiset



```
ex D:\handout\c++11-test-DevC++\Test-STL>
select: 6
how many elements: 1000000

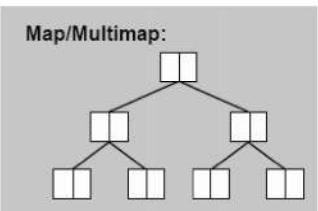
test_multiset<...>.....
milli-seconds : 6609
multiset.size()= 1000000
multiset.max_size()= 214748364
target <0~32767>: 23456
::find(), milli-seconds : 203
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

```
313 void test_multiset(long& value)
314 {
315     cout << "\ntest_multiset()..... \n";
316
317     multiset<string> c;
318     char buf[10];
319     clock_t timeStart = clock();
320     for(long i=0; i< value; ++i)
321     {
322         try {
323             sprintf(buf, 10, "%d", rand());
324             c.insert(string(buf));
325         }
326         catch(exception& p) {
327             cout << "i=" << i << " " << p.what() << endl; // abort();
328         }
329     }
330     cout << "milli-seconds : " << (clock()-timeStart) << endl; //
331     cout << "multiset.size()= " << c.size() << endl;
332     cout << "multiset.max_size()= " << c.max_size() << endl;
333
334     string target = get_a_target_string();
335     {
336         timeStart = clock();
337         auto pItem = ::find(c.begin(), c.end(), target); //比 c.find(...) 慢很多
338         cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
339         if (pItem != c.end())
340             cout << "found, " << *pItem << endl;
341         else
342             cout << "not found! " << endl;
343     }
344     {
345         timeStart = clock(); //比 ::find(...) 快很多
346         auto pItem = c.find(target);
347         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
348         if (pItem != c.end())
349             cout << "found, " << *pItem << endl;
350         else
351             cout << "not found! " << endl;
352     }
353 }
```

插入比较慢，查找很快



## 使用容器 multimap



```
ex D:\handout\C++11-test-DevC++\Test-STL>
select: 7
how many elements: 1000000

test_multimap<>. .....
milli-seconds : 4812
multimap.size()= 1000000
multimap.max_size()= 178956970
target <0~32767>: 23456
c.find(), milli-seconds : 0
found, value=29247
```

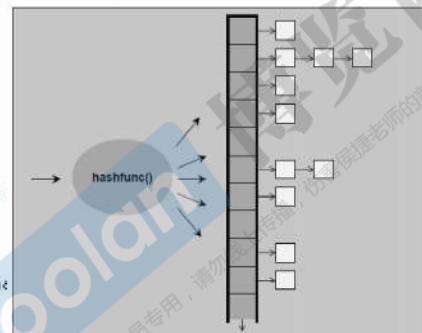
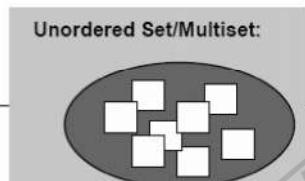
```
364 namespace jj07
365 {
366     void test_multimap(long& value)
367     {
368         cout << "\ntest_multimap()..... \n";
369
370         multimap<long, string> c;
371         char buf[10];
372
373         clock_t timeStart = clock();
374         for(long i=0; i< value; ++i)
375         {
376             try {
377                 snprintf(buf, 10, "%d", rand());
378                 //multimap 不可使用 [] 做 insertion
379                 c.insert(pair<long, string>(i,buf));
380             }
381             catch(exception& p) {
382                 cout << "i=" << i << " " << p.what() << endl; //
383                 abort();
384             }
385         }
386         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
387         cout << "multimap.size()= " << c.size() << endl;
388         cout << "multimap.max_size()= " << c.max_size() << endl;
389
390         long target = get_a_target_long();
391         timeStart = clock();
392         auto pItem = c.find(target);
393         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
394         if (pItem != c.end())
395             cout << "found, value=" << (*pItem).second << endl;
396         else
397             cout << "not found! " << endl;
398     }
399 }
```

bucket的数目一定比元素多。内部实现：当元素的数目等于bucket的数目了，bucket数翻倍

插入用的时间比红黑树的少

## 使用容器 unordered\_multiset

```
408 namespace jj08
409 {
410     void test_unordered_multiset(long& value)
411     {
412         cout << "\ntest_unordered_multiset()..... \n";
413
414         unordered_multiset<string> c;
415         char buf[10];
416
417         clock_t timeStart = clock();
418         for(long i=0; i< value; ++i)
419         {
420             try {
421                 snprintf(buf, 10, "%d", rand());
422                 c.insert(string(buf));
423             }
424             catch(exception& p) {
425                 cout << "i=" << i << " " << p.what();
426                 abort();
427             }
428         }
429         cout << "milli-seconds : " << (clock()-timeStart) << endl; // 
430         cout << "unordered_multiset.size()= " << c.size() << endl;
431         cout << "unordered_multiset.max_size()= " << c.max_size() << endl;
432         cout << "unordered_multiset.bucket_count()= " << c.bucket_count() << endl;
433         cout << "unordered_multiset.load_factor()= " << c.load_factor() << endl;
434         cout << "unordered_multiset.max_load_factor()= " << c.max_load_factor() << endl;
435         cout << "unordered_multiset.max_bucket_count()= " << c.max_bucket_count() << endl;
436         for (unsigned i=0; i< 20; ++i)
437             cout << "bucket #" << i << " has " << c.bucket_size(i) << " elements.\n";
438     }
}
```



```
ex D:\handout\c++11-test-DevC++\Test-STL\test-stl.exe
select: 8
how many elements: 1000000
test_unordered_multiset().....
milli-seconds : 4406
unordered_multiset.size()= 1000000
unordered_multiset.max_size()= 357913941
unordered_multiset.bucket_count()= 1056323
unordered_multiset.load_factor()= 0.94668
unordered_multiset.max_load_factor()= 1
unordered_multiset.max_bucket_count()= 357913941
bucket #0 has 0 elements.
bucket #1 has 0 elements.
bucket #2 has 0 elements.
bucket #3 has 0 elements.
bucket #4 has 0 elements.
bucket #5 has 0 elements.
bucket #6 has 0 elements.
bucket #7 has 0 elements.
bucket #8 has 0 elements.
bucket #9 has 0 elements.
bucket #10 has 0 elements.
bucket #11 has 0 elements.
bucket #12 has 24 elements.
bucket #13 has 0 elements.
bucket #14 has 0 elements.
bucket #15 has 0 elements.
bucket #16 has 0 elements.
bucket #17 has 0 elements.
bucket #18 has 0 elements.
bucket #19 has 0 elements.
target <0x32767>: 23456
:::find(), milli-seconds : 109
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

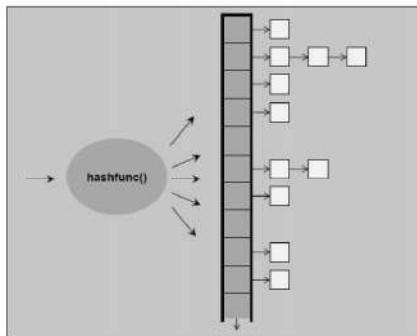
## 使用容器 `unordered_multiset`

```
439 string target = get_a_target_string();
440 {
441     timeStart = clock();
442     auto pItem = ::find(c.begin(), c.end(), target); //比 c.find(...) 快很多
443     cout << "::find()", milli-seconds : " << (clock()-timeStart) << endl;
444     if (pItem != c.end())
445         cout << "found, " << *pItem << endl;
446     else
447         cout << "not found! " << endl;
448 }
449 {
450     timeStart = clock();
451     auto pItem = c.find(target); //比 ::find(...) 快很多
452     cout << "c.find()", milli-seconds : " << (clock()-timeStart) << endl;
453     if (pItem != c.end())
454         cout << "found, " << *pItem << endl;
455     else
456         cout << "not found! " << endl;
457 }
458 }
459 }
460 }
```

```
c:\D:\handout\c++11-test-DevC++\Test-STL\test-stl.exe
select: 8
how many elements: 1000000

test_unordered_multiset().....
milli-seconds : 4406
unordered_multiset.size()= 1000000
unordered_multiset.max_size()= 357913941
unordered_multiset.bucket_count()= 1056323
unordered_multiset.load_factor()= 0.94668
unordered_multiset.max_load_factor()= 1
unordered_multiset.max_bucket_count()= 357913941
bucket #0 has 0 elements.
bucket #1 has 0 elements.
bucket #2 has 0 elements.
bucket #3 has 0 elements.
bucket #4 has 0 elements.
bucket #5 has 0 elements.
bucket #6 has 0 elements.
bucket #7 has 0 elements.
bucket #8 has 0 elements.
bucket #9 has 0 elements.
bucket #10 has 0 elements.
bucket #11 has 0 elements.
bucket #12 has 24 elements.
bucket #13 has 0 elements.
bucket #14 has 0 elements.
bucket #15 has 0 elements.
bucket #16 has 0 elements.
bucket #17 has 0 elements.
bucket #18 has 0 elements.
bucket #19 has 0 elements.
target <0'32767>: 23456
::find(), milli-seconds : 109
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

## 使用容器 `unordered_multimap`

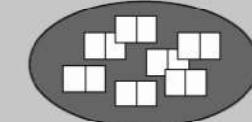


```
ex>D:\handout\c++11-test\DevC++\Test-STL\test-stl.exe
select: 9
how many elements: 1000000

test_unordered_multimap().....
milli-seconds : 4313
unordered_multimap.size()= 1000000
unordered_multimap.max_size()= 357913941
target <0^32767>: 23456
c.find(), milli-seconds : 0
found, value=15962
```

```
470 namespace jj09
471 {
472     void test_unordered_multimap(long& value)
473     {
474         cout << "\ntest_unordered_multimap()..... \n";
475
476         unordered_multimap<long, string> c;
477         char buf[10];
478
479         clock_t timeStart = clock();
480         for(long i=0; i< value; ++i)
481         {
482             try {
483                 sprintf(buf, 10, "%d", rand());
484                 //multimap 不可用 [] 进行 insertion
485                 c.insert(pair<long,string>(i,buf));
486             }
487             catch(exception& p) {
488                 cout << "i=" << i << " " << p.what() << endl; // abort();
489             }
490         }
491         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
492         cout << "unordered_multimap.size()= " << c.size() << endl;
493         cout << "unordered_multimap.max_size()= " << c.max_size() << endl;
494
495
496         long target = get_a_target_long();
497         timeStart = clock();
498         auto pItem = c.find(target);
499         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
500         if (pItem != c.end())
501             cout << "found, value=" << (*pItem).second << endl;
502         else
503             cout << "not found! " << endl;
504     }
505 }
506 }
```

Unordered Map/Multimap:

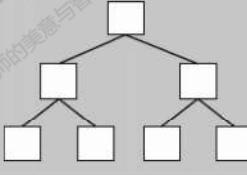


## 不可重复

### 使用容器 set

```
641 namespace jj13
642 {
643     void test_set(long& value)
644     {
645         cout << "\ntest_set()..... \n";
646         set<string> c;
647         char buf[10];
648
649         clock_t timeStart = clock();
650         for(long i=0; i< value; ++i)
651         {
652             try {
653                 snprintf(buf, 10, "%d", rand());
654                 c.insert(string(buf));
655             } catch(exception& p) {
656                 cout << "i=" << i << " " << p.what() << endl;
657                 abort();
658             }
659         }
660         cout << "milli-seconds : " << (clock()-timeStart) << endl;
661         cout << "set.size()= " << c.size() << endl;
662         cout << "set.max_size()= " << c.max_size() << endl;
663
664         string target = get_a_target_string();
665         {
666             timeStart = clock();
667             auto pItem = ::find(c.begin(), c.end(), target); //比 c.find(...) 慢很多
668             cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
669             if (pItem != c.end())
670                 cout << "found, " << *pItem << endl;
671             else
672                 cout << "not found! " << endl;
673         }
674     }
675 }
```

676 {  
677 timeStart = clock();  
678 auto pItem = c.find(target); //比 ::find(...) 快很多  
679 cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;  
680 if (pItem != c.end())  
681 cout << "found, " << \*pItem << endl;  
682 else  
683 cout << "not found! " << endl;  
684 }  
685 }

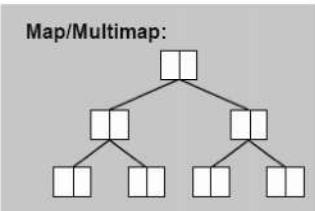
Set/Multiset:  


```
D:\handout\c++11-test-DevC++\Test-STL
select: 13
how many elements: 1000000

test_set().....
milli-seconds : 3922
set.size()= 32768
set.max_size()= 214748364
target (<0~32767): 23456
::find(), milli-seconds : 0
found, 23456
c.find(), milli-seconds : 0
found, 23456
```



## 使用容器 map



```
ex D:\handout\c++11-test-DevC++\Test-STI
select: 14
how many elements: 1000000

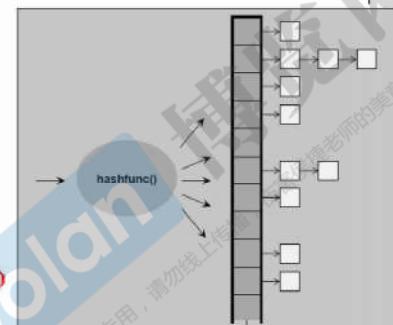
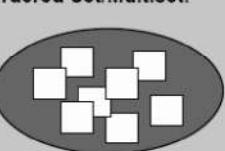
test_map().....
milli-seconds : 4890
map.size()= 1000000
map.max_size()= 178956970
target <0~32767>: 23456
c.find(), milli-seconds : 0
found, value=19128
```

```
696 namespace jj14
697 {
698     void test_map(long& value)
699     {
700         cout << "\ntest_map()..... \n";
701
702         map<long, string> c;
703         char buf[10];
704
705         clock_t timeStart = clock();
706         for(long i=0; i< value; ++i)
707         {
708             try {
709                 sprintf(buf, 10, "%d", rand());
710                 c[i] = string(buf);
711             }
712             catch(exception& p) {
713                 cout << "i=" << i << " " << p.what() << endl;    //
714                 abort();
715             }
716         }
717         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
718         cout << "map.size()= " << c.size() << endl;
719         cout << "map.max_size()= " << c.max_size() << endl;
720
721         long target = get_a_target_long();
722         timeStart = clock();
723         auto pItem = c.find(target);
724         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
725         if (pItem != c.end())
726             cout << "found, value=" << (*pItem).second << endl;
727         else
728             cout << "not found! " << endl;
729     }
730 }
```

## 使用容器 `unordered_set`

```
739 namespace jj15
740 {
741     void test_unordered_set(long& value)
742     {
743         cout << "\ntest_unordered_set().....\n";
744
745         unordered_set<string> c;
746         char buf[10];
747
748         clock_t timeStart = clock();
749         for(long i=0; i< value; ++i)
750         {
751             try {
752                 sprintf(buf, 10, "%d", rand());
753                 c.insert(string(buf));
754             }
755             catch(exception& p) {
756                 cout << "i=" << i << " " << p.what();
757                 abort();
758             }
759         }
760         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
761         cout << "unordered_set.size()= " << c.size() << endl;
762         cout << "unordered_set.max_size()= " << c.max_size() << endl;
763         cout << "unordered_set.bucket_count()= " << c.bucket_count() << endl;
764         cout << "unordered_set.load_factor()= " << c.load_factor() << endl;
765         cout << "unordered_set.max_load_factor()= " << c.max_load_factor() << endl;
766         cout << "unordered_set.max_bucket_count()= " << c.max_bucket_count() << endl;
767         for (unsigned i=0; i< 20; ++i) {
768             cout << "bucket #" << i << " has " << c.bucket_size(i) << " elements.\n";
769     }
```

Unordered Set/Multiset:



```
D:\handout\C++11-test\DevC++\VTest-STL\test-stl.exe
select: 15
how many elements: 1000000

test_unordered_set().....
milli-seconds : 2891
unordered_set.size()= 32768
unordered_set.max_size()= 357913941
unordered_set.bucket_count()= 62233
unordered_set.load_factor()= 0.526537
unordered_set.max_load_factor()= 1
unordered_set.max_bucket_count()= 357913941
bucket #0 has 1 elements.
bucket #1 has 1 elements.
bucket #2 has 1 elements.
bucket #3 has 0 elements.
bucket #4 has 3 elements.
bucket #5 has 0 elements.
bucket #6 has 0 elements.
bucket #7 has 1 elements.
bucket #8 has 0 elements.
bucket #9 has 0 elements.
bucket #10 has 0 elements.
bucket #11 has 1 elements.
bucket #12 has 0 elements.
bucket #13 has 2 elements.
bucket #14 has 1 elements.
bucket #15 has 1 elements.
bucket #16 has 1 elements.
bucket #17 has 1 elements.
bucket #18 has 0 elements.
bucket #19 has 0 elements.
target <0^32767>: 23456
::find(), milli-seconds : 0
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

## 使用容器 unordered\_set

```
770 string target = get_a_target_string();
771 {
772     timeStart = clock();
773     auto pItem = ::find(c.begin(), c.end(), target); //比 c.find(...) 慢很多
774     cout << "::find(), milli-seconds : " << (clock() - timeStart) << endl;
775     if (pItem != c.end())
776         cout << "found, " << *pItem << endl;
777     else
778         cout << "not found! " << endl;
779 }
780
781 {
782     timeStart = clock();
783     auto pItem = c.find(target); //比 ::find(...) 快很多
784     cout << "c.find(), milli-seconds : " << (clock() - timeStart) << endl;
785     if (pItem != c.end())
786         cout << "found, " << *pItem << endl;
787     else
788         cout << "not found! " << endl;
789 }
790
791 }
```

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 15
how many elements: 1000000

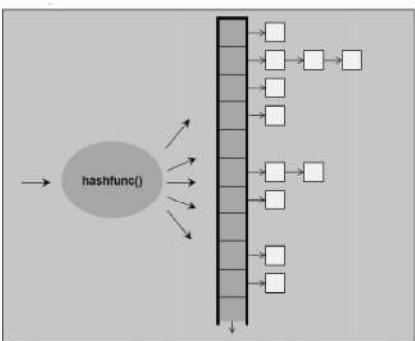
test_unordered_set().....
milli-seconds : 2891
unordered_set.size()= 32768
unordered_set.max_size()= 357913941
unordered_set.bucket_count()= 62233
unordered_set.load_factor()= 0.526537
unordered_set.max_load_factor()= 1
unordered_set.max_bucket_count()= 357913941
bucket #0 has 1 elements.
bucket #1 has 1 elements.
bucket #2 has 1 elements.
bucket #3 has 0 elements.
bucket #4 has 3 elements.
bucket #5 has 0 elements.
bucket #6 has 0 elements.
bucket #7 has 1 elements.
bucket #8 has 0 elements.
bucket #9 has 0 elements.
bucket #10 has 0 elements.
bucket #11 has 1 elements.
bucket #12 has 0 elements.
bucket #13 has 2 elements.
bucket #14 has 1 elements.
bucket #15 has 1 elements.
bucket #16 has 1 elements.
bucket #17 has 1 elements.
bucket #18 has 0 elements.
bucket #19 has 0 elements.
target <0x32767>: 23456
::find(), milli-seconds : 0
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

— 侯捷 —

38



## 使用容器 unordered\_map

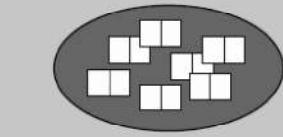


```
D:\handout\c++11-test\DevC++\Test-STL\test-stl.c
select: 16
how many elements: 1000000

test_unordered_map().....
milli-seconds : 3797
unordered_map.size()= 1000000
unordered_map.max_size()= 357913941
target <0~32767>: 23456
c.find(), milli-seconds : 0
found, value=12218
```

```
801 namespace jj16
802 {
803     void test_unordered_map(long& value)
804     {
805         cout << "\ntest_unordered_map()..... \n";
806
807         unordered_map<long, string> c;
808         char buf[10];
809
810         clock_t timeStart = clock();
811         for(long i=0; i< value; ++i)
812         {
813             try {
814                 sprintf(buf, 10, "%d", rand());
815                 c[i] = string(buf);
816             }
817             catch(exception& p) {
818                 cout << "i=" << i << " " << p.what() << endl;
819                 abort();
820             }
821         }
822         cout << "milli-seconds : " << (clock()-timeStart) << endl;
823         cout << "unordered_map.size()= " << c.size() << endl;
824         cout << "unordered_map.max_size()= " << c.max_size() << endl;
825
826
827         long target = get_a_target_long();
828         timeStart = clock();
829         //auto pItem = find(c.begin(), c.end(), target); //map 不適用 ::find()
830         auto pItem = c.find(target);
831
832         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
833         if (pItem != c.end())
834             cout << "found, value=" << (*pItem).second << endl;
835         else
836             cout << "not found! " << endl;
837     }
838 }
```

Unordered Map/Multimap:





使用容器 `hash_set`

`hash_map`

`hash_multiset`

`hash_multimap`

旧容器





## 使用分配器 allocator

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp>>
class vector : protected _Vector_base<_Tp, _Alloc>
```

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp>>
class list : protected _List_base<_Tp, _Alloc>
```

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp>>
class deque : protected _Deque_base<_Tp, _Alloc>
```

```
template<typename _Key, typename _Compare = std::less<_Key>,
         typename _Alloc = std::allocator<_Key>>
class set
```

```
template <typename _Key, typename _Tp, typename _Compare = std::less<_Key>,
          typename _Alloc = std::allocator<std::pair<const _Key, _Tp>>>
class map
```

```
template<class _Value,
         class _Hash = hash<_Value>,
         class _Pred = std::equal_to<_Value>,
         class _Alloc = std::allocator<_Value>>
class unordered_set
```

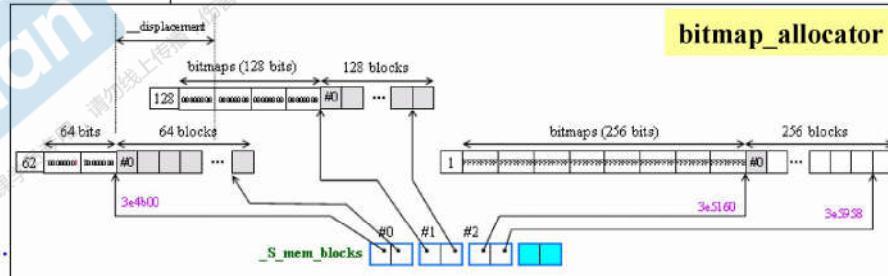
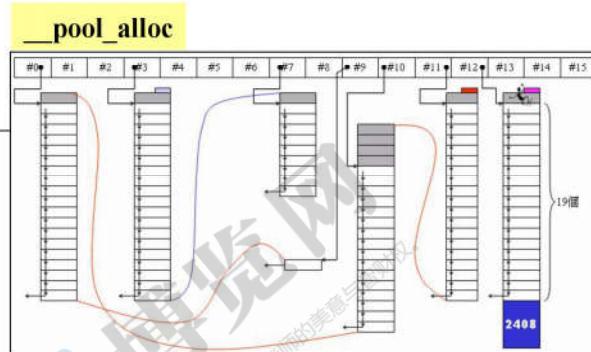
```
template<class _Key, class _Tp,
         class _Hash = hash<_Key>,
         class _Pred = std::equal_to<_Key>,
         class _Alloc = std::allocator<std::pair<const _Key, _Tp>>>
class unordered_map
```

## 使用分配器 allocator

```

916 #include <list>
917 #include <stdexcept>
918 #include <string>
919 #include <cstdlib>           //abort()
920 #include <cstdio>             //snprintf()
921 #include <algorithm>          //find()
922 #include <iostream>
923 #include <ctime>
924
925 #include <cstddef>
926 #include <memory>            //內含 std::allocator
927 //欲使用 std::allocator 以外的 allocator, 得自行 #include <ext>...
928 #include <ext\array_allocator.h>
929 #include <ext\mt_allocator.h>
930 #include <ext\debug_allocator.h>
931 #include <ext\pool_allocator.h>
932 #include <ext\bitmap_allocator.h>
933 #include <ext\malloc_allocator.h>
934 #include <ext\new_allocator.h>
935 namespace jj20
936 {
937     void test_list_with_special_allocator()
938     {
939         cout << "\ntest_list_with_special_allocator()....."
940
941         list<string, allocator<string>> c1;
942         list<string, __gnu_cxx::__malloc_allocator<string>> c2;
943         list<string, __gnu_cxx::__new_allocator<string>> c3;
944         list<string, __gnu_cxx::__pool_alloc<string>> c4;
945         list<string, __gnu_cxx::__mt_alloc<string>> c5;
946         list<string, __gnu_cxx::__bitmap_allocator<string>> c6;

```



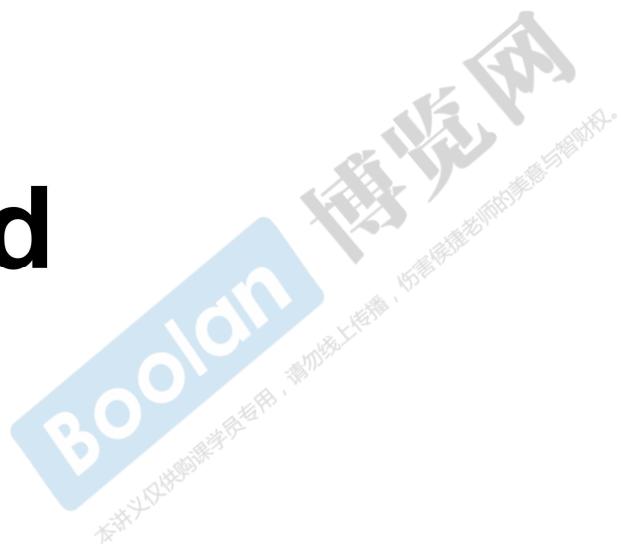
## 使用分配器 allocator

```
948 int choice;
949 long value;
950
951     cout << "select: ";
952     cin >> choice;
953     if ( choice != 0 ) {
954         cout << "how many elements: ";
955         cin >> value;
956     }
957
958     char buf[10];
959     clock_t timeStart = clock();
960     for(long i=0; i< value; ++i)
961     {
962         try {
963             snprintf(buf, 10, "%d", i);
964             switch (choice)
965             {
966                 case 1 : c1.push_back(string(buf));
967                             break;
968                 case 2 : c2.push_back(string(buf));
969                             break;
970                 case 3 : c3.push_back(string(buf));
971                             break;
972                 case 4 : c4.push_back(string(buf));
973                             break;
974                 case 5 : c5.push_back(string(buf));
975                             break;
976                 case 6 : c6.push_back(string(buf));
977                             break;
978                 default:
979                     break;
980             }
981         } catch(exception& p) {
982             cout << "i=" << i << " " << p.what() << endl;
983             abort();
984         }
985     }
986
987     cout << "a lot of push_back(), milli-seconds : "
988         << (clock()-timeStart) << endl;
```

实际上没有自己调用分配器的需要

```
989 //test all allocators' allocate() & deallocate();
990 int* p;
991 allocator<int> alloc1;
992 p = alloc1.allocate(1);
993 alloc1.deallocate(p,1);
994
995 __gnu_cxx::malloc_allocator<int> alloc2;
996 p = alloc2.allocate(1);
997 alloc2.deallocate(p,1);
998
999 __gnu_cxx::new_allocator<int> alloc3;
1000 p = alloc3.allocate(1);
1001 alloc3.deallocate(p,1);
1002
1003 __gnu_cxx::__pool_alloc<int> alloc4;
1004 p = alloc4.allocate(2);
1005 alloc4.deallocate(p,2);
1006
1007 __gnu_cxx::__mt_alloc<int> alloc5;
1008 p = alloc5.allocate(1);
1009 alloc5.deallocate(p,1);
1010
1011 __gnu_cxx::bitmap_allocator<int> alloc6;
1012 p = alloc6.allocate(3);
1013 alloc6.deallocate(p,3); |
```

# The End



— 侯捷 —

44

# C++ 標準庫

## 體系結構與內核分析

(C++ Standard Library — architecture & sources)

第二講



侯捷

— 侯捷 —

45

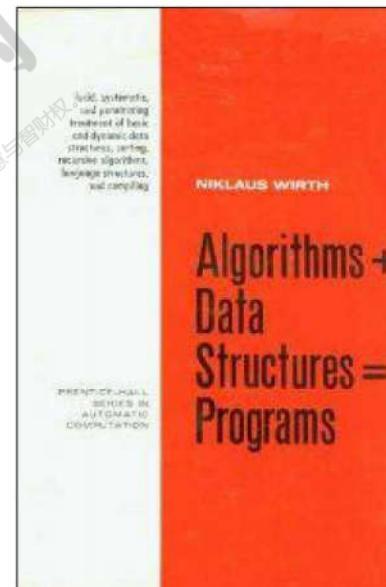
源碼之前  
了無秘密



## ■■■■ 你應具備的基礎

- C++ 基本語法
- 模板 (Templates) 基礎  
令你事半功倍
- 數據結構 (Data Structures) 和算法 (Algorithms) 概念，  
令你如魚得水

1976 book  
written by  
**Niklaus Wirth**



## ■■■■ 標準庫版本, Visual C++

(Visual C++ 6.0 畫面)

The screenshot shows the Microsoft Visual Studio 6.0 interface. The title bar reads "BoostSmartPtr - Microsoft Visual C++ - [E:\...\BoostSmartPtrTest.cpp]". The menu bar includes File, Edit, View, Insert, Project, Build, Tools, Window, Help. The toolbar has icons for New, Open, Save, Print, and others. The status bar at the bottom shows "Ready", "Ln 1, Col 1", "REC COL OVR READ", and "快速".

**Code Editor Content:**

```
// <Beyond the C++ Standard Library, An Introduction to Boost>, Chap1
#include <iostream>
#include <string>
#include "boost/scoped_ptr.hpp"
#include "boost\shared_ptr.hpp"

using namespace std;
using namespace boost;

//p12-
class pimpl_sample {
private:
    struct impl {
        void func() { cout << s << endl; }
        string s;
    };
    impl* pimpl;
public:
}
```

**Project Explorer:**

- Workspace 'BoostSmartPtr':
- BoostSmartPtr files
- Source Files
  - BoostSmartPtrTest.cpp
- Header Files
- Resource Files

**Properties Window:**

Configuration: BoostSmartPtr - Win32 Debug

BoostSmartPtr.exe - 0 error(s), 0 warning(s)

**Build Output Window:**

Build > Debug > Find in Files 1 > Find > | |

## 標準庫版本, Visual C++



— 侯捷 —

49

## 標準庫版本, GNU C++

(Dev-C++ 5.11 畫面; with GNU 4.9.2)

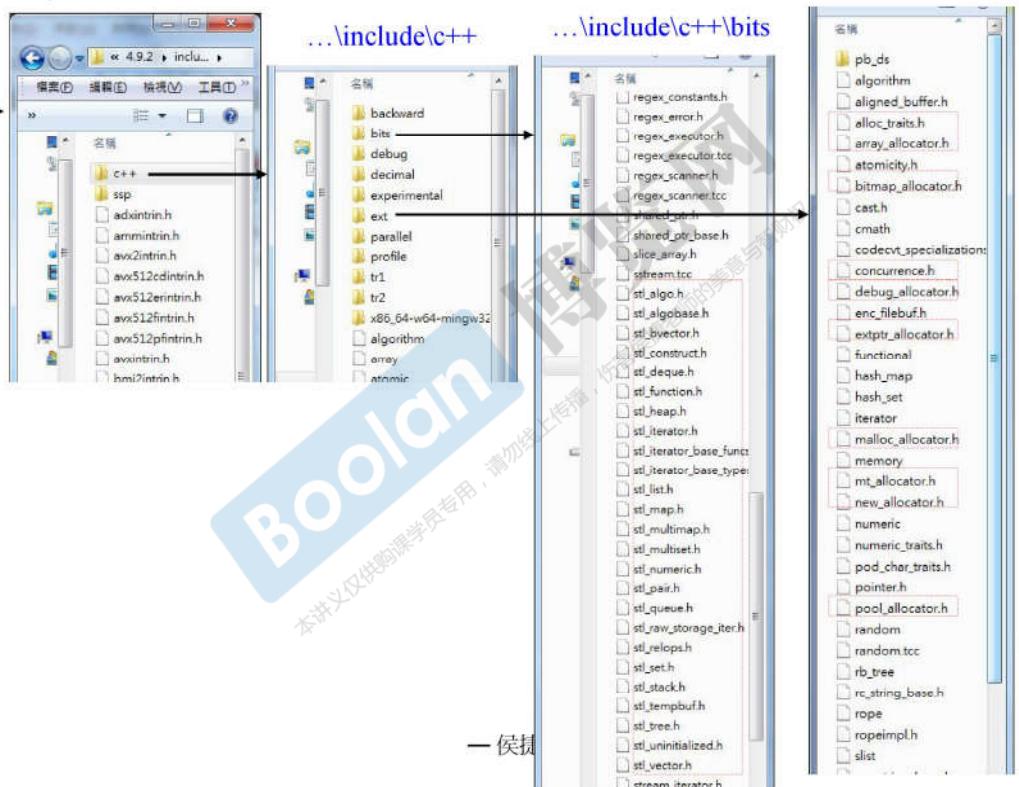
The screenshot shows the Dev-C++ 5.11 IDE interface. The title bar reads "C++11 - [C++11.dev] - Dev-C++ 5.11". The menu bar includes File, Edit, Search, View, Project, Execute, Tools, AS-Style, Window, Help. The toolbar has various icons for file operations. The main window displays the code editor with "TestC++11.cpp" selected. The code demonstrates different allocator types from the `<gnu_cxx>` namespace, including `malloc_allocator`, `pool_allocator`, `mt_allocator`, `bitmap_allocator`, `array_allocator`, `debug_allocator`, and `throw_allocator`. It also shows the use of `list`, `deque`, and `vector` containers with these allocators. The code ends with `cookie_test` calls. The status bar at the bottom shows "Line: 2161 Col: 1 Sel: 0 Lines: 2989 Length: 105275 Insert Done parsing 511 files in 1.281 seconds (398.91 files per second)".



## 標準庫版本, GNU C++

files layout

...\\4.9.2\\include →



## ■■■ OOP (Object-Oriented programming) vs. GP (Generic Programming)

OOP 企圖將 **datas** 和 **methods** 關聯在一起

```
template <class T,  
         class Alloc = alloc>  
class list {  
...  
    void sort();  
};
```

為什麼 list 不能使用 ::sort() 排序？

::sort(c.begin(), c.end());

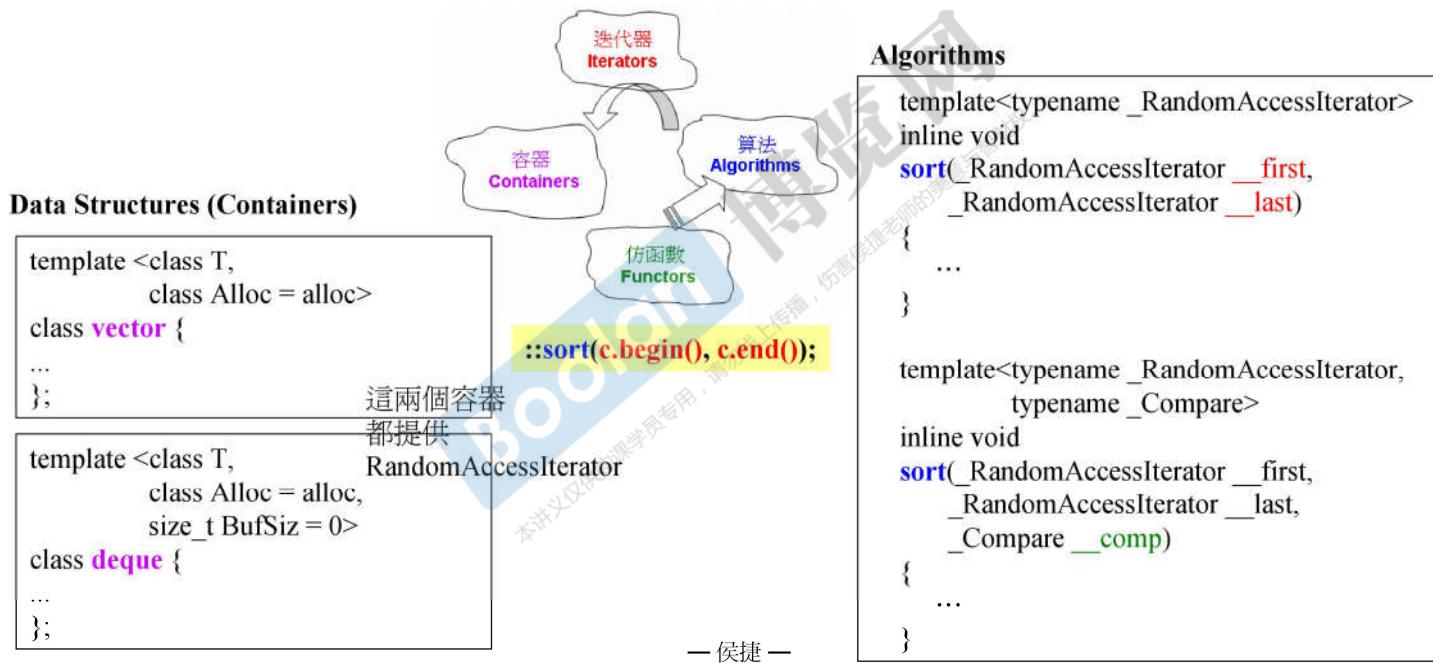
```
template <class RandomAccessIterator>  
inline void sort(RandomAccessIterator first, RandomAccessIterator last) {  
    if (first != last) {  
        __introsort_loop(first, last, value_type(first), __lg(last - first) * 2);  
        __final_insertion_sort(first, last);  
    }  
}
```

```
template <class RandomAccessIterator, class T, class Size>  
void __introsort_loop(RandomAccessIterator first,  
                      RandomAccessIterator last,  
                      T*,  
                      Size depth_limit) {  
...  
    RandomAccessIterator cut = __unguarded_partition  
        (first, last, T(__median(*first, *(first + (last - first)/2), *(last - 1))));  
...  
}
```

只有 RandomAccessIterator  
才能如此操作

## ■■■ OOP (Object-Oriented programming) vs. GP (Generic Programming)

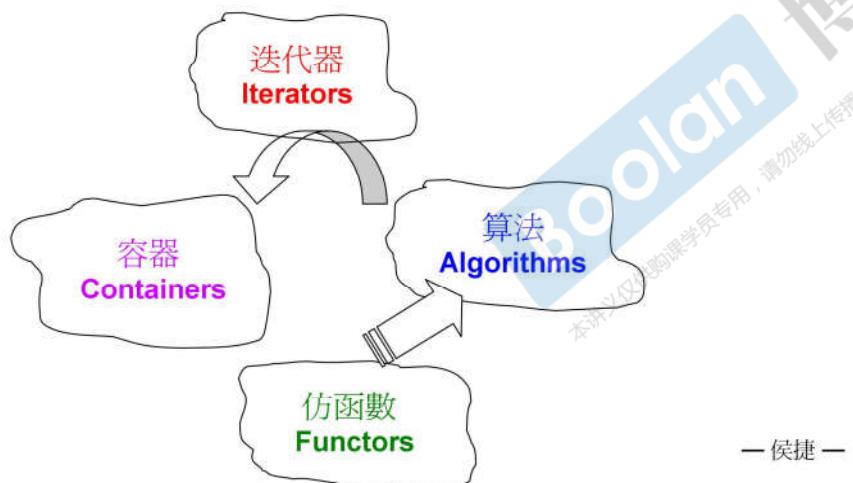
### GP 却是將 **datas** 和 **methods** 分開來



## ■■■■ OOP (Object-Oriented programming) vs. GP (Generic Programming)

採用 GP :

- Containers 和 Algorithms 團隊可各自閉門造車，其間以 Iterator 溝通即可。
- Algorithms 通過 Iterators 確定操作範圍，並通過 Iterators 取用 Container 元素。



— 侯捷 —

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

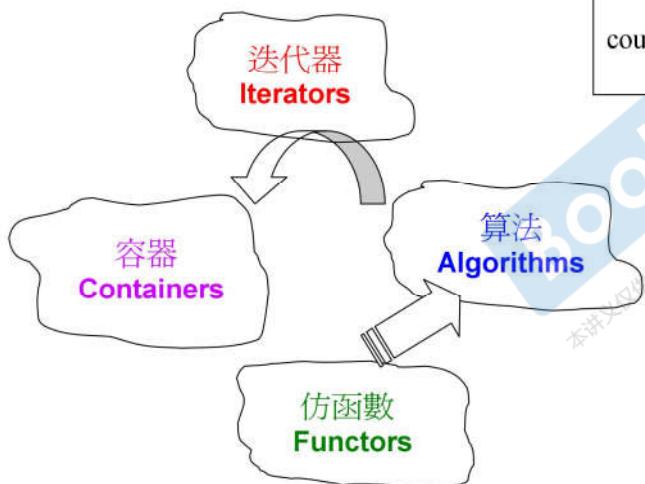
```
template <class T>
inline const T& max(const T& a, const T& b) {
    return a < b ? b : a;
}
```

```
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;
}
```

## ■■■ OOP (Object-Oriented programming) vs. GP (Generic Programming)

所有 algorithms，其內最終涉及元素本身的操作，無非就是比大小。



元素本身是否重载< 或提供comp仿函数对象

```
bool  
strLonger(const string& s1,  
          const string& s2)  
{ return s1.size() < s2.size(); }  
  
cout << "max of zoo and hello : "  
     << max(string("zoo"), string("hello")) << endl; //zoo  
  
cout << "longest of zoo and hello : "  
     << max(string("zoo"), string("hello"), strLonger) << endl; //hello
```

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

— 侯捷 —

55

## ■■■ 閱讀 C++ 標準庫 源碼 (source code) 之必要基礎

- **Operator Overloading** 操作符重載
- **Templates** 模板



## ■■■■ Operator Overloading, 操作符重載

<http://en.cppreference.com/w/cpp/language/operators>

### Overloaded operators

When an operator appears in an **expression**, and at least one of its operands has a **class type** or an **enumeration type**, then **overload resolution** is used to determine the user-defined function to be called among all the functions whose signatures match the following:

Expression	As member function	As non-member function	Example
@a	(a).operator@()	operator@(a)	<code>!std::cin calls std::cin.operator!()</code>
a@b	(a).operator@(b)	operator@(a, b)	<code>std::cout &lt;&lt; 42 calls std::cout.operator&lt;&lt;(42)</code>
a=b	(a).operator=(b)	cannot be non-member	<code>std::string s; s = "abc"; calls s.operator=( "abc" )</code>
a[b]	(a).operator[](b)	cannot be non-member	<code>std::map&lt;int, int&gt; m; m[1] = 2; calls m.operator[](1)</code>
a->	(a).operator->()	cannot be non-member	<code>std::unique_ptr&lt;S&gt; ptr(new S); ptr-&gt;bar() calls ptr.operator-&gt;()</code>
a@	(a).operator@(0)	operator@(a, 0)	<code>std::vector&lt;int&gt;::iterator i = v.begin(); i++ calls i.operator++(0)</code>

In this table, @ is a placeholder representing all matching operators: all prefix operators in @a, all postfix operators other than -> in a@, all infix operators other than = in a@b

## ■■■■ Operator Overloading, 操作符重載

<http://en.cppreference.com/w/cpp/language/operators>

Overloaded operators (but not the built-in operators) can be called using function notation:

```
std::string str = "Hello, ";
str.operator+=( "world"); // same as str += "world";
operator<<(operator<<(std::cout, str), '\n'); // same as std::cout << str << '\n';
```

### Restrictions

- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), and `?:` (ternary conditional) cannot be overloaded.
- New operators such as `**`, `<>`, or `&|` cannot be created.
- The overloads of operators `&&`, `||`, and `,` (comma) lose their special properties: short-circuit evaluation and `sequencing`.
- The overload of operator `->` must either return a raw pointer or return an object (by reference or by value), for which operator `->` is in turn overloaded.
- It is not possible to change the precedence, grouping, or number of operands of operators.

## Operator Overloading, 操作符重载

```
template<class T, class Ref, class Ptr>
struct __list_iterator {
    typedef __list_iterator<T, Ref, Ptr> self;
    typedef bidirectional_iterator_tag iterator_category; // (1)
    typedef T value_type; // (2)
    typedef Ptr pointer; // (3)
    typedef Ref reference; // (4)
    typedef __list_node<T>* link_type;
    typedef ptrdiff_t difference_type; // (5)

    link_type node;

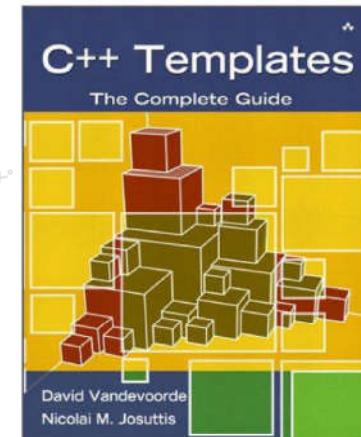
    reference operator*() const { return (*node).data; }
    pointer operator->() const { return &(operator*()); }
    self& operator++()
    self operator++(int)
    ...
},
```

## Class Templates, 類模板

```
template <typename T>
class complex
{
public:
    complex (T r = 0, T i = 0)
        : re (r), im (i)
    { }
    complex& operator += (const complex&);
    T real () const { return re; }
    T imag () const { return im; }
private:
    T re, im;

    friend complex& __doapl (complex*, const complex&);
};
```

```
{
    complex<double> c1(2.5,1.5);
    complex<int> c2(2,6);
    ...
}
```



## Function Templates, 函數模板

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

編譯器對 function template 進行  
實參推導 (argument deduction)

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

```
class stone  
{  
public:  
    stone(int w, int h, int we)  
        : _w(w), _h(h), _weight(we)  
    { }  
    bool operator<(const stone& rhs) const  
    { return _weight < rhs._weight; }  
private:  
    int _w, _h, _weight;  
};
```

實參推導的結果，T 為 stone，  
於是調用 stone::operator<()

## ■■■ Member Templates, 成員模板

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;
    pair() : first(T1()), second(T2()) {}
    pair(const T1& a, const T2& b) : first(a), second(b) {}

#define __STL_MEMBER_TEMPLATES
    template <class U1, class U2>
    pair(const pair<U1, U2>& p) : first(p.first), second(p.second) {}
#endif
};
```

有一个泛化的模板（普适的），但是现在针对某一种类型有更好的方法，可以使用特化，写出一种专属的方法

## Specialization, 特化

泛化

```
template <class type>
struct __type_traits {
    typedef __true_type   this_dummy_member_must_be_first;
    typedef __false_type  has_trivial_default_constructor;
    typedef __false_type  has_trivial_copy_constructor;
    typedef __false_type  has_trivial_assignment_operator;
    typedef __false_type  has_trivial_destructor;
    typedef __false_type  is_POD_type;
};
```

ref. G2.91 <type\_traits.h>

Plain Old Data

特化

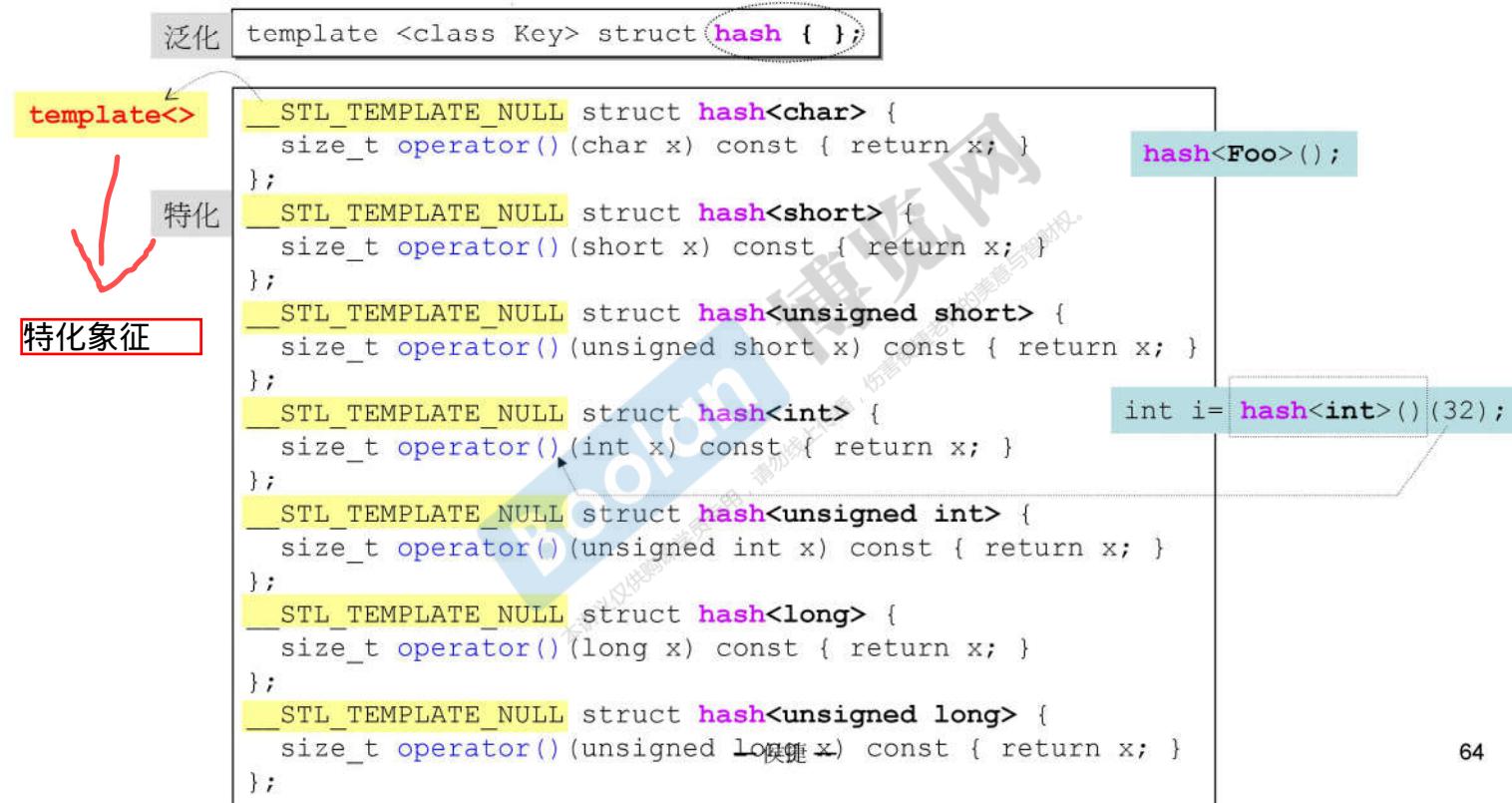
```
template<> struct __type_traits<int> {
    typedef __true_type   has_trivial_default_constructor;
    typedef __true_type   has_trivial_copy_constructor;
    typedef __true_type   has_trivial_assignment_operator;
    typedef __true_type   has_trivial_destructor;
    typedef __true_type   is_POD_type;
};
```

\_\_type\_traits<Foo>::has\_trivial\_destructor

特化

```
template<> struct __type_traits<double> {
    typedef __true_type   has_trivial_default_constructor;
    typedef __true_type   has_trivial_copy_constructor;
    typedef __true_type   has_trivial_assignment_operator;
    typedef __true_type   has_trivial_destructor;
    typedef __true_type   is_POD_type;
};
```

## Specialization, 特化



## ■■■■ Specialization, 特化

```
泛化 template<typename _Tp>
    class allocator;

    /// allocator<void> specialization.
    template<>
        class allocator<void>
    {
    public:
        typedef size_t      size_type;
        typedef ptrdiff_t   difference_type;
        typedef void*       pointer;
        typedef const void* const_pointer;
        typedef void        value_type;

        template<typename _Tp1>
            struct rebind
            { typedef allocator<_Tp1> other; };
    };
}
```

多个模板参数，不完全特化里面的参数，比如只绑定其中的一个

## ■■■■ Partial Specialization, 偏特化

ref. G2.9 <stl\_iterator.h>

泛化

```
template <class T,
          class Alloc = alloc>
class vector
{
    ...
};
```

偏特化

```
template<class Alloc>
class vector<bool, Alloc>
{
    ...
};
```

个数的偏特化

```
template <class Iterator>
struct iterator_traits { 泛化
    typedef typename Iterator::iterator_category iterator_category;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
};

//partial specialization for regular pointers
template <class T>
struct iterator_traits<T*> { 偏特化
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};

//partial specialization for regular const pointers
template <class T>
struct iterator_traits<const T*> { 偏特化
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef const T* pointer;
    typedef const T& reference;
};
```

范围的偏特化

## 分配器 allocators

先談 operator new() 和 malloc()

...\\vc98\\crt\\src\\newop2.cpp

```
void *operator new(size_t size, const std::nothrow_t&)
    _THROW0()
{
    // try to allocate size bytes
    void *p;
    while ((p = malloc(size)) == 0)
        { // buy more memory or return null pointer
        _TRY_BEGIN
            if (_callnewh(size) == 0) break;
            _CATCH(std::bad_alloc) return (0);
            _CATCH_END
        }
    return (p);
}
```

<new.h> of CB5

```
// inline versions of the nothrow_t versions of new & delete operators
inline void *_RTLENTRY operator new (size_t size, const std::nothrow_t &)
{
    size = size ? size : 1;
    return malloc(size);
}
```

malloc给你的，比你想要的空间多不少，存在一些额外的信息。所以对于小块内存分配，额外信息的比例就大

00000041
00790b30
00000079
0000000c
00000002
00000007
4 個 0xfd
fill 0xcd (ch)
4 個 0xfd
00000000 (pad)
00000000 (pad)
00000041

cookie记录整块的大小

## ■■■■ 分配器 allocators

VC6 STL 對 allocator 的使用

```
template<class _Ty, class _A = allocator<_Ty> >
class vector
{ ...
};
```

```
template<class _Ty, class _A = allocator<_Ty> >
class list
{ ...
};
```

```
template<class _Ty, class _A = allocator<_Ty> >
class deque
{ ...
};
```

```
template<class _K, class _Pr = less<_K>,
         class _A = allocator<_K> >
class set {
...
};
```

## ■■■ 分配器 allocators

VC6 所附的標準庫，其 `allocator` 實現如下 (`<xmemory>`)

```
template<class _Ty>
class allocator {
public:
    typedef _SIZT size_type;
    typedef _PDFT difference_type;
    typedef _Ty _FARQ *pointer;
    typedef _Ty value_type;
    pointer allocate(size_type _N, const void * );
    { return (_Allocate((difference_type)_N, (pointer)0)); }
    void deallocate(void _FARQ * _P, size_type)
    { operator delete(_P); }
};
```

```
#ifndef _FARQ
#define _FARQ
#define _PDFT
#define _SIZT
#endif
#define _POINTER_X(T, A) T _FARQ *
#define _REFERENCE_X(T, A) T _FARQ &
```

VC6+ 的 allocator 只是以 `::operator new` 和 `::operator delete` 完成 `allocate()` 和 `deallocate()`，沒有任何特殊設計。

其中用到的 `_Allocate()` 定義如下：

```
template<class _Ty> inline
_Ty _FARQ * _Allocate(_PDFT _N, _Ty _FARQ *)
{if (_N < 0) _N = 0;
 return ((_Ty _FARQ *)operator new((_SIZT)_N * sizeof (_Ty))); }
```

小技巧，不在乎传递什么  
临时对象  
//分配 512 ints.  
int\* p = **allocator<int>()**.allocate(512, (int\*)0);  
**allocator<int>()**.deallocate(p, 512);

## ■■■ 分配器 allocators

### BC5 STL 對 allocator 的使用

```
template <class T, class Allocator _RWSTD_COMPLEX_DEFAULT(allocator<T>) >
class vector ...

template <class T, class Allocator _RWSTD_COMPLEX_DEFAULT(allocator<T>) >
class list ...

template <class T, class Allocator _RWSTD_COMPLEX_DEFAULT(allocator<T>) >
class deque ...
```

↓  
# define \_RWSTD\_COMPLEX\_DEFAULT(a) = a <stdcomp.h>

```
template <class T, class Allocator = allocator<T> >
class vector ...

template <class T, class Allocator = allocator<T> >
class list ...

template <class T, class Allocator = allocator<T> >
class deque ...
```

## ■■■■ 分配器 allocators

BC5 所附的標準庫，其 `allocator` 實現如下 (`<memory.stl>`)

```
template <class T>
class allocator
{
public:
    typedef size_t           size_type;
    typedef ptrdiff_t        difference_type;
    typedef T*                pointer;
    typedef T                value_type;

    pointer allocate(size_type n, allocator<void>::const_pointer = 0) {
        pointer tmp =
            _RWSTD_STATIC_CAST(pointer, (::operator new
                (_RWSTD_STATIC_CAST(size_t, (n * sizeof(value_type))))));
        _RWSTD_THROW_NO_MSG(tmp == 0, bad_alloc);
        return tmp;
    }
    void deallocate(pointer p, size_type) {
        ::operator delete(p);
    } <new.h> of CB5
    ...
};
```

BC++ 的 allocator 只是以 `::operator new` 和 `::operator delete` 完成 `allocate()` 和 `deallocate()`，沒有任何特殊設計。

//分配 512 ints.  
int\* p = allocator<int>().allocate(512);  
allocator<int>().deallocate(p,512);

## ■■■■ 分配器 allocators

G2.9 所附的標準庫，其 allocator 實現如下 (<defalloc.h>)

GCC2.9 的 allocator 只是以 ::operator new 和 ::operator delete 完成 allocate() 和 deallocate()，沒有任何特殊設計。

```
template <class T>
class allocator {
public:
    typedef T      value_type;
    typedef T*     pointer;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    pointer allocate(size_type n) {
        return ::allocate((difference_type)n, (pointer)0);
    }
    void deallocate(pointer p) { ::deallocate(p); }
};
```

```
template <class T>
inline T* allocate(ptrdiff_t size, T*) {
    set_new_handler(0);
    T* tmp = (T*)
        ::operator new((size_t)(size*sizeof(T)));
    if (tmp == 0) {
        cerr << "out of memory" << endl;
        exit(1);
    }
    return tmp;
}
template <class T>
inline void deallocate(T* buffer)
    ::operator delete(buffer);
}
```

G++ <defalloc.h> 中有這樣的註釋：DO NOT USE THIS FILE unless you have an old container implementation that requires an allocator with the HP-style interface. SGI STL uses a different allocator interface. SGI-style allocators are not parametrized with respect to the object type; they traffic in void\* pointers. This file is not included by any other SGI STL header.

VC BC GC的allocator最終都是使用malloc分配，都需要內存額外開銷。通常区块是比较小的，所以額外開銷占比大

虽然提供了，但是G++裏面實現的容器從來不用這個分配器。

精巧的设计。alloc是另一种分配器了。

## ■■■■ 分配器 allocators G2.9 STL 對 allocator 的使用

```
template <class T, class Alloc = alloc>
class vector {
    ...
};
```

```
template <class T, class Alloc = alloc>
class list {
    ...
};
```

```
template <class T, class Alloc = alloc,
          size_t BufSiz = 0>
class deque {
    ...
};
```

```
//分配 512 bytes.
void* p = alloc::allocate(512); //也可alloc().allocate(512);
alloc::deallocate(p,512);
```

```
template <class Key,
          class T,
          class Compare = less<Key>,
          class Alloc = alloc>
class map {
    ...
};
```

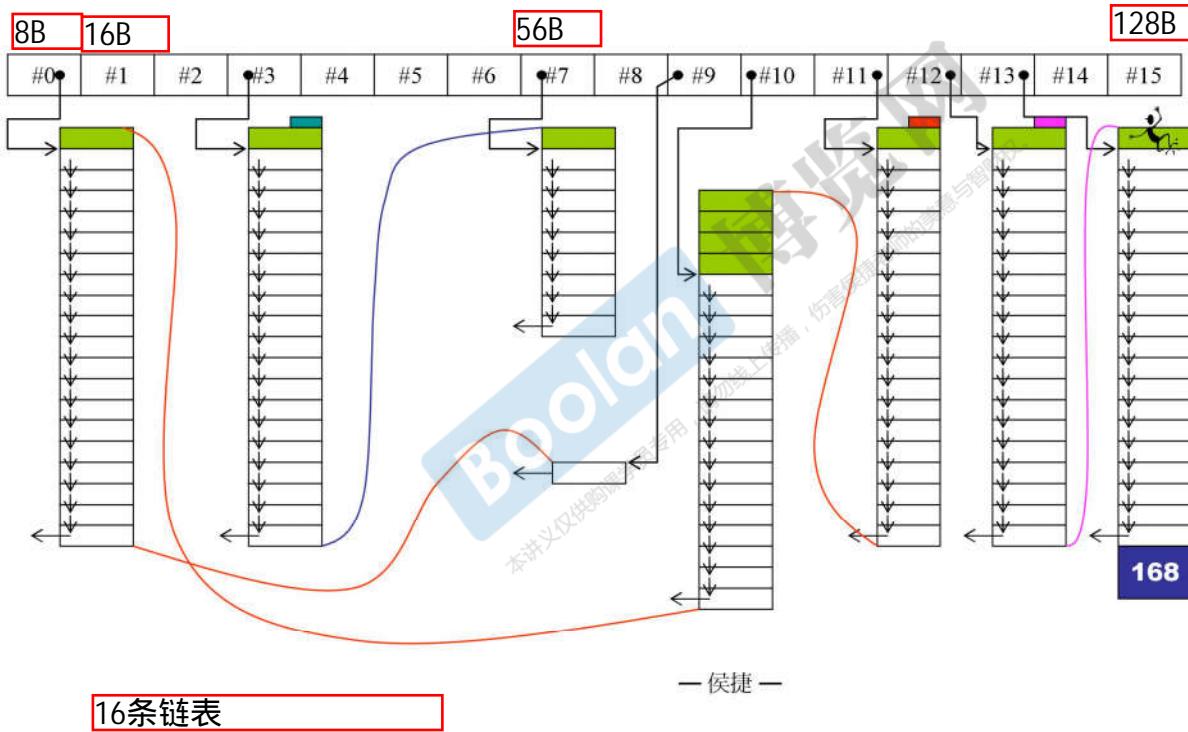
```
template <class Key,
          class Compare = less<Key>,
          class Alloc = alloc>
class set {
    ...
};
```

比如一个list容器，如果使用malloc那种方式的分配器，一次malloc一个节点，100w个的话，就需要100w份额外的空间。（vector应该还好，一次可以分配一大块）

## 分配器 allocators

所有分配内存的最后的动作一定是malloc，主要的诉求就是减少malloc的次数，因为malloc有额外开销（如记录内存块大小，使得free的时候不需要传大小），现在要减少就是这个额外的开销。（由于malloc是给分配格式各样的内存时使用的，分配的内存有大有小，所以需要cookie记录）  
容器的元素大小是一样的，所以没有必要每个元素都记着，所以在容器的应用情况下可以不需要cookie。g++的分配器就是从这个地方着手。  
粗略算下，使用这种分配器，对于100w个元素，可省 $4 \times 2 \times 100\text{wB}$

G2.9 所附的標準庫，其 `alloc` 實現如下 (`<stl_alloc.h>`)



## ■■■ 分配器 allocators

### G4.9 STL 對 allocator 的使用

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp>>
class vector : protected _Vector_base<_Tp, _Alloc>
{
    ...
};
```

## ■■■■ 分配器 allocators

G4.9 所附的標準庫，其 `allocator` 實現如下

4.9版本又換成了原來調用`malloc`的那種分配器  
原因不知道

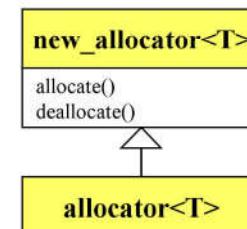
```
template<typename _Tp>                                <bits/allocator.h>
class allocator: public __allocator_base<_Tp>
{
    ...
};

# define __allocator_base __gnu_cxx::new_allocator
```

```
template<typename _Tp>                                <bits/new_allocator.h>
class new_allocator
{
    ...
    pointer allocate(size_type __n, const void* = 0) {
        if (__n > this->max_size())
            std::__throw_bad_alloc();
        return static_cast<_Tp*>
            (::operator new(__n * sizeof(_Tp)));
    }

    void deallocate(pointer __p, size_type)
    { ::operator delete(__p); }
    ...
};
```



## ■■■■ 分配器 allocators

G4.9 所附的標準庫，有許多 extention allocators，  
其中 `_pool_alloc` 就是 G2.9 的 `alloc`。

用例：

```
vector<string, __gnu_cxx::__pool_alloc<string>> vec;
```

```
template<typename _Tp>
class __pool_alloc : private __pool_alloc_base
{
    ...
};

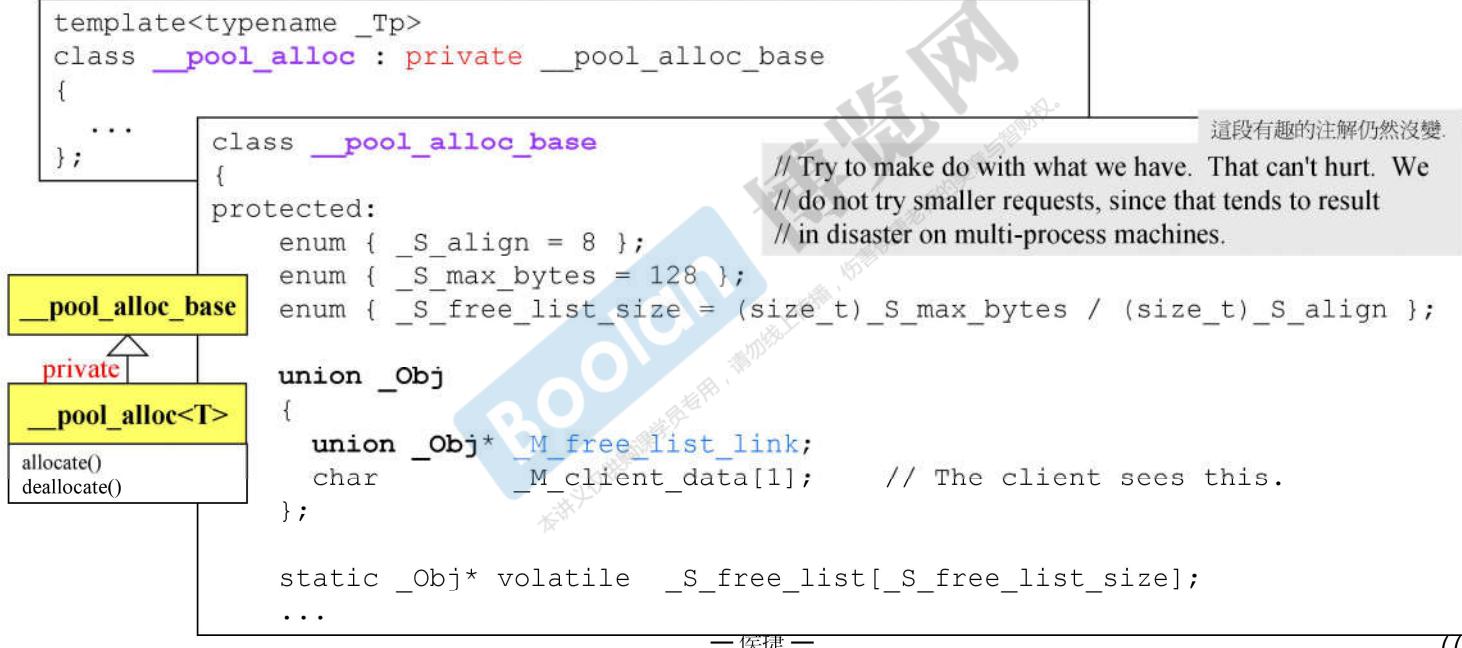
class __pool_alloc_base
{
protected:
    enum { __S_align = 8 };
    enum { __S_max_bytes = 128 };
    enum { __S_free_list_size = (size_t) __S_max_bytes / (size_t) __S_align };

    union __Obj
    {
        union __Obj* __M_free_list_link;
        char __M_client_data[1]; // The client sees this.
    };

    static __Obj* volatile __S_free_list[__S_free_list_size];
    ...
}
```

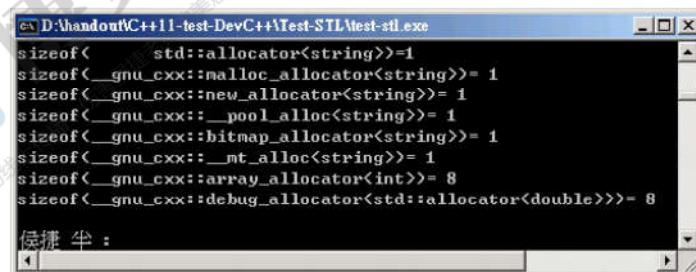
這段有趣的注解仍然沒變。  
// Try to make do with what we have. That can't hurt. We  
// do not try smaller requests, since that tends to result  
// in disaster on multi-process machines.

private  
`__pool_alloc<T>`  
allocate()  
deallocate()

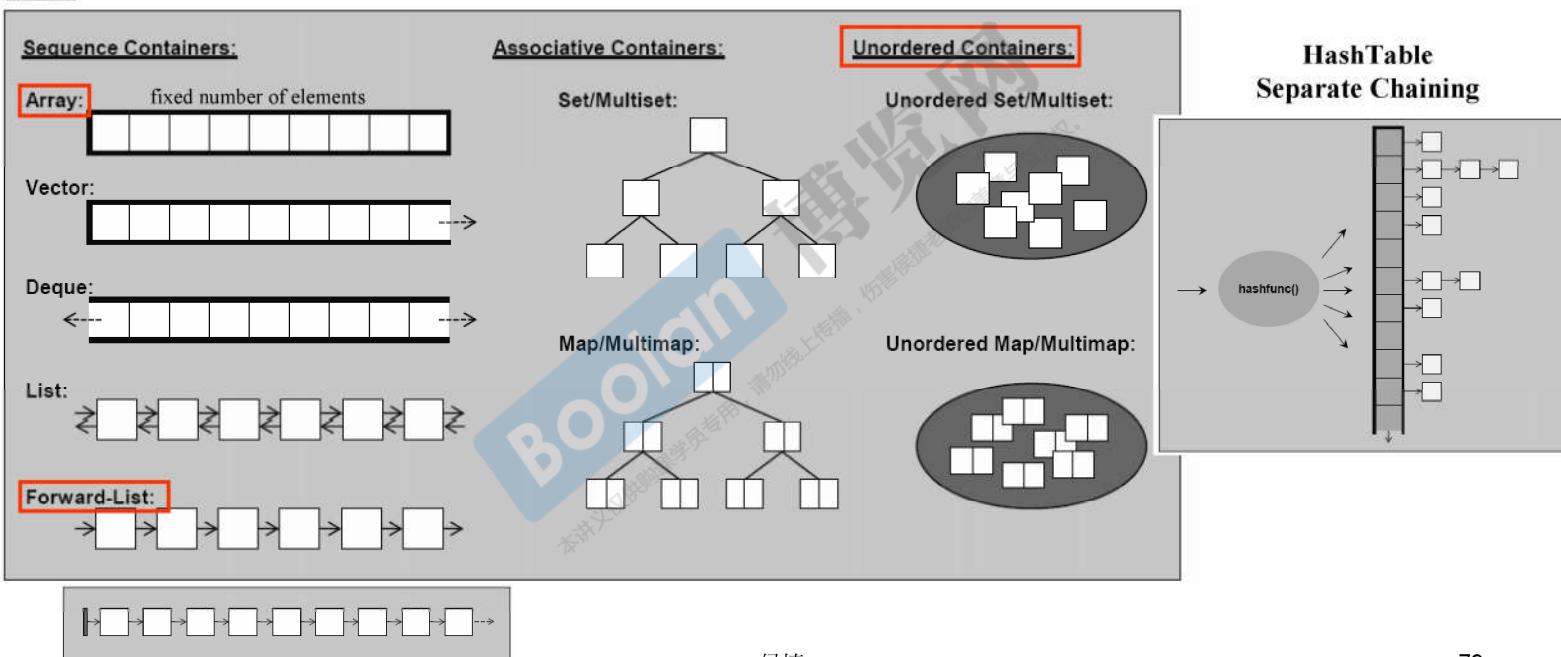


## ///// 迭代器, sizeof()

```
cout << "sizeof(      std::allocator<string>)=" << sizeof(std::allocator<string>) << endl;           //1 (理論值 0)
cout << "sizeof(_gnu_cxx::malloc_allocator<string>)=" << sizeof(_gnu_cxx::malloc_allocator<string>) << endl; //1 (理論值 0)
cout << "sizeof(_gnu_cxx::new_allocator<string>)=" << sizeof(_gnu_cxx::new_allocator<string>) << endl;        //1 (理論值 0)
cout << "sizeof(_gnu_cxx::_pool_alloc<string>)=" << sizeof(_gnu_cxx::_pool_alloc<string>) << endl;        //1 (理論值 0)
cout << "sizeof(_gnu_cxx::bitmap_allocator<string>)=" << sizeof(_gnu_cxx::bitmap_allocator<string>) << endl; //1 (理論值 0)
cout << "sizeof(_gnu_cxx::_mt_alloc<string>)=" << sizeof(_gnu_cxx::_mt_alloc<string>) << endl;          //1 (理論值 0)
cout << "sizeof(_gnu_cxx::array_allocator<int>)=" << sizeof(_gnu_cxx::array_allocator<int>) << endl;         //8
cout << "sizeof(_gnu_cxx::debug_allocator<std::allocator<double>>)"= << sizeof(_gnu_cxx::debug_allocator<std::allocator<double>>) << endl; //8
//=> 因為它有一個 ptr 指向 array 和一個 size_t 表示消耗到 array 哪兒
```



## 容器 – 結構與分類

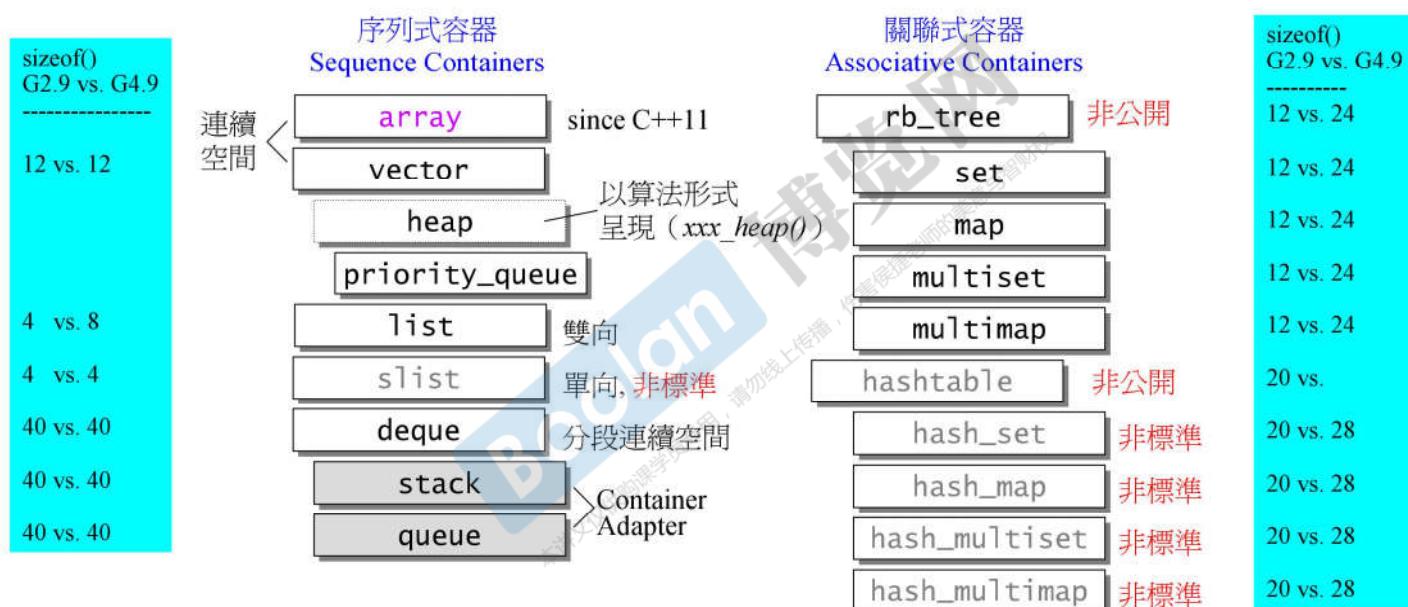


— 侯捷 —

## 容器, 結構與分類

32bit 电脑上

本圖以縮排形式表達“基層與衍生層”的關係。  
這裡所謂衍生，並非繼承（inheritance）而是複合（composition）。



在 C++11 中，slist 名為 `forward_list`，`hash_set`, `hash_map` 名為 `unordered_set`, `unordered_map`，  
`hash_multiset`, `hash_multimap` 名為 `unordered_multiset`, `unordered_multimap`；且新添 `array`。

## 容器和其迭代器, sizeof()

```
cout << "sizeof(array<int,100>)= " << sizeof(array<int,100>) << endl; //400
cout << "sizeof(vector<int>)= " << sizeof(vector<int>) << endl; //12
cout << "sizeof(list<int>)= " << sizeof(list<int>) << endl; //8
cout << "sizeof(forward_list<int>)= " << sizeof(forward_list<int>) << endl; //4
cout << "sizeof(deque<int>)= " << sizeof(deque<int>) << endl; //40
cout << "sizeof(stack<int>)= " << sizeof(stack<int>) << endl; //40
cout << "sizeof(queue<int>)= " << sizeof(queue<int>) << endl; //40
cout << "sizeof(set<int>)= " << sizeof(set<int>) << endl; //24
cout << "sizeof(map<int,int>)= " << sizeof(map<int,int>) << endl; //24
cout << "sizeof(multiset<int>)= " << sizeof(multiset<int>) << endl; //24
cout << "sizeof(multimap<int,int>)= " << sizeof(multimap<int,int>) << endl; //24
cout << "sizeof(unordered_set<int>)= " << sizeof(unordered_set<int>) << endl;
cout << "sizeof(unordered_map<int,int>)= " << sizeof(unordered_map<int,int>) << endl;
cout << "sizeof(unordered_multiset<int>)= " << sizeof(unordered_multiset<int>) << endl;
cout << "sizeof(unordered_multimap<int,int>)= " << sizeof(unordered_multimap<int,int>) << endl;

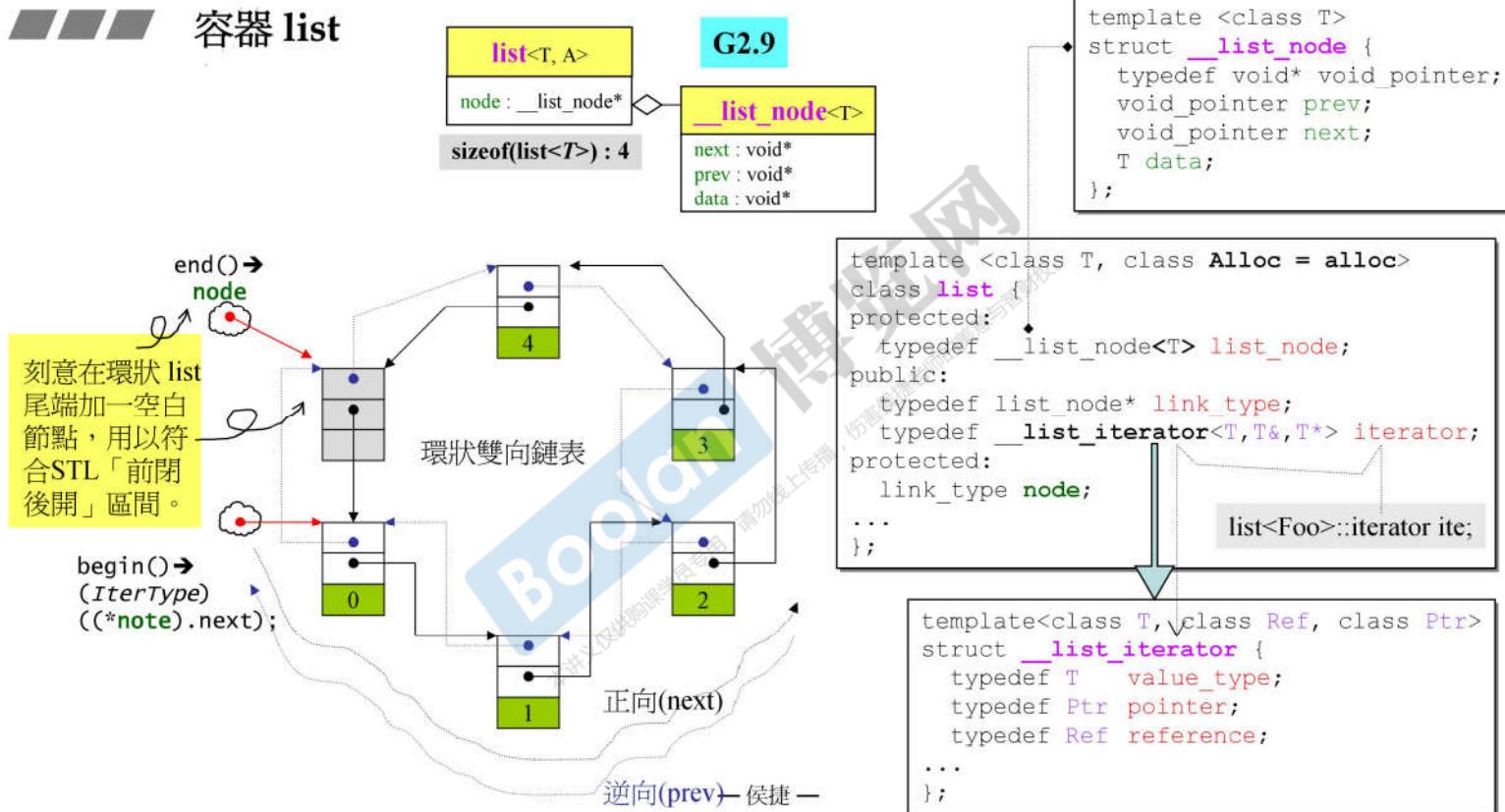
cout << "sizeof(array<int,100>::iterator)= " << sizeof(array<int,100>::iterator) << endl;
cout << "sizeof(vector<int>::iterator)= " << sizeof(vector<int>::iterator) << endl;
cout << "sizeof(list<int>::iterator)= " << sizeof(list<int>::iterator) << endl;
cout << "sizeof(forward_list<int>::iterator)= " << sizeof(forward_list<int>::iterator) << endl;
cout << "sizeof(deque<int>::iterator)= " << sizeof(deque<int>::iterator) << endl; //Er
// cout << "sizeof(stack<int>::iterator)= " << sizeof(stack<int>::iterator) << endl; //Er
// cout << "sizeof(queue<int>::iterator)= " << sizeof(queue<int>::iterator) << endl; //Er
cout << "sizeof(set<int>::iterator)= " << sizeof(set<int>::iterator) << endl;
cout << "sizeof(map<int,int>::iterator)= " << sizeof(map<int,int>::iterator) << endl;
cout << "sizeof(multiset<int>::iterator)= " << sizeof(multiset<int>::iterator) << endl;
cout << "sizeof(multimap<int,int>::iterator)= " << sizeof(multimap<int,int>::iterator) << endl; //4
cout << "sizeof(unordered_set<int>::iterator)= " << sizeof(unordered_set<int>::iterator) << endl; //4
cout << "sizeof(unordered_map<int,int>::iterator)= " << sizeof(unordered_map<int,int>::iterator) << endl; //4
cout << "sizeof(unordered_multiset<int>::iterator)= " << sizeof(unordered_multiset<int>::iterator) << endl; //4
cout << "sizeof(unordered_multimap<int,int>::iterator)= " << sizeof(unordered_multimap<int,int>::iterator) << endl; //4
```

D:\handout\c++11-test\DevC++\Test-STL\test-stl.exe

```
test_components_size().....
sizeof(array<int,100>)= 400
sizeof(vector<int>)= 12
sizeof(list<int>)= 8
sizeof(forward_list<int>)= 4
sizeof(deque<int>)= 40
sizeof(stack<int>)= 40
sizeof(queue<int>)= 40
sizeof(set<int>)= 24
sizeof(map<int,int>)= 24
sizeof(multiset<int>)= 24
sizeof(multimap<int,int>)= 24
sizeof(unordered_set<int>)= 28
sizeof(unordered_map<int,int>)= 28
sizeof(unordered_multiset<int>)= 28
sizeof(unordered_multimap<int,int>)= 28
sizeof(array<int,100>::iterator)= 4
sizeof(vector<int>::iterator)= 4
sizeof(list<int>::iterator)= 4
sizeof(forward_list<int>::iterator)= 4
sizeof(deque<int>::iterator)= 16
sizeof(stack<int>::iterator)= 4
sizeof(queue<int>::iterator)= 4
sizeof(set<int>::iterator)= 4
sizeof(map<int,int>::iterator)= 4
sizeof(multiset<int>::iterator)= 4
sizeof(multimap<int,int>::iterator)= 4
sizeof(unordered_set<int>::iterator)= 4
sizeof(unordered_map<int,int>::iterator)= 4
sizeof(unordered_multiset<int>::iterator)= 4
sizeof(unordered_multimap<int,int>::iterator)= 4
```

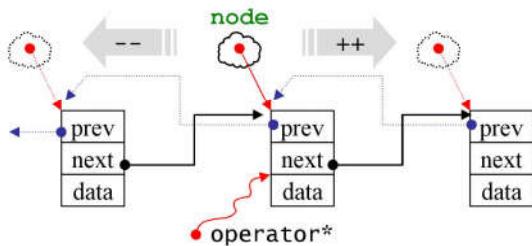
G4.9

## 容器 list



所有的容器，除了array和vector之外，他们都必须定义为一个class，重载++等运算符。（泛型指针）

## list's iterator



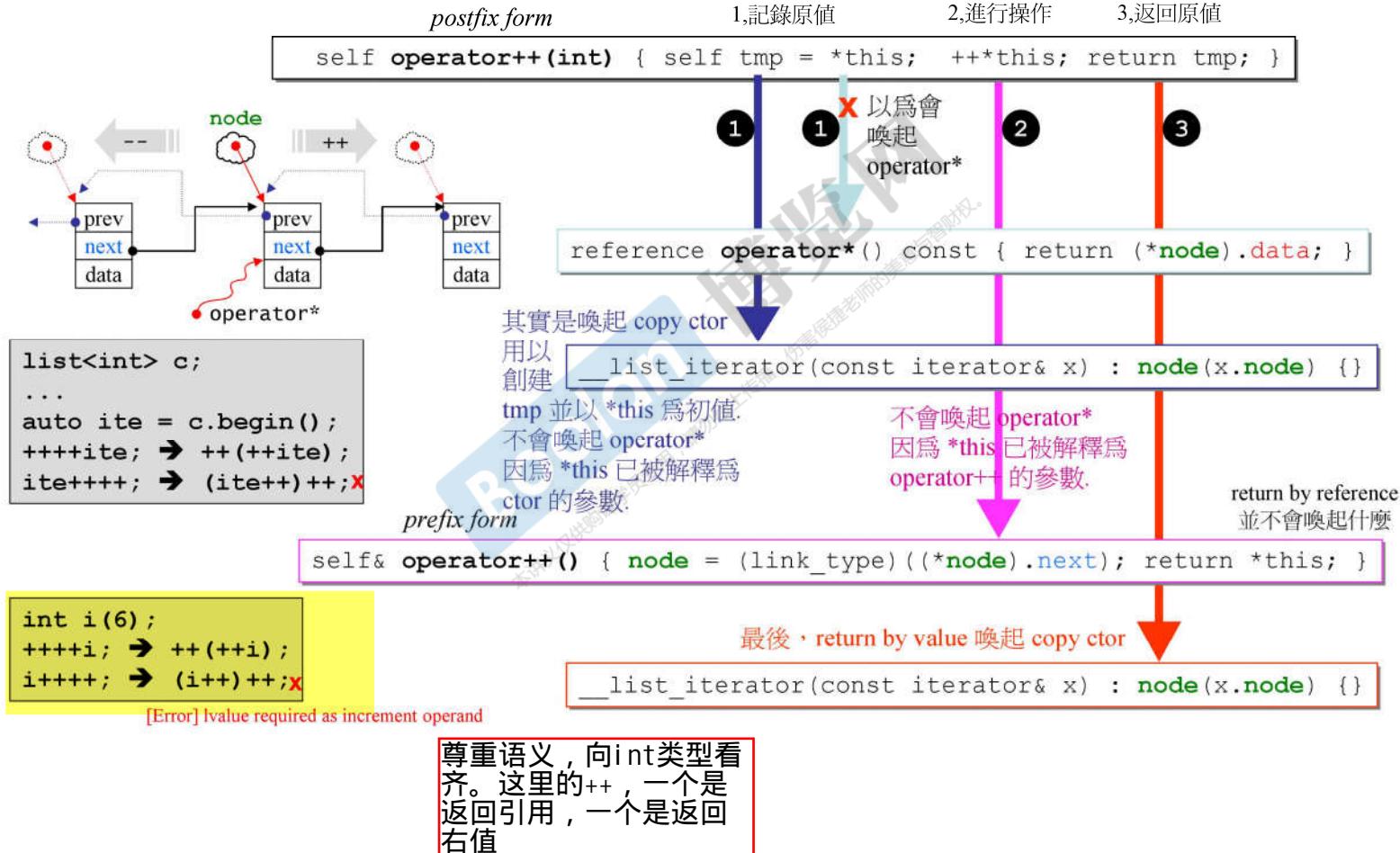
```
template <class T>
struct __list_node {
    typedef void* void_pointer;
    void_pointer prev;
    void_pointer next;
    T data;
};
```

```
template<class T, class Ref, class Ptr>
struct __list_iterator {
    typedef __list_iterator<T, Ref, Ptr> self;
    typedef bidirectional_iterator_tag iterator_category; // (1)
    typedef T value_type; // (2)
    typedef Ptr pointer; // (3)
    typedef Ref reference; // (4)
    typedef __list_node<T>* link_type;
    typedef ptrdiff_t difference_type; // (5)

    link_type node;

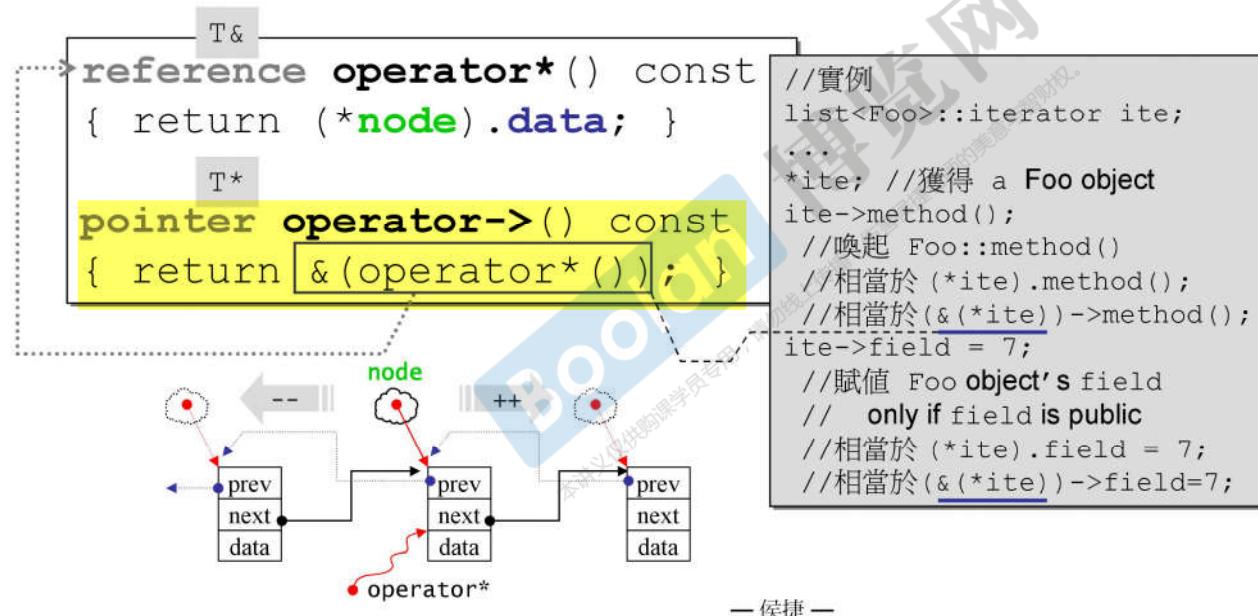
    reference operator*() const { return (*node).data; }
    pointer operator->() const { return &(operator*()); }
    self& operator++()
    {
        node = (link_type)((*node).next); return *this;
    }
    self operator++(int)
    {
        self tmp = *this; ++*this; return tmp;
    }
    ...
};
```

## list's iterator



## list's iterator

當你對某個 type 實施 `operator->`，而該 type 並非 built-in ptr 時，編譯器會做一件很有趣的事：在找出 user-defined `operator->` 並將它施行於該 type 後，編譯器會對執行結果再次施行 `operator->`。編譯器不斷執行這動作直至觸及 a pointer to a built-in type，然後才進行成員存取。



— 侯捷 —

## ■■■ list's iterator

```
template <class T,  
         class Alloc = alloc>  
class list {  
public:  
    typedef __list_iterator<T,T&,T*> iterator;  
    ...  
};
```

G2.9

```
template<class T,  
        class Ref, class Ptr>  
struct __list_iterator  
{  
    ...  
    typedef Ptr pointer; // (3)  
    typedef Ref reference; // (4)  
    ...  
};
```

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

```
template<typename _Tp,  
        typename _Alloc = std::allocator<_Tp>>  
class list : protected __List_base<_Tp, _Alloc>  
{  
public: typedef __List_iterator<_Tp> iterator;  
    ...  
};
```

G4.9

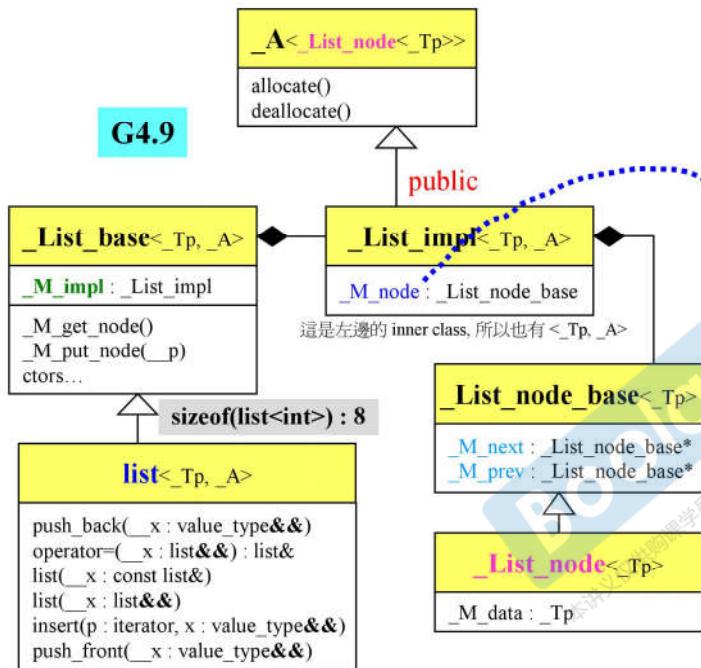
```
template<typename _Tp>  
struct __List_iterator  
{  
    ...  
    _Tp* pointer; // (3)  
    _Tp& reference; // (4)  
    ...  
};
```

```
struct __List_node_base  
{  
    ...  
    __List_node_base* _M_next;  
    __List_node_base* _M_prev;  
};
```

G4.9 相較於 G2.9：  
• 模板參數只有一個 (易理解)  
• node 結構有其 parent  
• node 的成員的 type 較精確  
— 快捷 —

```
template<typename _Tp>  
struct __List_node  
    : public __List_node_base  
{  
    ...  
    _Tp _M_data;  
};
```

## 容器 list



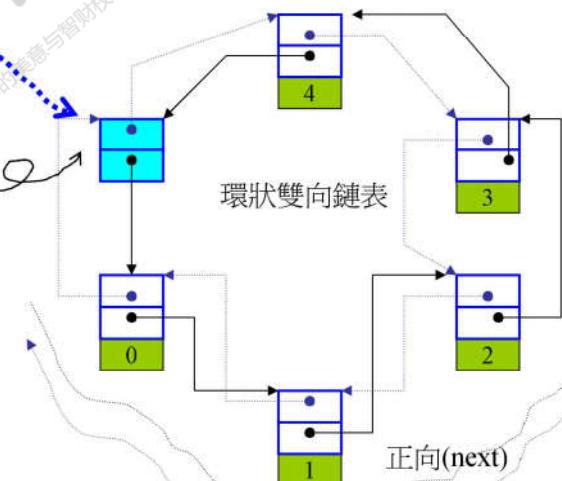
```

iterator
begin() _GLIBCXX_NOEXCEPT
{ return iterator(this->_M_impl._M_node._M_next); }

iterator
end() _GLIBCXX_NOEXCEPT
{ return iterator(&this->_M_impl._M_node); }

```

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。



— 侯捷 —

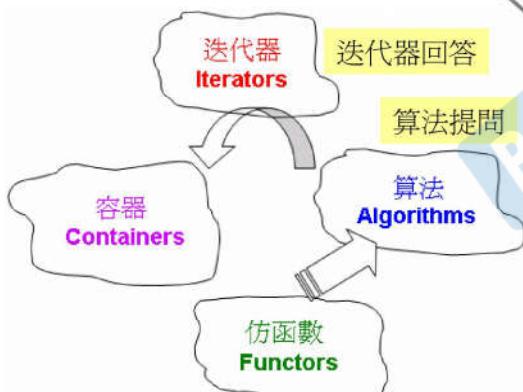
87

## Iterator 需要遵循的原則

```
template<typename _ForwardIterator>
inline void
rotate(_ForwardIterator __first,
       _ForwardIterator __middle,
       _ForwardIterator __last)
{
    ...
    std::rotate(__first, __middle, __last,
               std::iterator_category(__first));
}
```

這張圖可看出，  
rotate() 需要知道  
iterators 的三個  
associated types

```
/*
 * This function is not a part of the C++ standard but is syntactic
 * sugar for internal library use only.
 */
template<typename _Iter>
inline typename iterator_traits<_Iter>::iterator_category
__iterator_category(const _Iter&)
{ return typename iterator_traits<_Iter>::iterator_category(); }
```



```
// This is a helper function for the rotate algorithm.
template<typename _RandomAccessIterator>
void
__rotate(_RandomAccessIterator __first,
         _RandomAccessIterator __middle,
         _RandomAccessIterator __last,
         random_access_iterator_tag)
{
    ...
    typedef typename iterator_traits<_RandomAccessIterator>::difference_type _Distance;
    typedef typename iterator_traits<_RandomAccessIterator>::value_type _ValueType;
    _Distance __n = __last - __first;
    _Distance __k = __middle - __first;
    ...
    for (;;) {
        if (__k < __n - __k) {
            if (__is_pod(_ValueType) && __k == 1) {
                _ValueType __t = __GLIBCXX_MOVE(*__p)
            }
        }
    }
}
```

如果這是 RAI

iterators 必須有能力回答 algorithms 的提問

這樣的提問在 C++ 標準庫開發過程中設計出 5 種，本例出現 3 種。另兩種從未在 C++ 標準庫中被使用過：reference 和 pointer。

迭代器必须提供这五种  
类型

## Iterator 必須提供的 5 種 associated types

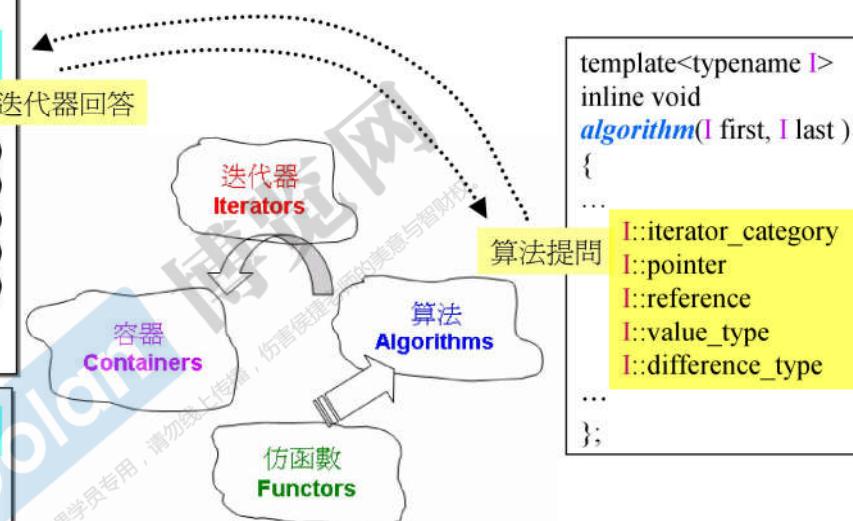
表示他是  
双向  
链表

```
template<class T, class Ref, class Ptr>
struct __list_iterator
{
    typedef bidirectional_iterator_tag iterator_category; // (1)
    typedef T value_type; // (2)
    typedef Ptr pointer; // (3)
    typedef Ref reference; // (4)
    typedef ptrdiff_t difference_type; // (5)
    ...
};
```

G2.9

```
template<typename _Tp>
struct _List_iterator
{
    typedef std::bidirectional_iterator_tag iterator_category;
    typedef _Tp value_type;
    typedef _Tp* pointer;
    typedef _Tp& reference;
    typedef ptrdiff_t difference_type;
    ...
};
```

G4.9



但，如果 iterator 並不是個 class 呢？  
例如 native pointer  
(它被視為一種退化的 iterator)。

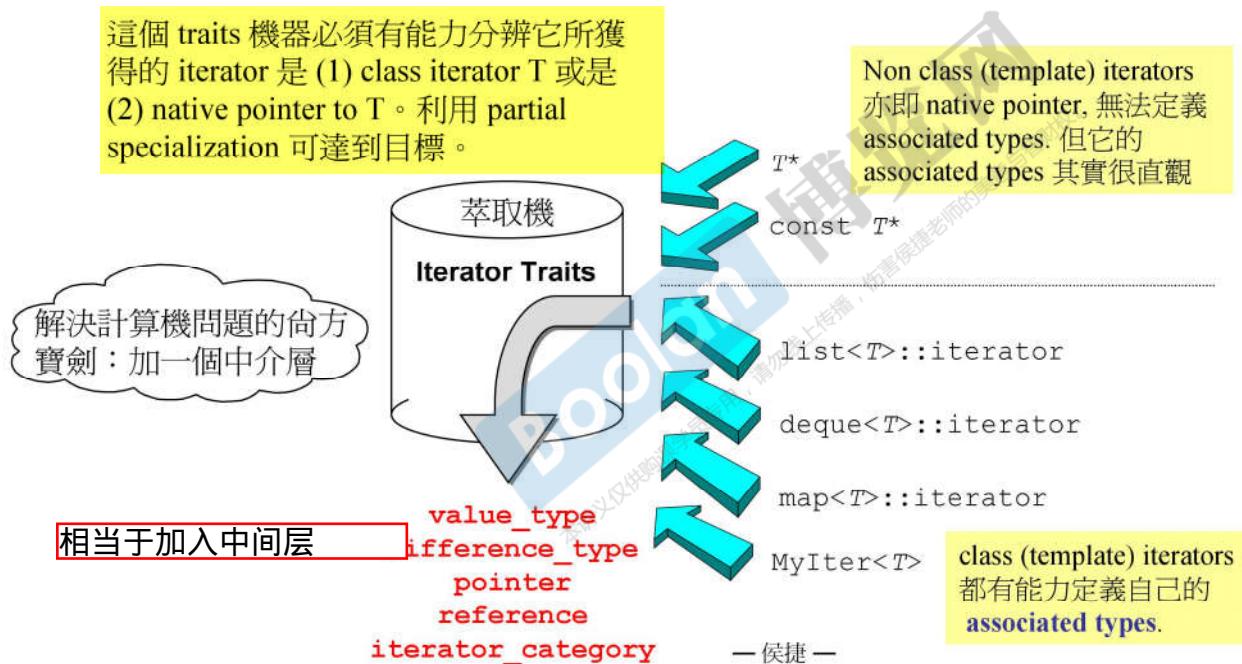
— 侯捷 —

89

问题：如果是迭代器是  
指针，指针无法回答问  
题，怎么办——通过萃  
取机解决

## Traits 特性,特徵,特質

Iterator Traits 用以分離 class iterators 和 non-class iterators



## /// iterator\_traits

Iterator Traits 用以分離 class iterators 和 non-class iterators

```
1 template <class I>
  struct iterator_traits { //traits 是特性之意
    typedef typename I::value_type value_type; //如果 I 是 class iterator 進入這裡
  };

2 //兩個 partial specialization: 偏特化
template <class T>
struct iterator_traits<T*> { //如果 I 是 pointer to T 進入這裡
  typedef T value_type;
};

3 template <class T>
struct iterator_traits<const T*> {
  typedef T value_type; //注意是 T 而不是 const T //如果 I 是 pointer to const T 進入這裡
}; 需要注意为什么不是 const
```

於是當需要知道 I 的 value type 時便可這麼寫：

```
template<typename I,...>
void algorithm(...) {
  typename iterator_traits<I>::value_type v1; //候選
```

value\_type 的主要目的是用來聲明變量，而聲明一個無法被賦值的變量沒什麼用，所以 iterator (即便是 constant iterator) 的 value type 不應加上 const。iterator 若是 const int\*，其 value\_type 應該是 int 而非 const int.

## 回答五个问题

### |||| 完整的 iterator\_traits

ref G2.9 <stl\_iterator.h>

```
template <class I>
struct iterator_traits {
    typedef typename I::iterator_category iterator_category;
    typedef typename I::value_type value_type;
    typedef typename I::difference_type difference_type;
    typedef typename I::pointer pointer;
    typedef typename I::reference reference;
};

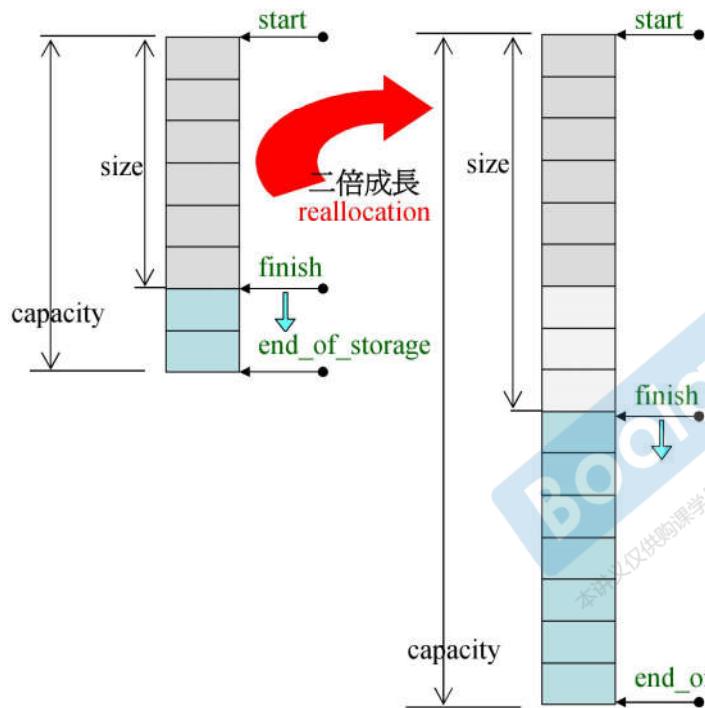
//partial specialization for regular pointers
template <class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};

//partial specialization for regular const pointers
template <class T>
struct iterator_traits<const T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef const T* pointer;
    typedef const T& reference;
};
```

## ■■■ 各式各樣的 Traits

- **type traits** <.../C++/type\_traits>
- **iterator traits** <.../C++/bits/stl\_iterator.h>
- **char traits** <.../C++/bits/char\_traits.h>
- **allocator traits** <.../C++/bits/alloc\_traits.h>
- **pointer traits** <.../C++/bits/ptr\_traits.h>
- **array traits** <.../C++/array>

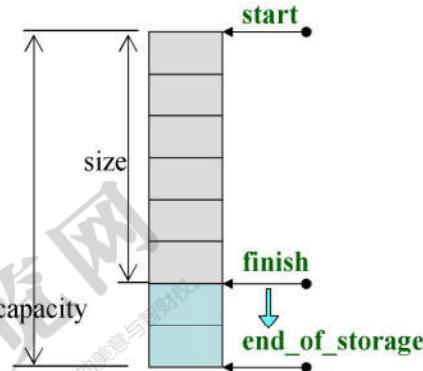
## 容器 vector



```
template <class T, class Alloc = alloc>
class vector {
public:
    typedef T           value_type;
    typedef value_type* iterator; //T*
    typedef value_type& reference;
    typedef size_t      size_type;
protected:
    iterator start;
    iterator finish;
    iterator end_of_storage;
public:
    iterator begin() { return start; }
    iterator end() { return finish; }
    size_type size() const
    { return size_type(end() - begin()); }
    size_type capacity() const
    { return size_type(end_of_storage - begin()); }
    bool empty() const { return begin() == end(); }
    reference operator[](size_type n)
    { return *(begin() + n); }
    reference front() { return *begin(); }
    reference back() { return *(end() - 1); }
};
```

## 容器 vector

```
void push_back(const T& x) {
    if (finish != end_of_storage) { //尚有備用空間
        construct(finish, x);      //全局函數
        ++finish;                  //調整水位高度
    }
    else //已無備用空間
        insert_aux(end(), x);
}
```



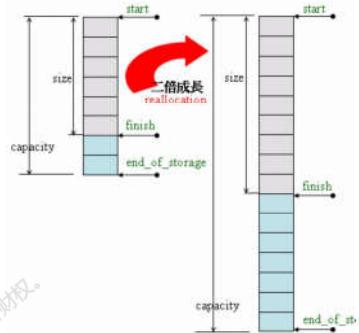
```
template <class T, class Alloc>
void vector<T,Alloc>::insert_aux(iterator position, const T& x) {
    if (finish != end_of_storage) { //尚有備用空間
        //在備用空間起始處建構一個元素，並以vector最後一個元素值為其初值。
        construct(finish, *(finish - 1));
        ++finish; //調整水位。
        T x_copy = x;
        copy_backward(position, finish - 2, finish - 1);
        *position = x_copy;
    }
    else { //已無備用空間
        繼下頁
    }
}
```

## 容器 vector

```
else { //已無備用空間
    const size_type old_size = size();
    const size_type len = old_size != 0 ? [2 * old_size] : 1;
    //以上分配原則：如果原大小為0，則分配 1 (個元素大小)；  

    //如果原大小不為0，則分配原大小的兩倍，  

    //前半段用來放置原數據，後半段準備用來放置新數據。
    iterator new_start = data_allocator::allocate(len);
    iterator new_finish = new_start;
    try {
        ...
    } catch(...) {
        ...
    }
    //解構並釋放原 vector
    destroy(begin(), end());
    deallocate();
    //調整迭代器，指向新vector
    start = new_start;
    finish = new_finish;
    end_of_storage = new_start + len;
}
```



```
try {
    //將原vector的內容拷貝到新vector
    new_finish = uninitialized_copy(start,
                                    position, new_start);
    construct(new_finish, x); //為新元素設初值x
    ++new_finish; //調整水位。
    //拷貝安插點後的原內容（因它也可能被insert(p,x)呼叫）
    new_finish = uninitialized_copy(position,
                                    finish, new_finish);
}
catch(...) {
    // "commit or rollback" semantics.
    destroy(new_start, new_finish);
    data_allocator::deallocate(new_start, len);
    throw;
}
```

因为push\_back可能被insert调用，所以需要将安插点之后的拷贝。

每一次空间成长，可能  
大量调用拷贝构造函数  
和析构函数，使用  
vector的时候需要注意

## vector's iterator

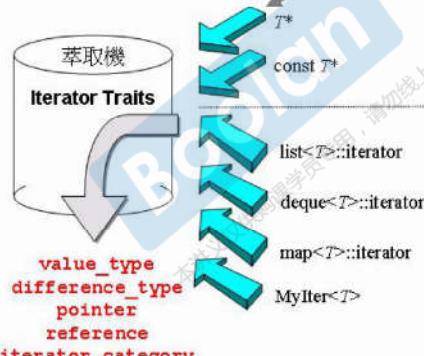
vector的迭代器实际就是指针

```
template <class T, class Alloc = alloc>
class vector {
public:
    typedef T value_type;
    typedef value_type* iterator; //T*
    ...
};
```

G2.9

```
vector<int> vec;
...
vector<int>::iterator ite
= vec.begin();
```

- ① iterator\_traits<ite>::  
iterator\_category
- ② iterator\_traits<ite>::  
difference\_type
- ③ iterator\_traits<ite>::  
value\_type



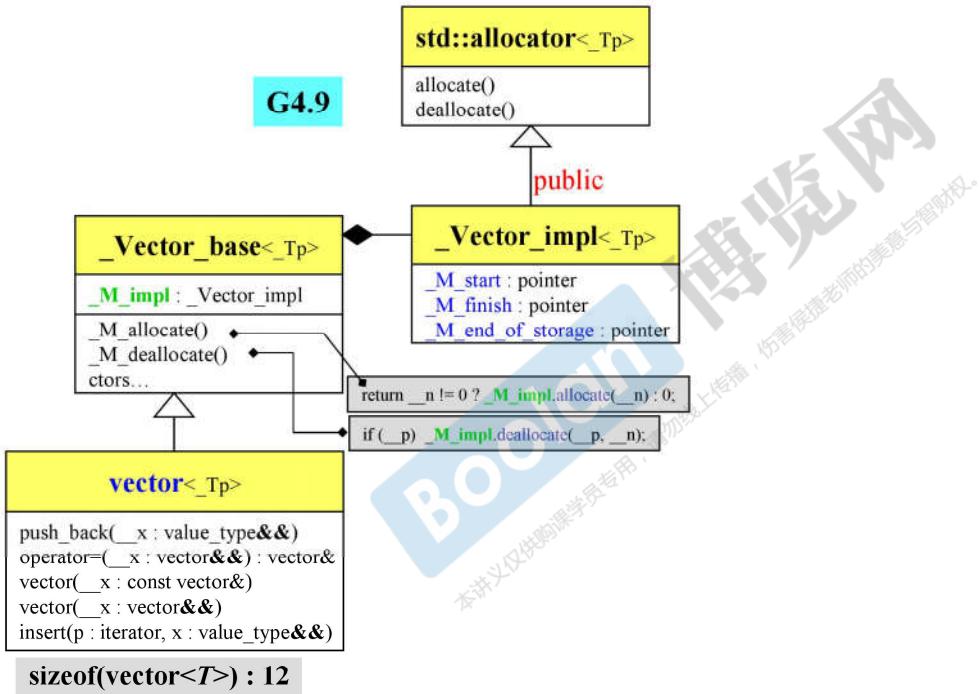
```
template <class I>
struct iterator_traits {
    typedef typename I::iterator_category iterator_category;
    typedef typename I::value_type      value_type;
    typedef typename I::difference_type difference_type;
    typedef typename I::pointer        pointer;
    typedef typename I::reference     reference;
};

//partial specialization for regular pointers
template <class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T      value_type;
    typedef ptrdiff_t difference_type;
    typedef T*    pointer;
    typedef T&    reference;
};

//partial specialization for regular const pointers
template <class T>
struct iterator_traits<const T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T      value_type;
    typedef ptrdiff_t difference_type;
    typedef const T* pointer;
    typedef const T& reference;
};
```

— 侯捷 —

## 容器 vector



## vector's iterator

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp>>
class vector : protected _Vector_base<_Tp, _Alloc>
{
...
typedef _Vector_base<_Tp, _Alloc> _Base;
typedef typename _Base::pointer pointer;
typedef _gnu_cxx::normal_iterator<pointer, vector> iterator;
```

G4.9

```
using std::iterator_traits;
using std::iterator;
template<typename _Iterator, typename _Container>
class _normal_iterator
{
protected:
    _Iterator _M_current;
public:
    typedef _Iterator iterator_type;
    typedef typename _traits_type::iterator_category iterator_category;
    typedef typename _traits_type::value_type value_type;
    typedef typename _traits_type::difference_type difference_type;
    typedef typename _traits_type::reference reference;
    typedef typename _traits_type::pointer pointer;
```

\_GLIBCXX\_CONSTEXPR **\_normal\_iterator()** \_GLIBCXX\_NOEXCEPT  
: **\_M\_current**(**\_Iterator**(0)) {}

本頁追蹤 **vector::iterator**，發現它就是 **\_Tp\*** 外覆一個 iterator adapter，使能支持 5 associated types

```
template<typename _Tp, typename _Alloc>
struct _Vector_base
{
    ...
    typedef typename _gnu_cxx::alloc_traits<_Alloc>::template
        rebind<_Tp>::other _Tp_alloc_type;
    typedef typename _gnu_cxx::alloc_traits<_Tp_alloc_type>::pointer pointer
```

**vector<T>::iterator**

**\_M\_current** : **\_Tp\***

```
template<typename _Tp>
class allocator: public _glibcxx_base_allocator<_Tp>
{
public:
    ...
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef _Tp* pointer;
    typedef const _Tp* const_pointer;
    typedef _Tp& reference;
    typedef const _Tp& const_reference;
    typedef _Tp value_type;
    ...
    // Inherit everything else.
};
```

9

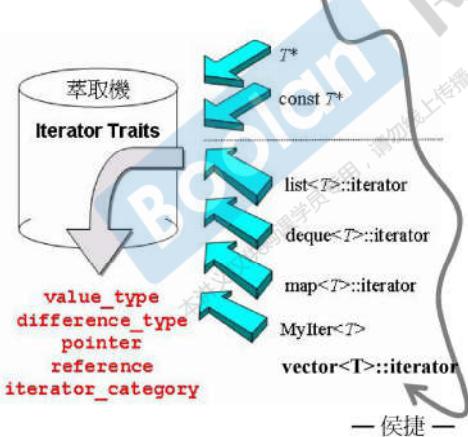
## vector's iterator

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp>>
class vector : protected _Vector_base<_Tp, _Alloc>
{
...
typedef _Vector_base<_Tp, _Alloc> _Base;
typedef typename _Base::pointer pointer;
typedef __gnu_cxx::__normal_iterator<pointer, vector> iterator;
```

G4.9

```
vector<int> vec;
...
vector<int>::iterator ite
= vec.begin();
```

- ① **iterator\_traits<ite>::iterator\_category**
- ② **iterator\_traits<ite>::difference\_type**
- ③ **iterator\_traits<ite>::value\_type**



本頁追蹤 **vector::iterator**，發現它就是 **\_Tp\*** 外覆一個 iterator adapter，使能支持 5 associated types

**vector<T>::iterator**  
**\_M\_current : \_Tp\***

```
template<typename _Iterator, typename _Container>
class _normal_iterator
{
protected:
    _Iterator _M_current;
    typedef iterator_traits<_Iterator> _traits_type;
public:
    typedef _Iterator iterator_type;
    typedef typename _traits_type::iterator_category iterator_category;
    typedef typename _traits_type::value_type value_type;
    typedef typename _traits_type::difference_type difference_type;
    typedef typename _traits_type::reference reference;
    typedef typename _traits_type::pointer pointer;
    ...
}
```

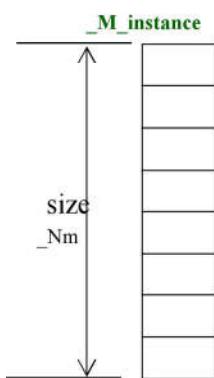
亂七八糟，捨近求遠，何必如此 !!

## 容器 array

TR1

array<\_Tp, \_Nm>

\_M\_instance : \_Tp[\_Nm]



```
template<typename _Tp, std::size_t _Nm>
struct array
{
    typedef _Tp           value_type;
    typedef _Tp*          pointer;
    typedef value_type*   iterator;    其 iterator 是 native pointer,
                                     (G2.9 vector 也是如此)

    // Support for zero-sized arrays mandatory.
    value_type _M_instance[_Nm ? _Nm : 1];

    iterator begin()
    { return iterator(&_M_instance[0]); }

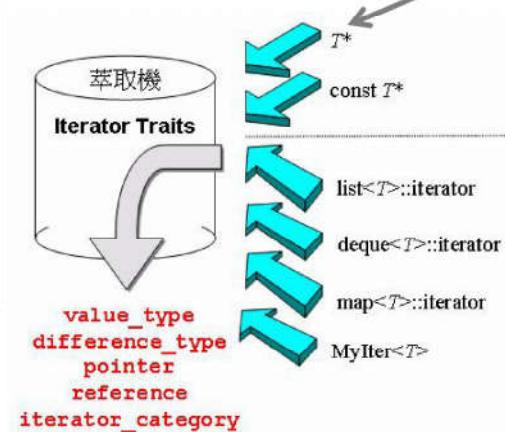
    iterator end()
    { return iterator(&_M_instance[_Nm]); }

    ...
}
```

沒有ctor, 沒有dtor

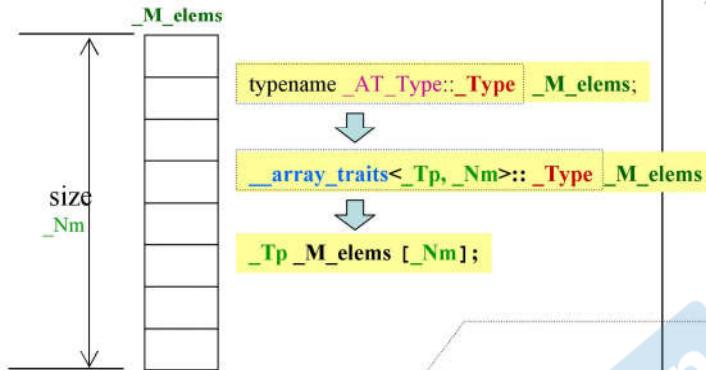
— 侯捷 —

```
array<int, 10> myArray;
auto ite = myArray.begin();
// array<int, 10>::iterator ite = ...
ite += 3;
cout << *ite;
```



## 容器 array

G4.9



template<typename Tp, std::size\_t Nm>  
struct array

```
{
  typedef Tp value_type;
  typedef value_type* pointer;
  typedef value_type& reference;
  typedef value_type* iterator;
  typedef std::size_t size_type;
```

```
int a[100]; //OK
int[100] b; //fail
typedef int T[100];
T c; //OK
```

// Support for zero-sized arrays mandatory.

```
typedef _GLIBCXX_STD_C:: array_traits<Tp, Nm> AT_Type;
typename AT_Type:: Type M_elems;
```

// No explicit construct/copy/destroy for aggregate type.

```
iterator begin() noexcept { return iterator(data()); }
```

```
iterator end() noexcept { return iterator(data() + Nm); }
```

```
constexpr size_type size() const noexcept { return Nm; }
```

```
reference operator[](size_type n) noexcept
```

```
{ return AT_Type:: S_ref(M_elems, n); }
```

沒有邊檢

```
reference at(size_type n) {
```

```
  if (n >= Nm) std:: throw_out_of_range_fmt(...);
```

```
  return AT_Type:: S_ref(M_elems, n);
```

```
}
```

```
pointer data() noexcept
```

```
{ return std:: addressof(AT_Type:: S_ref(M_elems, 0)); }
```

```
...
```

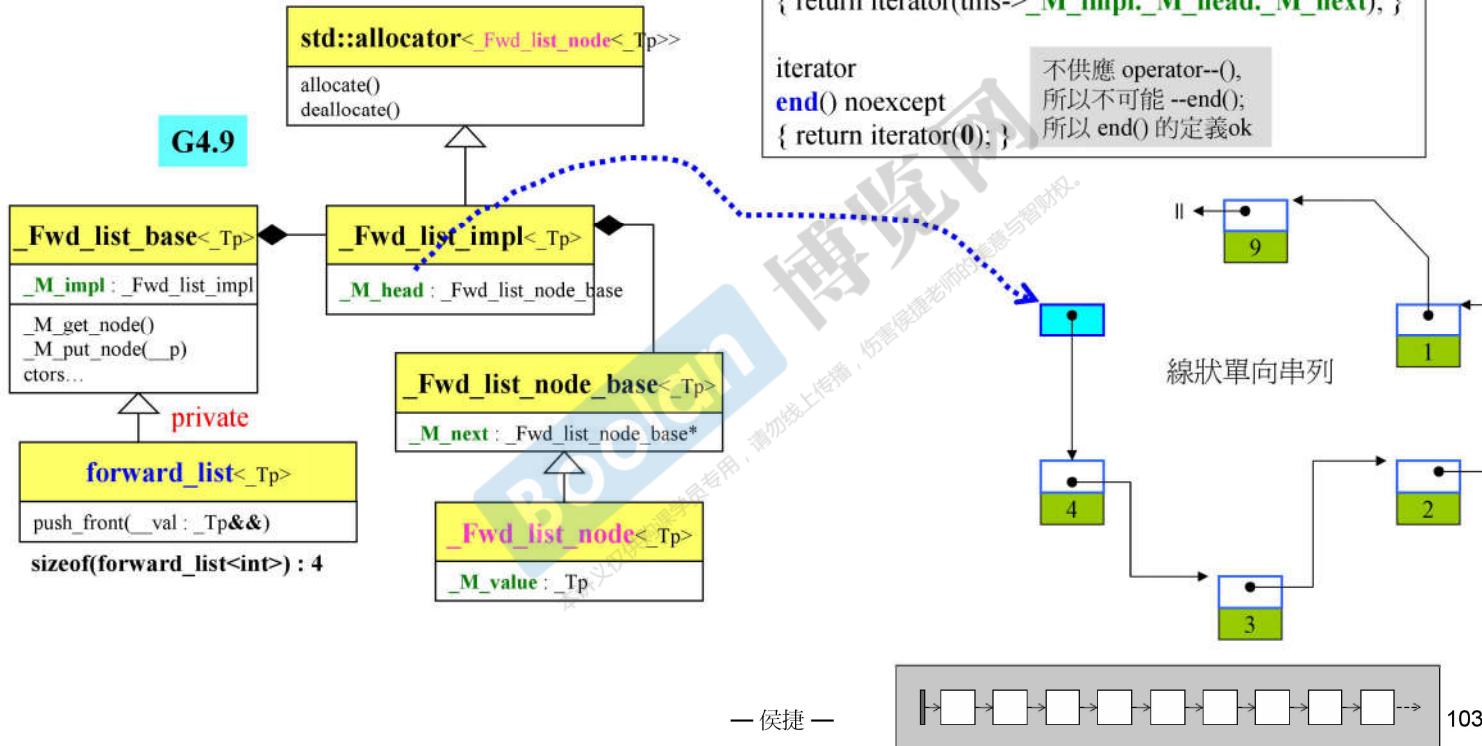
```
};
```

```
template<typename Tp, std::size_t Nm>
struct array_traits
{
  typedef Tp Type[_Nm];
```

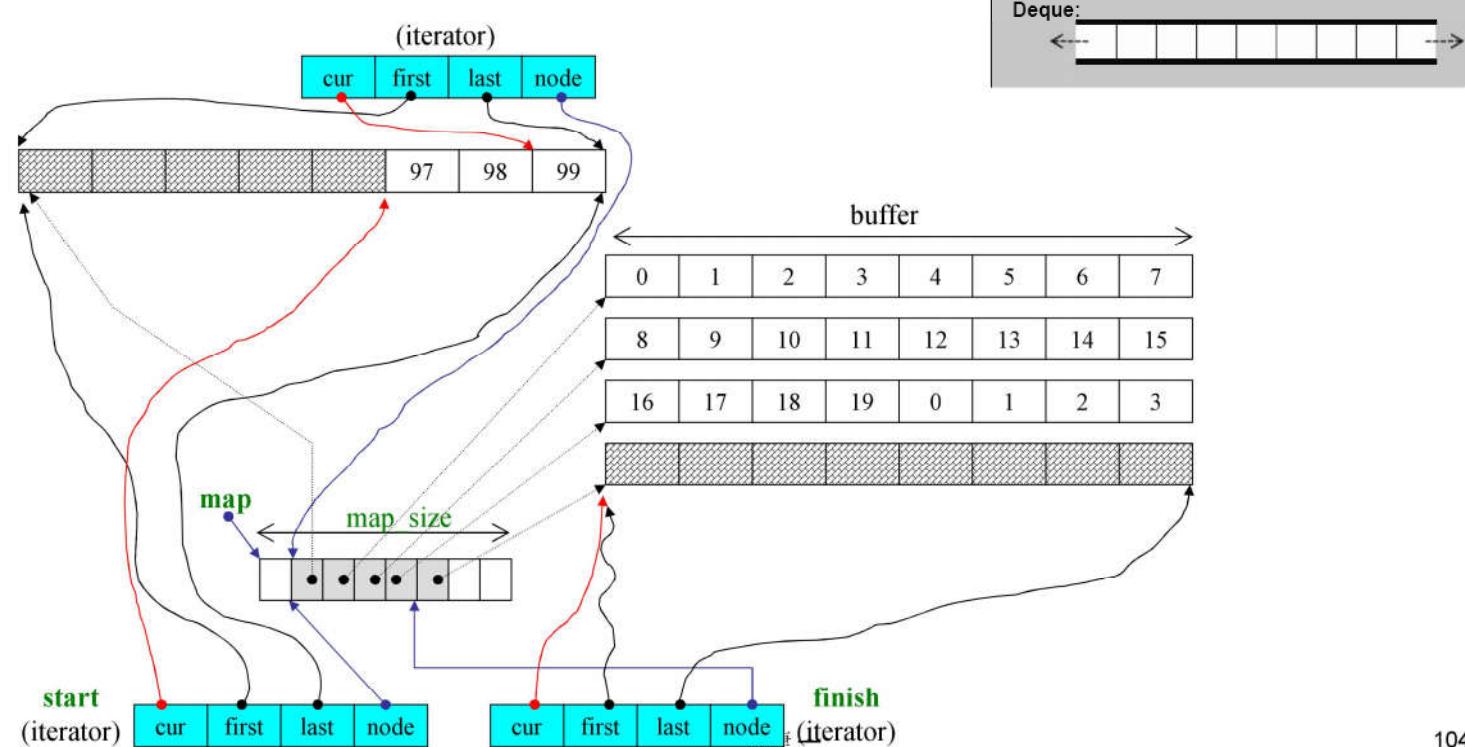
```
  static constexpr Tp&
```

```
  _S_ref(const Type& t, std::size_t n) noexcept
  { return const_cast<Tp&>(t[n]); }
};
```

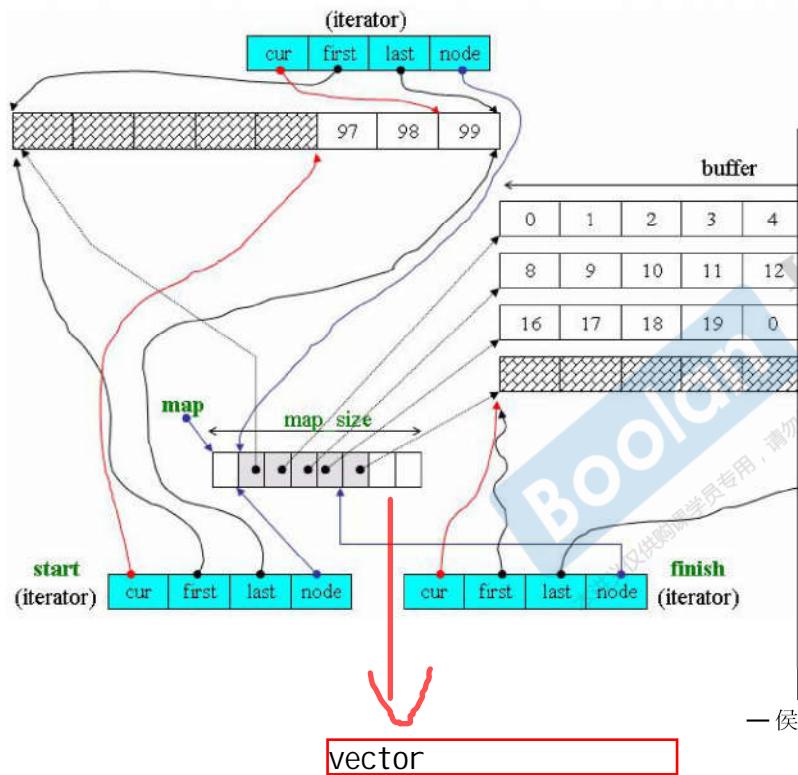
## 容器 forward\_list



## 容器 deque



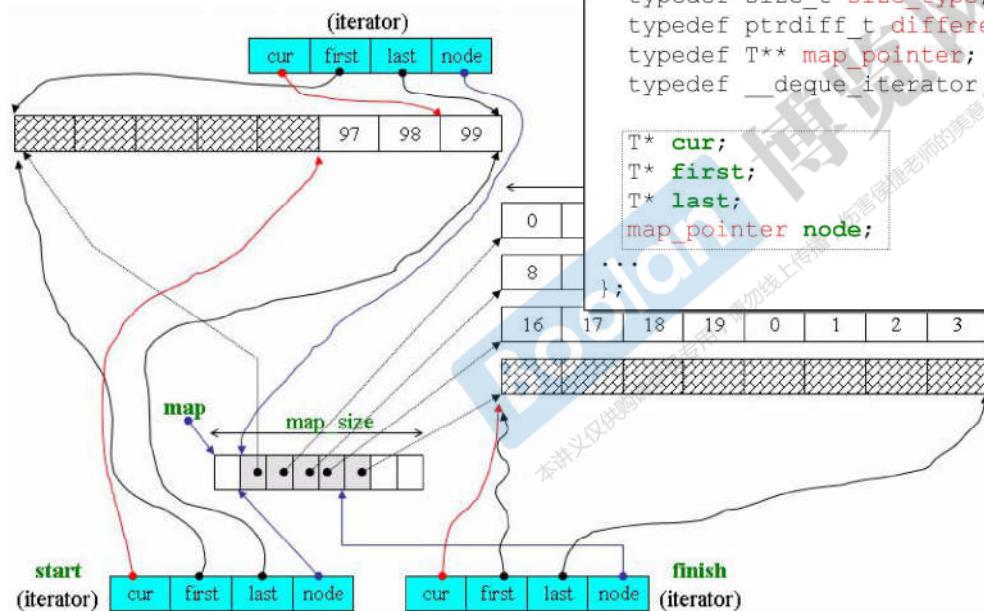
## 容器 deque



```
// 如果 n 不為 0，傳回 n，表示 buffer size 由使用者自定。  
// 如果 n 為 0，表示 buffer size 使用預設值，那麼  
//   如果 sz (sizeof(value_type)) 小於 512，傳回 512/sz，  
//   如果 sz 不小於 512，傳回 1。  
inline size_t __deque_buf_size(size_t n, size_t sz)  
{  
    return n != 0 ? n : (sz < 512 ? size_t(512 / sz) : size_t(1));  
}
```

```
template <class T, class Alloc=alloc,  
         size_t BufSiz=0>  
class deque {  
public:  
    // 所謂 buffer size 是指每個 buffer 容納的元素個數  
protected:  
    typedef T value_type;  
    typedef __deque_iterator<T, T&, T*, BufSiz> iterator;  
protected:  
    pointer* map_pointer; // T**  
protected:  
    iterator start;  
    iterator finish;  
    map_pointer map;  
    size_type map_size;  
public:  
    iterator begin() { return start; }  
    iterator end() { return finish; }  
    size_type size() const { return finish - start; }  
    ...  
};
```

## ///// deque'iterator



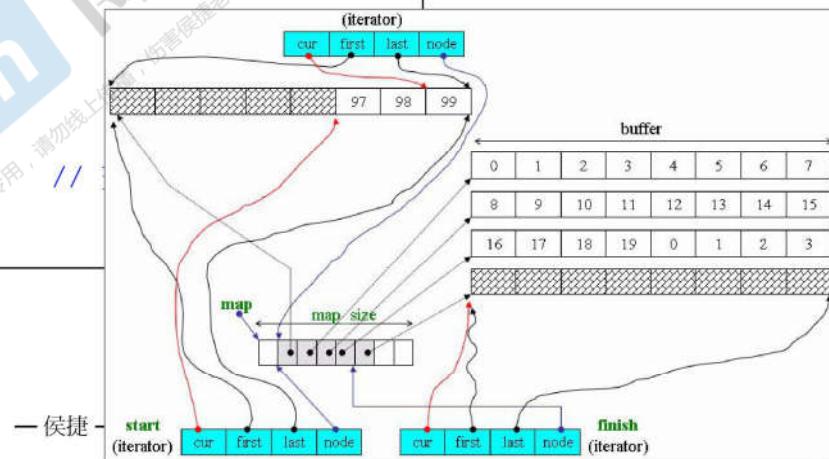
```
template <class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_iterator {
    typedef random_access_iterator_tag iterator_category; // (1)
    typedef T value_type; // (2)
    typedef Ptr pointer; // (3)
    typedef Ref reference; // (4)
    typedef size_t size_type;
    typedef ptrdiff_t difference_type; // (5)
    typedef T** map_pointer;
    typedef __deque_iterator self;
};
```

## deque<T>::insert()

插入会判断当前离尾端  
还是首端比较近

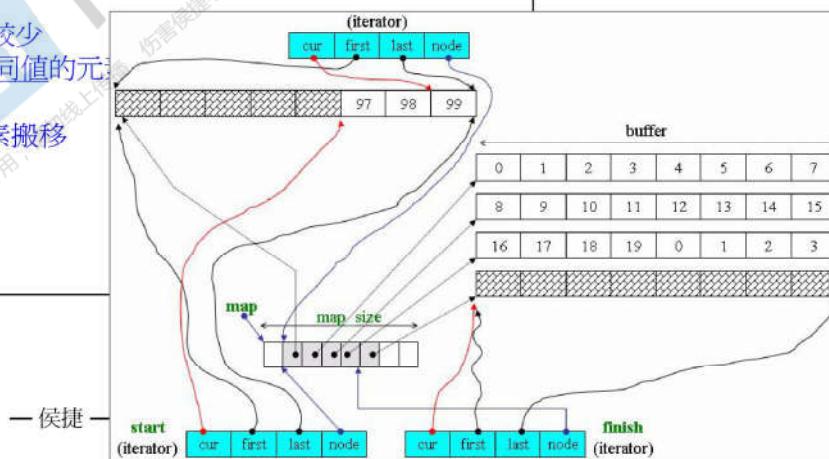
```
// 在position 處安插一個元素，其值為 x
iterator insert(iterator position, const value_type& x) {
    if (position.cur == start.cur) { //如果安插點是deque最前端
        push_front(x);           //交給push_front()做
        return start;
    }
    else if (position.cur == finish.cur) { //如果安插點是deque最尾端
        push_back(x);            //交給push_back()做
        iterator tmp = finish;
        --tmp;
        return tmp;
    }
    else {
        return insert_aux(position, x);
    }
}
```

下頁



## ■■■■ deque<T>::insert()

```
template <class T, class Alloc, size_t BufSize>
typename deque<T, Alloc, BufSize>::iterator
deque<T, Alloc, BufSize>::insert_aux(iterator pos, const value_type& x) {
    difference_type index = pos - start;           // 安插點之前的元素個數
    value_type x_copy = x;
    if (index < size() / 2) {                      // 如果安插點之前的元素個數較少
        push_front(front());                        // 在最前端加入與第一元素同值的元素。
        ...
        copy(front2, pos1, front1);                // 元素搬移
    }
    else {                                         // 安插點之後的元素個數較少
        push_back(back());                         // 在尾端加入與最末元素同值的元素。
        ...
        copy_backward(pos, back2, back1);          // 元素搬移
    }
    *pos = x_copy;      // 在安插點上設定新值
    return pos;
}
```



## deque 如何模擬連續空間

全都是 deque iterators 的功勞

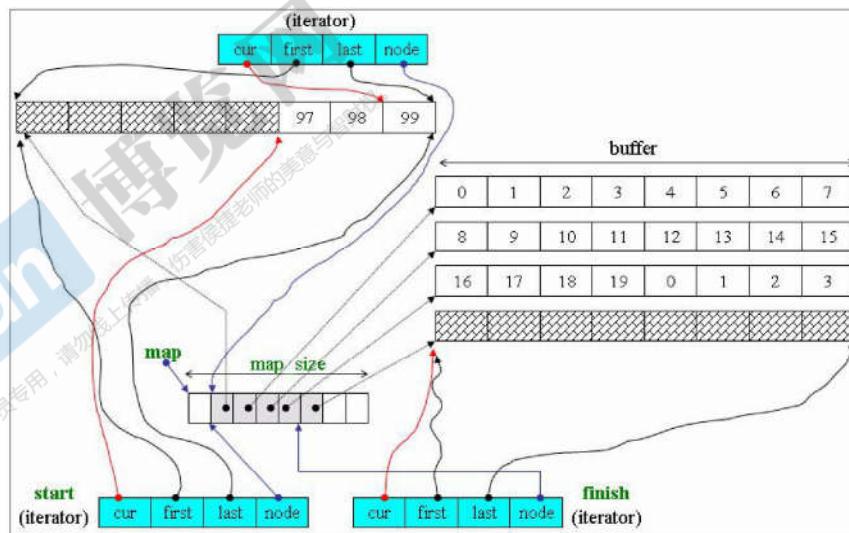
```
reference operator[](size_type n)
{
    return start[difference_type(n)];
}

reference front()
{ return *start; }

reference back()
{
    iterator tmp = finish;
    --tmp;
    return *tmp;
}

size_type size() const
{ return finish - start; }

bool empty() const
{ return finish == start; }
```



## deque 如何模擬連續空間

全都是 deque iterators 的功勞

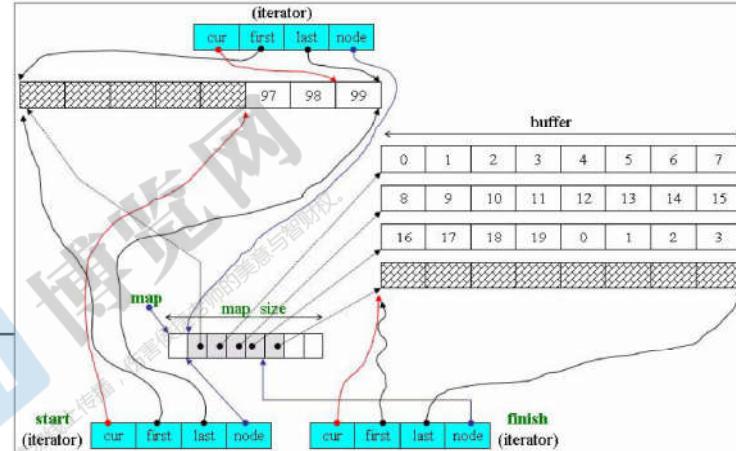
```
reference operator*() const
{ return *cur; }

pointer operator->() const
{ return &(operator*()); }
```

```
//兩根iterators之間的距離相當於
//(1)兩根iterators間的buffers的總長度 +
//(2)itr至其buffer末尾的長度 +
//(3)x 至其buffer起頭的長度
difference_type
operator-(const self& x) const
{
    return difference_type(buffer_size())
        * (node - x.node - 1) +
        (cur - first) + (x.last - x.cur);
}
```

末尾(當前) buffer  
的元素量

起始 buffer  
的元素量



首尾 buffers 之間的  
buffers 數量

— 侯捷

每个node里边装多少元  
素

## deque 如何模擬連續空間 全都是 deque iterators 的功勞

```
self& operator++() {
    ++cur; //切換至下一元素。
    if (cur == last) { //如果抵達緩衝區尾端,
        set_node(node + 1); //就跳至下一節點(緩衝區)
        cur = first; // 的起點。
    }
    return *this;
}

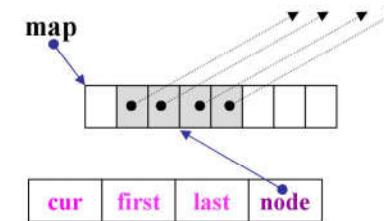
self operator++(int) {
    self tmp = *this;
    ++*this;
    return tmp;
}
```

用后++调用前++

```
self& operator--() {
    if (cur == first) { //如果目前在緩衝區起頭,
        set_node(node - 1); //就跳至前一節點(緩衝區)
        cur = last; // 的最末端。
    }
    --cur;
    return *this;
}

self operator--(int) {
    self tmp = *this;
    --*this;
    return tmp;
}
```

```
void set_node(map_pointer new_node) {
    node = new_node;
    first = *new_node;
    last = first +
           difference_type(buffer_size());
```



## deque 如何模擬連續空間 全都是 deque iterators 的功勞

```
self& operator+=(difference_type n) {
    difference_type offset = n + (cur - first);
    if (offset >= 0 && offset < difference_type(buffer_size()))
        //目標位置在同一緩衝區內
        cur += n;
    else {
        //目標位置不在同一緩衝區內
        difference_type node_offset =
            offset > 0 ? offset / difference_type(buffer_size())
                         : -difference_type((-offset - 1) / buffer_size()) - 1;
        // 切換至正確的緩衝區
        set_node(node + node_offset);
        // 切換至正確的元素
        cur = first + (offset - node_offset * difference_type(buffer_size()));
    }
    return *this;
}

self operator+(difference_type n) const {
    self tmp = *this;
    return tmp += n;
}
```

112

+借助+=实现

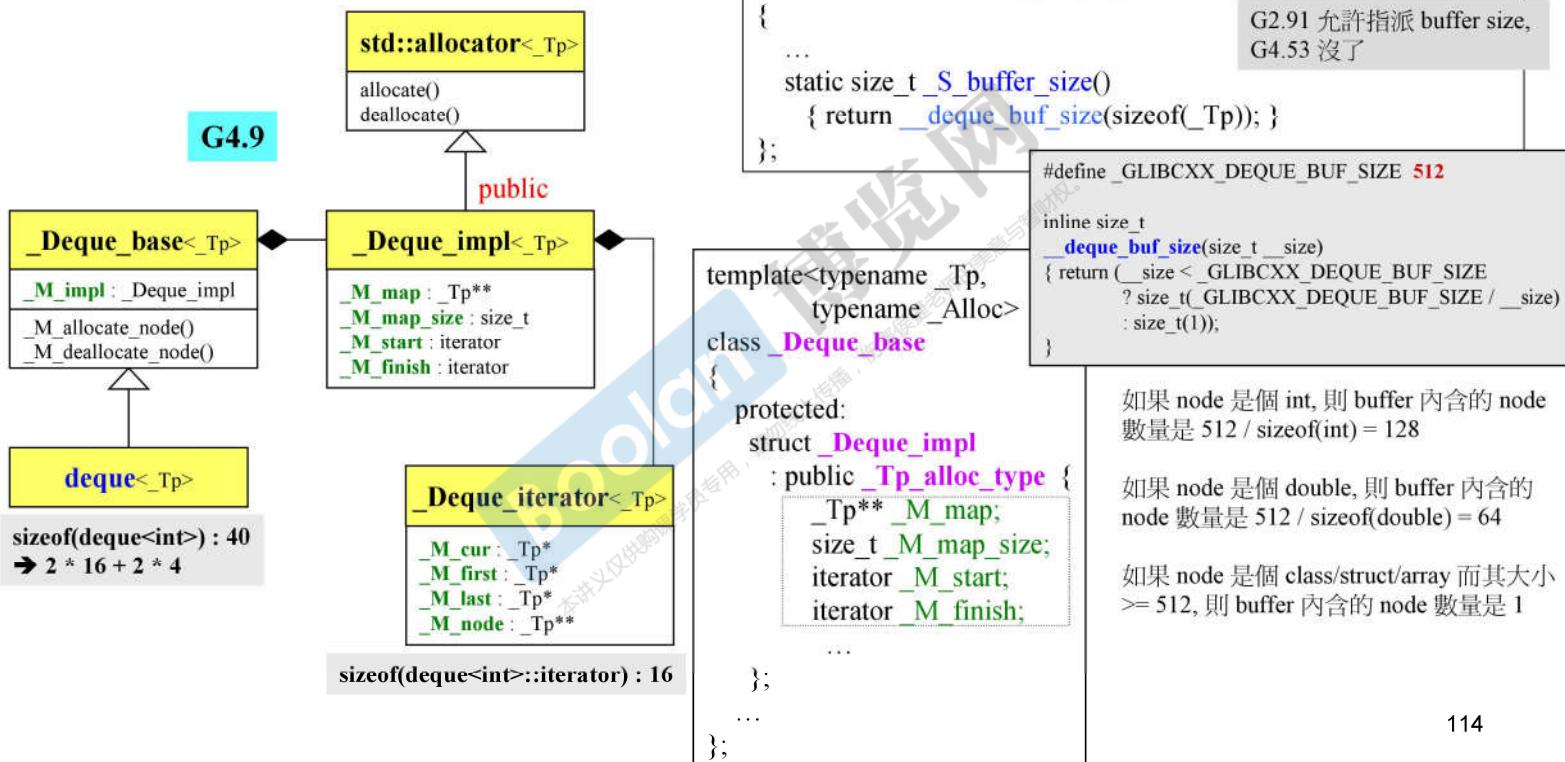
## deque 如何模擬連續空間 全都是 deque iterators 的功勞

```
self& operator-=(difference_type n)
{ return *this += -n; }

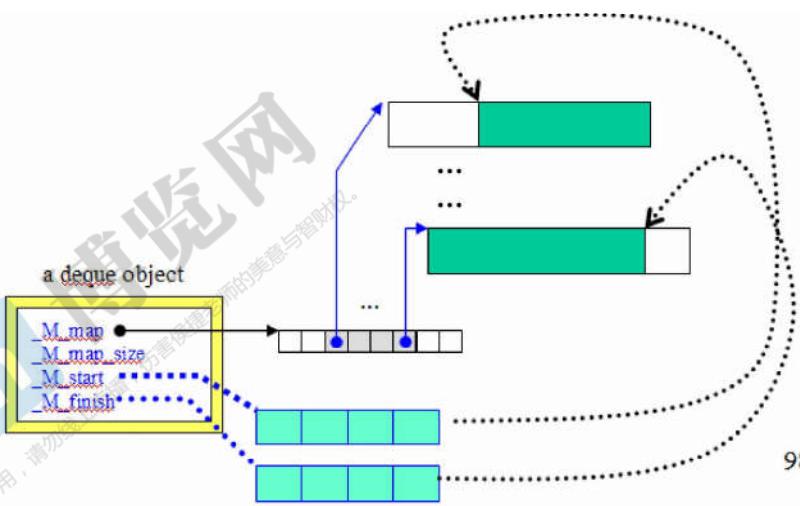
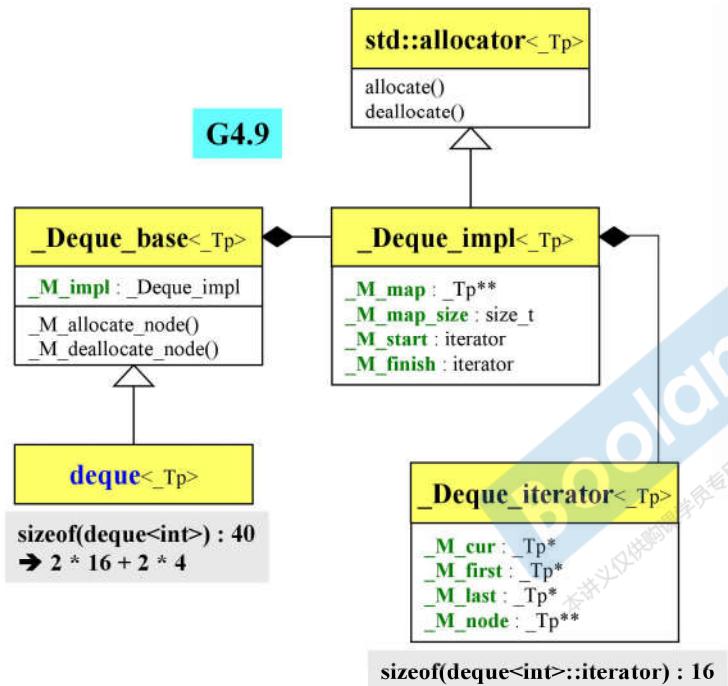
self operator-(difference_type n) const
{
    self tmp = *this;
    return tmp -= n;
}

reference operator[](difference_type n) const
{ return *(*this + n); }
```

## deque

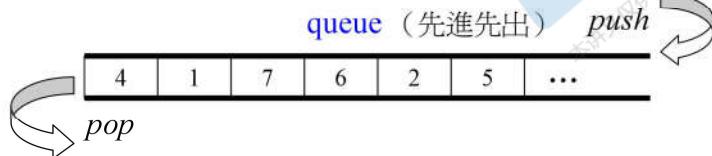


## ///// deque



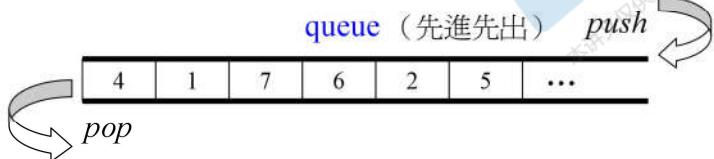
## 容器 queue

内含一个deque，封锁住  
某些功能



```
template <class T, class Sequence=deque<T>>
class queue {
...
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:
    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(); }
};
```

## 容器 stack



```
template <class T, class Sequence=deque<T>>
class 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:
    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 和 stack, 關於其 iterator 和底層結構

stack 或 queue 都不允許遍歷，  
也不提供 iterator。

```
stack<string>::iterator ite; // [Error] 'iterator' is not a member of 'std::stack<std::basic_string<char>>'
```

```
queue<string>::iterator ite; // [Error] 'iterator' is not a member of 'std::queue<std::basic_string<char>>'
```

stack 和 queue 都可選擇 list 或 deque  
做為底層結構。

```
stack<string, list<string>> c;
for(long i=0; i< 10; ++i) {
    sprintf(buf, 10, "%d", rand());
    c.push(string(buf));
}
cout << "stack.size()= " << c.size() << endl;
cout << "stack.top()= " << c.top() << endl;
c.pop();
cout << "stack.size()= " << c.size() << endl;
cout << "stack.top()= " << c.top() << endl;
```

```
queue<string, list<string>> c;
for(long i=0; i< 10; ++i) {
    sprintf(buf, 10, "%d", rand());
    c.push(string(buf));
}
cout << "queue.size()= " << c.size() << endl;
cout << "queue.front()= " << c.front() << endl;
cout << "queue.back()= " << c.back() << endl;
c.pop();
cout << "queue.size()= " << c.size() << endl;
cout << "queue.front()= " << c.front() << endl;
cout << "queue.back()= " << c.back() << endl;
```

一候

## queue 和 stack, 關於其 iterator 和底層結構

queue 不可選擇 vector 做為底層結構

```
queue<string, vector<string>> c;
for(long i=0; i< 10; ++i) {
    sprintf(buf, 10, "%d", rand());
    c.push(string(buf));
}
cout << "queue.size()= " << c.size() << endl;
cout << "queue.front()= " << c.front() << endl;
cout << "queue.back()= " << c.back() << endl;
//c.pop(); //Error: 'class std::vector<std::basic_string<char> >' has no member named 'pop_front'
cout << "queue.size()= " << c.size() << endl;
cout << "queue.front()= " << c.front() << endl;
cout << "queue.back()= " << c.back() << endl;
```

stack 可選擇 vector 做為底層結構

```
stack<string, vector<string>> c;
for(long i=0; i< 10; ++i) {
    sprintf(buf, 10, "%d", rand());
    c.push(string(buf));
}
cout << "stack.size()= " << c.size() << endl;
cout << "stack.top()= " << c.top() << endl;
c.pop();
cout << "stack.size()= " << c.size() << endl;
cout << "stack.top()= " << c.top() << endl;
```

如果没用到这一行，编译是可以通过的。因为其他的没有用到不支持的pop\_front操作

## queue 和 stack, 關於其 iterator 和底層結構

stack 和 queue 都不可選擇 set 或 map 做為底層結構

```
stack<string, set<string>> c;
for(long i=0; i< 10; ++i) {
    snprintf(buf, 10, "%d", rand());
    c.push(string(buf));
}
cout << "stack.size()= " << c.size() << endl;
cout << "stack.top()= " << c.top() << endl;
c.pop();
cout << "stack.size()= " << c.size() << endl;
cout << "stack.top()= " << c.top() << endl;
```

編譯器 無異議



[Error] 'class std::set<std::basic\_string<char> >' has no member named 'push\_back'  
[Error] 'class std::set<std::basic\_string<char> >' has no member named 'back'  
[Error] 'class std::set<std::basic\_string<char> >' has no member named 'pop\_back'

stack<string, map<string>> c;	[Error] template argument...
queue<string, map<string>> c;	[Error] template argument ...

## 容器 rb\_tree

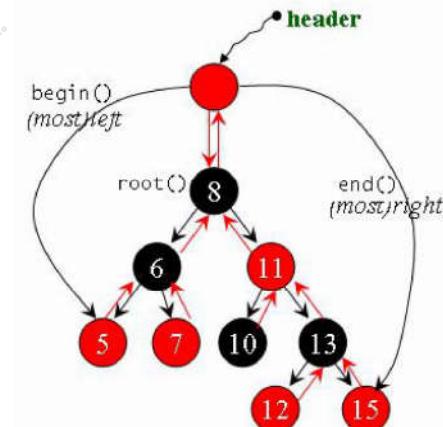
Red-Black tree (紅黑樹) 是平衡二元搜尋樹 (balanced binary search tree) 中常被使用的一種。  
平衡二元搜尋樹的特徵：排列規則有利 search 和 insert，並保持適度平衡—無任何節點過深。

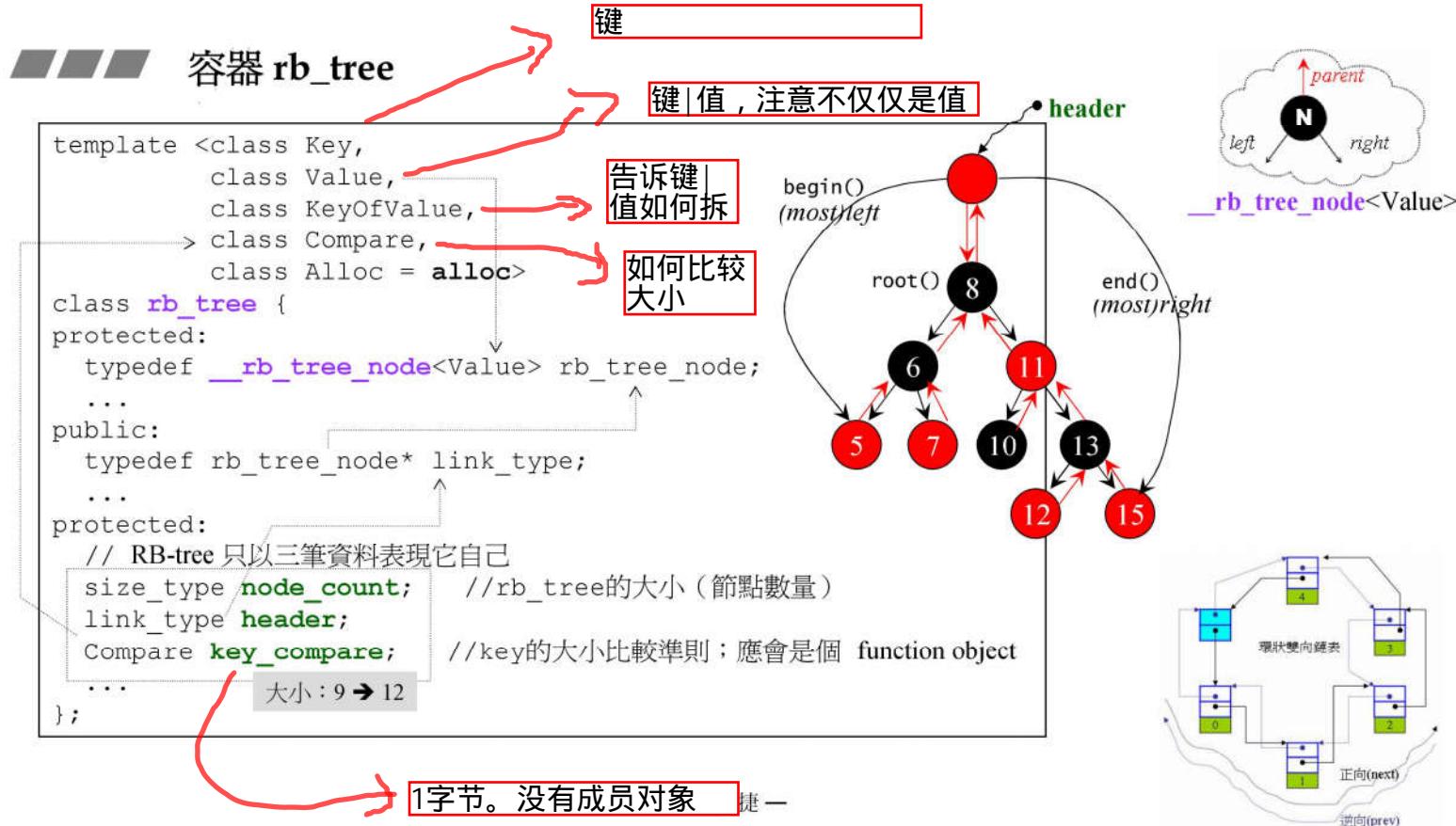
rb\_tree 提供“遍歷”操作及 iterators。

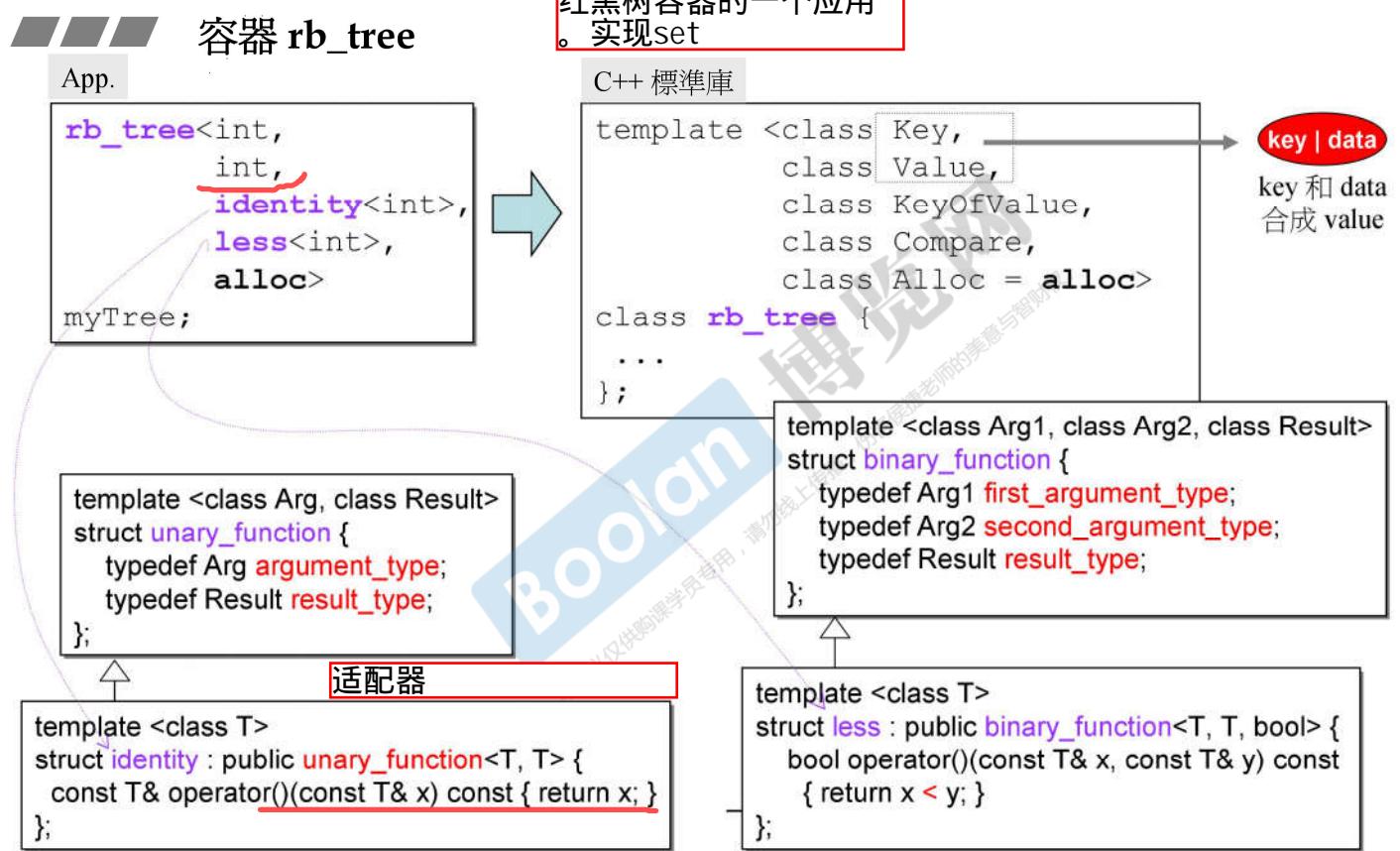
按正常規則 (`++ite`) 遍歷，便能獲得排序狀態 (sorted)。

我們不應使用 rb\_tree 的 iterators 改變元素值（因為元素有其嚴謹排列規則）。編程層面 (programming leve) 並未阻絕此事。如此設計是正確的，因為 rb\_tree 即將為 set 和 map 服務（做為其底部支持），而 map 允許元素的 data 被改變，只有元素的 key 才是不可被改變的。

rb\_tree 提供兩種 insertion 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。







## 容器 rb\_tree, 用例

G2.9

```
//測試 rb_tree
rb_tree<int, int, identity<int>, less<int>> itree;
cout << itree.empty() << endl;    //1
cout << itree.size() << endl;     //0

itree.insert_unique(3);
itree.insert_unique(8);
itree.insert_unique(5);
itree.insert_unique(9);
itree.insert_unique(13);
itree.insert_unique(5);  //no effect, since using insert_unique().
cout << itree.empty() << endl;    //0
cout << itree.size() << endl;     //5
cout << itree.count(5) << endl;   //1

itree.insert_equal(5);
itree.insert_equal(5);
cout << itree.size() << endl;    //7, since using insert_equal().
cout << itree.count(5) << endl;   //3
```

## 容器 `_Rb_tree`, 用例

G4.9

```
//測試 rb_tree
_Rb_tree<int, int, _Identity<int>, less<int>> itree;
cout << itree.empty() << endl; //1
cout << itree.size() << endl; //0

itree._M_insert_unique(3);
itree._M_insert_unique(8);
itree._M_insert_unique(5);
itree._M_insert_unique(9);
itree._M_insert_unique(13);
itree._M_insert_unique(5); //no effect, since using _M_insert_unique().
cout << itree.empty() << endl; //0
cout << itree.size() << endl; //5
cout << itree.count(5) << endl; //1

itree._M_insert_equal(5);
itree._M_insert_equal(5);
cout << itree.size() << endl; //7, since using _M_insert_equal().
cout << itree.count(5) << endl; //3
```

## 容器 \_Rb\_tree, 用例

G4.9

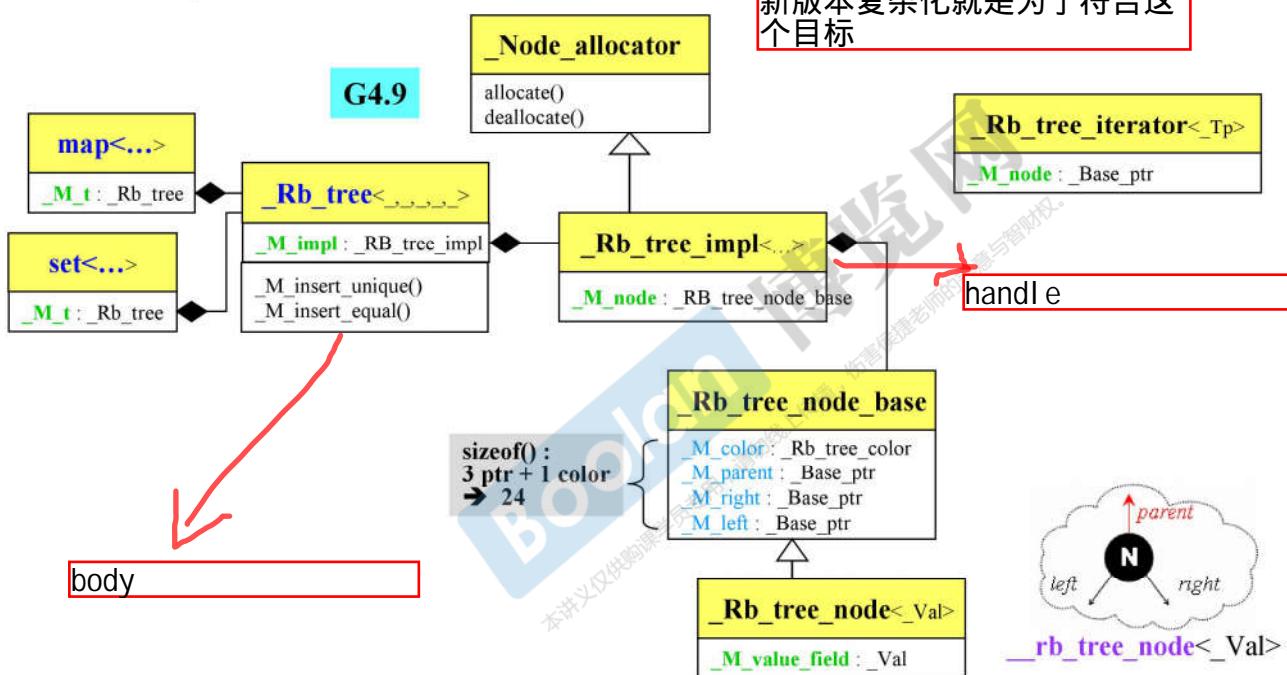
```
_Rb_tree<int, int, _Identity<int>, less<int>> itree;
cout << itree.empty() << endl; //1
cout << itree.size() << endl; //0

itree._M_insert_unique(3);
itree._M_insert_unique(8);
itree._M_insert_unique(5);
itree._M_insert_unique(9);
itree._M_insert_unique(13);
itree._M_insert_unique(5); //no effect, since using insert_unique().
cout << itree.empty() << endl; //0
cout << itree.size() << endl; //5
cout << itree.count(5) << endl; //1

itree._M_insert_equal(5);
itree._M_insert_equal(5);
cout << itree.size() << endl; //7, since using insert_equal().
cout << itree.count(5) << endl; //3
```

```
c:\ D:\handoutC++11-test... test_Rb_tree().....  
1  
0  
0  
5  
1  
7  
3
```

## 容器 \_Rb\_tree



— 侯捷 —

127

## 容器 set, multiset

set/multiset 以 rb\_tree 為底層結構，因此有「元素自動排序」特性。  
排序的依據是 key，而 set/multiset 元素的 value 和 key 合一：  
value 就是 key。

key | data  
key 和 data  
合成 value

set/multiset 提供“遍歷”操作及 iterators。  
按正常規則 (`++ite`) 遍歷，便能獲得排序狀態 (sorted)。

我們無法使用 set/multiset 的 iterators 改變元素值（因為 key 有其嚴謹排列規則）。set/multiset 的 iterator 是其底部的 RB tree 的 const-iterator，就是為了禁止 user 對元素賦值。

set 元素的 key 必須獨一無二，因此其 `insert()` 用的是 rb\_tree 的 `insert_unique()`。  
multiset 元素的 key 可以重複，因此其 `insert()` 用的是 rb\_tree 的 `insert_equal()`。

## 容器 set

```
template <class Key,
          class Compare = less<Key>,
          class Alloc = alloc>
class set {
public:
    // typedefs:
    typedef Key key_type;
    typedef Key value_type;
    typedef Compare key_compare;
    typedef Compare value_compare;
private:
    typedef rb_tree<key_type, value_type,
                    identity<value type>, key_compare, Alloc> rep_type;
    rep_type t;
public:
    typedef typename rep_type::const_iterator iterator;
...
};
```

key | data

key 和 data  
合成 value

set<int> iset;

set<  
int,  
less<int>,  
alloc  
> iset;

template <  
int,  
int,  
identity<int>,  
less<int>,  
alloc  
>  
class rb\_tree;

set 控制不能通过迭代器修改数据的地方

set 的所有操作，都轉呼叫底層 t 的操作。  
從這層意義看，set 未嘗不是個 container adapter。

## 容器 set, in VC6

VC6 不提供 identity()，那麼其 set 和 map 如何使用 RB-tree ?

```
template<class _K, class _Pr = less<_K>,
         class _A = allocator<_K>>
class set {
public:
    typedef set<_K, _Pr, _A> _Myt;
    typedef _K value_type;
    struct _Kfn : public unary_function<value_type, _K> {
        const _K& operator()(const value_type& _X) const
        {return (_X); }
    };
    typedef _Pr value_compare;
    typedef _K key_type;
    typedef _Pr key_compare;
    typedef _A allocator_type;
    typedef _Tree<_K, value_type, _Kfn, _Pr, _A> _Imp;
    ...
protected:
    _Imp _Tr;
};
```

VC6

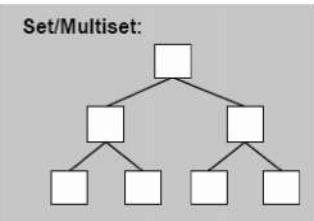
G2.9

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};
```

```
template <class T>
struct identity : public unary_function<T, T> {
    const T& operator()(const T& x) const { return x; }
};
```



## 使用容器 multiset



```
ex D:\handout\c++11-test-DevC++\Test-STL>
select: 6
how many elements: 1000000

test_multiset<>.....
milli-seconds : 6609
multiset.size()= 1000000
multiset.max_size()= 214748364
target <0~32767>: 23456
::find(), milli-seconds : 203
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

```
313 void test_multiset(long& value)
314 {
315     cout << "\ntest_multiset()..... \n";
316     !
317     multiset<string> c;
318     char buf[10];
319     clock_t timeStart = clock();
320     for(long i=0; i< value; ++i)
321     {
322         try {
323             sprintf(buf, 10, "%d", rand());
324             c.insert(string(buf));
325         }
326         catch(exception& p) {
327             cout << "i=" << i << " " << p.what() << endl; //
328             abort();
329         }
330     }
331     cout << "milli-seconds : " << (clock()-timeStart) << endl; //
332     cout << "multiset.size()= " << c.size() << endl;
333     cout << "multiset.max_size()= " << c.max_size() << endl;
334
335     string target = get_a_target_string();
336     {
337         timeStart = clock();
338         auto pItem = ::find(c.begin(), c.end(), target); //比 c.find(...) 慢很多
339         cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
340         if (pItem != c.end())
341             cout << "found, " << *pItem << endl;
342         else
343             cout << "not found! " << endl;
344     }
345     {
346         timeStart = clock(); //比 ::find(...) 快很多
347         auto pItem = c.find(target);
348         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
349         if (pItem != c.end())
350             cout << "found, " << *pItem << endl;
351         else
352             cout << "not found! " << endl;
353     }
354 }
```

## 容器 map, multimap

map/multimap 以 rb\_tree 為底層結構，因此有「元素自動排序」特性。  
排序的依據是 key。

key | data  
key 和 data  
合成 value

map/multimap 提供“遍歷”操作及 iterators。  
按正常規則 (++ite) 遍歷，便能獲得排序狀態 (sorted)。

我們無法使用 map/multimap 的 iterators 改變元素的 key  
(因為 key 有其嚴謹排列規則)，但可以用它來改變元素  
的 data。因此 map/multimap 內部自動將 user 指定的 key  
type 設為 const，如此便能禁止 user 對元素的 key 賦值。

```
template <class Key, class T, ...>
class map {
    typedef pair<const Key, T> value_type;
    typedef
        rb_tree<key_type,value_type,...> rep_type;
    rep_type t;
    ...
}
```

map 元素的 key 必須獨一無二，因此其 insert() 用的是 rb\_tree 的 insert\_unique()。  
multimap 元素的 key 可以重複，因此其 insert() 用的是 rb\_tree 的 insert\_equal()。

## 容器 map

```
template <class Key,
          class T,
          class Compare = less<Key>,
          class Alloc=alloc>
class map {
public:
    typedef Key key_type;
    typedef T data_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef Compare key_compare;
private:
    typedef rb_tree<key_type, value_type,
                    select1st<value_type>, key_compare, Alloc> rep_type;
    rep_type t;
public:
    typedef typename rep_type::iterator iterator;
...
};
```

对比set的做法



注意这里的select1st , g++独有的

imap;

map<int, string, less<int>, alloc> imap;

key 和 data  
合成 value

template <  
int;  
pair<const int, string>  
select1st<pair<const int, string>>,  
less<int>,  
alloc  
>  
class rb\_tree;

## 容器 map, in VC6

VC6 不提供 select1st()，那麼其 map 如何使用 RB-tree ?

```
template<class _K, class _Ty, class _Pr = less<_K>, VC6  
         class _A = allocator<_Ty>>  
class map {  
public:  
    typedef map<_K, _Ty, _Pr, _A> _Myt;  
    typedef pair<const _K, _Ty> value_type;  
    struct _Kfn : public unary_function<value_type, _K> {  
        const _K& operator()(const value_type& _X) const  
        {return (_X.first); }  
    };  
    ...  
    typedef _Tree<_K, value_type, _Kfn, _Pr, _A> _Imp;  
    ...  
protected:  
    _Imp _Tr;  
};
```

G2.9

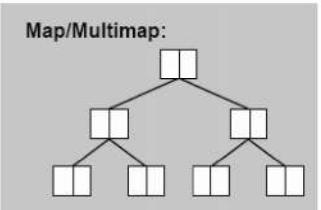
```
template <class Arg, class Result>  
struct unary_function {  
    typedef Arg argument_type;  
    typedef Result result_type;  
};
```

template <class Pair>

struct select1st :

```
public unary_function<Pair, typename Pair::first_type> {  
    const typename Pair::first_type&  
    operator()(const Pair& x) const  
    { return x.first; }  
};
```

## 使用容器 multimap



```
ex D:\handout\c++11-test-DevC++\Test-STL\m  
select: 7  
how many elements: 1000000  
  
test_multimap<>.  
milli-seconds : 4812  
multimap.size()= 1000000  
multimap.max_size()= 178956970  
target <0~32767>: 23456  
c.find(), milli-seconds : 0  
found, value=29247
```

```
364 namespace jj07  
365 {  
366     void test_multimap(long& value)  
367     {  
368         cout << "\ntest_multimap()..... \n";  
369  
370         multimap<long, string> c;  
371         char buf[10];  
372  
373         clock_t timeStart = clock();  
374         for(long i=0; i< value; ++i)  
375         {  
376             try {  
377                 snprintf(buf, 10, "%d", rand());  
378                 //multimap 不可使用 [] 做 insertion  
379                 c.insert(pair<long,string>(i,buf));  
380             }  
381             catch(exception& p) {  
382                 cout << "i=" << i << " " << p.what() << endl; //  
383                 abort();  
384             }  
385         }  
386         cout << "milli-seconds : " << (clock()-timeStart) << endl; //  
387         cout << "multimap.size()= " << c.size() << endl;  
388         cout << "multimap.max_size()= " << c.max_size() << endl;  
389  
390         long target = get_a_target_long();  
391         timeStart = clock();  
392         auto pItem = c.find(target);  
393         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;  
394         if (pItem != c.end())  
395             cout << "found, value=" << (*pItem).second << endl;  
396         else  
397             cout << "not found! " << endl;  
398     }  
399 }
```

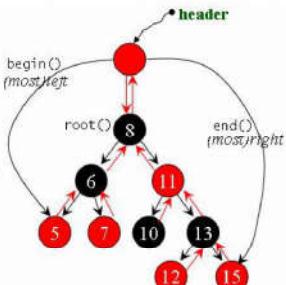
## 容器 map, 獨特的 operator[ ]

G4.9

```

mapped_type&
operator[](const key_type& __k)
{
    // concept requirements
    __glbcxx_function_requires(_DefaultConstructibleCon
    iterator __i = lower_bound(__k);
    // __i->first is greater than or equivalent to __k.
    if (__i == end() || key_comp)(__k, (*__i).first))
#if __cplusplus >= 201103L
        __i = _M_t._M_emplace_hint_unique(__i, std::piecewise_construct
                                            std::tuple<const key_type>
                                            std::tuple<>());
#else
        __i = insert(__i, value_type(__k, mapped_type()));
#endif
    return (*__i).second;
}

```



// [23.3.1.2] element access

```

/**
 * @brief Subscript (@c []) access to %map data.
 * @param __k The key for which data should be retrieved.
 * @return A reference to the data of the (key,data) %pair.
 *
 * Allows for easy lookup with the subscript (@c [])
 * operator. Returns data associated with the key specified in
 * subscript. If the key does not exist, a pair with that key
 * is created using default values, which is then returned.
 *
 * Lookup requires logarithmic time.
 */

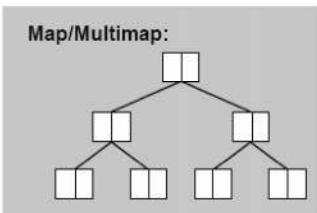
```

- `lower_bound` 是二分搜尋 (binary search) 的一種版本，試圖在 sorted  $[first, last]$  中尋找元素 `value`。若  $[first, last]$  擁有與 `value` 相等的元素(s)，便返回一個 `iterator` 指向其中第一個元素。如果沒有這樣的元素存在，便返回「假設該元素存在時應該出現的位置」。也就是說它會返回 `iterator` 指向第一個「不小於 `value`」的元素。如果 `value` 大於  $[first, last]$  內的任何元素，將返回 `last`。換句話說 `lower_bound` 返回的是「不破壞排序得以安插 `value` 的第一個適當位置」。

— 侯捷 —



## 使用容器 map



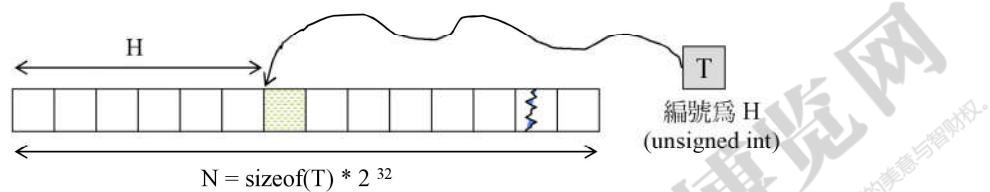
```
ex D:\handout\c++11-test-DevC++\Test-STI
select: 14
how many elements: 1000000

test_map().....
milli-seconds : 4890
map.size()= 1000000
map.max_size()= 178956970
target <0~32767>: 23456
c.find(), milli-seconds : 0
found, value=19128
```

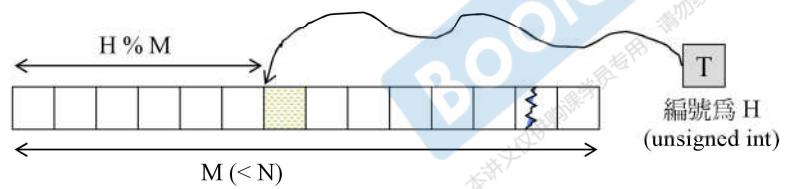
```
696 namespace jj14
697 {
698     void test_map(long& value)
699     {
700         cout << "\ntest_map()..... \n";
701
702         map<long, string> c;
703         char buf[10];
704
705         clock_t timeStart = clock();
706         for(long i=0; i< value; ++i)
707         {
708             try
709             {
710                 sprintf(buf, 10, "%d", rand());
711                 c[i] = string(buf);
712             }
713             catch(exception& p)
714             {
715                 cout << "i=" << i << " " << p.what() << endl;    //
716                 abort();
717             }
718         }
719         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
720         cout << "map.size()= " << c.size() << endl;
721         cout << "map.max_size()= " << c.max_size() << endl;
722
723         long target = get_a_target_long();
724         timeStart = clock();
725         auto pItem = c.find(target);
726         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
727         if (pItem != c.end())
728             cout << "found, value=" << (*pItem).second << endl;
729         else
730             cout << "not found! " << endl;
731     }
732 }
```

## 容器 hashtable

空間足夠時：

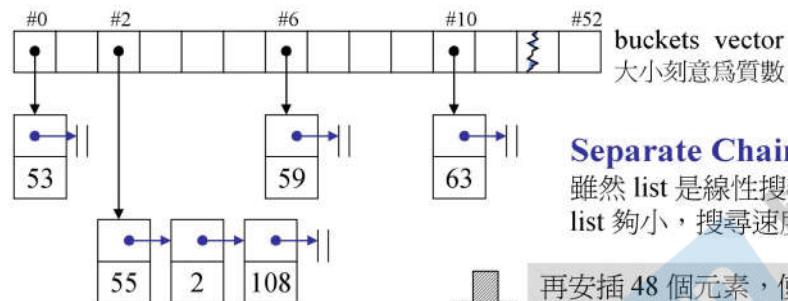


空間不足時：



## 容器 hashtable

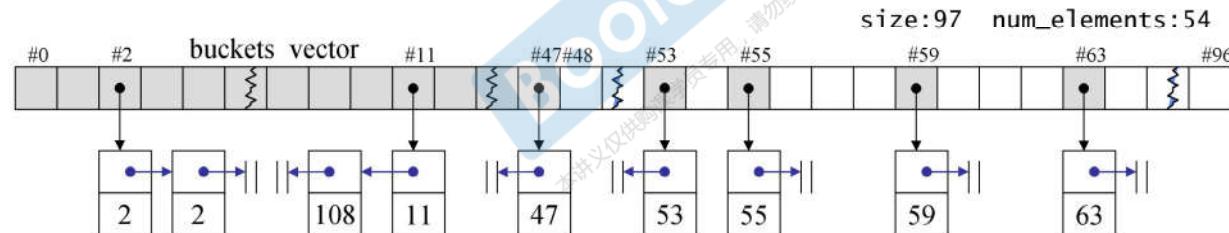
`insert_unique()`, 次序: 59, 63, 108, 2, 53, 55 num\_elements: 6



### 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 53，於是 rehashing

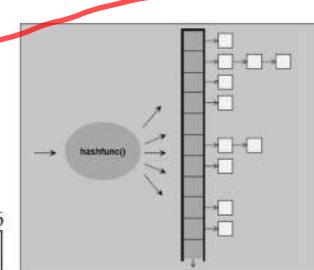


我們可以使用 hashtable iterators 改變元素的 data，但不能改變元素的 key  
(因為 hashtable 根據 key 實現嚴謹的元素排列)。

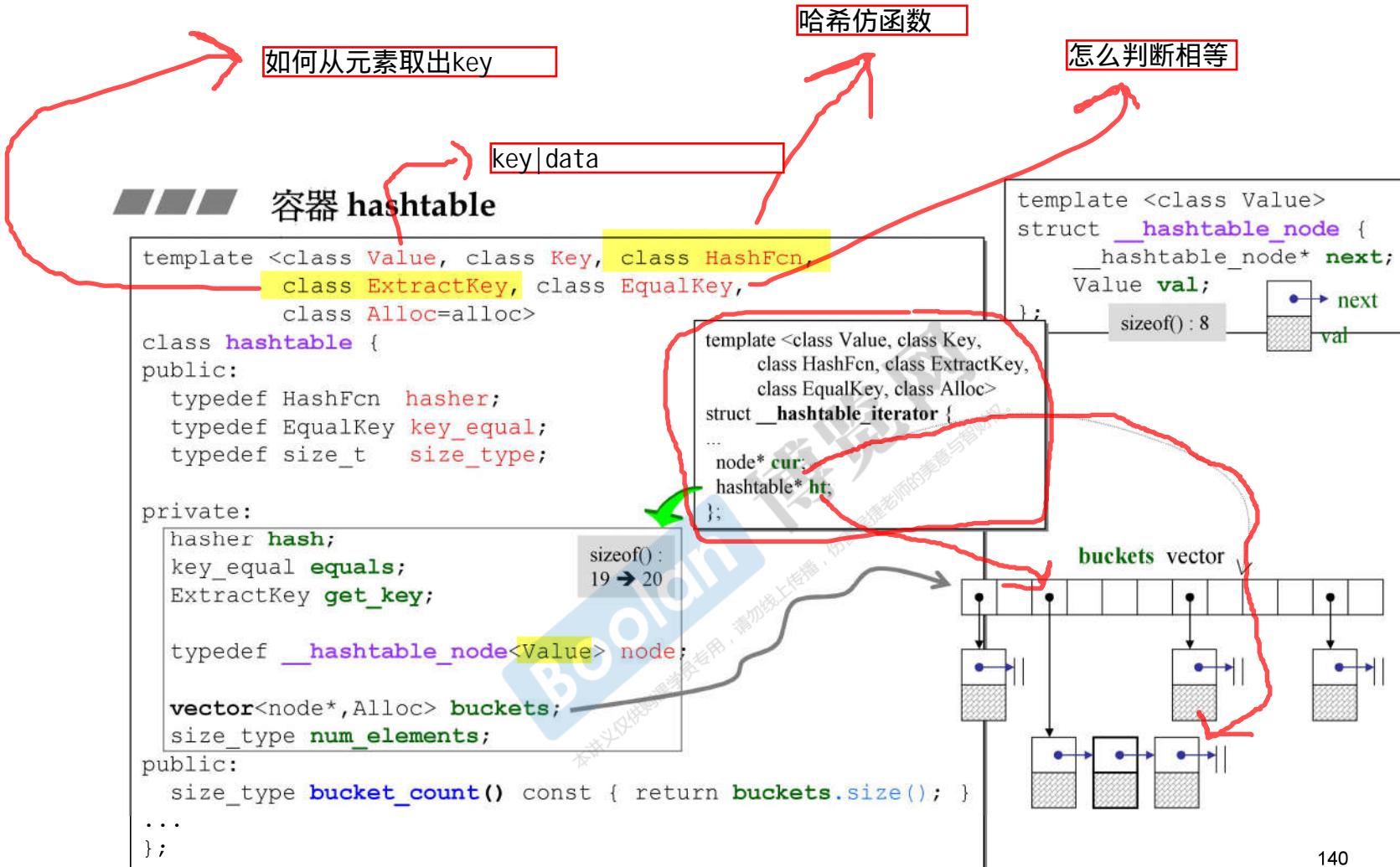
```
static const unsigned long
__stl_prime_list[__stl_num_primes] =
{
    53, 97, 193, 389, 769,
    1543, 3079, 6151, 12289, 24593,
    49157, 98317, 196613, 393241, 786433,
    1572869, 3145739, 6291469, 12582917, 25165843,
    50331653, 100663319, 201326611, 402653189,
    805306457, 1610612741, 3221225473ul, 4294967291ul
};
```

G2.9

x G4.9



扩充到两倍后最近的质数，53-->97。扩充后将原有各个值打散，所有元素重新算一遍



## 容器 hashtable

App.

```
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");
```

比較兩個 c-string 是否相等，有 strcmp() 可用，但它傳回 -1,0,1，不是傳回 bool，所以必須加一層外套

```
struct eqstr {  
    bool operator()(const char* s1,  
                    const char* s2) const  
    { return strcmp(s1,s2) == 0; }  
};
```

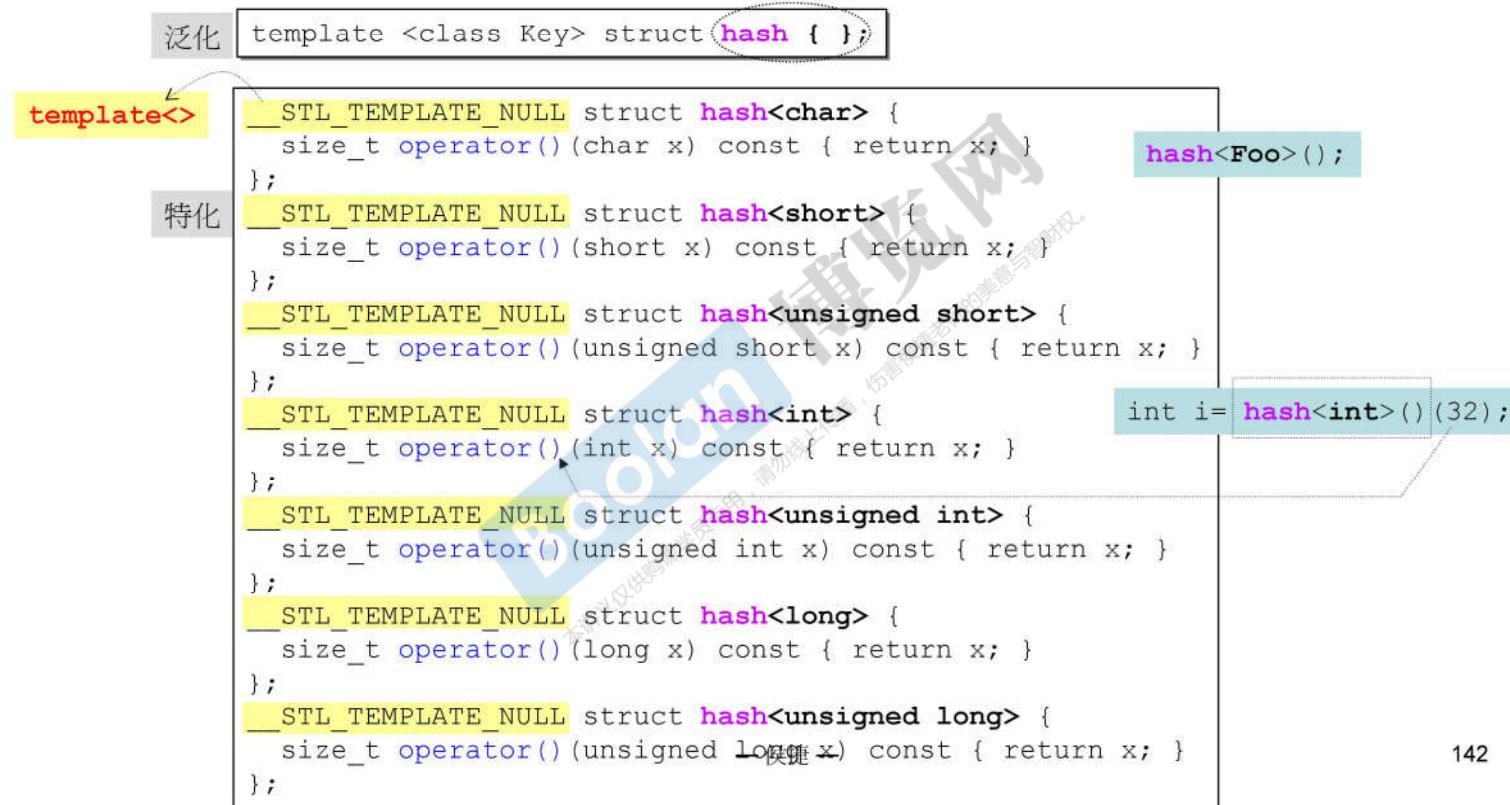
C++ 標準庫

```
template <class Value,  
         class Key,  
         class HashFcn,  
         class ExtractKey,  
         class EqualKey,  
         class Alloc=alloc>  
class hashtable {  
public:  
    typedef HashFcn    hasher;  
    typedef EqualKey   key_equal;  
    typedef size_t     size_type;  
private:  
    hasher    hash;  
    key_equal equals;  
    ExtractKey get_key;  
    ...
```

下頁

— 侯捷 —

## hash-function, hash-code



## hash-function, hash-code

```
inline size_t __stl_hash_string(const char* s)
{
    unsigned long h = 0;
    for ( ; *s; ++s) {
        h = 5*h + *s;           // 本例若 s 指向 "abc"，計算得 h 為
                                // 5 * (5 * 'a' + 'b') + 'c'
    }
    return size_t(h);
}
__STL_TEMPLATE_NULL struct hash<char*>
{
    size_t operator()(const char* s) const { return __stl_hash_string(s); }
};
__STL_TEMPLATE_NULL struct hash<const char*>
{
    size_t operator()(const char* s) const { return __stl_hash_string(s); }
};
```

注意，標準庫 **G2.9** 沒有提供現成的  
`hash<std::string>`  
**G4.9** 有提供，在  
...\\4.9.2\\include\\c++\\bits\\basic\_string.h

hash function 的目的，就是希望根據元素值算出一個 hash code（一個可進行 modulus 運算的值），使得元素經 hash code 映射之後能夠「夠雜夠亂夠隨機」地被置於 hashtable 內。愈是雜亂，愈不容易發生碰撞。

## modulus 運算

```
iterator find(const key_type& key)
{ size_type n = bkt_num_key(key); //花落何家
...
size_type count(const key_type& key) const
{ const size_type n = bkt_num_key(key);
...
template <class V, class K, class HF, class ExK,
          class EqK, class A>
__hashtable_iterator<V, K, HF, ExK, EqK, A>&
__hashtable_iterator<V, K, HF, ExK, EqK, A>::operator++()
{
...
size_type bucket = ht->bkt_num(old->val);
...
}
```

1 bkt\_num\_key(const key\_type& key) const  
{  
 return bkt\_num\_key(key, buckets.size());  
}  
size\_type  
bkt\_num(const value\_type& obj) const  
{  
 return bkt\_num\_key(get\_key(obj));  
}

template <>  
struct hash<char\*>  
{ ... };

size\_type  
bkt\_num\_key(const key\_type& key, size\_t n) const  
{  
 return hash(key) % n;  
}  
這個 hash 不是前頁出現的 struct hash ,  
而是 class hashtable 中的 hasher hash 。  
size\_type  
bkt\_num(const value\_type& obj, size\_t n) const  
{  
 return bkt\_num\_key(get\_key(obj), n);  
}

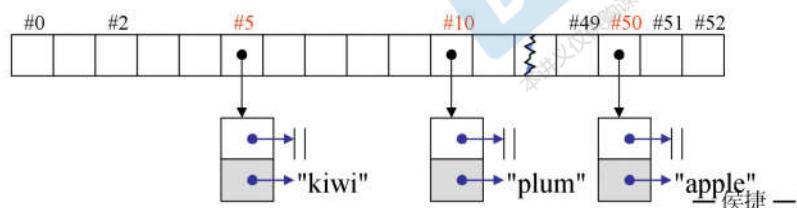
template <...>  
class hashtable {  
private:  
 hasher hash;  
... };

—侯捷—

## hash-function, hash-code

App.

```
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");
```



$$\begin{aligned}\text{hash code of "kiwi"} &= 5 * (5 * (5 * 'k' + 'i') + 'w') + 'i' \\ &= 5 * (5 * (5 * 107 + 105) + 119) + 105 \\ &= 16700 \% 53 = 5\end{aligned}$$

$$\begin{aligned}\text{hash code of "plum"} &= 5 * (5 * (5 * 'p' + 'l') + 'u') + 'm' \\ &= 5 * (5 * (5 * 112 + 108) + 117) + 109 \\ &= 17394 \% 53 = 10\end{aligned}$$

$$\begin{aligned}\text{hash code of "apple"} &= 5 * (5 * (5 * (5 * 'a' + 'p') + 'p') + 'l') + 'e' \\ &= 5 * (5 * (5 * (5 * 97 + 112) + 112) + 108) + 101 \\ &= 78066 \% 53 = 50\end{aligned}$$

↑

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

## ///// hashtable 用例 G2.9

```
7 template<> struct hash<string>
8 {
9     size_t operator()(string s) const {
10         return __stl_hash_string(s.c_str());
11     }
12 };
```

```
50 hashtable< string,
51             string,
52             hash<string>,
53             identity<string>,
54             equal_to<string>,
55             alloc
56         >
57         sht(50, hash<string>(), equal_to<string>());
58
59 cout<< sht.size() << endl;           // 0
60 cout<< sht.bucket_count() << endl;    // 53
61
62 // my goal!
63 hashtable< pair<const string,int>,
64             string,
65             hash<string>,
66             selectist< pair<const string,int> >,
67             equal_to<string>,
68             alloc
69         >
70         siht(100, hash<string>(), equal_to<string>());
71
72 cout<< siht.size() << endl;           // 0
73 cout<< siht.bucket_count() << endl;    // 193
74
75 siht.insert_unique( make_pair(string("jjhou"),95) );
76 siht.insert_unique( make_pair(string("sabrina"),90) );
77 siht.insert_unique( make_pair(string("mjchen"),85) );
78 cout<< siht.size() << endl;           // 3
79 cout<< siht.bucket_count() << endl;    // 193
80 cout << siht.find(string("sabrina"))->second << endl; //90
81 cout << siht.find(string("jjhou"))->second << endl;   //95
82 cout << siht.find(string("mjchen"))->second << endl; //85
```

## hashtable 用例 G4.9

```
1355     _Hashtable< string,
1356             string,
1357             hash<string>,
1358             _Identity<string>,
1359             equal_to<string>,
1360             > // [Error] wrong number of template arguments (6, should be 10)
1361     sht(50, hash<string>(), equal_to<string>());
1362
1363     cout<< sht.size() << endl;           // 0
1364     cout<< sht.bucket_count() << endl;    // 53
1365
1366 // my goal!
1367     _Hashtable< pair<const string,int>,
1368             string,
1369             hash<string>,
1370             _Selectlist< pair<const string,int> >,
1371             equal_to<string>,
1372             > // [Error] wrong number of template arguments (6, should be 10)
1373     siht(100, hash<string>(), equal_to<string>());
1374
1375     cout<< siht.size() << endl;           // 0
1376     cout<< siht.bucket_count() << endl;    // 193
1377
1378     siht.insert_unique( make_pair(string("jjhou"),95) );
1379     siht.insert_unique( make_pair(string("sabrina"),90) );
1380     siht.insert_unique( make_pair(string("mjchen"),85) );
1381     cout<< siht.size() << endl;           // 3
1382     cout<< siht.bucket_count() << endl;    // 193
1383     cout << siht.find(string("sabrina"))->second << endl; //90
1384     cout << siht.find(string("jjhou"))->second << endl;   //95
1385     cout << siht.find(string("mjchen"))->second << endl; //85
```

```
/// std::hash specialization for string.
template<>
...\\4.9.2\\include\\c++\\bits\\basic_string.h
► struct hash<string>
: public __hash_base<size_t, string>
{
size_t
operator()(const string& __s) const noexcept
{ return std::__Hash_impl::hash(__s.data(), __s.length()); }
};

template<>
struct __is_fast_hash<hash<string>> : std::false_type
{ };
```



容器 **hash\_set, hash\_multiset**

容器 **hash\_map, hash\_multimap**



## ||||| **unordered 容器**

Before C++11

- **hash\_set**
- **hash\_multiset**
- **hash\_map**
- **hash\_multimap**



Since C++11

- **unordered\_set**
- **unordered\_multiset**
- **unordered\_map**
- **unordered\_multimap**

G4.9

```
template <typename T,
         typename Hash = hash<T>,
         typename EqPred = equal_to<T>,
         typename Allocator = allocator<T> >
class unordered_set;
```

```
template <typename T,
         typename Hash = hash<T>,
         typename EqPred = equal_to<T>,
         typename Allocator = allocator<T> >
class unordered_multiset;
```

```
template <typename Key, typename T,
         typename Hash = hash<T>,
         typename EqPred = equal_to<T>,
         typename Allocator = allocator<pair<const Key, T> > >
class unordered_map;
```

```
template <typename Key, typename T,
         typename Hash = hash<T>,
         typename EqPred = equal_to<T>,
         typename Allocator = allocator<pair<const Key, T> > >
class unordered_multimap;
```



容器 `unordered_set`, `unordered_multiset`

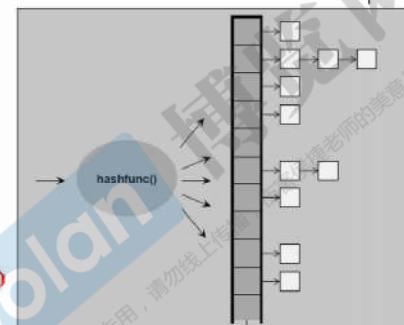
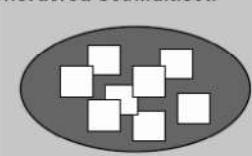
容器 `unordered_map`, `unordered_multimap`



## 使用容器 `unordered_set`

```
739 namespace jj15
740 {
741     void test_unordered_set(long& value)
742     {
743         cout << "\ntest_unordered_set().....\n";
744
745         unordered_set<string> c;
746         char buf[10];
747
748         clock_t timeStart = clock();
749         for(long i=0; i< value; ++i)
750         {
751             try {
752                 sprintf(buf, 10, "%d", rand());
753                 c.insert(string(buf));
754             }
755             catch(exception& p) {
756                 cout << "i=" << i << " " << p.what();
757                 abort();
758             }
759         }
760         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
761         cout << "unordered_set.size()= " << c.size() << endl;
762         cout << "unordered_set.max_size()= " << c.max_size() << endl;
763         cout << "unordered_set.bucket_count()= " << c.bucket_count() << endl;
764         cout << "unordered_set.load_factor()= " << c.load_factor() << endl;
765         cout << "unordered_set.max_load_factor()= " << c.max_load_factor() << endl;
766         cout << "unordered_set.max_bucket_count()= " << c.max_bucket_count() << endl;
767         for (unsigned i=0; i< 20; ++i) {
768             cout << "bucket #" << i << " has " << c.bucket_size(i) << " elements.\n";
769         }
    }
```

Unordered Set/Multiset:



```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 15
how many elements: 1000000

test_unordered_set().....
milli-seconds : 2891
unordered_set.size()= 32768
unordered_set.max_size()= 357913941
unordered_set.bucket_count()= 62233
unordered_set.load_factor()= 0.526537
unordered_set.max_load_factor()= 1
unordered_set.max_bucket_count()= 357913941
bucket #0 has 1 elements.
bucket #1 has 1 elements.
bucket #2 has 1 elements.
bucket #3 has 0 elements.
bucket #4 has 3 elements.
bucket #5 has 0 elements.
bucket #6 has 0 elements.
bucket #7 has 1 elements.
bucket #8 has 0 elements.
bucket #9 has 0 elements.
bucket #10 has 0 elements.
bucket #11 has 1 elements.
bucket #12 has 0 elements.
bucket #13 has 2 elements.
bucket #14 has 1 elements.
bucket #15 has 1 elements.
bucket #16 has 1 elements.
bucket #17 has 1 elements.
bucket #18 has 0 elements.
bucket #19 has 0 elements.
target <0^32767>: 23456
::find(), milli-seconds : 0
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

## 使用容器 `unordered_set`

```
770 string target = get_a_target_string();
771 {
772     timeStart = clock();
773     auto pItem = ::find(c.begin(), c.end(), target); // 比 c.find(...) 慢很多
774     cout << "::find(), milli-seconds : " << (clock() - timeStart) << endl;
775     if (pItem != c.end())
776         cout << "found, " << *pItem << endl;
777     else
778         cout << "not found! " << endl;
779 }
780
781 {
782     timeStart = clock();
783     auto pItem = c.find(target); // 比 ::find(...) 快很多
784     cout << "c.find(), milli-seconds : " << (clock() - timeStart) << endl;
785     if (pItem != c.end())
786         cout << "found, " << *pItem << endl;
787     else
788         cout << "not found! " << endl;
789 }
790
791 }
```

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 15
how many elements: 1000000

test_unordered_set<>.....
milli-seconds : 2891
unordered_set.size()= 32768
unordered_set.max_size()= 357913941
unordered_set.bucket_count()= 62233
unordered_set.load_factor()= 0.526537
unordered_set.max_load_factor()= 1
unordered_set.max_bucket_count()= 357913941
bucket #0 has 1 elements.
bucket #1 has 1 elements.
bucket #2 has 1 elements.
bucket #3 has 0 elements.
bucket #4 has 3 elements.
bucket #5 has 0 elements.
bucket #6 has 0 elements.
bucket #7 has 1 elements.
bucket #8 has 0 elements.
bucket #9 has 0 elements.
bucket #10 has 0 elements.
bucket #11 has 1 elements.
bucket #12 has 0 elements.
bucket #13 has 2 elements.
bucket #14 has 1 elements.
bucket #15 has 1 elements.
bucket #16 has 1 elements.
bucket #17 has 1 elements.
bucket #18 has 0 elements.
bucket #19 has 0 elements.
target <0x32767>: 23456
::find(), milli-seconds : 0
found, 23456
c.find(), milli-seconds : 0
found, 23456
```



The End



# C++ 標準庫

## 體系結構與內核分析

(C++ Standard Library — architecture & sources)

第三講



侯捷

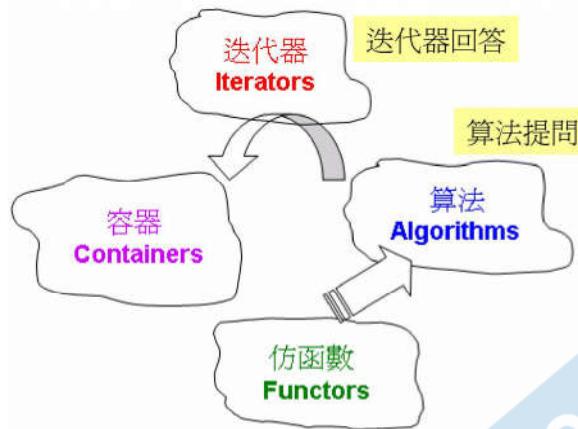
— 侯捷 —

154

源碼之前  
了無秘密



## ■■■■ C++標準庫的算法，是什麼東西？



Algorithms 看不見 Containers，對其一無所知；所以，它所需要的一切信息都必須從 Iteratrors 取得，而 Iterators (由 Containers 供應) 必須能夠回答 Algorithm 的所有提問，才能搭配該 Algorithm 的所有操作。

— 侯捷 —

如果没法回答，编译到  
那一行将报错

從語言層面講

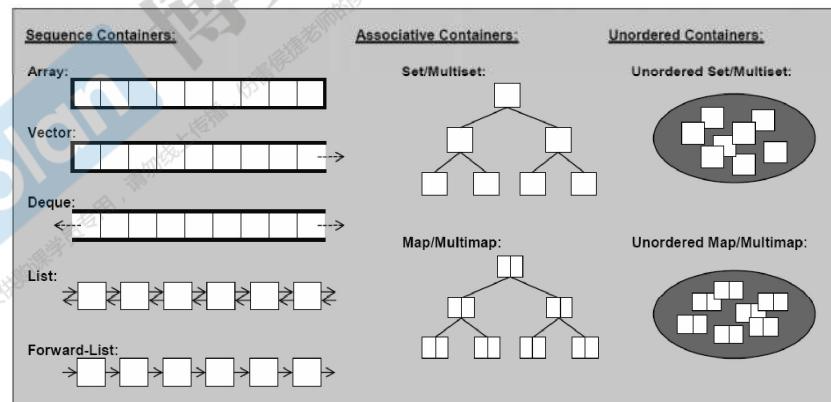
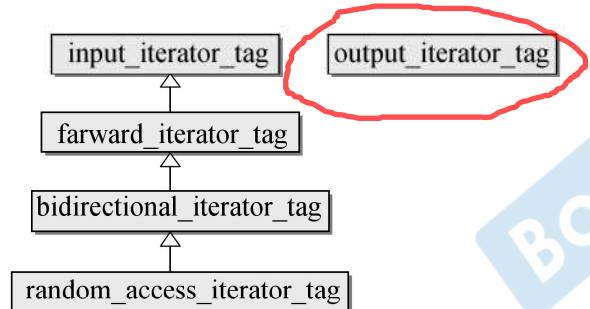
- 容器 Container 是個 class template
- 算法 Algorithm 是個 function template
- 迭代器 Iterator 是個 class template
- 仿函數 Functor 是個 class template
- 適配器 Adapter 是個 class template
- 分配器 Allocator 是個 class template

```
template<typename Iterator>
Algorithm(Iterator itr1, Iterator itr2)
{
    ...
}
```

```
template<typename Iterator, typename Cmp>
Algorithm(Iterator itr1, Iterator itr2, Cmp comp)
{
    ...
}
```

## ■■■ 各種容器的 iterators 的 iterator\_category

```
// 五種 iterator category
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```



## ■■■■ 各種容器的 iterators 的 iterator\_category



```
void _display_category(random_access_iterator<I>)
{ cout << "random_access_iterator" << endl; }
void _display_category(bidirectional_iterator<I>)
{ cout << "bidirectional_iterator" << endl; }
void _display_category(forward_iterator_tag)
{ cout << "forward_iterator" << endl; }
void _display_category(output_iterator_tag)
{ cout << "output_iterator" << endl; }
void _display_category(input_iterator_tag)
{ cout << "input_iterator" << endl; }

template<typename I>
void display_category(I itr)
{
    typename iterator_traits<I>::iterator_category cagy;
    _display_category(cagy);
}
```

```
cout << "\ntest_iterator_category()..... \n";
display_category(array<int,10>::iterator());
display_category(vector<int>::iterator());
display_category(list<int>::iterator());
display_category(forward_list<int>::iterator());
display_category(deque<int>::iterator());

display_category(set<int>::iterator());
display_category(map<int,int>::iterator());
display_category(multiset<int>::iterator());
display_category(multimap<int,int>::iterator());
display_category(unordered_set<int>::iterator());
display_category(unordered_map<int,int>::iterator());
display_category(unordered_multiset<int>::iterator());
display_category(unordered_multimap<int,int>::iterator());

display_category(istream_iterator<int>());
display_category(ostream_iterator<int>(cout,""));
```

```
D:\handout\c++11-test...
random_access_iterator
random_access_iterator
bidirectional_iterator
forward_iterator
random_access_iterator
bidirectional_iterator
bidirectional_iterator
bidirectional_iterator
forward_iterator
forward_iterator
forward_iterator
input_iterator
output_iterator
```

— 侯捷 —

## ■■■ 各種容器的 iterators 的 iterator\_category 的 typeid

```
#include <typeinfo> // typeid

void _display_category(random_access_iterator_tag)
{
    cout << "random_access_iterator" << endl; }

void _display_category(bidirectional_iterator_tag)
{
    cout << "bidirectional_iterator" << endl; }

void _display_category(forward_iterator_tag)
{
    cout << "forward_iterator" << endl; }

void _display_category(output_iterator_tag)
{
    cout << "output_iterator" << endl; }

void _display_category(input_iterator_tag)
{
    cout << "input_iterator" << endl; }

template<typename I>
void display_category(I itr)
{
    typename iterator_traits<I>::iterator_category cagy;
    _display_category(cagy);

    cout << "typeid(itr).name()= " << typeid(itr).name() << endl << endl;
    //The output depends on library implementation.
    //The particular representation pointed by the
    //returned value is implementation-defined.
    //and may or may not be different for different types.
}
```

```
random_access_iterator
typeid(itr).name()= Pi

random_access_iterator
typeid(itr).name()= N9_gnu_cxx17_normal_iteratorIPiSt6vectorIiSaIiEEEE

bidirectional_iterator
typeid(itr).name()= St14_List_iteratorIiE

forward_iterator
typeid(itr).name()= St18_Fwd_list_iteratorIiE

random_access_iterator
typeid(itr).name()= St15_Deque_iteratorIiRiPiE

bidirectional_iterator
typeid(itr).name()= St23_Rb_tree_const_iteratorIiE

bidirectional_iterator
typeid(itr).name()= St17_Rb_tree_iteratorISt4pairIKiiEE

bidirectional_iterator
typeid(itr).name()= St23_Rb_tree_const_iteratorIiE

bidirectional_iterator
typeid(itr).name()= St17_Rb_tree_iteratorISt4pairIKiiEE

forward_iterator
typeid(itr).name()= NSt8_detailI4_Node_iteratorIiLb1ELb0EEEE

forward_iterator
typeid(itr).name()= NSt8_detailI4_Node_iteratorISt4pairIKiiELb0ELb0EEEE

forward_iterator
typeid(itr).name()= NSt8_detailI4_Node_iteratorIiLb1ELb0EEEE

forward_iterator
typeid(itr).name()= NSt8_detailI4_Node_iteratorISt4pairIKiiELb0ELb0EEEE

input_iterator
typeid(itr).name()= St16istream_iteratorIicSt11char_traitsIcEiE

output_iterator
typeid(itr).name()= St16ostream_iteratorIicSt11char_traitsIcEE
```

— 侯捷 —

## istream\_iterator 的 iterator\_category

```
display_category(istream_iterator<int>());
display_category(ostream_iterator<int>(cout, ""));
```

```
template<typename _Category,
         typename _Tp,
         typename _Distance = ptrdiff_t,
         typename _Pointer = _Tp*,
         typename _Reference = _Tp&>
struct iterator
{
    typedef _Category iterator_category;
    typedef _Tp      value_type;
    typedef _Distance difference_type;
    typedef _Pointer pointer;
    typedef _Reference reference;
};
```

```
template<typename _Tp,
         typename _CharT = char,
         typename _Traits = char_traits<_CharT>,
         typename _Dist = ptrdiff_t>
class istream_iterator
    : public iterator<input_iterator_tag, _Tp, _Dist, const _Tp*, const _Tp&>
{
```

1      2      3      4      5

**G2.9**

```
template <class T,
          class Distance = ptrdiff_t>
class istream_iterator {
public:
    typedef input_iterator_tag iterator_category;
    ...
}
```

**G3.3**

```
template <class _Tp,
          class _CharT = char,
          class _Traits = char_traits<_CharT>,
          class _Dist = ptrdiff_t>
class istream_iterator {
public:
    typedef input_iterator_tag iterator_category;
    ...
}
```

`random_access_iterator  
random_access_iterator  
bidirectional_iterator  
forward_iterator  
random_access_iterator  
bidirectional_iterator  
bidirectional_iterator  
bidirectional_iterator  
forward_iterator  
forward_iterator  
forward_iterator  
forward_iterator  
input_iterator  
output_iterator`

## ostream\_iterator 的 iterator\_category

```
display_category(istream_iterator<int>());
display_category(ostream_iterator<int>(cout, ""));
```

```
template<typename _Category,
         typename _Tp,
         typename _Distance = ptrdiff_t,
         typename _Pointer = _Tp*,
         typename _Reference = _Tp&>
struct iterator
{
    typedef _Category iterator_category;
    typedef _Tp value_type;
    typedef _Distance difference_type;
    typedef _Pointer pointer;
    typedef _Reference reference;
};
```

```
template<typename _Tp, typename _CharT = char,
         typename _Traits = char_traits<_CharT> >
class ostream_iterator
: public iterator<output_iterator_tag, void, void, void, void>
{
```

G2.9

```
template <class T>
class ostream_iterator {
public:
    typedef output_iterator_tag iterator_category;
```

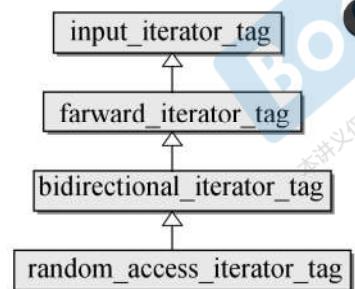
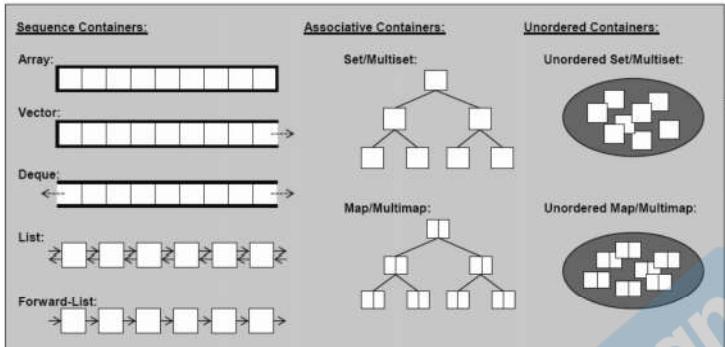
G3.3

```
template <class _Tp,
         class _CharT = char,
         class _Traits = char_traits<_CharT> >
class ostream_iterator {
public:
    typedef output_iterator_tag iterator_category;
    ...
```

D:\handout\c++11-test... \D:\

```
random_access_iterator
random_access_iterator
bidirectional_iterator
forward_iterator
random_access_iterator
bidirectional_iterator
bidirectional_iterator
bidirectional_iterator
bidirectional_iterator
forward_iterator
forward_iterator
forward_iterator
forward_iterator
input_iterator
output_iterator
```

## iterator\_category 對 算法的影響



如果是random的，直接  
减了就可以得到距离了  
，如果不是需要 $O(n)$ ，  
通过 $++$

```

template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last,
           input_iterator_tag) {
    iterator_traits<InputIterator>::difference_type n = 0;
    while (first != last) {
        ++first; ++n;
    }
    return n;
}

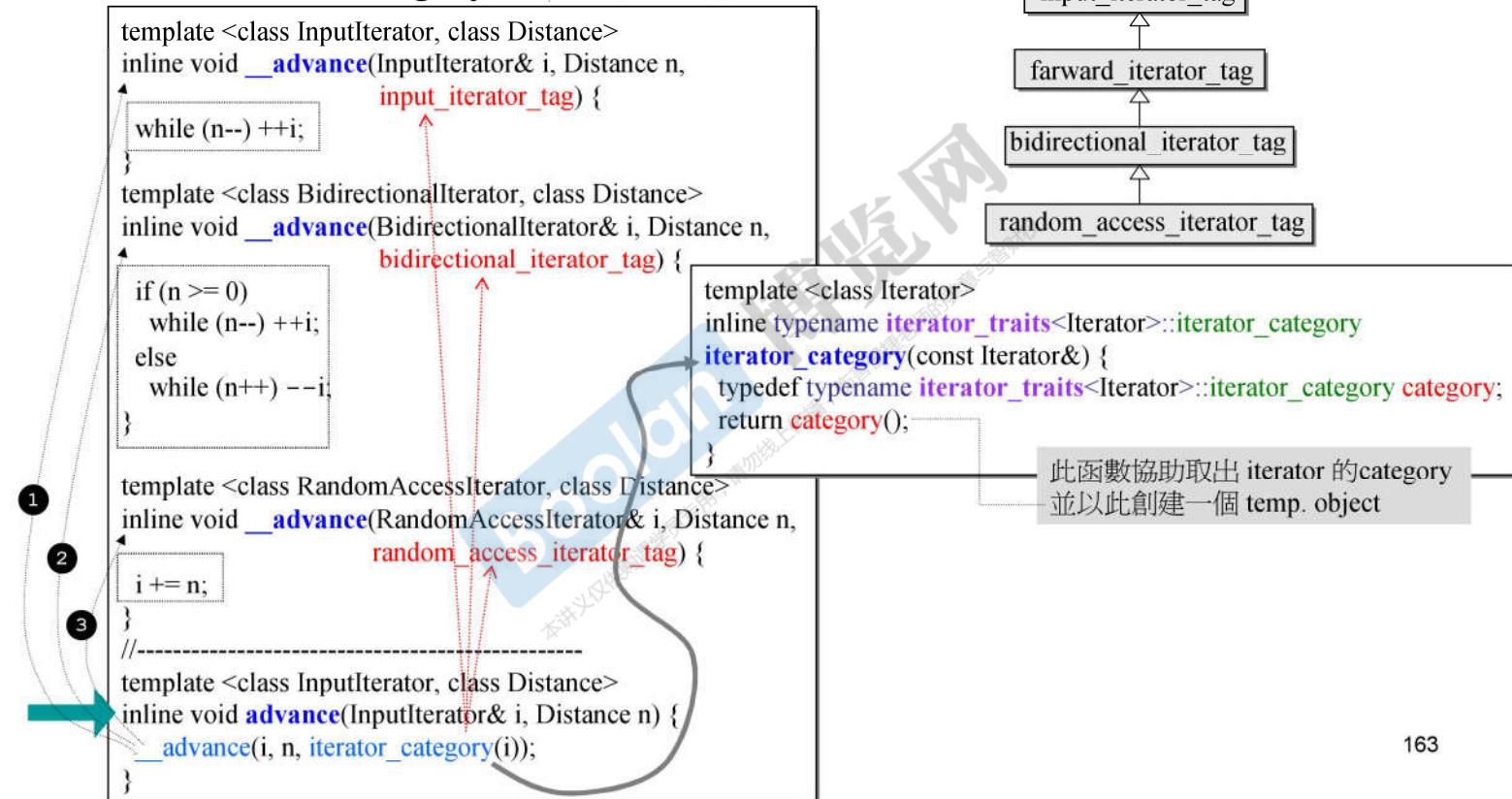
template <class RandomAccessIterator>
inline iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first, RandomAccessIterator last,
           random_access_iterator_tag) {
    return last - first;
}
// -----
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last) {
    typedef typename iterator_traits<InputIterator>::iterator_category category;
    return __distance(first, last, category());
}
```

temp. object

萃取机，对迭代器进行  
“提问”。牛逼



## iterator\_category 對算法的影響



太牛逼了

## iterator\_category 和 type traits 對算法的影響

```
template<class InputIterator,  
        class OutputIterator>  
OutputIterator  
copy (InputIterator first,  
      InputIterator last,  
      OutputIterator result)  
{  
    while (first!=last) {  
        *result = *first;  
        ++result; ++first;  
    }  
    return result;  
}
```



function template 沒有所謂特化；  
這兒用的是重載手法。

泛化：generalization  
特化：specialization  
強化：refinement

泛化 → `_copy_dispatch()`  
`<InputIterator, InputIterator>`

特化 → `memmove()`  
`(const char*, const char*)` 低階動作  
速度極快

特化 → `memmove()`  
`(const wchar_t*, const wchar_t*)`

泛化 → `_copy()`  
`<InputIterator, InputIterator>`

特化 → `_copy_t()`  
`<T*, T*>`

特化 → `_copy_t()`  
`<const T*, T*>`

以 iterators 是否相等來決定  
for-loop 是否繼續；速度較慢

for(; first != last; ...)  
`<InputIterator, InputIterator>`

強化 → `<RandomAccessIterator, RandomAccessIterator>`

強化 → `memmove()`  
`has trivial op=()`

`has non-trivial op=()`

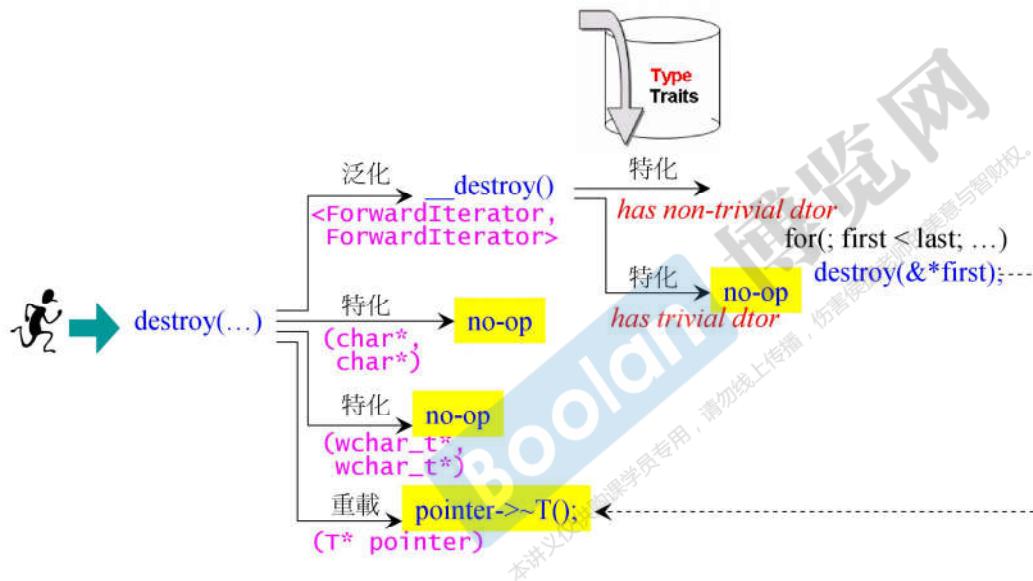
`_copy_d()`  
for(...; n < last-first; ...)  
以 n 決定 for-loop 次數；  
速度較快

— 侯捷 —

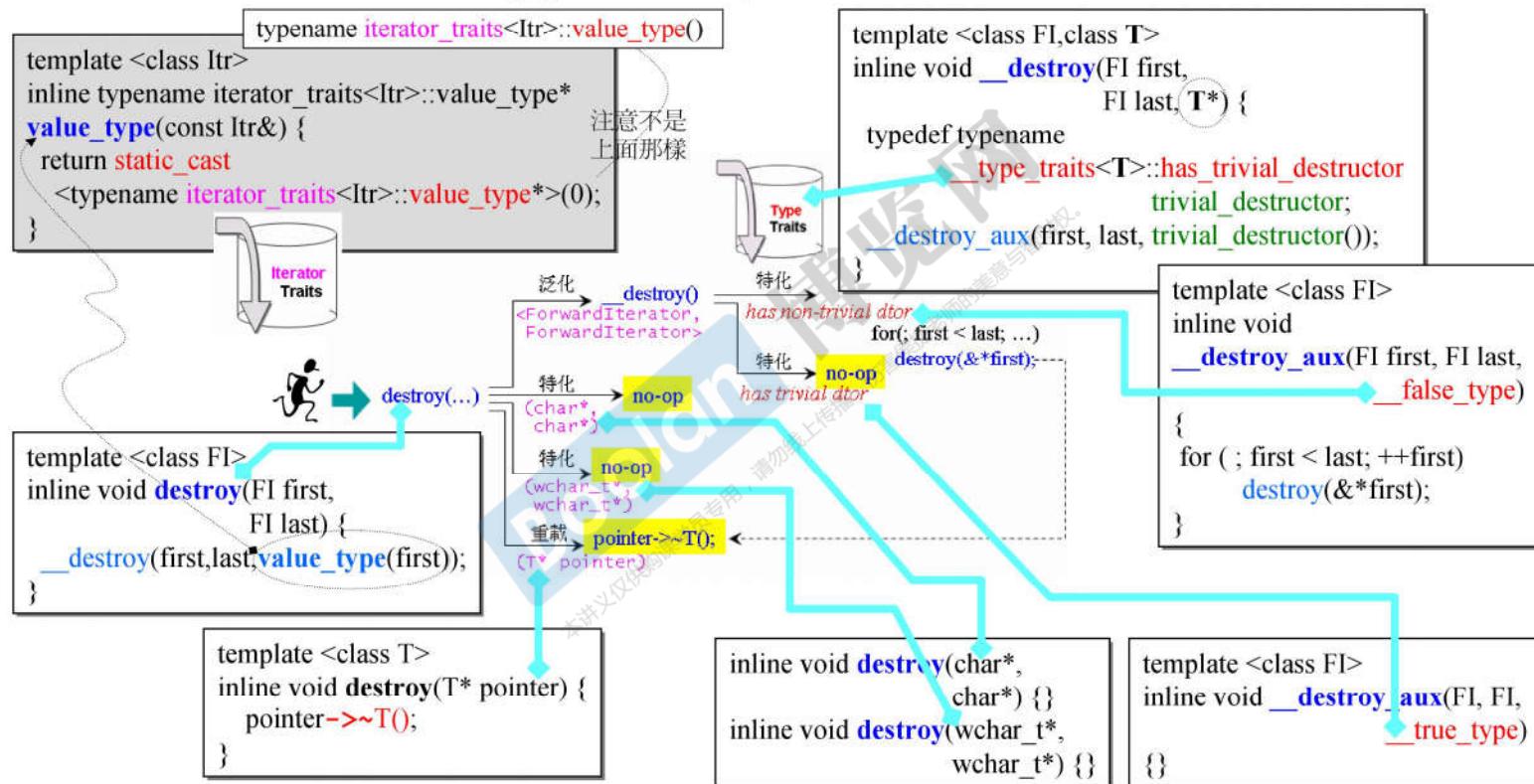
164

第4讲

## iterator traits 和 type traits 對算法的影響



## iterator traits 和 type traits 對算法的影響



## iterator traits 和 type traits 對算法的影響

```
template <class InputIterator, class OutputIterator>
inline OutputIterator __unique_copy(InputIterator first,
                                    InputIterator last,
                                    OutputIterator result,
                                    output_iterator_tag) {
    // output iterator 有其特別侷限，  

    // 所以處理前先探求其 value type.  

    return __unique_copy(first, last, result, value_type(first));
}
```

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator __unique_copy(InputIterator first,
                            InputIterator last,
                            OutputIterator result, T*) {
    T value = *first;
    *result = value;
    while (++first != last)
        if (value != *first) {
            value = *first;
            *++result = value;
        }
    return ++result;
}
```

由於 output iterator (例 **ostream\_iterator**) 是 **write-only**，無法像 forward iterator 那般可以 **read**，所以不能有類似(右側)  
`*result!=*first` 的動作，因此需設計出(左側)專屬版本。

避免了read的动作

不能读

```
template <class Itr>
inline typename iterator_traits<Itr>::value_type*
value_type(const Itr&) {
    return static_cast<typename iterator_traits<Itr>::value_type*>(0);
}
```

注意不是下面這樣  
**typename iterator\_traits<Itr>::value\_type()**

```
template <class InputIterator, class ForwardIterator>
ForwardIterator __unique_copy(InputIterator first,
                             InputIterator last,
                             ForwardIterator result,
                             forward_iterator_tag) {
    *result = *first; // 登錄第一元素
    while (++first != last) // 遍歷整個區間
        if (*result != *first) {
            *++result = *first;
        }
    return ++result;
}
```

算法源码里对迭代器的分类没有强制限制，只有暗示。因为算法是一种模板，模板就是可以接收各式各样的类型。

## /// 算法源碼中對 iterator\_category 的“暗示”

```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last) {
    typedef typename
        iterator_traits<InputIterator>::iterator_category category;
    return __distance(first, last, category());
}
```

```
template <class ForwardIterator>
inline void rotate(ForwardIterator first, ForwardIterator middle,
                  ForwardIterator last) {
    if (first == middle || middle == last) return;
    __rotate(first, middle, last, distance_type(first),
             iterator_category(first));
}
```

```
template <class RandomAccessIterator>
inline void sort(RandomAccessIterator first,
                 RandomAccessIterator last) {
    if (first != last) {
        __introsort_loop(first, last, value_type(first), __lg(last - first) * 2);
        __final_insertion_sort(first, last);
    }
}
```

通过模板参数命名  
暗示

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

```
template <class BidirectionalIterator,
         class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                           BidirectionalIterator last,
                           OutputIterator result) {
    while (first != last) {
        --last;
        *result = *last;
        ++result;
    }
    return result;
}
```

區分哪些是C++標準庫提供的算法

## 先前示例中出現的算法

```
qsort(c.data(), ASIZE, sizeof(long), compareLongs);  
long* pItem =  
(long*)bsearch(&target, (c.data()), ASIZE,  
                sizeof(long), compareLongs);
```

這是 C 函數.

這是 C++ 標準庫提供的 algorithms, 以函數的形式呈現

```
cout << count_if(vi.begin(), vi.end(),  
                  not1(bind2nd(less<int>(), 40)));  
  
auto ite = find(c.begin(), c.end(), target);  
  
sort(c.begin(), c.end());
```

```
template<typename Iterator>  
std::Algorithm(Iterator itr1, Iterator itr2, ...)  
{  
    ...  
}
```

## 算法 accumulate

```
template <class InputIterator,
          class T>
T accumulate(InputIterator first,
             InputIterator last,
             T init)
{
    for ( ; first != last; ++first)
        //將元素累加至初值 init 身上
        init = init + *first;
    return init;
}

template <class InputIterator,
          class T,
          class BinaryOperation>
T accumulate(InputIterator first,
             InputIterator last,
             T init,
             BinaryOperation binary_op)
{
    for ( ; first != last; ++first)
        //對元素「累計算」至初值 init 身上
        init = binary_op(init, *first);
    return init;
}
```

```
1717 #include <iostream>      // std::cout
1718 #include <functional>    // std::minus
1719 #include <numeric>       // std::accumulate
1720 namespace jj34
1721 {
1722     int myfunc (int x, int y) {return x+2*y;}
1723
1724     struct myclass {
1725         int operator()(int x, int y) {return x+3*y;}
1726     } myobj;
1727
1728     void test_accumulate()
1729     {
1730         int init = 100;
1731         int nums[] = {10,20,30};
1732
1733         cout << "using default accumulate: ";
1734         cout << accumulate(nums,nums+3,init); //160
1735         cout << '\n';
1736
1737         cout << "using functional's minus: ";
1738         cout << accumulate(nums, nums+3, init, minus<int>()); //40
1739         cout << '\n';
1740
1741         cout << "using custom function: ";
1742         cout << accumulate(nums, nums+3, init, myfunc); //220
1743         cout << '\n';
1744
1745         cout << "using custom object: ";
1746         cout << accumulate(nums, nums+3, init, myobj); //280
1747         cout << '\n';
1748     }
1749 }
```

可调用对象

括号不是调用，是初始化

## //// 算法 for\_each

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

## range-based for statement

(since C++11)

```
for( decl : coll ) {
    statement
}
```

```
for( int i : {2,3,5,7,9,13,17,19} ) {
    cout << i << endl;
}
```

```
1751 #include <iostream>      // std::cout
1752 #include <algorithm>     // std::for_each
1753 #include <vector>        // std::vector
1754 namespace jj35
1755 {
1756     void myfunc(int i) {
1757         cout << ' ' << i;
1758     }
1759
1760     struct myclass {
1761         void operator()(int i) { cout << ' ' << i; }
1762     } myobj;
1763
1764     void test_for_each()
1765     {
1766         vector<int> myvec;
1767         myvec.push_back(10);
1768         myvec.push_back(20);
1769         myvec.push_back(30);
1770
1771         for_each (myvec.begin(), myvec.end(), myfunc);
1772         cout << endl;           //output: 10 20 30
1773
1774         for_each (myvec.begin(), myvec.end(), myobj);
1775         cout << endl;           //output: 10 20 30
1776
1777         //since C++11, range-based for- statement
1778         for (auto& elem : myvec)
1779             elem += 5;
1780
1781         for (auto elem : myvec)
1782             cout << ' ' << elem;   //output: 15 25 35
1783     }
1784 }
```

## ■■■ 算法 replace, replace\_if, replace\_copy

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first,
             ForwardIterator last,
             const T& old_value,
             const T& new_value) {
    //範圍內所有等同於 old_value 者都以 new_value 取代
    for ( ; first != last; ++first)
        if (*first == old_value)
            *first = new_value;
}
```

```
template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first,
                ForwardIterator last,
                Predicate pred,
                const T& new_value) {
    //範圍內所有滿足 pred() 為 true 之元素都以 new_value 取代
    for ( ; first != last; ++first)
        if (pred(*first))
            *first = new_value;
}
```

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first,
                           InputIterator last,
                           OutputIterator result,
                           const T& old_value,
                           const T& new_value) {
    //範圍內所有等同於 old_value 者都以 new_value 放至新區間，
    //不符合者原值放入新區間。
    for ( ; first != last; ++first, ++result)
        *result =
            *first == old_value ? new_value : *first;
    return result;
}
```

## //// 算法 count, count\_if

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last,
       const T& value) {
    //以下定義一個初值為 0 的計數器 n
    typename iterator_traits<InputIterator>::difference_type n = 0;
    for ( ; first != last; ++first)           //遍歷(循序搜尋)
        if (*first == value)                 //如果元素值和 value 相等
            ++n;                         //計數器累加1
    return n;
}
```

```
template <class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last,
           Predicate pred) {
    //以下定義一個初值為 0 的計數器 n
    typename iterator_traits<InputIterator>::difference_type n = 0;
    for ( ; first != last; ++first) //遍歷(循序搜尋)
        if (pred(*first))          //如果元素帶入 pred 的結果為 true
            ++n;               //計數器累加1
    return n;
}
```

容器**不帶**成員函數 **count()**：  
array, vector, list, forward\_list, deque,

容器**帶有**成員函數 **count()**：  
set / multiset,  
map / multimap,  
unordered\_set / unordered\_multiset  
unordered\_map / unordered\_multimap

## ■■■ 算法 find, find\_if

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

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

容器**不帶**成員函數 find()：  
array, vector, list, forward\_list, deque,

容器**帶有**成員函數 find()：  
set / multiset,  
map / multimap  
unordered\_set / unordered\_multiset  
unordered\_map / unordered\_multimap

## 算法 sort

```
1791 bool myfunc (int i,int j) { return (i<j); }
1792
1793 struct myclass {
1794     bool operator() (int i,int j) { return (i<j);}
1795     } myobj;
1796
1797 bool test_sort()
1798 {
1799     int myints[] = {32,71,12,45,26,80,53,33};
1800     vector<int> myvec(myints, myints+8);           // 32 71 12 45 26 80 53 33
1801
1802     // using default comparison (operator <):
1803     sort(myvec.begin(), myvec.begin()+4);           //(12 32 45 71)26 80 53 33
1804
1805     // using function as comp
1806     sort(myvec.begin()+4, myvec.end(), myfunc);    // 12 32 45 71(26 33 53 80)
1807
1808     // using object as comp
1809     sort(myvec.begin(), myvec.end(), myobj);        //(12 26 32 33 45 53 71 80)
1810
1811     // print out content:
1812     cout << "myvec contains:";
1813     for (auto elem : myvec)             //C++11 range-based for statement
1814         cout << ' ' << elem ;          //output: 12 26 32 33 45 53 71 80
1815
1816     // using reverse iterators and default comparison (operator <):
1817     sort(myvec.rbegin(), myvec.rend());
1818
1819     // print out content:
1820     cout << "myvec contains:";
1821     for (auto elem : myvec)             //C++11 range-based for statement
1822         cout << ' ' << elem ;          //output: 80 71 53 45 33 32 26 12
1823 }
```

容器不帶成員函數 sort() :

array, vector, deque,  
set / multiset  
map / multimap  
unordered\_set / unordered\_multiset  
unordered\_map / unordered\_multimap

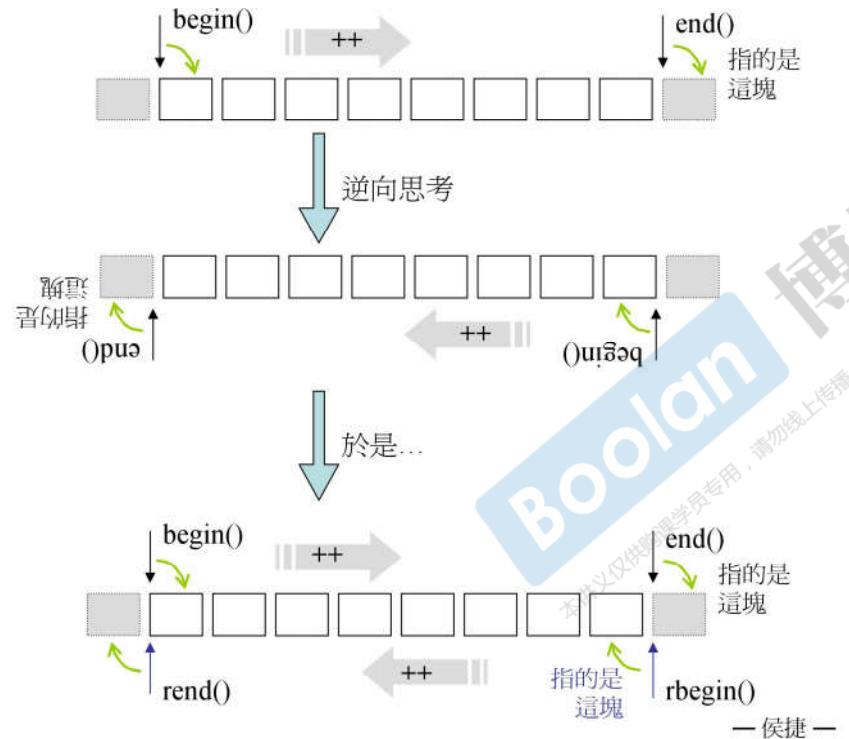
本身就是  
sort了

遍歷自然形成 sorted 狀態

容器帶有成員函數 sort() :

list, forward\_list

## 關於 reverse iterator, rbegin(), rend()



```
reverse_iterator  
rbegin()  
{ return reverse_iterator(end()); }  
  
reverse_iterator  
rend()  
{ return reverse_iterator(begin()); }
```

這是個 iterator adapter

— 侯捷 —



## 算法 binary\_search

Test if value exists in **sorted** sequence

```
template <class ForwardIterator, class T>
bool binary_search (ForwardIterator first,
                    ForwardIterator last,
                    const T& val)
{
    first = std::lower_bound(first, last, val);
    return (first!=last && !(val < *first));
}
```



獲得的 iterator 所指的位置  
既非 end, 目標值 val 亦不小於首元  
素 (sorted range 之首元素值最小)

侯捷檢討：先判斷 !(val < \*first) 然後才  
調用 **lower\_bound()**，將獲得較佳效率。  
因為進入 **lower\_bound()** 後乃由中間元素  
開始比較，雖說二分搜尋也快，畢竟都  
不必要了。

advance 和 distance 好好  
体会下，为什么不直接 -和+

```
template <class ForwardIterator, class T>
ForwardIterator
lower_bound (ForwardIterator first,
               ForwardIterator last,
               const T& val)
{
    ForwardIterator it;
    iterator_traits<ForwardIterator>::difference_type count, step;
    count = distance(first, last);
    while (count>0)
    {
        it = first; step=count/2; advance(it,step);
        if (*it < val) { // or: if (comp(*it, val)), for version (2)
            first = ++ it;
            count -= step+1;
        }
        else count=step;
    }
    return first;
}
```

low=**lower\_bound** (v.begin(), v.end(), 20);  
up=**upper\_bound** (v.begin(), v.end(), 20);

Returns an iterator pointing to the first element in the range [first,last) which does not compare less than val. The elements are compared using operator< for the first version, and comp for the second. The elements in the range shall already be sorted according to this same criterion (operator< or comp), or at least partitioned with respect to val.

## 仿函數 functors

标准库提供的三大类  
functor

```
//算術類 (Arithmetic)
template <class T>
struct plus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const
    { return x + y; }
};

template <class T>
struct minus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const
    { return x - y; }
};
...
```

```
template<typename Iterator, typename Cmp>
Algorithm(Iterator itr1, Iterator itr2, Cmp comp)
{
    ...
}
```

```
//邏輯運算類 (Logical)
template <class T>
struct logical_and : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const
    { return x && y; }
};
...
這就是融入 STL 的條件
```

```
//相對關係類 (Relational)
template <class T>
struct equal_to : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const
    { return x == y; }
};
```

```
template <class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const
    { return x < y; }
};
— 侯捷 —
```

只为算法服务

你参与度比较高的部分  
融入到STL中

## 仿函数 functors

```
template <class T>
struct identity : public unary_function<T, T> {
    const T& operator()(const T& x) const { return x; }
};
```

G2.9

GNU C++ 獨有, 非標準

```
template <class Pair>
struct select1st : public unary_function<Pair, typename Pair::first_type> {
    const typename Pair::first_type& operator()(const Pair& x) const
    {
        return x.first;
    }
};
```

```
template <class Pair>
struct select2nd : public unary_function<Pair, typename Pair::second_type> {
    const typename Pair::second_type& operator()(const Pair& x) const
    {
        return x.second;
    }
};
```

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;
    pair() : first(T1()), second(T2()) {}
    pair(const T1& a, const T2& b)
        : first(a), second(b) {}
};
```

...\\4.9.2\\include\\c++\\ext

```
template <class _Tp> G4.9
struct identity
    : public std::_Identity<_Tp> {};

template <class _Pair>
struct select1st
    : public std::_Select1st<_Pair> {};

template <class _Pair>
struct select2nd
    : public std::_Select2nd<_Pair> {};
```

G4.9

```
template <class T>
struct _Identity;
```

```
template <class Pair>
struct _Select1st;
```

```
template <class Pair>
struct _Select2nd{};
```

## 仿函數 functors

```
// using default comparison (operator <):  
sort(myvec.begin(), myvec.end());  
  
// using function as comp  
sort(myvec.begin(), myvec.end(), myfunc);  
  
// using object as comp  
sort(myvec.begin(), myvec.end(), myobj);  
  
// using explicitly default comparison (operator <):  
sort(myvec.begin(), myvec.end(), less<int>());  
  
// using another comparision criteria (operator >):  
sort(myvec.begin(), myvec.end(), greater<int>());
```

```
struct myclass {  
    bool operator()(int i, int j) { return (i < j); }  
} myobj;  
bool myfunc (int i, int j) { return (i < j); }
```

沒有繼承，也就意味着沒有融入STL整個體系結構里邊。雖然在sort可以運作，但是繼續下去做某些事情可能就失敗了。STL對functor的要求沒有全部達到，不能被適配器改造

```
//相對關係類 (Relational)  
template <class T>  
struct greater : public binary_function<T, T, bool> {  
    bool operator()(const T& x, const T& y) const  
    { return (x > y); }  
};  
  
template <class T>  
struct less : public binary_function<T, T, bool> {  
    bool operator()(const T& x, const T& y) const  
    { return (x < y); }  
};
```

兩個操作數

## 仿函數 functors 的可適配 (adaptable) 條件

1个操作数    2个操作数

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

有点像萃取器。用来其他操作问“问题”。让仿函数可以被适配

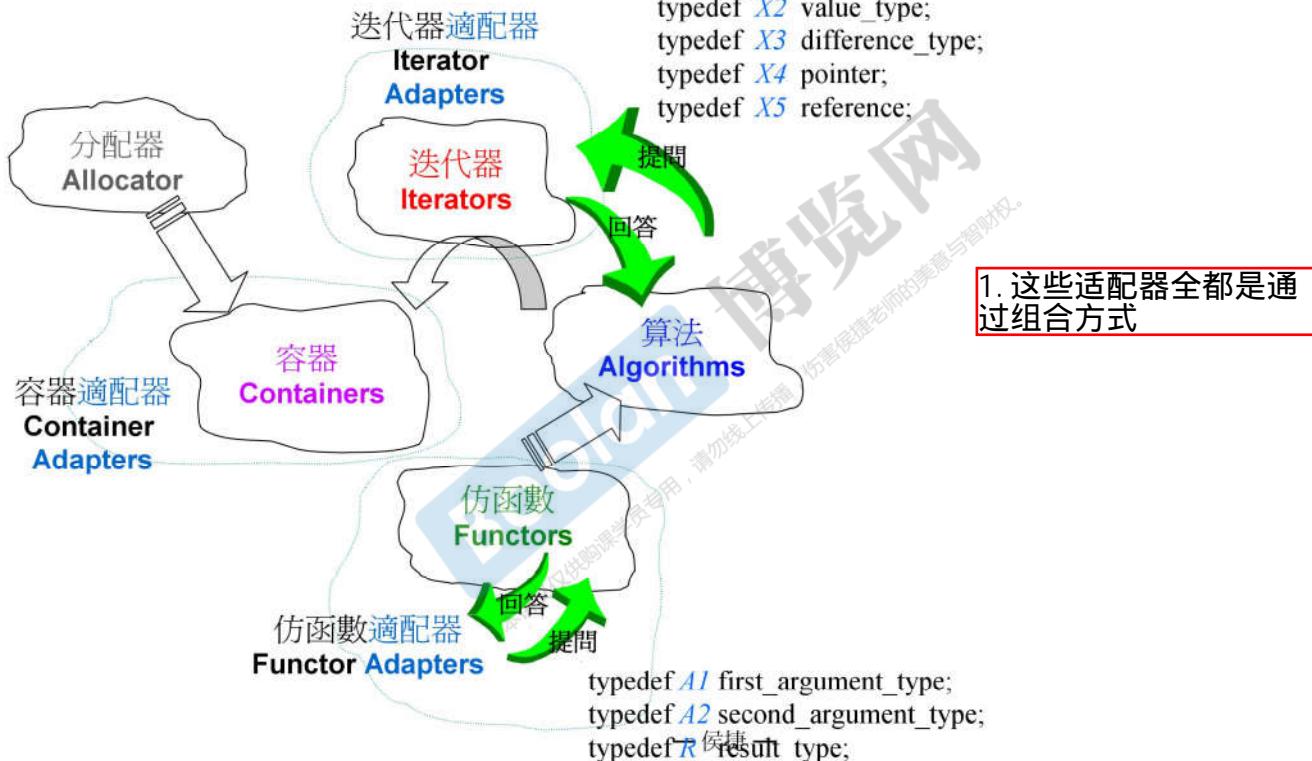
STL 規定每個 Adaptable Function 都應挑選適當者繼承之 (因為 Function Adapter 將會提問)。例如：

```
template <class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const
    { return x < y; }
};
```

於是 less<int> 便有了三個 typedef，分別是：

```
typedef int first_argument_type;
typedef int second_argument_type;
typedef bool result_type;
```

## 存在多種 Adapters



## 容器適配器 : stack, queue

```
template <class T, class Sequence=deque<T>>
class 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:
    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(); }
};
```

```
template <class T, class Sequence=deque<T>>
class queue {
...
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:
    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(); }
};
```

被bind取代了

细节：这里必须是整数，模板需要提供  
检查，不然会runtime\_error(最害怕的)  
(检查部分为灰色部分代码)

牛逼，33集建议多看

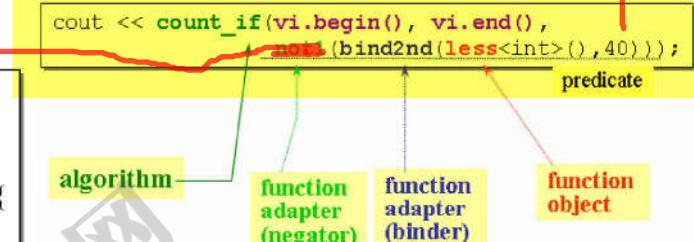
## 函数适配器：binder2nd

```
// 輔助函數，讓 user 得以方便使用 binder2nd<Op>：  
// 編譯器會自動推導 Op 的 type  
template <class Operation, class T>  
inline binder2nd<Operation> bind2nd(const Operation& op, const T& x) {  
    typedef typename Operation::second_argument_type arg2_type;  
    return binder2nd<Operation>(op, arg2_type(x));  
}
```

告诉编译器，  
这是一个  
typename，可  
以通过编译

不能转换则出错

```
template <class InputIterator, class Predicate>  
typename iterator_traits<InputIterator>::difference_type  
count_if(InputIterator first, InputIterator last,  
         Predicate pred) {  
    //以下定義一個初值為 0 的計數器  
    typename iterator_traits<InputIterator>::difference_type n = 0;  
    for ( ; first != last; ++first) //遍歷  
        if (pred(*first)) //如果元素帶入 pred 的結果為 true  
            ++n; //計數器累加1  
    return n;  
}
```



// 以下將某個 Adaptable Binary function 轉換為 Unary Function

```
template <class Operation>  
class binder2nd  
: public unary_function<typename Operation::first_argument_type,  
                       typename Operation::result_type> {  
protected:  
    Operation op; // 內部成員，分別用以記錄算式和第二實參  
    typename Operation::second_argument_type value;  
public:  
    // constructor  
    binder2nd(const Operation& x,  
              const typename Operation::second_argument_type& y)  
        : op(x), value(y) {} // 將算式和第二實參記錄下來  
    typename Operation::result_type  
>operator()(const typename Operation::first_argument_type& x) const {  
    return op(x, value); // 實際呼叫算式並取 value 為第二實參  
}
```

适配器问仿函数 问题，  
检查value的值

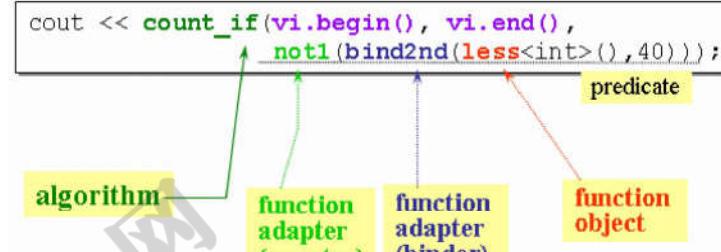
如果要直接使用，需要指  
定模板参数，但是需要功  
力比较深才能做到，所以  
用模板函数做对外接口。

重点：其实只是把要“绑定”的参数记下来，也没有真正绑定，等到要调用的时候再传进去。

## /// 函數適配器 : not1

```
// 輔助函式，使 user 得以方便使用 unary_negate<Pred>
template <class Predicate>
inline unary_negate<Predicate> not1(const Predicate& pred) {
    return unary_negate<Predicate>(pred);
}
```

```
template <class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last,
         Predicate pred) {
    //以下定義一個初值為 0 的計數器
    typename
        iterator_traits<InputIterator>::difference_type n = 0;
    for ( ; first != last; ++first) //遍歷
        if (pred(*first)) //如果元素帶入 pred 的結果為 true
            ++n; //計數器累加1
    return n;
}
```



```
// 以下取某個 Adaptable Predicate 的邏輯負值 (logical negation)
template <class Predicate>
class unary_negate
    : public unary_function<typename Predicate::argument_type, bool> {
protected:
    Predicate pred; // 內部成員
public:
    //constructor
    explicit unary_negate(const Predicate& x) : pred(x) {}
    bool operator()(const typename Predicate::argument_type& x) const {
        return !pred(x); // 將 pred 的運算結果 "取否" (negate)
    }
};
```

## 新型适配器, bind

...\\include\\c++\\backward\\backward\_warning.h

```
/*
```

A list of **valid replacements** is as follows:

Use:

```
<sstream>, basic_stringbuf  
<sstream>, basic_istringstream  
<sstream>, basic_ostringstream  
<sstream>, basic_stringstream  
<unordered_set>, unordered_set  
<unordered_set>, unordered_multiset  
<unordered_map>, unordered_map  
<unordered_map>, unordered_multimap  
<functional>, bind  
<functional>, bind  
<functional>, bind  
<functional>, bind  
<memory>, unique_ptr
```

```
*/
```

Instead of:

```
<strstream>, strstreambuf  
<strstream>, istrstream  
<strstream>, ostrstream  
<strstream>, strstream  
<ext/hash_set>, hash_set  
<ext/hash_set>, hash_multiset  
<ext/hash_map>, hash_map  
<ext/hash_map>, hash_multimap  
<functional>, binder1st  
<functional>, binder2nd  
<functional>, bind1st  
<functional>, bind2nd  
<memory>, auto_ptr
```



## 新型適配器, bind

Since C++11

<http://www.cplusplus.com/reference/functional/bind?kw=bind>



```
#include <functional> // std::bind
```

```
2768 // a function: (also works with function object:
2769 //           std::divides<double> my_divide;
2770 double my_divide (double x, double y)
2771 { return x / y; }
2772
2773 struct MyPair {
2774     double a,b;
2775     double multiply() { return a * b; }
2776     //member function 其實有個 argument: this
2777 };
```

std::bind 可以綁定：

1. functions
2. function objects
3. member functions, \_1 必須是某個 object 地址.
4. data members, \_1 必須是某個 object 地址.

返回一個 function object ret. 調用 ret 相當於  
調用上述 1,2,3，或相當於取出 4.

```
2781 using namespace std::placeholders; // adds visibility of _1, _2, _3,...
2782
2783 // binding functions:
2784 auto fn_five = bind (my_divide,_1,5); // returns 10/2
2785 cout << fn_five() << '\n'; // 5
2786
2787 auto fn_half = bind (my_divide,_1,_2); // returns x/2
2788 cout << fn_half(10) << '\n'; // 5
2789
2790 auto fn_invert = bind (my_divide,_2,_1); // returns y/x
2791 cout << fn_invert(10,2) << '\n'; // 0.2
2792
2793 auto fn_rounding = bind<int> (my_divide,_1,_2); // returns int(x/y)
2794 cout << fn_rounding(10,3) << '\n'; // 3

// binding members:
MyPair ten_two {10,2}; //member function 其實有個 argument: this
auto bound_memfn = bind(&MyPair::multiply, _1); // returns x.multiply()
cout << bound_memfn(ten_two) << '\n'; // 20

auto bound_memdata = bind(&MyPair::a, ten_two); // returns ten_two.a
cout << bound_memdata() << '\n'; // 10

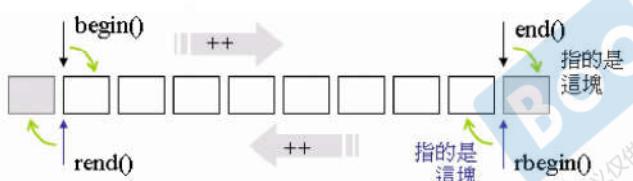
auto bound_memdata2 = bind(&MyPair::b, _1); // returns x.b
cout << bound_memdata2(ten_two) << '\n'; // 2

vector<int> v {15,37,94,50,73,58,28,98};
int n = count_if(v.cbegin(), v.cend(), not1(bind2nd(less<int>(),50)));
cout << "n= " << n << endl; //5

auto fn_ = bind(less<int>(), _1, 50);
cout << count_if(v.cbegin(), v.cend(), fn_) << endl; //3
cout << count_if(v.begin(), v.end(), bind(less<int>(), _1, 50)) << endl; //3
```

## 迭代器適配器： reverse\_iterator

```
reverse_iterator  
rbegin()  
{ return reverse_iterator(end()); }  
  
reverse_iterator  
rend()  
{ return reverse_iterator(begin()); }
```

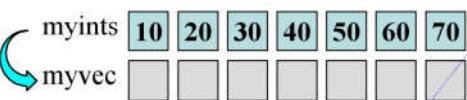


```
template <class Iterator>  
class reverse_iterator  
{  
protected:  
    Iterator current; // 對應之正向迭代器  
public:  
    // 逆向迭代器的5種 associated types 都和其對應之正向迭代器相同  
    typedef typename iterator_traits<Iterator>::iterator_category iterator_category;  
    typedef typename iterator_traits<Iterator>::value_type value_type;  
    ...  
    typedef Iterator iterator_type; // 代表正向迭代器  
    typedef reverse_iterator<Iterator> self; // 代表逆向迭代器  
  
public:  
    explicit reverse_iterator(iterator_type x) : current(x) {}  
    reverse_iterator(const self& x) : current(x.current) {}  
  
    iterator_type base() const { return current; } // 取出對應的正向迭代器  
    reference operator*() const { Iterator tmp = current; return *tmp--; }  
    // 以上為關鍵所在。對逆向迭代器取值，就是  
    // 將「對應之正向迭代器」退一位取值。  
    pointer operator->() const { return &(operator*()); } // 意義同上。  
  
    // 前進變成後退，後退變成前進  
    self& operator++() { --current; return *this; }  
    self& operator--() { ++current; return *this; }  
    self operator+(difference_type n) const { return self(current - n); }  
    self operator-(difference_type n) const { return self(current + n); }  
};
```

## 迭代器適配器： inserter

```
int myints[] = {10, 20, 30, 40, 50, 60, 70};
vector<int> myvec(7);

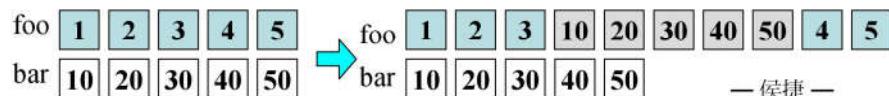
copy(myints, myints+7, myvec.begin());
```



```
list<int> foo, bar;
for (int i=1; i<=5; i++)
{ foo.push_back(i); bar.push_back(i*10);

list<int>::iterator it = foo.begin();
advance(it, 3);

copy(bar.begin(), bar.end(), inserter(foo, it));
```



```
template<class InputIterator,
         class OutputIterator>
OutputIterator
copy (InputIterator first,
      InputIterator last,
      OutputIterator result)
{
    while (first!=last) {
        *result = *first;
        ++result; ++first;
    }
    return result;
}
```

// 這個 adapter 將 iterator 的賦值 (assign) 操作改變為  
// 安插 (insert) 操作，並將 iterator 右移一個位置。如此便可  
// 讓 user 連續執行「表面上 assign 而實際上 insert」的行為。

```
template <class Container>
class inserter {
protected:
    Container* container; // 底層容器
    typename Container::iterator iter;
public:
    typedef output_iterator_tag iterator_category; // 注意類型
    inserter(Container& x, typename Container::iterator i)
        : container(&x), iter(i) {}

    inserter<Container>&
    operator=(const typename Container::value_type& value) {
        iter = container->insert(iter, value); // 關鍵：轉調用 insert()
        ++iter; // 令 insert iterator 永遠隨其 target 貼身移動
        return *this;
    }
};

// 輔助函式，幫助 user 使用 inserter。
template <class Container, class Iterator>
inline inserter<Container>
inserter(Container& x, Iterator i) {
    typedef typename Container::iterator iter;
    return inserter<Container>(x, iter(i));
}
```

— 侯捷 —

## X 適配器 : ostream\_iterator

```
// ostream_iterator example
#include <iostream> // std::cout
#include <iterator> // std::ostream_iterator
#include <vector> // std::vector
#include <algorithm> // std::copy

int main () {
    std::vector<int> myvector;
    for (int i=1; i<10; ++i) myvector.push_back(i*10);

    std::ostream_iterator<int> out_it (std::cout, " ");
    std::copy ( myvector.begin(), myvector.end(), out_it );
    return 0;
}
```

```
template<class InputIterator, class OutputIterator>
OutputIterator
copy (InputIterator first, InputIterator last,
      OutputIterator result)
{
    while (first != last) {
        *result = *first;
        ++result; ++first;
    }
    return result;
}
```

這就相當於  
cout << \*first;  
cout << ", "  
;

```
template <class T, class charT=char, class traits=char_traits<charT> >
class ostream_iterator :
public iterator<output_iterator_tag, void, void, void, void>
{
basic_ostream<charT,traits>* out_stream;
const charT* delim;

public:
typedef charT char_type;
typedef traits traits_type;
typedef basic_ostream<charT,traits> ostream_type;
ostream_iterator(ostream_type& s) : out_stream(&s), delim(0) {}
ostream_iterator(ostream_type& s, const charT* delimiter)
: out_stream(&s), delim(delimiter) {}
ostream_iterator(const ostream_iterator<T,charT,traits>& x)
: out_stream(x.out_stream), delim(x.delim) {}
~ostream_iterator() {}
ostream_iterator<T,charT,traits>& operator=(const T& value) {
    *out_stream << value;
    if (delim!=0) *out_stream << delim;
    return *this;
}

ostream_iterator<T,charT,traits>& operator*() { return *this; }
ostream_iterator<T,charT,traits>& operator++() { return *this; }
ostream_iterator<T,charT,traits>& operator++(int) { return *this; }
};
```

## X 適配器 : istream\_iterator

```
// istream_iterator example  
#include <iostream> // std::cin, std::cout  
#include <iterator> // std::istream_iterator  
  
int main () {  
    double value1, value2;  
    std::cout << "Please, insert two values: ",  
    std::istream_iterator<double> eos; // end-of-stream iterator  
    std::istream_iterator<double> iit (std::cin); // stdin iterator  
    if (iit!=eos) value1=*iit;  
  
    ++iit;  
    if (iit!=eos) value2=*iit;  
  
    std::cout << value1 << "*" << value2 << "="  
        << (value1* value2) << '\n';  
    return 0;  
}
```

例 1

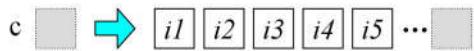
這就相當於  
cin >> value;

這就相當於  
return value;

```
template <class T, class charT=char, class traits=char_traits<charT>,  
         class Distance=ptrdiff_t>  
class istream_iterator :  
public iterator<input_iterator_tag, T, Distance, const T*, const T&>  
{  
    basic_istream<charT,traits>* in_stream;  
    T value;  
public:  
    typedef charT char_type;  
    typedef traits traits_type;  
    typedef basic_istream<charT,traits> istream_type;  
    istream_iterator() : in_stream(0) {}  
    istream_iterator(istream_type& s) : in_stream(&s) { ++*this; }  
    istream_iterator(const istream_iterator<T,charT,traits,Distance>& x)  
        : in_stream(x.in_stream), value(x.value) {}  
    ~istream_iterator() {}  
    const T& operator*() const { return value; }  
    const T* operator->() const { return &value; }  
    istream_iterator<T,charT,traits,Distance>& operator++()  
    {  
        if (in_stream && !(*in_stream >> value)) in_stream=0;  
        return *this;  
    }  
    istream_iterator<T,charT,traits,Distance> operator++(int){  
        istream_iterator<T,charT,traits,Distance> tmp = *this;  
        ++*this;  
        return tmp;  
    };
```

一旦創建立刻 read,  
可能令 user 吃驚

## ■■■ X 適配器 : istream\_iterator



```
istream_iterator<int> iit(cin), eos; 例 2  
copy(iit, eos, inserter(c, c.begin()));
```

```
template<class InputIterator, class  
OutputIterator>  
OutputIterator  
copy (InputIterator first, InputIterator last,  
      OutputIterator result)  
{  
    while (first != last) {  
        *result = *first;  
        ++result; ++first;  
    }  
    return result;  
}
```

```
template <class T, class charT=char, class traits=char_traits<charT>,  
         class Distance=ptrdiff_t>  
class istream_iterator :  
public iterator<input_iterator_tag, T, Distance, const T*, const T&>  
{  
    basic_istream<charT,traits>* in_stream;  
    T value;  
public:  
    typedef charT char_type;  
    typedef traits traits_type;  
    typedef basic_istream<charT,traits> istream_type;  
    istream_iterator(): in_stream(0) {}  
    istream_iterator(istream_type& s): in_stream(&s) { ++*this; }  
    istream_iterator(const istream_iterator<T,charT,traits,Distance>& x)  
        : in_stream(x.in_stream), value(x.value) {}  
    ~istream_iterator() {}  
    const T& operator*() const { return value; }  
    const T* operator->() const { return &value; }  
    istream_iterator<T,charT,traits,Distance>& operator++()  
    { if (in_stream && !(*in_stream >> value)) in_stream=0;  
        return *this;  
    }  
    istream_iterator<T,charT,traits,Distance> operator++(int){  
        istream_iterator<T,charT,traits,Distance> tmp = *this;  
        ++*this;  
        return tmp;  
    };
```

一旦創建立刻 **read**,  
可能令 user 吃驚

## 閱讀 C++ 標準庫源代碼 – 意義與價值

使用一個東西，  
卻不明白它的道理，  
不高明！



The End



# C++ 標準庫

## 體系結構與內核分析

(C++ Standard Library — architecture & sources)

第四講



侯捷



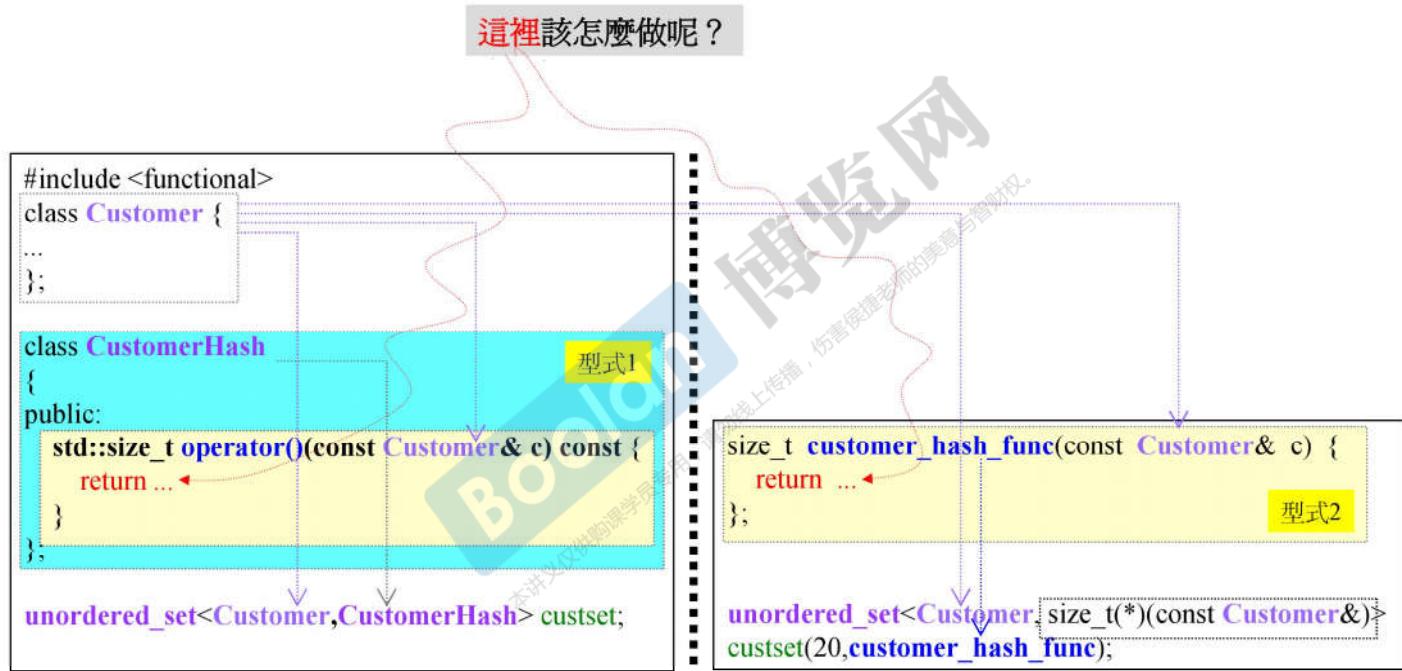
# 勿在浮沙築高台



— 侯捷 —

196

## ■■■■ 一個萬用的 Hash Function



## 一個萬用的 Hash Function

```
//a naive approach : simply add all hash values for  
//those attributes that are relevant for the hash function.  
  
class CustomerHash {  
public:  
    std::size_t operator()(const Customer& c) const {  
        return std::hash<std::string>()(c.fname) +  
               std::hash<std::string>()(c.lname) +  
               std::hash<long>()(c.no);  
    }  
};
```

```
#include <functional>  
template <typename T>  
④ inline void hash_combine(size_t& seed, const T& val) {  
    seed ^= std::hash<T>()(val) + 0x9e3779b9  
    + (seed<<6) + (seed>>2);  
}  
  
③ // auxiliary generic functions  
last template <typename T>  
inline void hash_val(size_t& seed, const T& val) {  
    hash_combine(seed, val);  
}
```

using **variadic templates** allows calling `hash_val()` with an arbitrary number of elements of any type to process a hash value out of all these values

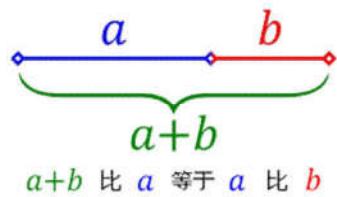
```
class CustomerHash {  
public:  
    std::size_t operator()(const Customer& c) const {  
        return hash_val(c.fname, c.lname, c.no);  
    }  
};
```

② template <typename **T**, typename... **Types**>  
 ↗ inline void **hash\_val** (size\_t& **seed**,  
 recursive const **T**& **val**, const **Types**&... **args**) {  
 hash\_combine(**seed**, **val**);  
 hash\_val(**seed**, **args**...);  
 }  
 ↗ 逐一取 **val**  
 ↗ 改變 **seed**  
 ↗ (pass by reference)  
  
① // auxiliary generic function  
template <typename... **Types**>  
inline size\_t hash\_val(const **Types**&... **args**) {  
 size\_t **seed** = 0;  
 hash\_val(**seed**, **args**...);  
 return **seed**; ↗ **seed** 最終就被視為 hash code  
}

— 侯捷 —

## ■■■ 一個萬用的 Hash Function

黃金比例，又稱黃金比，是一種數學上的比例關係。黃金分割具有嚴格的比例性、藝術性、和諧性，蘊藏著豐富的美學價值。應用時一般取0.618或1.618，就像圓周率在應用時取3.14一樣。黃金分割早存在於大自然中，呈現於不少動物和植物外觀。現今很多工業產品、電子產品、建築物或藝術品均普遍應用黃金分割，呈現其功能性與美觀性。常用希臘字母 $\varphi$ 表示黃金比值，用代數式表達就是： $\frac{a+b}{a} = \frac{a}{b} \equiv \varphi$



0x9e3779b9 / 0xFFFFFFFF = 0.618033.....  
[http://en.wikipedia.org/wiki/Golden\\_ratio](http://en.wikipedia.org/wiki/Golden_ratio)

List of numbers - Irrational and suspected irrational numbers	
$\gamma$	$\zeta(3)$
$\sqrt{2}$	$\sqrt{3}$
$\sqrt{5}$	$\varphi$
$\rho$	$\delta_S$
$e$	$\pi$
$\delta$	
Binary	1.1001111000110111011...
Decimal	1.6180339887498948482...
Hexadecimal	1.9E3779B97F4A7C15F39...
Continued fraction	$1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ddots}}}}$
Algebraic form	$\frac{1 + \sqrt{5}}{2}$
Infinite series	$\frac{13}{8} + \sum_{n=0}^{\infty} \frac{(-1)^{(n+1)}(2n+1)!}{(n+2)!n!4^{(2n+3)}}$

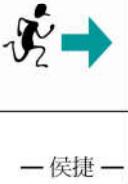
## /// 一個萬用的 Hash Function

```
488 //From boost (functional/hash):
489 template <typename T>
490 inline void hash_combine (size_t& seed,
491                         const T& val)
492 {
493     seed ^= hash<T>()(val) +
494         0x9e3779b9 +
495         (seed<<6) +
496         (seed>>2);
497 }
498
499 //auxiliary generic functions to create a hash value using a seed
500 template <typename T>
501 inline void hash_val (size_t& seed, const T& val)
502 {
503     hash_combine(seed, val);
504 }
505
506 template <typename T, typename... Types>
507 inline void hash_val (size_t& seed,
508                       const T& val,
509                       const Types&... args)
510 {
511     hash_combine(seed, val);
512     hash_val(seed, args...);
513 }
```

↓

```
515 //auxiliary generic function
516 template <typename... Types>
517 inline size_t hash_val (const Types&... args)
518 {
519     size_t seed = 0;
520     hash_val(seed, args...);
521     return seed;
522 }
523
524 class CustomerHash {
525 public:
526     size_t operator()(const Customer& c) const {
527         return hash_val(c.fname, c.lname, c.no);
528     }
529 };
```

— 侯捷 —



## ■■■ 一個萬用的 Hash Function

```
CustomerHash hh;
cout << "bucket position of Ace = " << hh(Customer("Ace", "Hou", 1L)) % 11 << endl; //2
cout << "bucket position of Sabri = " << hh(Customer("Sabri", "Hou", 2L)) % 11 << endl; //4
cout << "bucket position of Stacy = " << hh(Customer("Stacy", "Chen", 3L)) % 11 << endl; //10
cout << "bucket position of Mike = " << hh(Customer("Mike", "Tseng", 4L)) % 11 << endl; //2
cout << "bucket position of Paili = " << hh(Customer("Paili", "Chen", 5L)) % 11 << endl; //9
cout << "bucket position of Light = " << hh(Customer("Light", "Shiau", 6L)) % 11 << endl; //6
cout << "bucket position of Shally = " << hh(Customer("Shally", "Hwung", 7L)) % 11 << endl; //2

for (unsigned i=0; i<set3.bucket_count(); ++i) {
    cout << "bucket #" << i << " has " << set3.bucket_size(i) << " elements.\n";
}

//bucket #0 has 0 elements.
//bucket #1 has 0 elements.
//bucket #2 has 3 elements.
//bucket #3 has 0 elements.
//bucket #4 has 1 elements.
//bucket #5 has 0 elements.
//bucket #6 has 1 elements.
//bucket #7 has 0 elements.
//bucket #8 has 1 elements.
//bucket #9 has 1 elements.
```

B



```
unordered_set<Customer, CustomerHash> set3;
set3.insert( Customer("Ace", "Hou", 1L) );
set3.insert( Customer("Sabri", "Hou", 2L) );
set3.insert( Customer("Stacy", "Chen", 3L) );
set3.insert( Customer("Mike", "Tseng", 4L) );
set3.insert( Customer("Paili", "Chen", 5L) );
set3.insert( Customer("Light", "Shiau", 6L) );
set3.insert( Customer("Shally", "Hwung", 7L) );
cout << "set3 current bucket_count: " << set3.bucket_count() << endl; //11
```

## ■■■■ 以 struct hash 偏特化形式 實現 Hash Function

```
template <typename T,  
         typename Hash = hash<T>,  
         typename EqPred = equal_to<T>,  
         typename Allocator = allocator<T> >  
class unordered_set;  
  
template <typename T,  
         typename Hash = hash<T>,  
         typename EqPred = equal_to<T>,  
         typename Allocator = allocator<T> >  
class unordered_multiset;  
  
template <typename Key, typename T,  
         typename Hash = hash<T>,  
         typename EqPred = equal_to<T>,  
         typename Allocator = allocator<pair<const Key, T> > >  
class unordered_map;  
  
template <typename Key, typename T,  
         typename Hash = hash<T>,  
         typename EqPred = equal_to<T>,  
         typename Allocator = allocator<pair<const Key, T> > >  
class unordered_multimap;
```

G4.9

## 以 struct hash 偏特化形式實現 Hash Function

```
class MyString {  
private:  
    char* _data;  
    size_t _len;  
...  
};  
  
namespace std //必須放在 std 內  
{  
template<>  
struct hash<MyString> //為了 unordered containers  
{  
    size_t  
    operator()(const MyString& s) const noexcept  
    { return hash<string>()(string(s.get())); } //借用 hash<string>  
};  
}  
  
/// std::hash specialization for string.  
template<>  
struct hash<string> ...\\4.9.2\\include\\c++\\bits\\basic_string.h  
: public __hash_base<size_t, string>  
{  
    size_t  
    operator()(const string& s) const noexcept  
    { return std::__Hash_impl::hash(s.data(), s.length()); }  
};
```

## tuple, 用例



```
cout << "string, sizeof = " << sizeof(string) << endl;           //4
cout << "double, sizeof = " << sizeof(double) << endl;             //8
cout << "float, sizeof = " << sizeof(float) << endl;                //4
cout << "int, sizeof = " << sizeof(int) << endl;                  //4
cout << "complex<double>, sizeof = " << sizeof(complex<double>) << endl; //16

// tuples
// create a four-element tuple
// - elements are initialized with default value (0 for fundamental types)
tuple<string, int, int, complex<double>> t;
cout << "sizeof = " << sizeof(t) << endl;      //32, why not 28?

// create and initialize a tuple explicitly
tuple<int, float, string> t1(41, 6.3, "nico");
cout << "tuple<int, float, string>, sizeof = " << sizeof(t1) << endl; //12
// iterate over elements:
cout << "t1: " << get<0>(t1) << ' ' << get<1>(t1) << ' ' << get<2>(t1) << endl;

// create tuple with make_tuple()
auto t2 = make_tuple(22, 44, "stacy");

// assign second value in t2 to t1
get<1>(t1) = get<1>(t2);
```

```
// comparison and assignment
// - including type conversion from tuple<int,int,const char*>
//   to tuple<int,float,string>
if (t1 < t2) { // compares value for value
    cout << "t1 < t2" << endl;
} else {
    cout << "t1 >= t2" << endl;
}
t1 = t2; // OK, assigns value for value
cout << "t1: " << t1 << endl;

tuple<int, float, string> t3(77, 1.1, "more light");
int i1;
float f1;
string s1;
tie(i1,f1,s1) = t3; //assigns values of t to i,f, and s

typedef tuple<int, float, string> TupleType;
cout << tuple_size<TupleType>::value << endl; // yields 3
tuple_element<1,TupleType>::type f1 = 1.0; // yields float
typedef tuple_element<1,TupleType>::type T;
```

— 侯捷 —

## tuple 元之組合, 數之組合

### G4.8 節錄並簡化

```

template<typename... Values> class tuple;
template<> class tuple<> { };

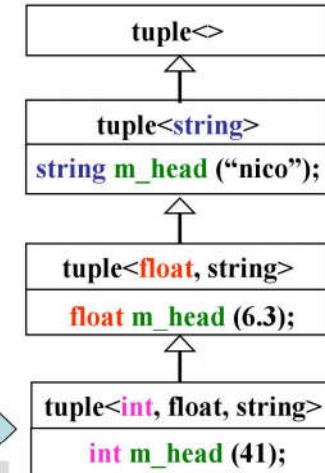
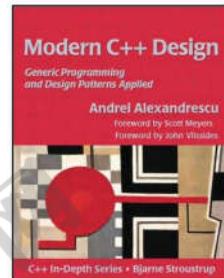
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
: private tuple<Tail...>
{
    typedef tuple<Tail...> inherited;
public:
    tuple() { }
    tuple(Head v, Tail... vtail) {
        : m_head(v), inherited(vtail...) { }
    }
    typename Head::type head() { return m_head; }
    inherited& tail() { return *this; }
protected:
    Head m_head;
};

```

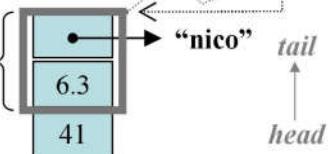
呼叫 base ctor 並予參數  
(不是創建 temp object)

注意這是 initialization list

return 後  
轉型為 inherited,  
獲得的是 — 侯捷 —



例：tuple<int,float,string>  
`t(41, 6.3, "nico");`  
`t.head() → 獲得 41`  
`t.tail() → 獲得`  
`t.tail().head() → 獲得 6.3`  
`&(t.tail()) →`



## type traits

泛化

```
template <class type>
struct __type_traits {
    typedef __true_type
    typedef __false_type
    typedef __false_type
    typedef __false_type
    typedef __false_type
    typedef __false_type
};
```

G2.9

```
struct __true_type { };
struct __false_type { };
```

Plain Old Data

特化

```
template<> struct __type_traits<int> {
    typedef __true_type has_trivial_default_constructor;
    typedef __true_type has_trivial_copy_constructor;
    typedef __true_type has_trivial_assignment_operator;
    typedef __true_type has_trivial_destructor;
    typedef __true_type is_POD_type;
};
```

`__type_traits<Foo>::has_trivial_destructor`

特化

```
template<> struct __type_traits<double> {
    typedef __true_type has_trivial_default_constructor;
    typedef __true_type has_trivial_copy_constructor;
    typedef __true_type has_trivial_assignment_operator;
    typedef __true_type has_trivial_destructor;
    typedef __true_type is_POD_type;
};
```

# type traits

Since C++11

[http://www.cplusplus.com/reference/type\\_traits/?kw=type\\_traits](http://www.cplusplus.com/reference/type_traits/?kw=type_traits)



## Type properties

<code>is_abstract</code>	Is abstract class (class template )
<code>is_const</code>	Is const-qualified (class template )
<code>is_empty</code>	Is empty class (class template )
<code>is_literal_type</code>	Is literal type (class template )
<code>is_pod</code>	Is POD type (class template )
<code>is_polymorphic</code>	Is polymorphic (class template )
<code>is_signed</code>	Is signed type (class template )
<code>is_standard_layout</code>	Is standard-layout type (class template )
<code>is_trivial</code>	Is trivial type (class template )
<code>is_trivially_copyable</code>	Is trivially copyable (class template )
<code>is_unsigned</code>	Is unsigned type (class template )
<code>is_volatile</code>	Is volatile-qualified (class template )

## Primary type categories

<code>is_array</code>	Is array (class template )
<code>is_class</code>	Is non-union class (class template )
<code>is_enum</code>	Is enum (class template )
<code>is_floating_point</code>	Is floating point (class template )
<code>is_function</code>	Is function (class template )
<code>is_integral</code>	Is integral (class template )
<code>is_lvalue_reference</code>	Is lvalue reference (class template )
<code>is_member_function_pointer</code>	Is member function pointer (class template )
<code>is_member_object_pointer</code>	Is member object pointer (class template )
<code>is_pointer</code>	Is pointer (class template )
<code>is_rvalue_reference</code>	Is rvalue reference (class template )
<code>is_union</code>	Is union (class template )
<code>is_void</code>	Is void (class template )

## Composite type categories

<code>is_arithmetic</code>	Is arithmetic type (class template )
<code>is_compound</code>	Is compound type (class template )
<code>is_fundamental</code>	Is fundamental type (class template )
<code>is_member_pointer</code>	Is member pointer type (class template )
<code>is_object</code>	Is object type (class template )
<code>is_reference</code>	Is reference type (class template )
<code>is_scalar</code>	Is scalar type (class template )

## type traits

[http://www.cplusplus.com/reference/type\\_traits/?kw=type\\_traits](http://www.cplusplus.com/reference/type_traits/?kw=type_traits)



### trivial

瑣碎的, 平凡的, 平淡無奇的, 無關痛癢的, 無價值的, 不重要的

#### Type features

<code>has_virtual_destructor</code>	Has virtual destructor (class template )
<code>is_assignable</code>	Is assignable (class template )
<code>is_constructible</code>	Is constructible (class template )
<code>is_copy_assignable</code>	Is copy assignable (class template )
<code>is_copy_constructible</code>	Is copy constructible (class template )
<code>is_destructible</code>	Is destructible (class template )
<code>is_default_constructible</code>	Is default constructible (class template )
<code>is_move_assignable</code>	Is move assignable (class template )
<code>is_move_constructible</code>	Is move constructible (class template )
<code>is_trivially_assignable</code>	Is trivially assignable (class template )
<code>is_trivially_constructible</code>	Is trivially constructible (class template )
<code>is_trivially_copy_assignable</code>	Is trivially copy assignable (class template )
<code>is_trivially_copy_constructible</code>	Is trivially copy constructible (class template )
<code>is_trivially_destructible</code>	Is trivially destructible (class template )
<code>is_trivially_default_constructible</code>	Is trivially default constructible (class template )
<code>is_trivially_move_assignable</code>	Is trivially move assignable (class template )
<code>is_trivially_move_constructible</code>	Is trivially move constructible (class template )
<code>is_nothrow_assignable</code>	Is assignable throwing no exceptions (class template )
<code>is_nothrow_constructible</code>	Is constructible throwing no exceptions (class template )
<code>is_nothrow_copy_assignable</code>	Is copy assignable throwing no exceptions (class template )
<code>is_nothrow_copy_constructible</code>	Is copy constructible throwing no exceptions (class template )
<code>is_nothrow_destructible</code>	Is nothrow destructible (class template )
<code>is_nothrow_default_constructible</code>	Is default constructible throwing no exceptions (class template )
<code>is_nothrow_move_assignable</code>	Is move assignable throwing no exception (class template )
<code>is_nothrow_move_constructible</code>	Is move constructible throwing no exceptions (class template )

## type traits, 測試

```
//global function template
template <typename T>
void type_traits_output(const T& x)
{
    cout << "\nType traits for type : " << typeid(T).name() << endl;

    cout << "is_void\t" << is_void<T>::value << endl;
    cout << "is_integral\t" << is_integral<T>::value << endl;
    cout << "is_floating_point\t" << is_floating_point<T>::value << endl;
    cout << "is_arithmetic\t" << is_arithmetic<T>::value << endl;
    cout << "is_signed\t" << is_signed<T>::value << endl;
    cout << "is_unsigned\t" << is_unsigned<T>::value << endl;
    cout << "is_const\t" << is_const<T>::value << endl;
    cout << "is_VOLATILE\t" << is_VOLATILE<T>::value << endl;
    cout << "is_class\t" << is_class<T>::value << endl;
    cout << "is_function\t" << is_function<T>::value << endl;
    cout << "is_reference\t" << is_reference<T>::value << endl;
    cout << "is_lvalue_reference\t" << is_lvalue_reference<T>::value << endl;
    cout << "is_rvalue_reference\t" << is_rvalue_reference<T>::value << endl;
    cout << "is_pointer\t" << is_pointer<T>::value << endl;
    cout << "is_member_pointer\t" << is_member_pointer<T>::value << endl;
    cout << "is_member_object_pointer\t" << is_member_object_pointer<T>::value << endl;
    cout << "is_member_function_pointer\t" << is_member_function_pointer<T>::value << endl;
    cout << "is_fundamental\t" << is_fundamental<T>::value << endl;
    cout << "is_scalar\t" << is_scalar<T>::value << endl;
    cout << "is_object\t" << is_object<T>::value << endl;
    cout << "is_compound\t" << is_compound<T>::value << endl;
```



## type traits, 測試

```
61  /// A string of @c char
62  typedef basic_string<char> string;
67  /// A string of @c wchar_t
68  typedef basic_string<wchar_t> wstring;
77  /// A string of @c char16_t
78  typedef basic_string<char16_t> u16string;
80  /// A string of @c char32_t
81  typedef basic_string<char32_t> u32string;
110 // 21.3 Template class basic_string
111 template<typename _CharT, typename _Traits, typename _Alloc>
112     class basic_string
113 {
146     basic_string(const basic_string& __str);
512     basic_string(basic_string&& __str)
513     #if _GLIBCXX_FULLY_DYNAMIC_STRING == 0
514         noexcept // FIXME C++11: should always be noexcept.
515     #endif
516     : _M_dataplus(__str._M_dataplus)
517     {
546     ~basic_string() _GLIBCXX_NOEXCEPT
547     { _M_rep()>_M_dispose(this->get_allocator()); }
553     basic_string&
554     operator=(const basic_string& __str)
555     { return this->assign(__str); }
589     operator=(basic_string&& __str)
590     {
591         // NB: DR 1204.
592         this->swap(__str);
593         return *this;
594     }
}
```

```
type traits for type : Ss
is_void 0
is_integral 0
is_floating_point 0
is_arithmetic 0
is_signed 0
is_unsigned 0
is_const 0
is_volatile 0
is_class 1
is_function 0
is_reference 0
is_lvalue_reference 0
is_rvalue_reference 0
is_pointer 0
is_member_pointer 0
is_member_object_pointer 0
is_member_function_pointer 0
is_fundamental 0
is_scalar 0
is_object 1
is_compound 1
is_standard_layout 1
is_pod 0
is_literal_type 0
is_empty 0
is_polymorphic 0
is_abstract 0
```

```
has_virtual_destructor 0
is_default_constructible 1
is_copy_constructible 1
is_move_constructible 1
is_copy_assignable 1
is_move_assignable 1
is_destructible 1
is_trivial 0
__has_trivial_assign 0
__has_trivial_copy 0
__has_trivial_constructor 0
__has_trivial_destructor 0
is_trivially_destructible 0
is_nothrow_default_constructible 0
is_nothrow_copy_constructible 0
is_nothrow_move_constructible 0
is_nothrow_copy_assignable 0
is_nothrow_move_assignable 0
is_nothrow_destructible 1
```

## //// type traits, 測試

```
class Foo
{
private:
    int d1, d2;
};

type_traits_output(Foo());
```

```
type traits for type : N4jj243FooE
is_void 0
is_integral 0
is_floating_point 0
is_arithmetic 0
is_signed 0
is_unsigned 0
is_const 0
is_volatile 0
is_class 1
is_function 0
is_reference 0
is_lvalue_reference 0
is_rvalue_reference 0
is_pointer 0
is_member_pointer 0
is_member_object_pointer 0
is_member_function_pointer 0
is_fundamental 0
is_scalar 0
is_object 1
is_compound 1
is_standard_layout 1
is_pod 1
is_literal_type 1
is_empty 0
is_polymorphic 0
is_abstract 0
```

```
has_virtual_destructor 0
is_default_constructible 1
is_copy_constructible 1
is_move_constructible 1
is_copyAssignable 1
is_moveAssignable 1
is_destructible 1
is_trivial 1
__has_trivial_assign 1
__has_trivial_copy 1
__has_trivial_constructor 1
__has_trivial_destructor 1
is_trivially_destructible 1
is_nothrow_default_constructible 1
is_nothrow_copy_constructible 1
is_nothrow_move_constructible 1
is_nothrow_copyAssignable 1
is_nothrow_moveAssignable 1
is_nothrow_destructible 1
```

— 侯捷 —

## ■■■ type traits, 測試

```
class Goo
{
public:
    virtual ~Goo() { }
private:
    int d1, d2;
};

type_traits_output(Goo());
```

A polymorphic class is a  
class that declares or  
inherits a virtual function.

```
type traits for type : N4jj253GooE
is_void 0
is_integral 0
is_floating_point 0
is_arithmetic 0
is_signed 0
is_unsigned 0
is_const 0
is_volatile 0
is_class 1
is_function 0
is_reference 0
is_lvalue_reference 0
is_rvalue_reference 0
is_pointer 0
is_member_pointer 0
is_member_object_pointer 0
is_member_function_pointer 0
is_fundamental 0
is_scalar 0
is_object 1
is_compound 1
is_standard_layout 0
is_pod 0
is_literal_type 0
is_empty 0
is_polymorphic 1
is_abstract 0
```

```
has_virtual_destructor 1
is_default_constructible 1
is_copy_constructible 1
is_move_constructible 1
is_copy_assignable 1
is_move_assignable 1
is_destructible 1
is_trivial 0
__has_trivial_assign 0
__has_trivial_copy 0
__has_trivial_constructor 0
__has_trivial_destructor 0
is_trivially_destructible 0
is_nothrow_default_constructible 1
is_nothrow_copy_constructible 1
is_nothrow_move_constructible 1
is_nothrow_copy_assignable 1
is_nothrow_move_assignable 1
is_nothrow_destructible 1
```

— 侯捷 —

## //// type traits, 測試

```
class Zoo
{
public:
    Zoo(int i1, int i2) : d1(i1), d2(i2) { }
    Zoo(const Zoo&) = delete;
    Zoo(Zoo&&) = default;
    Zoo& operator=(const Zoo&) = default;
    Zoo& operator=(const Zoo&&) = delete;
    virtual ~Zoo() { }

private:
    int d1, d2;
};

type_traits_output(Zoo(1,2));
```

```
type traits for type : N4jj263ZooE
is_void 0
is_integral 0
is_floating_point 0
is_arithmetic 0
is_signed 0
is_unsigned 0
is_const 0
is_volatile 0
is_class 1
is_function 0
is_reference 0
is_lvalue_reference 0
is_rvalue_reference 0
is_pointer 0
is_member_pointer 0
is_member_object_pointer 0
is_member_function_pointer 0
is_fundamental 0
is_scalar 0
is_object 1
is_compound 1
is_standard_layout 0
is_pod 0
is_literal_type 0
is_empty 0
is_polymorphic 1
is_abstract 0
has_virtual_destructor 1
is_default_constructible 0
is_copy_constructible 0
is_move_constructible 1
is_copy_assignable 1
is_move_assignable 0
is_destructible 1
is_trivial 0
__has_trivial_assign 0
__has_trivial_copy 0
__has_trivial_constructor 0
__has_trivial_destructor 0
is_trivially_destructible 0
is_nothrow_default_constructible 0
is_nothrow_copy_constructible 0
is_nothrow_move_constructible 1
is_nothrow_copy_assignable 1
is_nothrow_move_assignable 0
is_nothrow_destructible 1
```

— 侯捷 —

## //// type traits, 測試

```
type_traits_output(complex<float>());
```

```
type traits for type : St7complexIfE
is_void 0
is_integral 0
is_floating_point 0
is_arithmetic 0
is_signed 0
is_unsigned 0
is_const 0
is_volatile 0
is_class 1
is_function 0
is_reference 0
is_lvalue_reference 0
is_rvalue_reference 0
is_pointer 0
is_member_pointer 0
is_member_object_pointer 0
is_member_function_pointer 0
is_fundamental 0
is_scalar 0
is_object 1
is_compound 1
is_standard_layout 1
is_pod 0
is_literal_type 1
is_empty 0
is_polymorphic 0
is_abstract 0
has_virtual_destructor 0
is_default_constructible 1
is_copy_constructible 1
is_move_constructible 1
is_copy_assignable 1
is_move_assignable 1
is_destructible 1
is_trivial 0
__has_trivial_assign 1
__has_trivial_copy 1
__has_trivial_constructor 0
__has_trivial_destructor 1
is_trivially_destructible 1
is_nothrow_default_constructible 1
is_nothrow_copy_constructible 1
is_nothrow_move_constructible 1
is_nothrow_copy_assignable 1
is_nothrow_move_assignable 1
is_nothrow_destructible 1
```

— 侯捷 —

## ■■■ type traits, 測試

```
type_traits_output(list<int>());
```

type traits for type : St4listIiSaIiEE	
is_void	0
is_integral	0
is_floating_point	0
is_arithmetic	0
is_signed	0
is_unsigned	0
is_const	0
is_volatile	0
is_class	1
is_function	0
is_reference	0
is_lvalue_reference	0
is_rvalue_reference	0
is_pointer	0
is_member_pointer	0
is_member_object_pointer	0
is_member_function_pointer	0
is_fundamental	0
is_scalar	0
is_object	1
is_compound	1
is_standard_layout	1
is_pod	0
is_literal_type	0
is_empty	0
is_polymorphic	0
is_abstract	0
has_virtual_destructor	0
is_default_constructible	1
is_copy_constructible	1
is_move_constructible	1
is_copy_assignable	1
is_move_assignable	1
is_destructible	1
is_trivial	0
__has_trivial_assign	0
__has_trivial_copy	0
__has_trivial_constructor	0
__has_trivial_destructor	0
is_trivially_destructible	0
is_nothrow_default_constructible	1
is_nothrow_copy_constructible	0
is_nothrow_move_constructible	1
is_nothrow_copy_assignable	0
is_nothrow_move_assignable	0
is_nothrow_destructible	1

— 侯捷 —

## type traits, 實現, is\_void

```
1423 // remove_const
1424 template<typename _Tp>
1425 struct remove_const
1426 { typedef _Tp type; };
1427
1428 template<typename _Tp>
1429 struct remove_const<_Tp const>
1430 { typedef _Tp type; };
1431
1432 // remove_volatile
1433 template<typename _Tp>
1434 struct remove_volatile
1435 { typedef _Tp type; };
1436
1437 template<typename _Tp>
1438 struct remove_volatile<_Tp volatile>
1439 { typedef _Tp type; };
1440
1441 // remove_cv
1442 template<typename _Tp>
1443 struct remove_cv
1444 {
1445     typedef typename
1446         remove_const<typename remove_volatile<_Tp>::type>::type
1447         type;
1448
1449 // add_const
1450 template<typename _Tp>
1451 struct add_const
1452 { typedef _Tp const type; };

167 : public false_type { };

168
169 template<>
170 struct __is_void_helper<void>
171 : public true_type { };

172
173 // is_void
174 template<typename _Tp>
175 struct __is_voidid
176 : public __is_void_helper<typename remove_cv<_Tp>::type>::type
177 { };
```

## type traits, 實現 is\_integral

```
179 template<typename>
180     struct __is_integral_helper
181     : public false_type { };
182
183 template<>
184     struct __is_integral_helper<bool>
185     : public true_type { };
186
187 template<>
188     struct __is_integral_helper<char>
189     : public true_type { };
190
191 template<>
192     struct __is_integral_helper<signed char>
193     : public true_type { };
194
195 template<>
196     struct __is_integral_helper<unsigned char>
197     : public true_type { };
...
221 ...
222     struct __is_integral_helper<int>
223     : public true_type { };
224
225 template<>
226     struct __is_integral_helper<unsigned int>
227     : public true_type { };
228
229 template<>
230     struct __is_integral_helper<long>
231     : public true_type { };
232
233 template<>
234     struct __is_integral_helper<unsigned long>
235     : public true_type { };
236
237 template<>
238     struct __is_integral_helper<long long>
239     : public true_type { };
240
241 template<>
242     struct __is_integral_helper<unsigned long long>
243     : public true_type { };
...
255 /// is_integral
256 template<typename _Tp>
257     struct __is_integral
258     : public __is_integral_helper<typename remove_cv<_Tp>::type>::type
259     { };
```

## ■■■■ type traits, 實現 is\_class, is\_union, is\_enum, is\_pod

```
367 // is_enum
368 template<typename _Tp>
369     struct is_enum
370     : public integral_constant<bool, __is_enum(_Tp)>
371     {};
372
373 // is_union
374 template<typename _Tp>
375     struct is_union
376     : public integral_constant<bool, __is_union(_Tp)>
377     {};
378
379 // is_class
380 template<typename _Tp>
381     struct is_class
382     : public integral_constant<bool, __is_class(_Tp)>
383     {};
```

```
617 // is_pod
618 // Could use is_standard_layout && is_trivial instead of the builtin.
619 template<typename _Tp>
620     struct is_pod
621     : public integral_constant<bool, __is_pod(_Tp)>
622     {};
```

藍色這些 `_is_xxx` 未曾出現於 C++ 標準庫源代碼

## ■■■■ type traits, 實現 is\_moveAssignable

```
1219 template<typename _Tp, bool = __is_referenceable(_Tp)::value>
1220     struct __is_moveAssignable_implementation;
1221
1222 template<typename _Tp>
1223     struct __is_moveAssignable_implementation<_Tp, false>
1224     : public false_type { };
1225
1226 template<typename _Tp>
1227     struct __is_moveAssignable_implementation<_Tp, true>
1228     : public isAssignable<_Tp&, _Tp&&>
1229     { };
1230
1231 /// is_moveAssignable
1232 template<typename _Tp>
1233     struct is_moveAssignable
1234     : public __is_moveAssignable_implementation<_Tp>
1235     { };
```

```
566 // Utility to detect referenceable types ([defns.referenceable]).
567
568 template<typename _Tp>
569     struct __isReferenceable
570     : public __or__<__isObject<_Tp>, __isReference<_Tp>>::type
571     { };
572
573 template<typename _Res, typename... _Args>
574     struct __isReferenceable<_Res(_Args...)>
575     : public true_type
576     { };
577
578 template<typename _Res, typename... _Args>
579     struct __isReferenceable<_Res(_Args.....)>
580     : public true_type
581     { };
```

## cout

```
class ostream : virtual public ios  
{  
public:  
    ostream& operator<<(char c);  
    ostream& operator<<(unsigned char c) { return (*this) << (char)c; }  
    ostream& operator<<(signed char c) { return (*this) << (char)c; }  
    ostream& operator<<(const char *s);  
    ostream& operator<<(const unsigned char *s) { return (*this) << (const char*)s; }  
    ostream& operator<<(const signed char *s) { return (*this) << (const char*)s; }  
    ostream& operator<<(const void *p);  
    ostream& operator<<(int n);  
    ostream& operator<<(unsigned int n);  
    ostream& operator<<(long n);  
    ostream& operator<<(unsigned long n);  
...  
}
```

G2.9  
iostream.h



```
class _IO_ostream_withassign : public ostream {  
public:  
    _IO_ostream_withassign& operator=(ostream&);  
    _IO_ostream_withassign& operator=(_IO_ostream_withassign& rhs)  
    { return operator= (static_cast<ostream&> (rhs)); }  
};  
  
extern _IO_ostream_withassign cout;
```

G2.9  
iostream.h

## cout

G4.9

```
template<typename _CharT, typename _Traits, typename _Alloc>
inline basic_ostream<_CharT, _Traits>&
operator<<(basic_ostream<_CharT, _Traits>& __os,
           const basic_string<_CharT, _Traits, _Alloc>& __str)
{ ... }
```

```
template<typename _Tp, typename _CharT, class _Traits>
basic_ostream<_CharT, _Traits>&
operator<<(basic_ostream<_CharT, _Traits>& __os, const complex<_Tp>& __x)
{ ... }
```

```
template<class _CharT, class _Traits>
inline basic_ostream<_CharT, _Traits>&
operator<<(basic_ostream<_CharT, _Traits>& __os, __out, thread::id __id)
{ ... }
```

```
template<typename _Ch, typename _Tr, typename _Tp, _Lock_policy _Lp>
inline std::basic_ostream<_Ch, _Tr>&
operator<<(std::basic_ostream<_Ch, _Tr>& __os,
           const __shared_ptr<_Tp, _Lp>& __p)
{ ... }
```

```
/**  
 * @brief Inserts a matched string into an output stream.  
 */
```

```
template<typename _Ch_type, typename _Ch_traits, typename _Bi_iter>
inline basic_ostream<_Ch_type, _Ch_traits>&
operator<<(basic_ostream<_Ch_type, _Ch_traits>& __os,
           const sub_match<_Bi_iter>& __m)
{ ... }
```

221

## moveable 元素對於 vector 速度效能的影響

D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

```
test, with moveable elements
construction, milli-seconds : 8547
size()= 3000000
8MyString --
CCtor=0 MCtor=7194303 CAsgn=0 MAsgn=0 Dtor=7194309 Ctor=3000006 DCtor=0
copy, milli-seconds : 3500 > 差別巨大
move copy, milli-seconds : 0
swap, milli-seconds : 0

test, with non-moveable elements
construction, milli-seconds : 14235
size()= 3000000
11MyStrNoMove --
CCtor=7194303 MCtor=0 CAsgn=0 MAsgn=0 Dtor=7194307 Ctor=3000004 DCtor=0
copy, milli-seconds : 2468 > 差別巨大
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

差別很大，有時候更是巨大得多

```
for(long i=0; i< value; ++i) {
    sprintf(buf, 10, "%d", rand());
    auto ite = c1.end();
    c1.insert(ite, V1type(buf));
}
```

— 侯捷 —

222

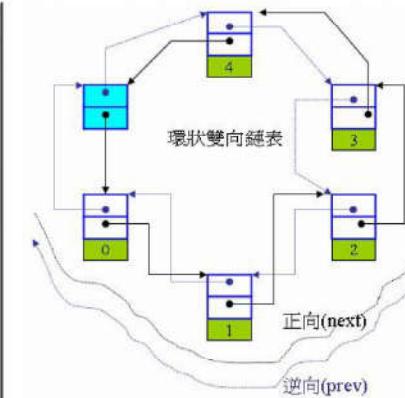
## moveable 元素對於 list 速度效能的影響

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

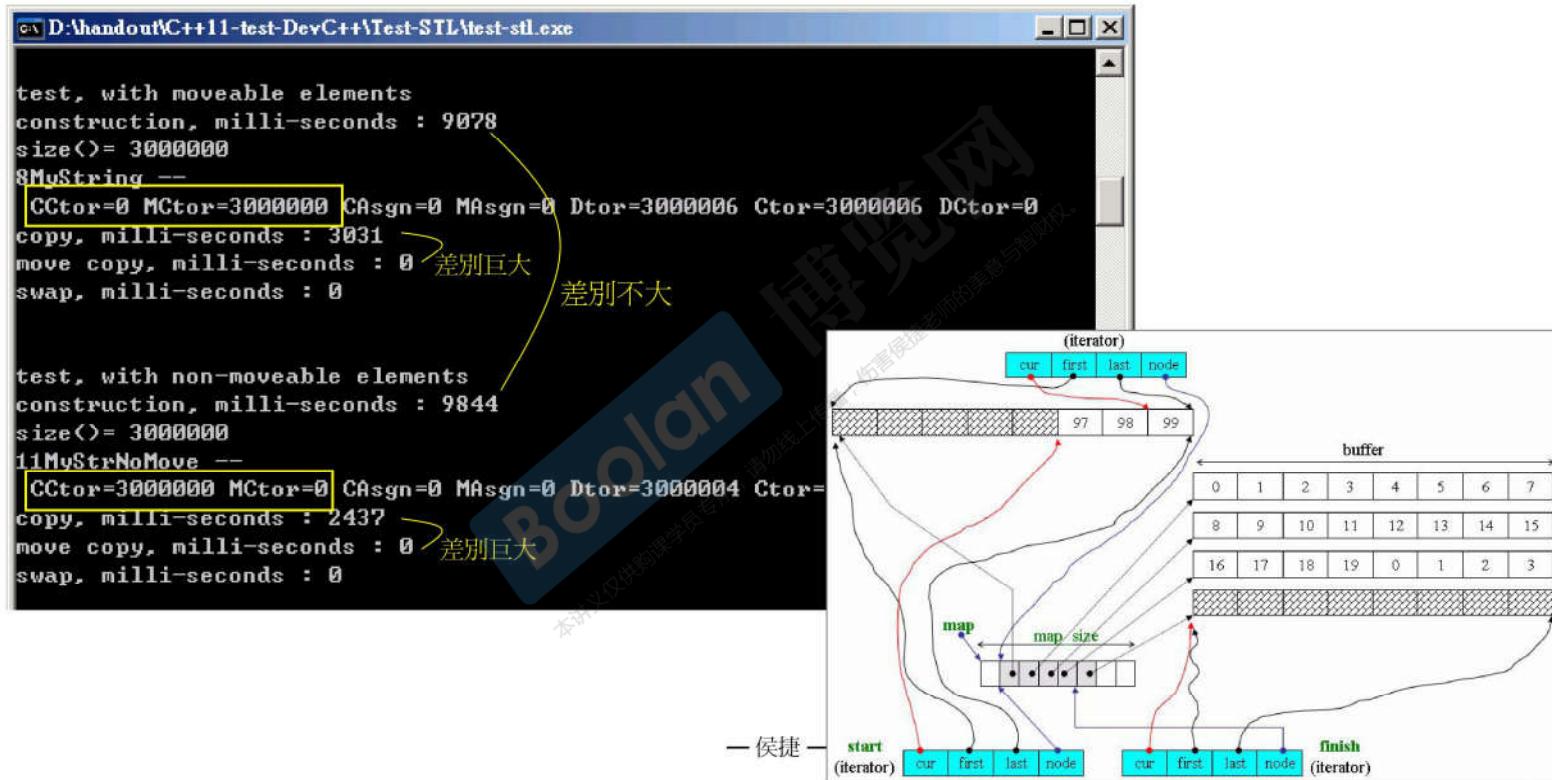
test, with moveable elements
construction, milli-seconds : 10765
size()= 3000000
8MyString --
[CCtor=0 MCtor=3000000 CAsgn=0 MAsgn=0 Dtor=3000006 Ctor=3000006 DCtor=0]
copy, milli-seconds : 4188 > 差別巨大
move copy, milli-seconds : 0
swap, milli-seconds : 0

差別不大

test, with non-moveable elements
construction, milli-seconds : 11016
size()= 3000000
11MyStrNoMove --
[CCtor=3000000 MCtor=0 CAsgn=0 MAsgn=0 Dtor=3000004 Ctor=3000004 DCtor=0]
copy, milli-seconds : 3906 > 差別巨大
move copy, milli-seconds : 0
swap, milli-seconds : 0
```



## moveable 元素對於 deque 速度效能的影響

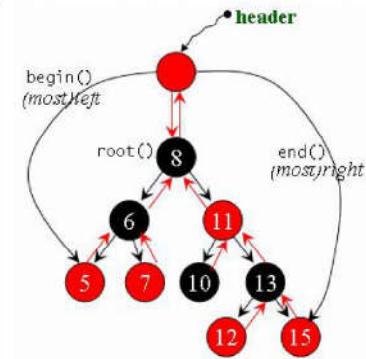


## moveable 元素對於 multiset 速度效能的影響

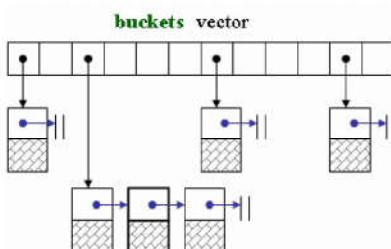
```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

test, with moveable elements
construction, milli-seconds : 74125
size()= 3000000
8MyString --
[CCtor=0 MCtor=3000000 CAsgn=0 MAsgn=0 Dtor=3000006 Ctor=3000006 DCtor=0]
copy, milli-seconds : 5438 > 差別巨大
move copy, milli-seconds : 0 差別不大
swap, milli-seconds : 0

test, with non-moveable elements
construction, milli-seconds : 74297
size()= 3000000
11MyStrNoMove --
[CCtor=3000000 MCtor=0 CAsgn=0 MAsgn=0 Dtor=3000004 Ctor=3000004 DCtor=0]
copy, milli-seconds : 4765 > 差別巨大
move copy, milli-seconds : 0 差別巨大
swap, milli-seconds : 0
```



## moveable 元素對於 unordered\_multiset 速度效能的影響



The diagram illustrates the internal structure of an `unordered_multiset`. It consists of an array of `buckets` and a `vector` of pointers. Each bucket contains a linked list of nodes, where each node has a pointer to the next node in the list. The `vector` contains pointers to the first node of each bucket.

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
test, with moveable elements
construction, milli-seconds : 23891
size()= 3000000
8MyString --
CCtor=0 MCtor=3000000 CAsgn=0 MASgn=0 Dtor=3000006 Ctor=3000006 DCtor=0
copy, milli-seconds : 7812 > 差別巨大
move copy, milli-seconds : 0 差別不大
swap, milli-seconds : 0

test, with non-moveable elements
construction, milli-seconds : 24672
size()= 3000000
11MuStrNoMove --
CCtor=3000000 MCtor=0 CAsgn=0 MASgn=0 Dtor=3000004 Ctor=3000004 DCtor=0
copy, milli-seconds : 7188 > 差別巨大
move copy, milli-seconds : 0 差別巨大
swap, milli-seconds : 0
```

本讲义仅供教学员使用，勿线上上传，伤害敏捷老师的美意与用心

## 寫一個 moveable class

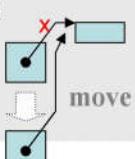
```
class MyString {
public:
    static size_t DCtor; //累計 default-ctor 呼叫次數
    static size_t Ctor; //累計 ctor 呼叫次數
    static size_t CCtor; //累計 copy-ctor 呼叫次數
    static size_t CAsgn; //累計 copy-asgn 呼叫次數
    static size_t MCtor; //累計 move-ctor 呼叫次數
    static size_t MAsgn; //累計 move-asgn 呼叫次數
    static size_t Dtor; //累計 dtor 呼叫次數
private:
    char* _data;
    size_t _len;
    void _init_data(const char *s) {
        _data = new char[_len+1];
        memcpy(_data, s, _len);
        _data[_len] = '\0';
    }
public:
    //default ctor
    MyString() : _data(NULL), _len(0) { ++DCtor; }

    //ctor
    MyString(const char* p) : _len(strlen(p)) {
        ++Ctor;
        _init_data(p);
    }
}
```

```
// copy ctor
MyString(const MyString& str) : _len(str._len) {
    ++CCtor;
    _init_data(str._data); //COPY
}

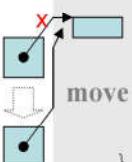
//move ctor, with "noexcept"
MyString(MyString&& str) noexcept
: _data(str._data), _len(str._len) {
    ++MCtor;
    str._len = 0;
    str._data = NULL; //避免 delete (in dtor)
}

//copy assignment
MyString& operator=(const MyString& str) {
    ++CAsgn;
    if (this != &str) {
        if (_data) delete _data;
        _len = str._len;
        _init_data(str._data); //COPY!
    }
    else {
    }
    return *this;
}
```



## 寫一個 moveable class

```
//move assignment
MyString& operator=(MyString&& str) noexcept {
    ++MASgn;
    if (this != &str) {
        if (_data) delete _data;
        _len = str._len;
        _data = str._data; //MOVE!
        str._len = 0;
        str._data = NULL; //避免 delete (in dtor)
    }
    return *this;
}
//dtor
virtual ~MyString() {
    ++Dtor;
    if (_data) {
        delete _data;
    }
}
bool
operator<(const MyString& rhs) const //為了 set
{
    return std::string(this->_data)
        < std::string(rhs._data);
    //借用現成事實：string 已能比較大小.
}
```



```
bool
operator==(const MyString& rhs) const //為了 set
{
    return std::string(this->_data)
        == std::string(rhs._data);
    //借用現成事實：string 已能判斷相等.
}

char* get() const { return _data; }

size_t MyString::DCtor=0;
size_t MyString::Ctor=0;
size_t MyString::CCtor=0;
size_t MyString::CASgn=0;
size_t MyString::MCtor=0;
size_t MyString::MASgn=0;
size_t MyString::Dtor=0;

namespace std //必須放在 std 內
{
template<>
struct hash<MyString> { //這是為了 unordered containers
    size_t
    operator()(const MyString& s) const noexcept
    {
        return hash<string>()(string(s.get()));
    }
    //借用現有的 hash<string>
    // (在 ...4.9.2\include\c++\bits\basic_string.h)
};
```

— 侯捷

## /// 测试函数

```
test, with moveable elements
construction, milli-seconds : 8547
size()= 3000000
8MyString --
  CCtor=0 MCtor=7194303 CAsgn=0 MAsgn=
copy, milli-seconds : 3500
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

```
#include <typeinfo> //typeid()
template<typename T>
void output_static_data(const T& myStr)
{
    cout << typeid(myStr).name() << " -- " << endl;
    cout << "CCtor=" << T::CCtor
        << "MCtor=" << T::MCtor
        << "CAsgn=" << T::CAsgn
        << "MAsgn=" << T::MAsgn
        << "Dtor=" << T::Dtor
        << "Ctor=" << T::Ctor
        << "DCtor=" << T::DCtor
        << endl;
}
```

```
template<typename M, typename NM>
void test_moveable(M c1, NM c2, long& value)
{
    char buf[10];
    1 //測試 moveable
    test_moveable(vector<MyString>(),
                  vector<MyStrNoMove>(),
                  value);
    //測試 non-moveable
    ...
}

typedef typename iterator_traits<typename M::iterator>::value_type V1type;
clock_t timeStart = clock();
for(long i=0; i< value; ++i) {
    sprintf(buf, 10, "%d", rand()); //隨機數, 放進 buf (轉換為字符串)
    auto ite = c1.end();           //定位尾端
    c1.insert(ite, V1type(buf));  //安插於尾端 (對 RB-tree 和 HT 這只是hint)
}
cout << "construction, milli-seconds ." << (clock()-timeStart) << endl;
cout << "size()= " << c1.size() << endl;
output_static_data(*c1.begin()));

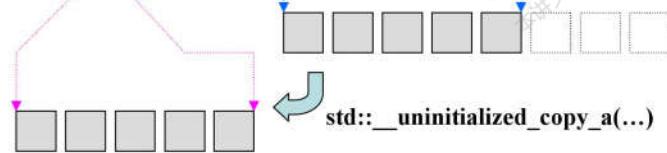
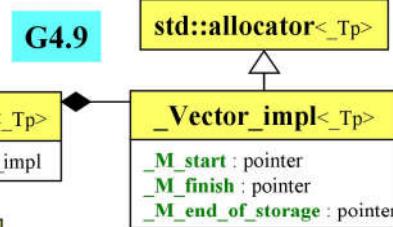
M c11(c1); //關於 std::move
M c12(std::move(c1)); //必須確保接下來不會再用到 c1
c11.swap(c12);
```

## vector 的 copy ctor

```

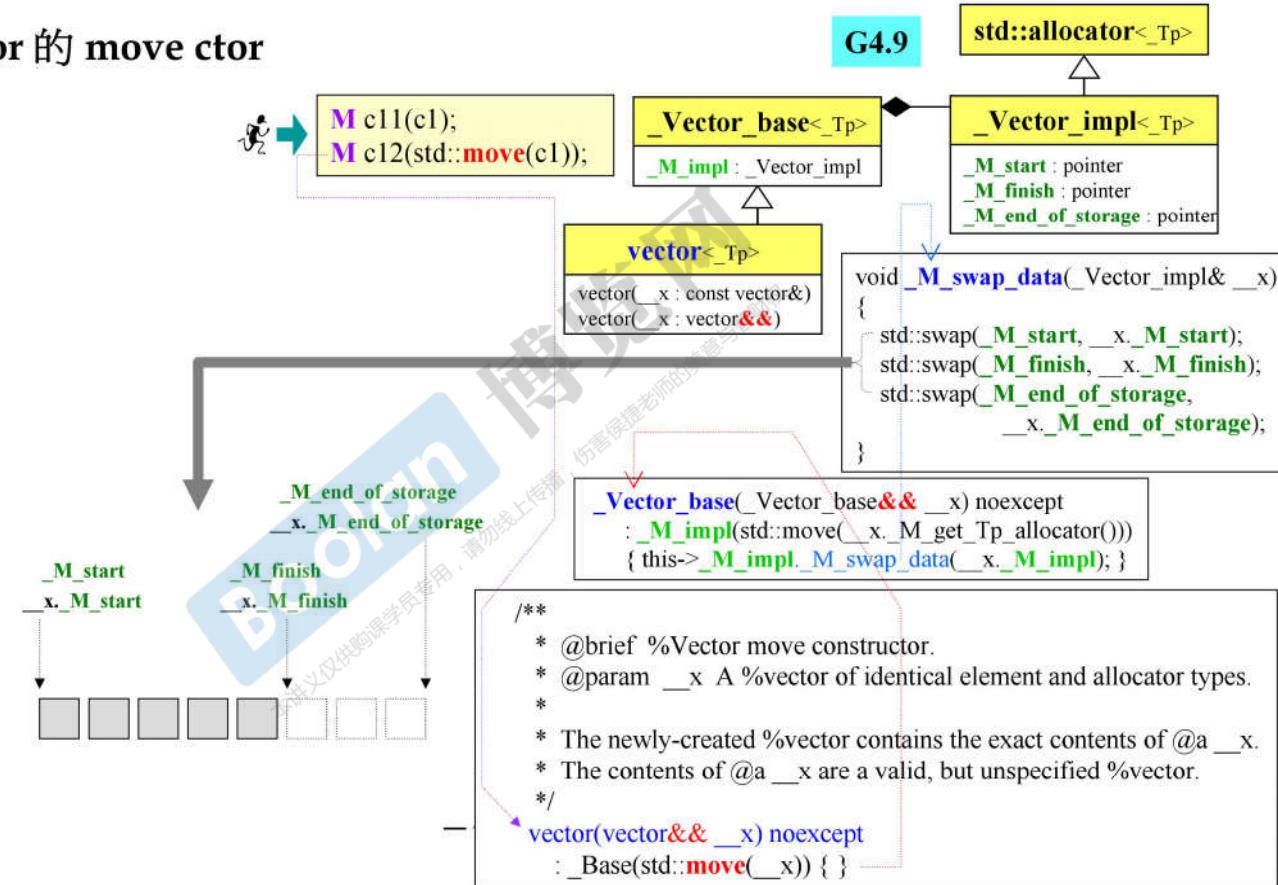

    /**
     * @brief %Vector copy constructor.
     * @param __x A %vector of identical element and allocator types.
     *
     * The newly-created %vector uses a copy of the allocation
     * object used by @a __x. All the elements of @a __x are copied,
     * but any extra memory in
     * @a __x (for fast expansion) will not be copied.
     */
    vector(const vector& __x)
        : _Base(__x.size(),
            __Alloc_traits::S_select_on_copy(__x._M_get_Tp_allocator()))
    { this->_M_impl._M_finish =
        std::__uninitialized_copy_a(__x.begin(), __x.end(),
            this->_M_impl._M_start,
            _M_get_Tp_allocator());
    }


```



— 侯捷 —

## vector 的 move ctor



## std::string 是否 moveable ?

```

579 #if __cplusplus >= 201103L
580 /**
581 * @brief Move assign the value of @a str to this string.
582 * @param __str Source string.
583 *
584 * The contents of @a str are moved into this string (without copying).
585 * @a str is a valid, but unspecified string.
586 */
587 // PR 58265, this should be noexcept.
588 basic_string&
589 operator=(basic_string&& __str)
590 {
591 // NB: DR 1204.
592 this->swap(__str);
593 return *this;
594 }

504 #if __cplusplus >= 201103L
505 /**
506 * @brief Move construct str.
507 * @param __str Source strin.
508 *
509 * The newly-created string contains the exact contents of @a __str.
510 * @a __str is a valid, but unspecified string.
511 */
512 basic_string(basic_string&& __str)
513 #if GLIBCXX_FULLY_DYNAMIC_STRING == 0
514 noexcept // FIXME C++11: should always be noexcept.
515 #endif
516 : _M_dataplus(__str._M_dataplus)
517 {
518 #if GLIBCXX_FULLY_DYNAMIC_STRING == 0
519 __str._M_data(_S_empty_rep()._M_refdata());
520 #else
521 __str._M_data(_S_construct(size_type(), _CharT(), get_allocator()));
522 #endif
523 }

```

<pre> 549 /** 550 * @brief Assign the value of @a str to this string. 551 * @param __str Source string. 552 */ 553 basic_string&amp; 554 operator=(const basic_string&amp; __str) 555 { return this-&gt;assign(__str); } </pre>	<pre> 456 /** 457 * @brief Construct string with copy of value of @a str. 458 * @param __str Source string. 459 */ 460 basic_string(const basic_string&amp; __str); </pre>
<pre> 168 template&lt;typename _CharT, typename _Traits, typename _Alloc&gt; 169 basic_string&lt;_CharT, _Traits, _Alloc&gt;; 170 basic_string(const basic_string&amp; __str) 171 : _M_dataplus(__str._M_rep()&gt;_M_grab(_Alloc(__str.get_allocator())), 172 __str._M_data(_M_allocate(_str.get_allocator(), 173 __str.get_allocator()))) 174 { } </pre>	



The End

