



**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。



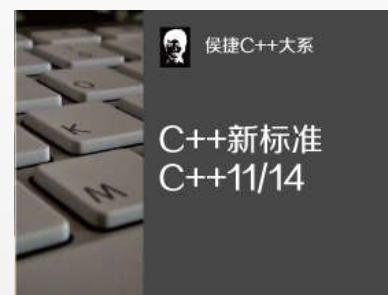
## C++面向对象开发

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++11-14

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 画面)

The screenshot shows the Microsoft Visual Studio IDE interface. The title bar reads "BoostSmartPtr - Microsoft Visual C++ - [EA...\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".

The left pane displays the "ClassView" and "FileView" windows. The "FileView" window shows the workspace structure:

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

The main code editor window contains C++ code:

```
// <Beyond the C++ Standard Library, An
#include <string>
#include <iostream>
#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:
}
```

The status bar at the bottom indicates "Configuration: BoostSmartPtr - Win32 Debug". Below that, it says "BoostSmartPtr.exe - 0 error(s), 0 warning(s)".

(Dev-C++ 5.11 畫面; with GNU 4.9.2)



## 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 is a banner stating "C++ is 30 years old" with the date "October 14, 1985". Below the banner, the page title "C++ reference" is displayed, along with links for "C++98, C++03, C++11, C++14". The main navigation menu on the left includes sections for "Language", "Headers", "Library concepts", "Utilities library", "Technical specifications", and "Standard library extensions". The "Language" section lists "ASCII chart", "Compiler support", "Preprocessor", "Keywords", "Operator precedence", "Escape sequences", and "Fundamental types". The "Headers" section lists "Type support", "Dynamic memory management", "Error handling", "Program Utilities", "Date and time", "bitset", "Function objects", "pair - tuple (C++11)", and "integer\_sequence (C++14)". The "Library concepts" section lists "Containers library", "Algorithms library", "Iterators library", "Numerics library", and "String library". The "Utilities library" section lists "Input/output library", "Localizations library", "Regular expressions library (C++11)", "Atomic operations library (C++11)", and "Thread support library (C++11)". The "Technical specifications" section lists "Standard library extensions (library fundamentals TS)" and "Standard library extensions v2 (library fundamentals TS v2)". The "Standard library extensions (library fundamentals TS)" section includes "optional - any - basic\_string\_view", "polymorphic allocators - search", and "source\_location - ostream\_joiner". The "Standard library extensions v2 (library fundamentals TS v2)" section includes "propagate\_const - not\_fn - observer\_ptr", "detection idiom - uniform\_container erasure", and "uniform\_allocator - uniform\_allocator\_traits".



## 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:

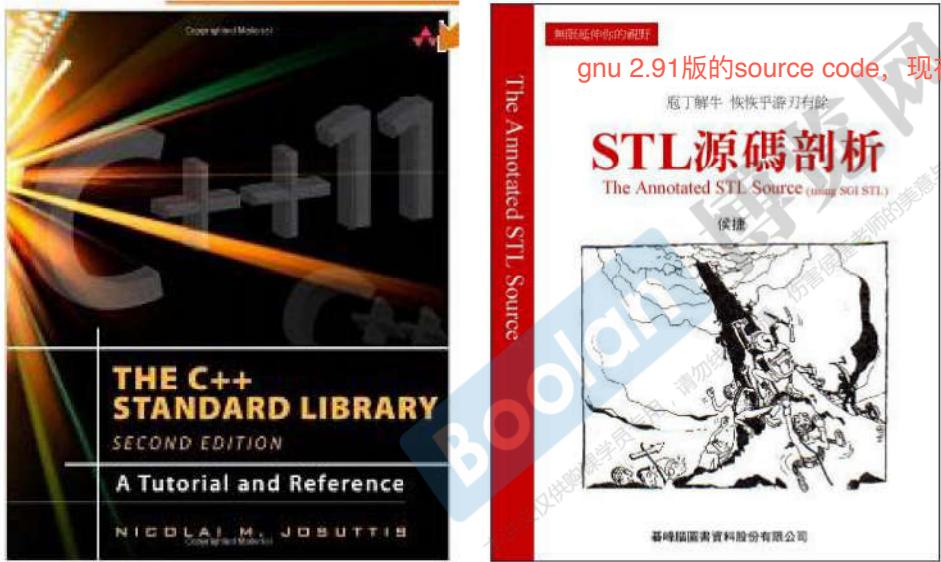
```
graph TD; std::Deque_base["std::Deque_base< _Tp, _Alloc >"] --> std::Deque_base["std::Deque_base< _Tp, _Alloc >"]; std::Deque_base --> std::deque["std::deque"]
```

List of all members.

**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

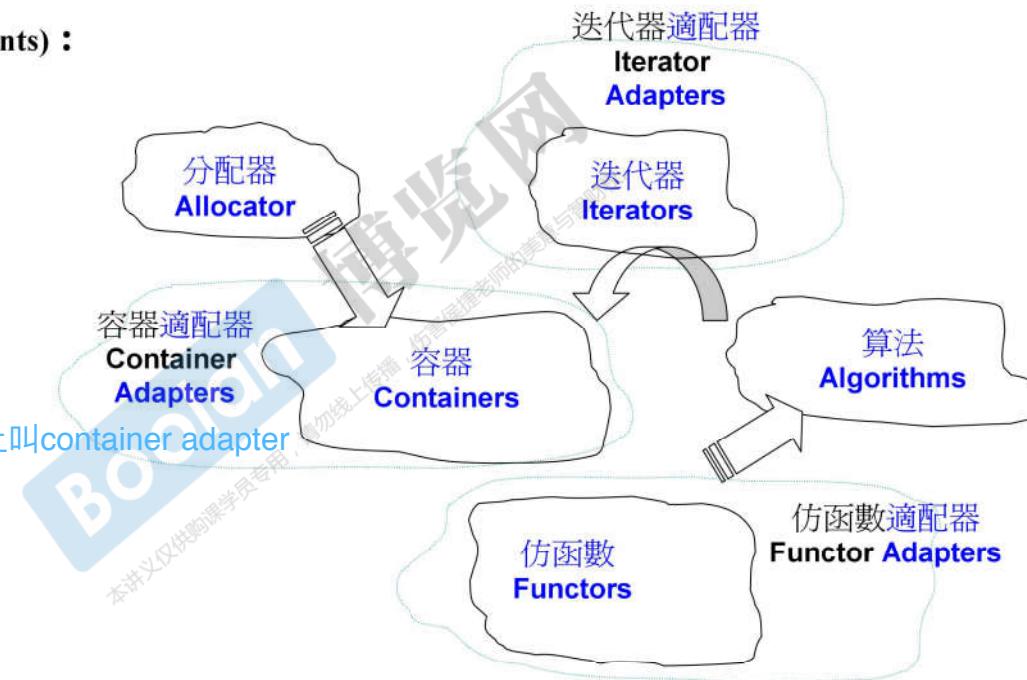
內有對 C++ 標準庫的各種測試

## ■■■■ 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)

stack & queue没有实际的数据结构，技术上叫container adapter



containers封装了内存的操作，所以需要allocator的支持

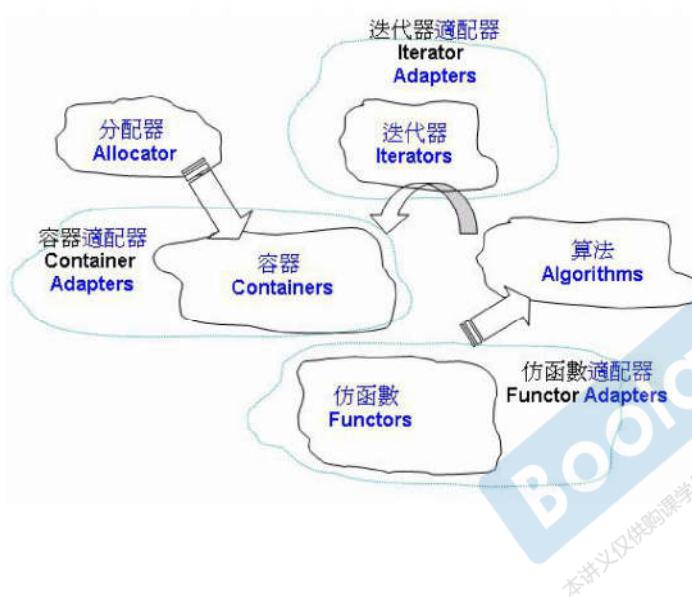
algorithms通过iterators操作containers中的数据

oo封装是数据和方法进行封装，而gp独立了containers和algorithms

functors作用像函数

adapters对containers/functors/iterators做转换

## 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
15         not1(bind2nd(less<int>(), 40)));
16
17 }
```

Annotations for the code:

- allocator**: Points to the `allocator<int>` template in line 11.
- container**: Points to the `vector` type in line 11.
- iterator**: Points to the range `vi.begin()` to `vi.end()` in line 13.
- algorithm**: Points to the `count_if` function in line 13.
- function adapter (negator)**: Points to the `not1` function in line 15.
- function adapter (binder)**: Points to the `bind2nd` function in line 15.
- predicate**: Points to the `less<int>()` function in line 15.
- function object**: Points to the argument `40` in line 15.

std::allocator<int>也是个template,每次分配一个int

— 侯捷 —

每个元素和40比，bind2nd()绑定第二参数，本来是a和b比，现在是a和40比

not1()后条件变成>=40

"条件"在stl中叫predicate判断式

## ■■■ 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。

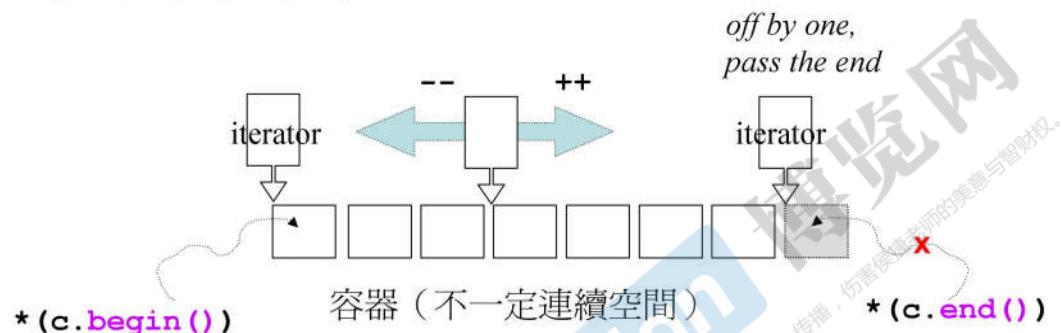
n要很大，才有意義

— 侯捷 —

14

## ■■■■ “前閉後開” 區間

[ ] ( ) [ ]



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

## |||| 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;  
}
```

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

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



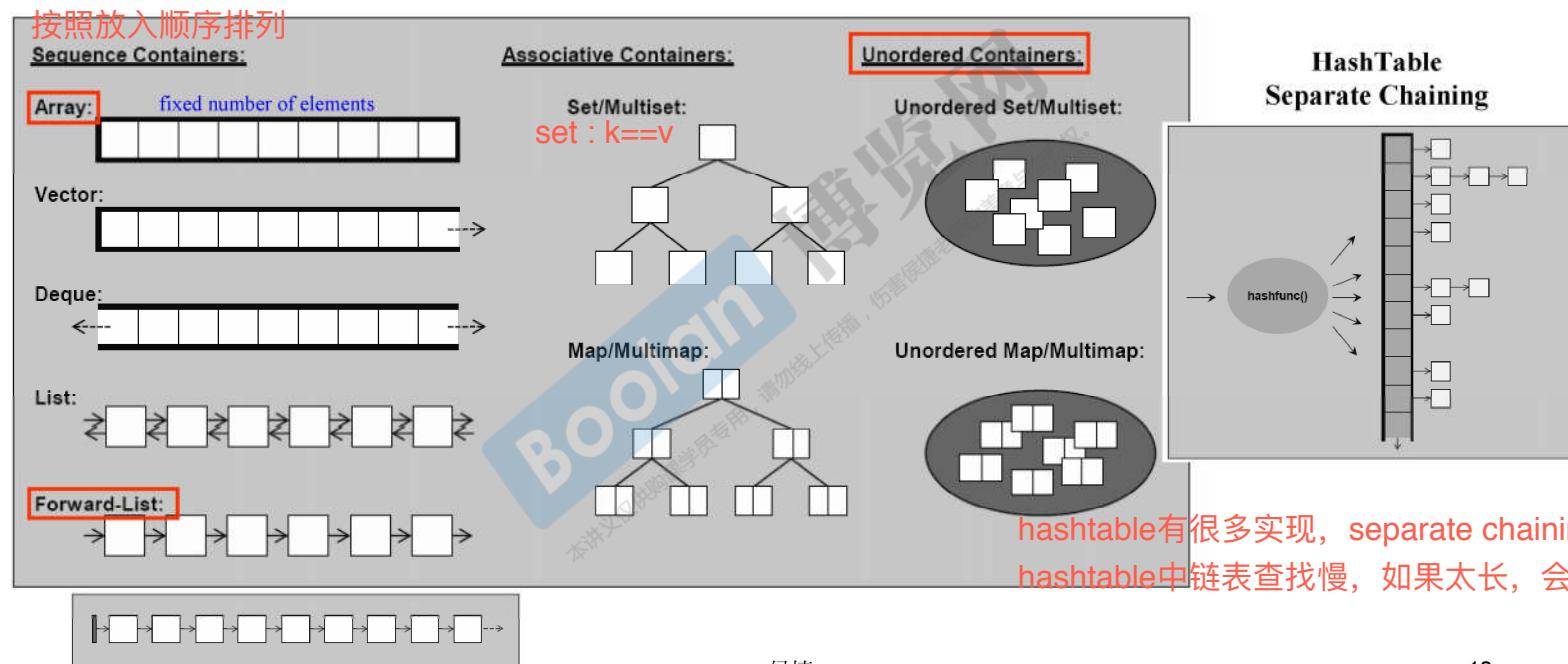
```
list<string> c;  
...  
auto ite = ::find(c.begin(), c.end(), target);
```

set/multiset/map/multimap : rb tree, 高度平衡二分树

平衡：左右自调整成差不多一样，不会出现查找一个很长的右臂等

标准库没规定实现方法，但因为rb tree好，所以各家编译器所带的标准库都用rb tree实现set/map等

## 容器 - 結構與分類



— 侯捷 —

18

associative containers :  $<k,v>$ 适合快速查找

unordered containers 属于 associative containers, 背后是hash table

unordered : 元素位置不定, 每次操作都可能引起变化

Array: 把语言级别的数组包装成class

deque: 两端可进可出可扩充

list: 双向环状链表, 环状是为了操作的便利

Forward-List比list省内存

## 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  }
```

c func : qsort()需要compare func  
qsort()快排

## 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; .data(),数组在内存中的起始地址
72
73         long target = get_a_target_long();
74
75         timeStart = clock(); clock(),从程序运行到运行到75行的ms
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
```

20

bsearch()要求有序，否则只能用sequence search，很慢



## 使用容器 vector

```

Vector: [ ] →

```

```

C:\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()>bssearch(), milli-seconds : 2765
found, 23456

```

vector成长空间是原有空间的2倍

```

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
100 vector<string> c; 一个字符串大小等于指针大小?
101 char buf[10];
102
103 clock_t timeStart = clock();
104 for(long i=0; i< value; ++i)
105 {
106 try {
107 snprintf(buf, 10, "%d", rand());
108 c.push_back(string(buf));
109 }
110 catch(exception& p) {
111 cout << "i=" << i << " " << p.what() << endl;
112 //曾经最高 i=58389486 then std::bad_alloc
113 abort(); abort(), 异常发生后, 用此方式退出程序
114 }
115 }
116 cout << "milli-seconds : " << (clock()-timeStart) << endl;
117 cout << "vector.size()= " << c.size() << endl;
118 cout << "vector.front()= " << c.front() << endl;
119 cout << "vector.back()= " << c.back() << endl;
120 cout << "vector.data()= " << c.data() << endl;
121 cout << "vector.capacity()= " << c.capacity() << endl;
122
123

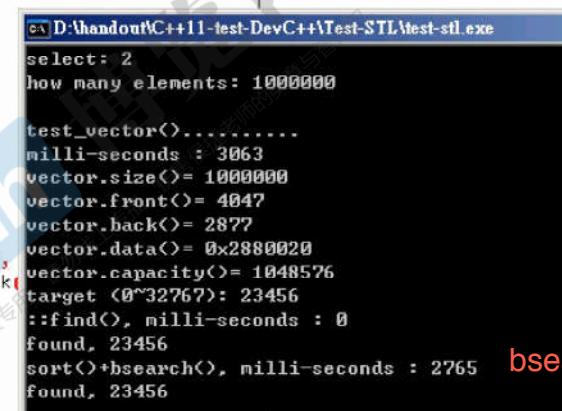
```

c++标准库有70多个算法，`find()`是模板函数，用法同一般函数  
`::find()`直接去全局找，不加`::`，先去局部再去全局



## 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); ::find(),逐一查找
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 }
```



```
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 bsearch(),c本身有
found, 23456
```

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

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

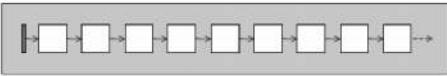
/\*\* Returns the size() of the largest possible %list. \*/
size\_type  
max\_size() const \_GLIBCXX\_NOEXCEPT  
{ return \_M\_get\_Node\_allocator().max\_size(); }

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

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

自己有, 最好用自己的sort()

## 使用容器 forward\_list



```
D:\handout\c++11-test\DevC++\Test-STL\test-s
select: 4
how many elements: 1000000

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
```

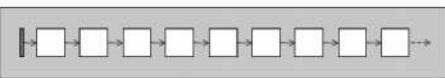
```
210 namespace jj04
211 {
212     void test_forward_list(long& value)
213     {
214         cout << "\ntest_forward_list()..... \n";
215
216         forward_list<string> c;
217         char buf[10];
218
219         clock_t timeStart = clock();
220         for(long i=0; i< value; ++i)
221         {
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         string target = get_a_target_string();
238         timeStart = clock();
239         auto pItem = ::find(c.begin(), c.end(), target);
240         cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl; //
241
242         if (pItem != c.end())
243             cout << "found, " << *pItem << endl;
244         else
245             cout << "not found! " << endl;
246
247         timeStart = clock();
248         c.sort();
249         cout << "c.sort(), milli-seconds : " << (clock()-timeStart) << endl; //
250
251     }
252 }
```

forward\_list只有push\_front()

没有访问最后一个元素的函数，很慢

gnu g++编译器所带的标准库中有个`slist`, 和`forward_list`一样  
`slist`很早之前就有, gnu自带的非标准的

## 使用容器 `slist`



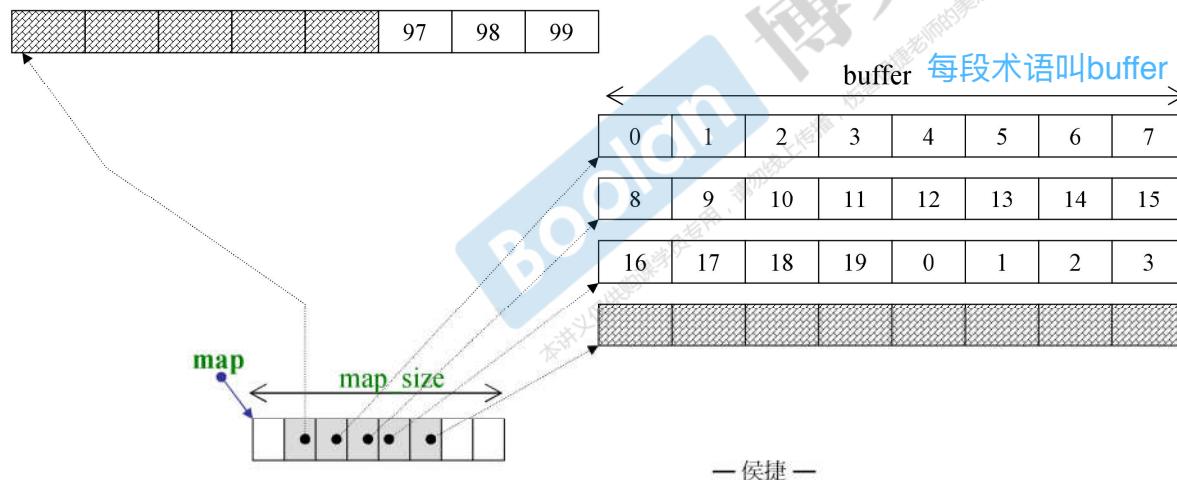
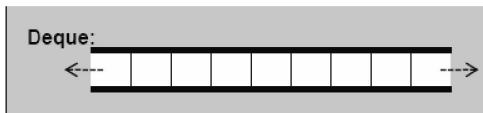
```
c:\D:\handout\c++11-test\DevC++\Test-S>
select: 10
how many elements: 1000000
test_slist()..... .
milli-seconds : 3046
```

```
508 #include <ext\slist>
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                 sprintf(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 }
```



两边可进可出可扩充

使用容器 **deque** 分段连续, **deque**号称段和段之间, 这是假象, 但必须做出



26

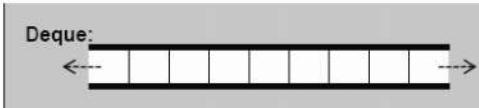
让使用者感觉段和段之间连续,

如: iterator i 指向98, ++到99, ++到99后面, 没意义, deque应该能知道走到最后, 并自动跳到0

若已有的buffer满了, 在push\_back时, 再分配新buffer, 并指向它。或push\_front

buffer大小

## 使用容器 deque



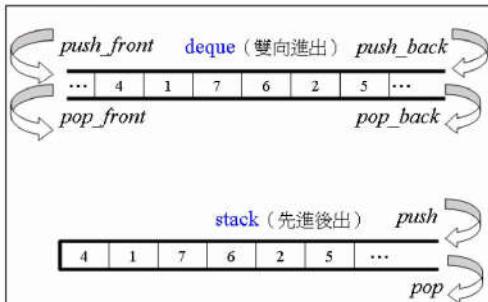
```
D:\handout\c++11-test\DevC++\Test-STL\test_deque.cpp
select: 5
how many elements: 1000000

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 }
```

以上是sequence container, 查找都需要时间  
association container查找很快

## 使用容器 stack



```
c:\D\handout\C++11-test-DevC++\Test
select: 17
how many elements: 300000

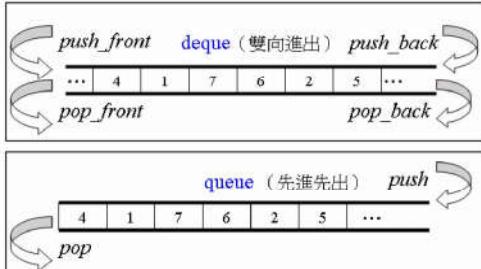
test_stack().....
milli-seconds : 812
stack.size()= 300000
stack.top()= 23929
stack.size()= 299999
stack.top()= 12911
```

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

— 侯捷 —

28

## 使用容器 queue



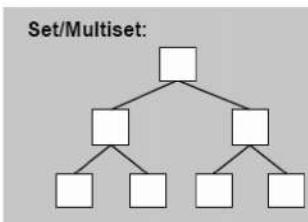
```
ex D:\handoutC++11-test-DevC++\Test-  
select: 18  
how many elements: 300000  
  
test_queue().....  
milli-seconds : 890  
queue.size()= 300000  
queue.front()= 12074  
queue.back()= 14585  
queue.size()= 299999  
queue.front()= 6888  
queue.back()= 14585
```

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

association container插入慢，查找快



## 使用容器 multiset



```

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

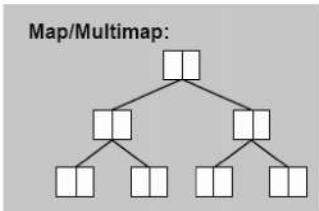
test_multiset<>. .....
milli-seconds : 6609
multiset.size()= 1000000
multiset.max_size()= 214748364
target <0~32767>: 23456
::find(), milli-seconds : 203
found, 23456
c.find(), milli-seconds : 0
found, 23456
  
```

```

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 } 
```

insert后已经排好序，6609ms花在排序上

## 使用容器 multimap



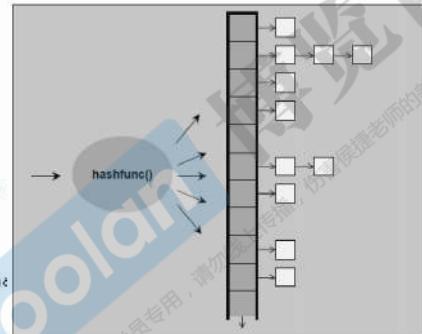
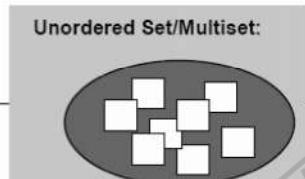
```
ex D:\handout\c++11-test-DevC++\Test-STL\m  
select: 7  
how many elements: 1000000  
  
test_multimap<>.  
milli-seconds : 4812  
multimap.size()= 1000000  
multimap.max_size()= 178956970  
target <0~32767>: 23456  
c.find(), milli-seconds : 0  
found, value=29247
```

```
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; Second是pair的源码  
396         else  
397             cout << "not found! " << endl;  
398     }  
399 }
```

以前gnu c提供的非标准的叫hash\_multiset

## 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     } 某个篮子有几个元素
```



```
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 <0x32767>: 23456
::find(), milli-seconds : 109
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

bucket\_count() > size()合理，肯定会多，出于设计的考虑，见第二讲

设计上，单个篮子不能放太多元素，否则循序查找会慢，所以经验上若元素格式 $\geq$ 篮子个数，篮子就要重新扩充为大约2倍  
load\_factor()，见第二讲

max\_load\_factor()永远=1

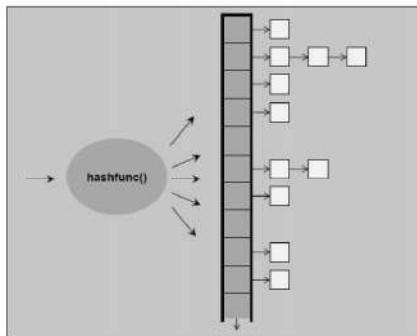
## 使用容器 `unordered_multiset`

```
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`

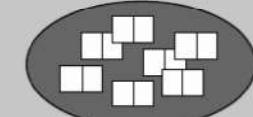


```
ex>D:\handout\c++11-test\DevC++\Test-STL\test-stl.exe
select: 9
how many elements: 1000000

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:



## 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:

```
graph TD; Root[ ] --- L1_1[ ]; Root --- L1_2[ ]; L1_1 --- L2_1[ ]; L1_1 --- L2_2[ ]; L1_2 --- L2_3[ ]; L1_2 --- L2_4[ ];
```

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

```
select: 13
how many elements: 1000000

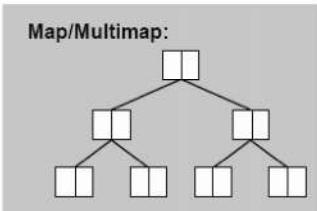
test_set().....
milli-seconds : 3922
set.size()= 32768
set.max_size()= 214748364
target <0~32767>: 23456
::find(), milli-seconds : 0
found, 23456
c.find(), milli-seconds : 0
found, 23456
```

35

set.size()=32768, int范围内的每个数在100万次的插入后，每个都有机会被访问  
重复放入会失败、不会死机



## 使用容器 map



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

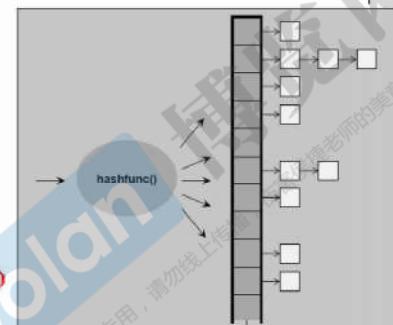
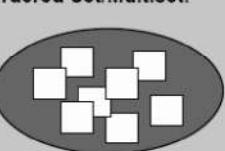
test_map().....
milli-seconds : 4890
map.size()= 1000000
map.max_size()= 178956970
target <0~32767>: 23456
c.find(), milli-seconds : 0
found, value=19128
```

```
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); // map会把i和buf变成pair
712             }
713             catch(exception& p)
714             {
715                 cout << "i=" << i << " " << p.what() << endl; // abort();
716             }
717         }
718         cout << "milli-seconds : " << (clock()-timeStart) << endl; //
719         cout << "map.size()= " << c.size() << endl;
720         cout << "map.max_size()= " << c.max_size() << endl;
721
722         long target = get_a_target_long();
723         timeStart = clock();
724         auto pItem = c.find(target);
725         cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
726         if (pItem != c.end())
727             cout << "found, value=" << (*pItem).second << endl;
728         else
729             cout << "not found! " << endl;
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

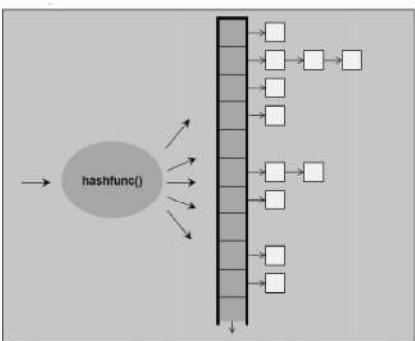
test_unordered_set().....
milli-seconds : 2891
unordered_set.size()= 32768
unordered_set.max_size()= 357913941
unordered_set.bucket_count()= 62233
unordered_set.load_factor()= 0.526537
unordered_set.max_load_factor()= 1
unordered_set.max_bucket_count()= 357913941
bucket #0 has 1 elements.
bucket #1 has 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`

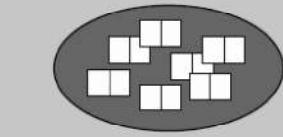


```
D:\handout\c++11-test\DevC++\Test-STL\test-stl.c
select: 16
how many elements: 1000000

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**

除了上述，还有**priority\_queue**<>和**heap**

上面4个是gnu c非标准实现，支持**unordered\_xxx**后仍存在，但可能头文件位置有变动

同**slist**，头文件变为#include <ext/slist>

非标准实现，因为重要，几乎各家厂商都实现，但c++1.0标准库没有实现

支持容器对内存的操作

现在开始，只讲gnu c下，vc不一定有或名字不同

## 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
```

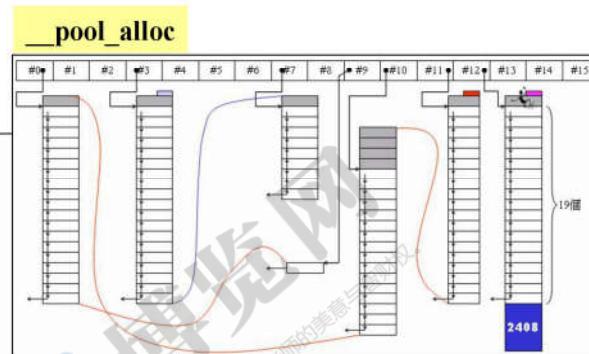
## 使用分配器 allocator

```

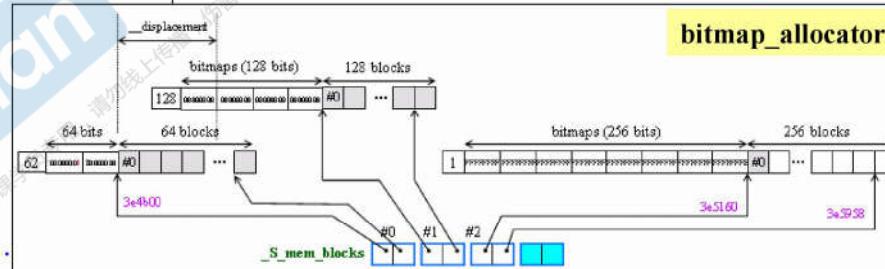
916 #include <list>
917 #include <stdexcept>
918 #include <string>
919 #include <cstdlib>           //abort()
920 #include <cstdio>            //snprintf()
921 #include <algorithm>          //find()
922 #include <iostream>
923 #include <ctime>
924
925 #include <cstddef>
926 #include <memory>             //内含 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;

```

不在std下



gnu c下allocator



42

mt : multi\_thread

<ext/xxx> : ext下都是扩充的，非标准的

单独使用allocator<>的情况没有，因为真正的工具是容器

最后都用malloc/free



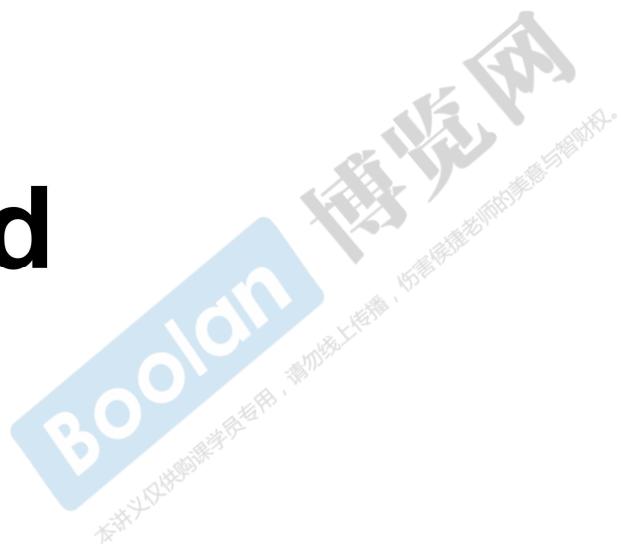
使用分配器  
allocator

```
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         sprintf(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 cout << "a lot of push_back(), milli-seconds : "
987 << (clock()-timeStart) << endl;
```

969  
970  
971 //test all allocators' allocate() & deallocate();
972 int\* p;
973 allocator<int> alloc1;
974 p = alloc1.allocate(1);
975 alloc1.deallocate(p,1);
976
977 \_\_gnu\_cxx::malloc\_allocator<int> alloc2;
978 p = alloc2.allocate(1);
979 alloc2.deallocate(p,1);
980
981 \_\_gnu\_cxx::new\_allocator<int> alloc3;
982 p = alloc3.allocate(1);
983 alloc3.deallocate(p,1);
984
985 \_\_gnu\_cxx::\_\_pool\_alloc<int> alloc4;
986 p = alloc4.allocate(2);
987 alloc4.deallocate(p,2);
988
989 \_\_gnu\_cxx::\_\_mt\_alloc<int> alloc5;
990 p = alloc5.allocate(1);
991 alloc5.deallocate(p,1);
992
993 \_\_gnu\_cxx::bitmap\_allocator<int> alloc6;
994 p = alloc6.allocate(3);
995 alloc6.deallocate(p,3); |

释放时需指定个数，所以一般不直接用allocator

# 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

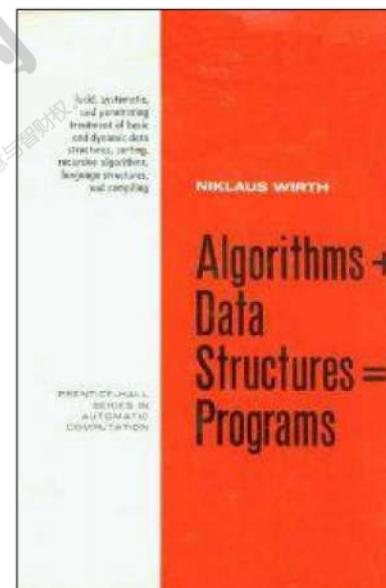
源碼之前  
了無秘密



## ■■■■ 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 says "Ready", "Ln 1, Col 1", "REC COL OVR READ", and "快速".

**Workspace 'BoostSmartPtr':**

- BoostSmartPtr files
- Source Files
  - BoostSmartPtrTest.cpp
- Header Files
- Resource Files

**Code Editor:**

```
// <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:
}
```

**Build Output:**

```
-----Configuration: BoostSmartPtr - Win32 Debug-----
BoostSmartPtr.exe - 0 error(s), 0 warning(s)
```

48

不同版本的标准库，接口一样，实现也差不多(如采用rb tree/hash table)，小部分不同  
gnu c++2.91版最好，4.9及以后基本稳定

## 標準庫版本, Visual C++

files layout

...\\include →



— 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 for "TestC++11.cpp". The code uses `#include <malloc\_allocator.h>` and `#include <gnu\_cxx/alloc.h>` to experiment with different allocator types. It includes definitions for `list`, `deque`, and `vector` using these allocators. The code also demonstrates the use of `sizeof` operator and `cookie` mechanism. The status bar at the bottom shows "Line: 2161 Col: 1 Set: 0 Lines: 2989 Length: 105275 Insert Done parsing 511 files in 1.281 seconds (398.91 files per second)".

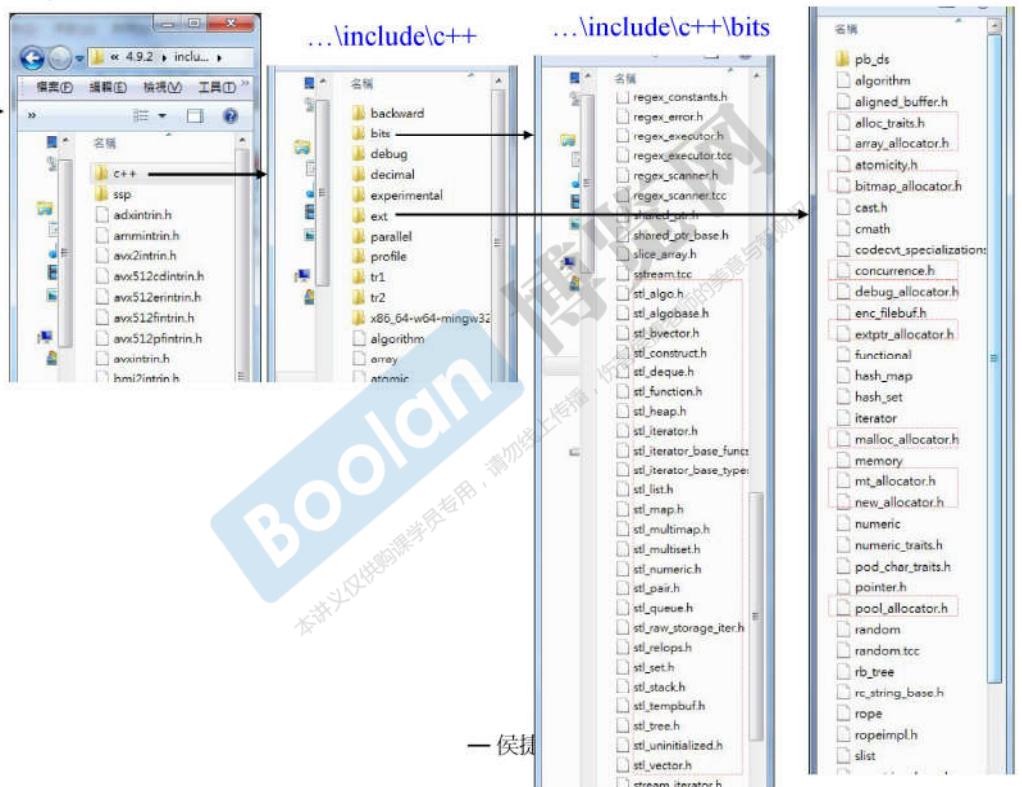
```
//從語法上試用各式各樣的 allocators
cout << sizeof(_gnu_cxx::malloc_allocator<int>) << endl; //1. 大小 1 因其實為 0, fields 都是 static
cout << sizeof(_gnu_cxx::__pool_alloc<int>) << endl; //1
cout << sizeof(_gnu_cxx::__mt_alloc<int>) << endl; //1
cout << sizeof(_gnu_cxx::bitmap_allocator<int>) << endl; //1
cout << sizeof(_gnu_cxx::array_allocator<int>) << endl; //8 ==> 因為它有一個 ptr 指向 array 和一個 size
cout << sizeof(_gnu_cxx::debug_allocator<std::allocator<double>>) << endl; //8
//cout << sizeof(_gnu_cxx::throw_allocator<int>) << endl; //只有 throw_allocator_base, throw_allocator
//清單容器
list<int, _gnu_cxx::malloc_allocator<int>> list_malloc;
deque<int, _gnu_cxx::debug_allocator<std::allocator<int>>> deque_debug;
vector<int, _gnu_cxx::__pool_alloc<int>> vector_pool;
//測試 cookie
cout << "sizeof(int) = " << sizeof(int) << endl; //4
cout << "sizeof(double) = " << sizeof(double) << endl; //8
cookie_test(std::allocator<int>(), 1); //指派 10h (表示無 cookie)
cookie_test(_gnu_cxx::malloc_allocator<int>(), 1); //指派 10h (表示有 cookie)
//
```



## 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

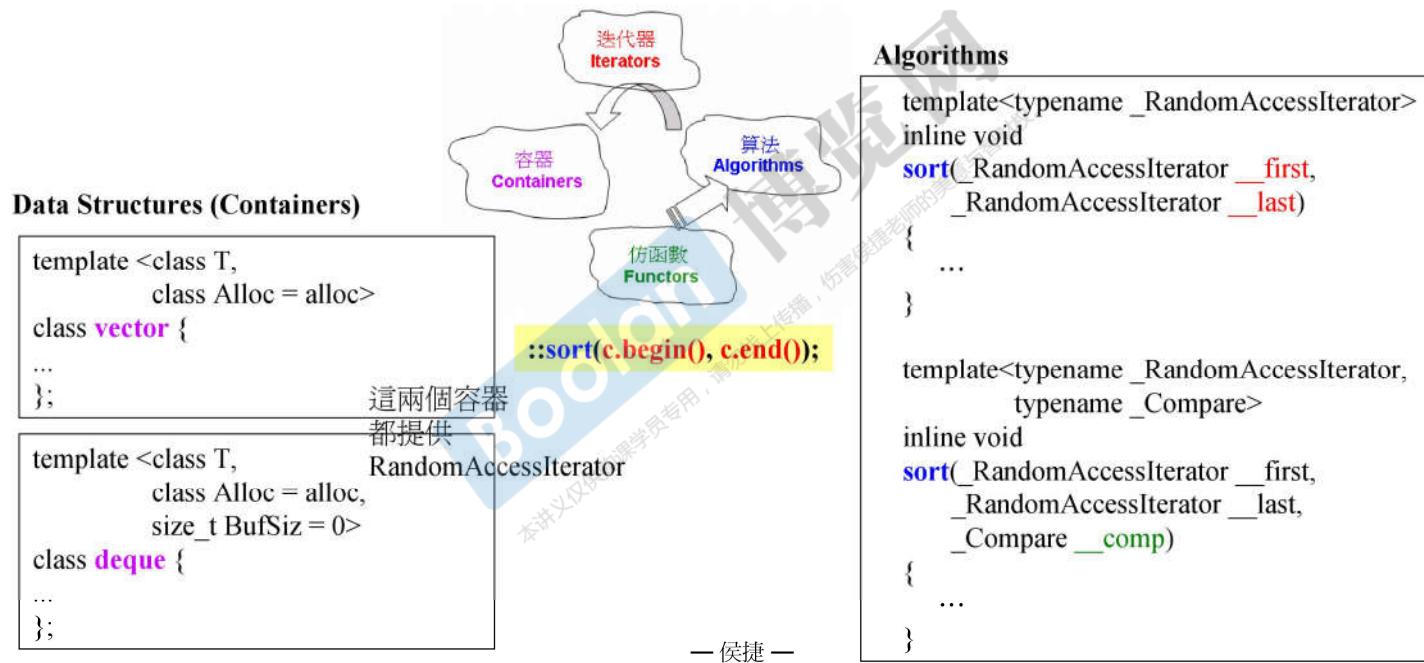
62

list的iterator不能做+-操作

std::sort()需要用RandomAccessIterator, 而list没提供此类iterator  
只能用list.sort()

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

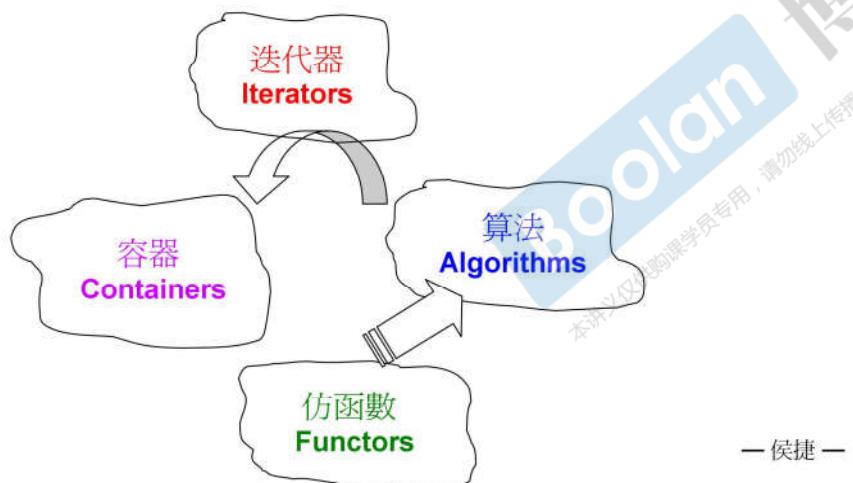
GP 却是將 **datas** 和 **methods** 分開來



## ■■■■ 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;
}
```

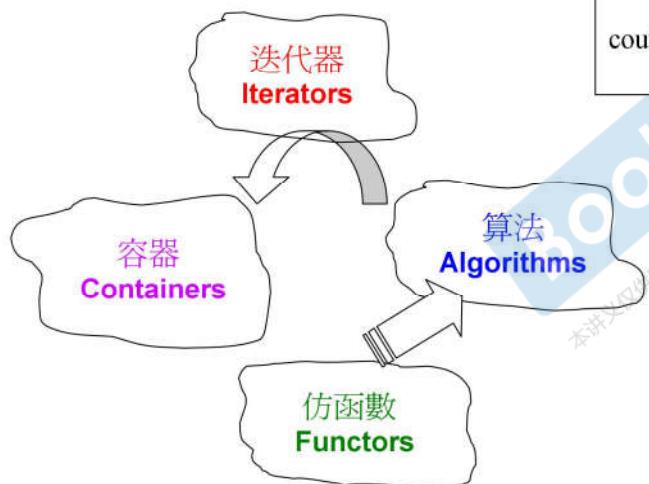
```
template <class T>
inline const T& max(const T& a, const T& b) {
    return a < b ? b : a;
}
```

```
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。



```
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 —

■■■■ 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

list的iterator

```
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;

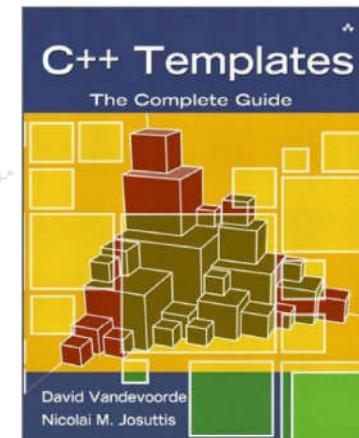
    reference operator*() const { return (*node).data; }
    pointer operator->() const { return &(operator*()); }
    self& operator++()
    self operator++(int)
    ...
},
```

## 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> class和typename一样  
inline  
const T& min(const T& a, const T& b)  
{  
    return b < a ? b : a; 确定T后，就能找到<，<作用于b  
}
```

```
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<()

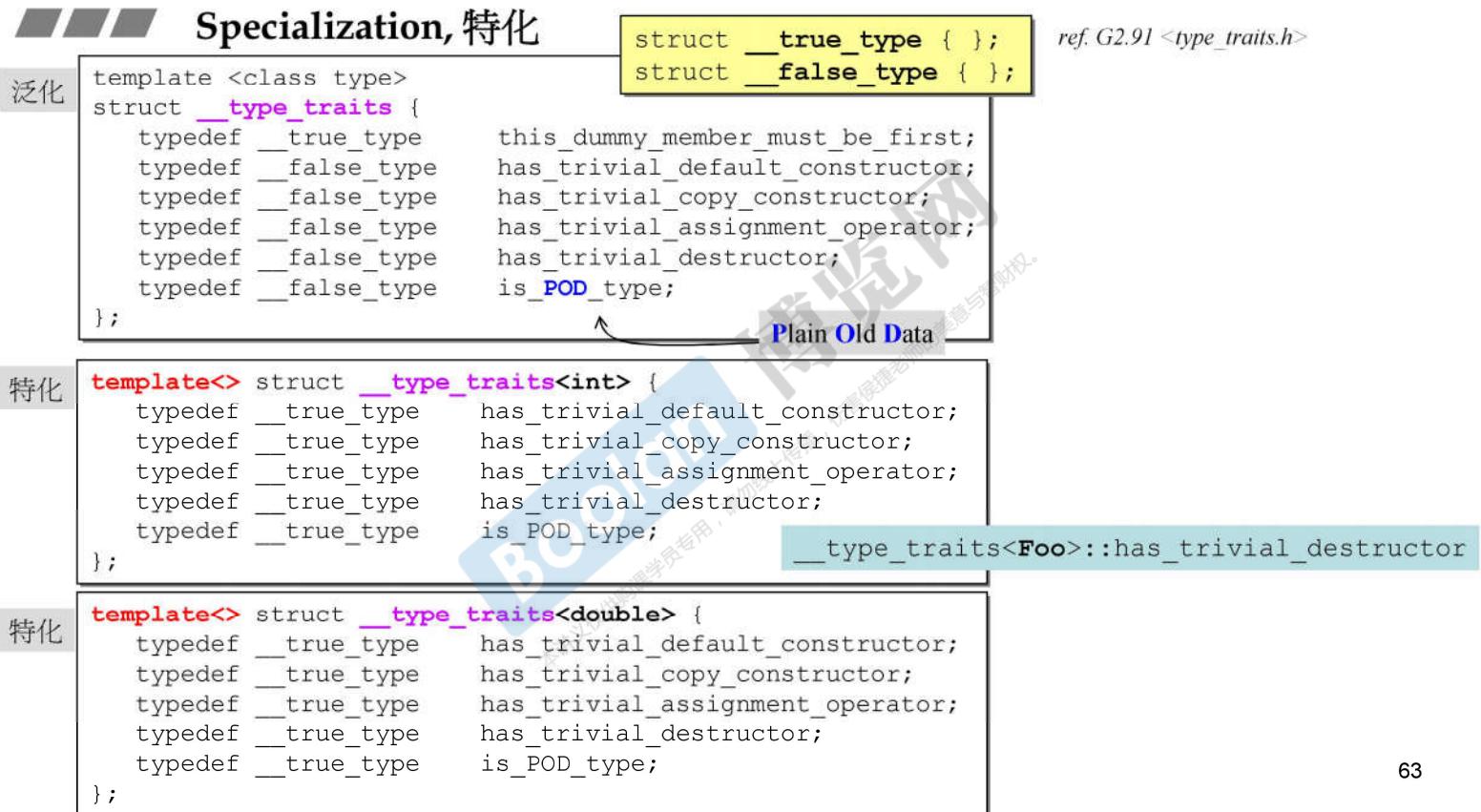
实参推导：确定T，所以不用写min<>()

## ■■■ Member Templates, 成員模板

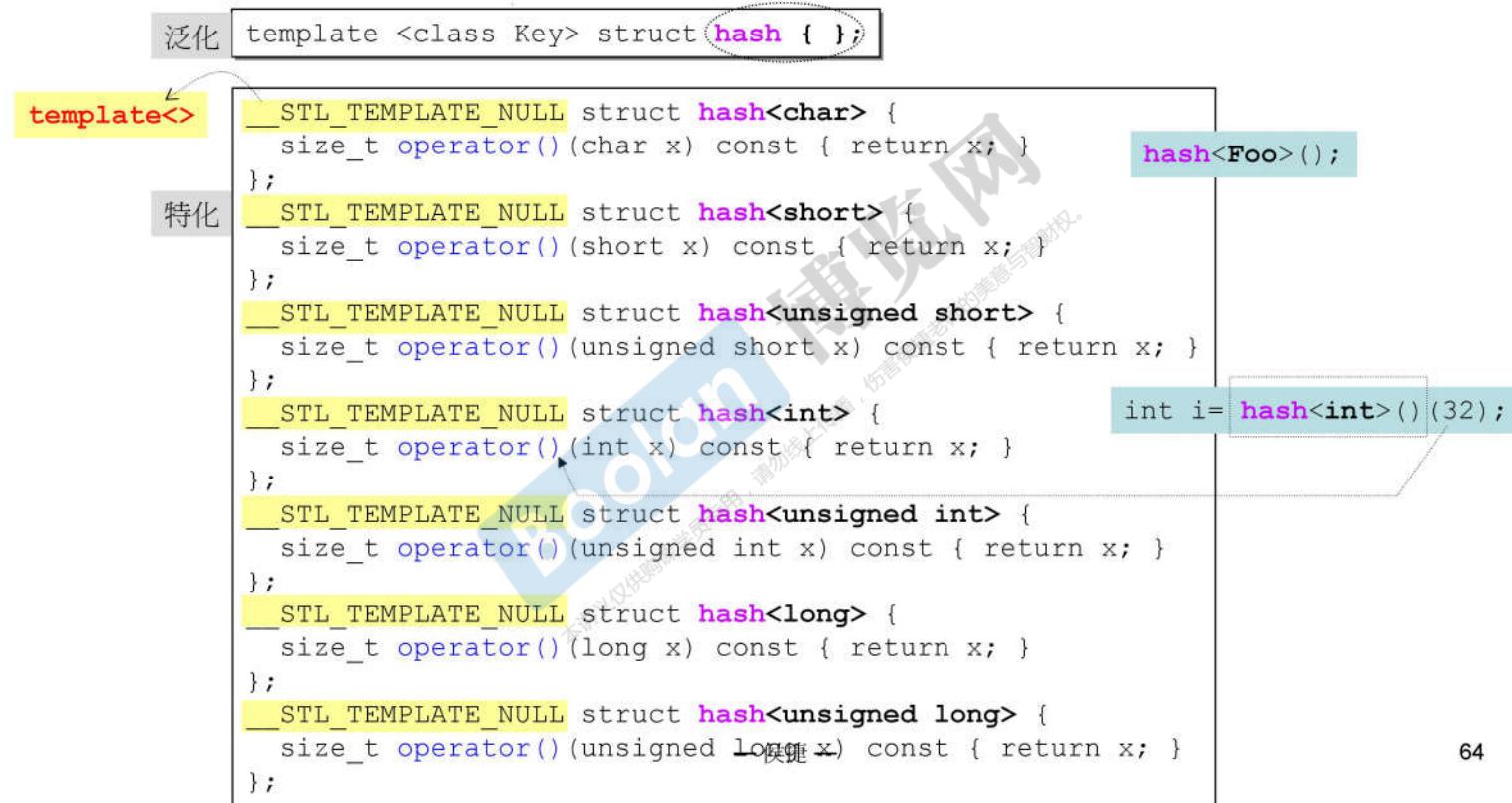
```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    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
};
```



## 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; };
        };
    
```

## ■■■■ Partial Specialization, 偏特化

数量的偏

泛化

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

偏特化

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

范围的偏

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

```
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;  
};
```

分配器的效率影响容器的效率，包括速度和空间

所有分配动作最终都调用c runtime library提供的malloc(), malloc()根据不同os再调用sc拿到mem

## ■■■ 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()分配mem，会附加其他分配后，再打包给你  
附加部分(overhead)固定

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

debug more下添加

size {  
要求的大小

alignment of bound

cookie记录整块大小(必要)，为了free()

## 不同平台(vc/gnu c), allocator的花样

### ■■■■ 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 *):接受任意东西
        { return (_Allocate((difference_type)_N, (pointer)0)); }
    void deallocate(void _FARQ * _P, size_type)
        { operator delete(_P); } free(_P)
};
```

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))); }
```

//分配 512 ints.  
int\* p = allocator<int>().allocate(512, (int\*)0);  
allocator<int>().deallocate(p, 512); 所以难用，容器用它还可以

69

vc6没有任何特殊设计？

如果，容器中放的元素(string大小4B)通常都很小，overhead占的比例大，若容器放10w个元素，则额外消耗很大，vc就是这样  
vc的allocator只是调用c的malloc/free分配释放mem

## ■■■■ 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。同 vc

比 vc 好点

//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。

同样叫allocator，标准库规格书规定

```
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.

同vc，现实overhead占用比例很大，不太好

gnu c的stl是SGI公司写的，user a different allocator => 虽然SGI虽然写了符合标准规格书的allocator，但自己的容器从来不同它即：有更好的allocator

## ■■■■ 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 {
    ...
};
```

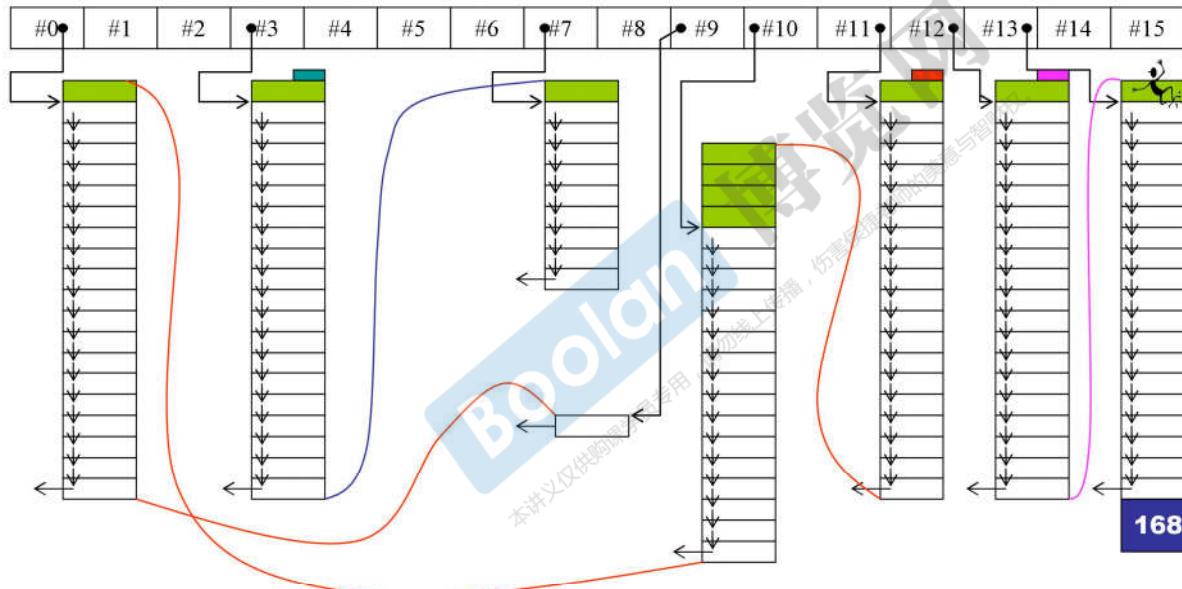
```
//分配 512 bytes.
void* p = alloc::allocate(512); //也可alloc().allocate(512);
alloc::deallocate(p,512);
```

```
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 {
    ...
};
```

## ■■■ 分配器 allocators

G2.9 所附的標準庫，其 `alloc` 實現如下 (`<stl_alloc.h>`) `alloc` 源碼見內存管理



— 侯捷 —

74

1. `alloc` 主要：尽量减少 `malloc` 调用次数，因为 `malloc` 每次都带 overhead，
2. overhead 中值得讲的 cookie：因为容器元素大小一样，所以不必每个元素都记录自己的大小，所以容器这种应用，不需要 cookie  $\leq gnu c$
3. 16 条链表，每条链表串起负责某种特定大小的区块，如：#0 负责 8B 大小的区块，#7 负责 7\*8B 大小的区块
4. 所有容器需要 mem 时，都向 `alloc` 要，容器元素大小被调整为  $8 \times n$ ，如：50B 调整为 56B，再找能负责的链表，看上面是否挂有 mem 块，若没有再 `malloc` 一大块并做切割，最后给出一小块，切割后用单向链表串起，切出来的每小块不带 cookie
5. cookie 在很多平台上，上下都有，所以消耗 8B。所以容器放 100w 个元素可省去 < 800wB (100w 元素都不带 cookie)，因为终究还得用 `malloc`
6. `alloc` 也有缺陷

## ■■■ 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>
{
    ...
};
```

不是alloc

之前试用了gnu c下的很多分配器，2.9的alloc在4.9里还存在吗？仍然被容器使用吗？

## ■■■■ 分配器 allocators

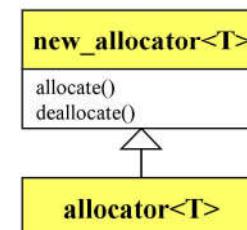
G4.9 所附的標準庫，其 `allocator` 實現如下

```
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); }
    ...
};
```



卧槽，又和vc一样了

但2.9的alloc还在，改名为

extention allocators, 如: <bitmap\_allocator><array\_allocator><debug\_allocator>...<pool\_alloc>

## ■■■■ 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.

`__pool_alloc_base`

`private`

`__pool_alloc<T>`

`allocate()`

`deallocate()`

`free_list`就是16条链表

— 便捷 —

77

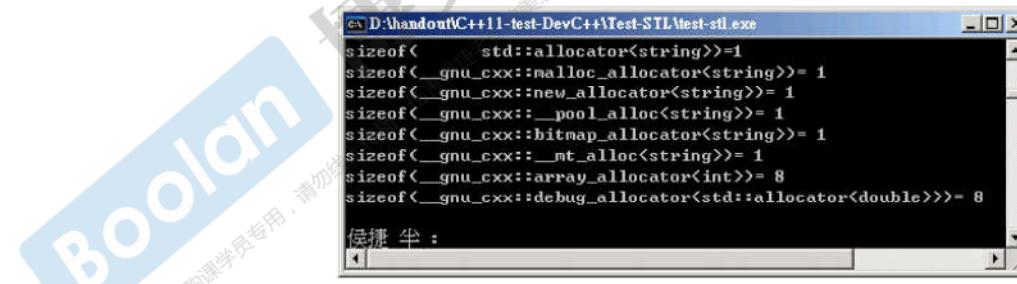
新版如4.5/4.9/5.1/5.2都是如上设计，没特点的和好的都在

## ///// 迭代器, sizeof()

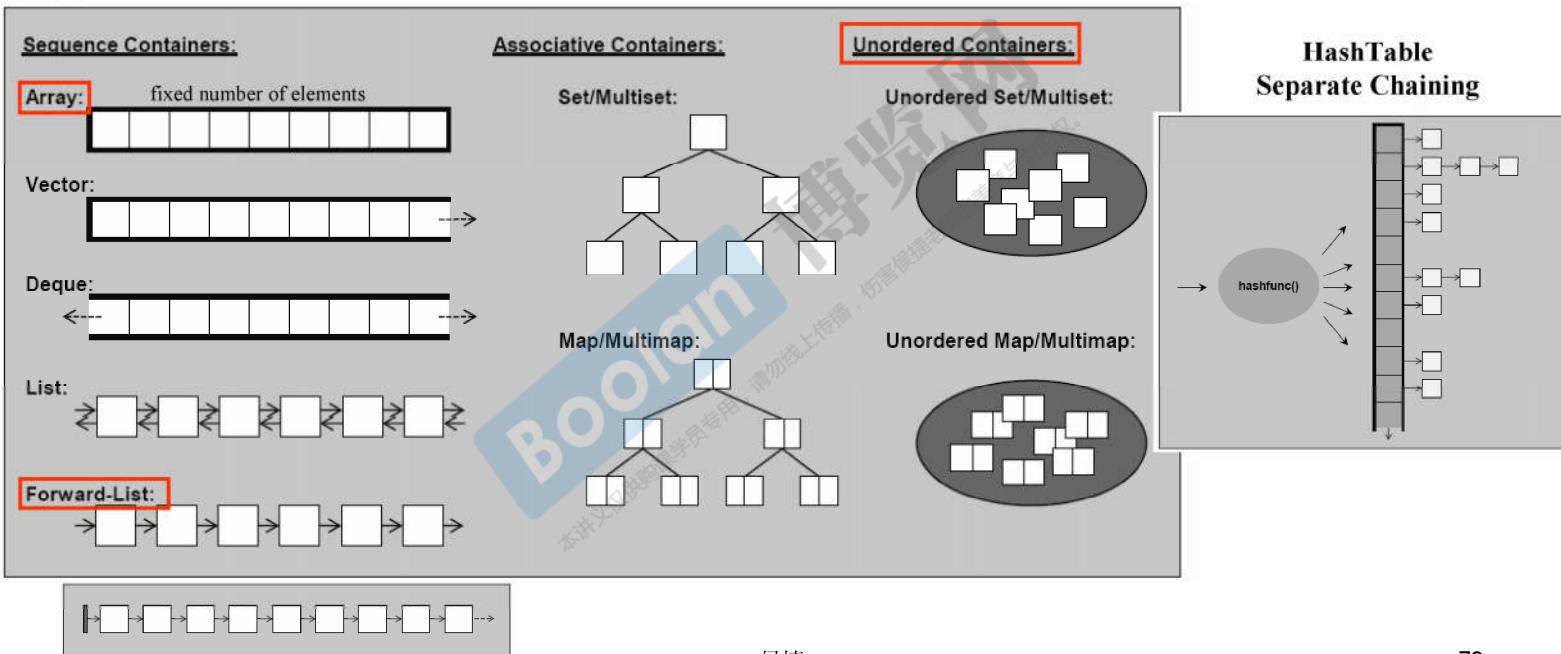
```

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 << "====> 因為它有一個 ptr 指向 array 和一個 size_t 表示消耗到 array 哪兒
cout << "sizeof(_gnu_cxx::debug_allocator<std::allocator<double>>)=" << sizeof(_gnu_cxx::debug_allocator<std::allocator<double>>) << endl;

```



## 容器 – 結構與分類



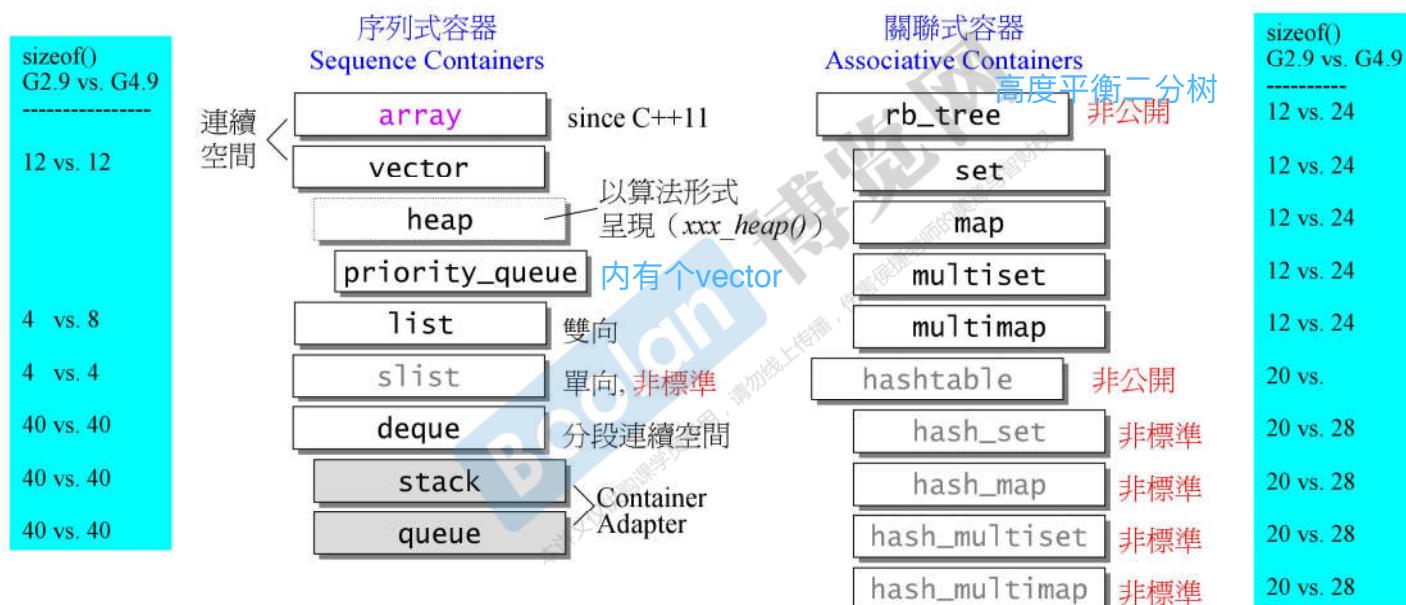
— 侯捷 —

## 容器, 結構與分類

早期标准库继承很少

4.9后单一容器本身就分为多个class的组合，多了不必要的复杂

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`。

80

class A; class B; 若A用到B的func, A可继承B / A拥有B

非标准: c++11之前各家编译器因为重要，所以实现了，现在改了名并放入了std

非公开: 文档没有，看源码才知道的

蓝色部分: 测试这些容器obj本身大小sizeof(), 和元素个数无关

## 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

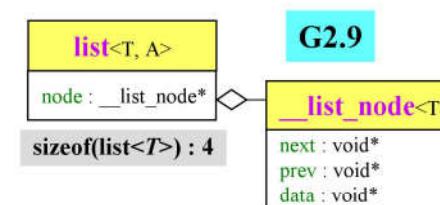
## 13. 深度探索list(上)

最简单的容器是array, vector, 最具代表性是list

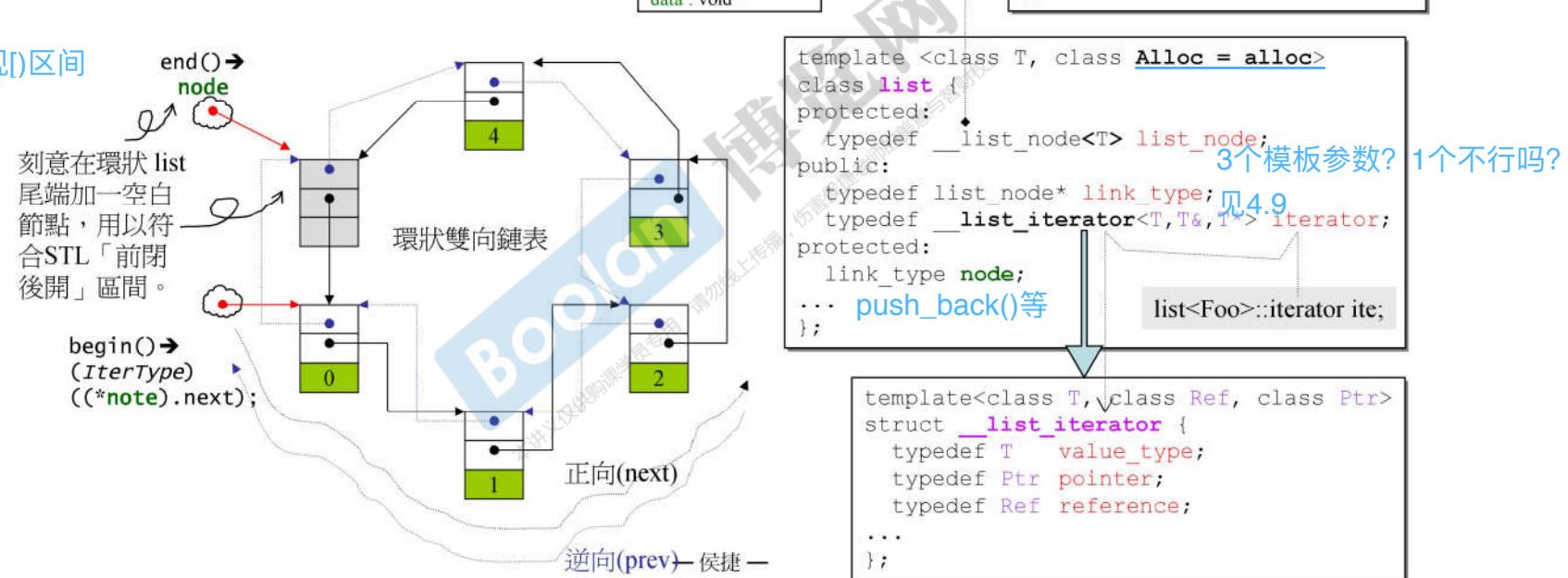
list中data只有link\_type node, 是指针, 所以sizeof(list<T>())==4

prev & next指向void, 通常应该指向自己这类obj, 这里指向void, 所以程序设计时需要转型  
list存储100w元素时, 每个元素还需要两个指针的mem, list向alloc要mem时是元素+两个指针

### 容器 list

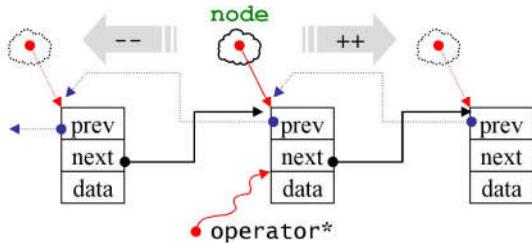


```
template <class T>
struct __list_node {
    typedef void* void_pointer;
    void_pointer prev;
    void_pointer next;
    T data;
};
```



list是非连续的, 所以其iterator不能是指针, 但希望iterator模拟指针动作, 所以iterator++后应该能找到next  
所以iterator必须是class才能足够聪明(除了vector&array), 所有容器都有typedef xxx iterator;

## 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; }
    ...
};
```

iterator为了模拟指针，有很多操作符重载

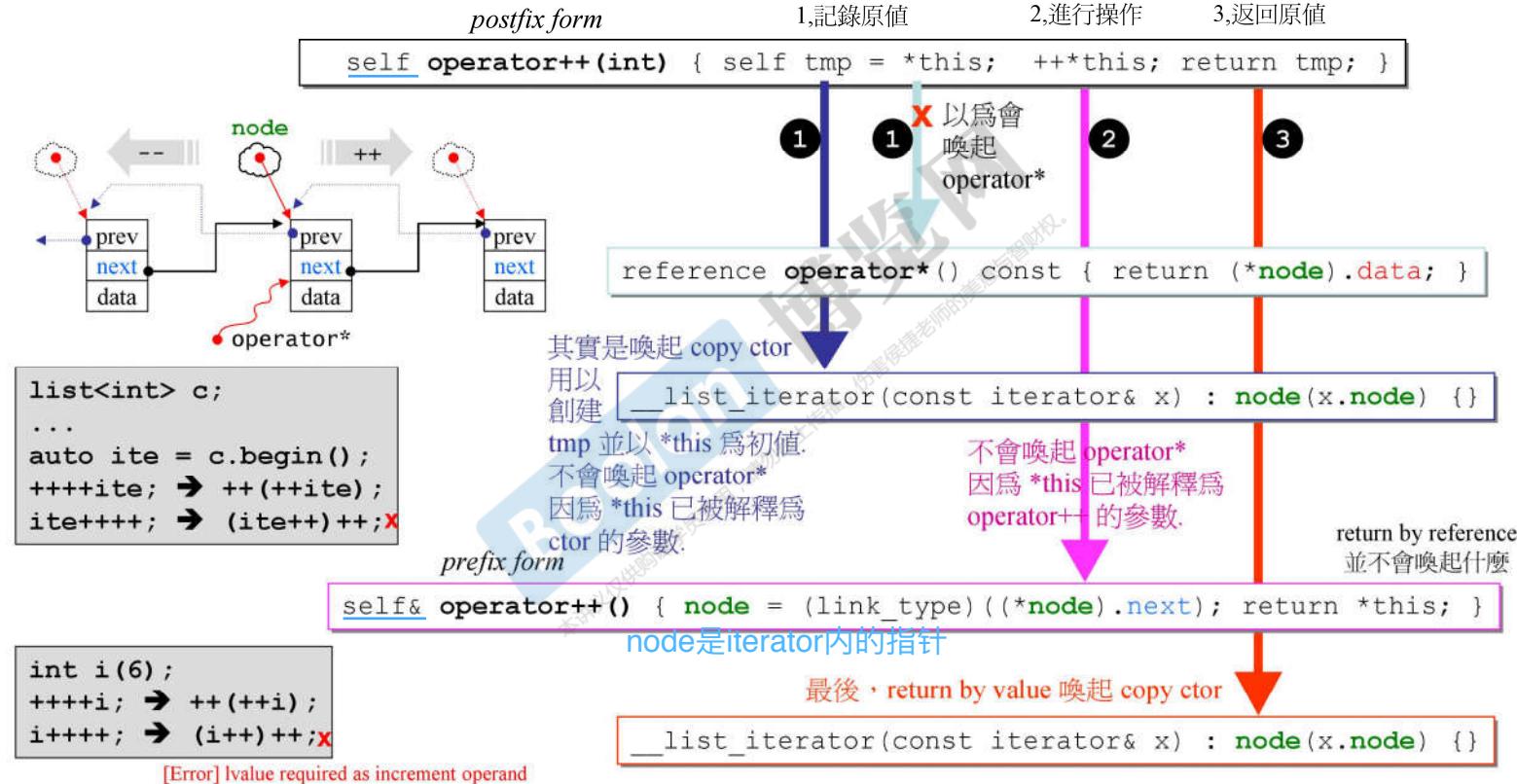
每个容器的iterator都必须有图中5个typedef和操作符重载  
operator++() & operator++(int)

— 侯捷 —

83

## list's iterator

postfix form这里符合可能已经被重载  
编译器在遇到\*之前，遇到了=



self和self& : operator要向int看齐，int怎么做，operator就怎么设计

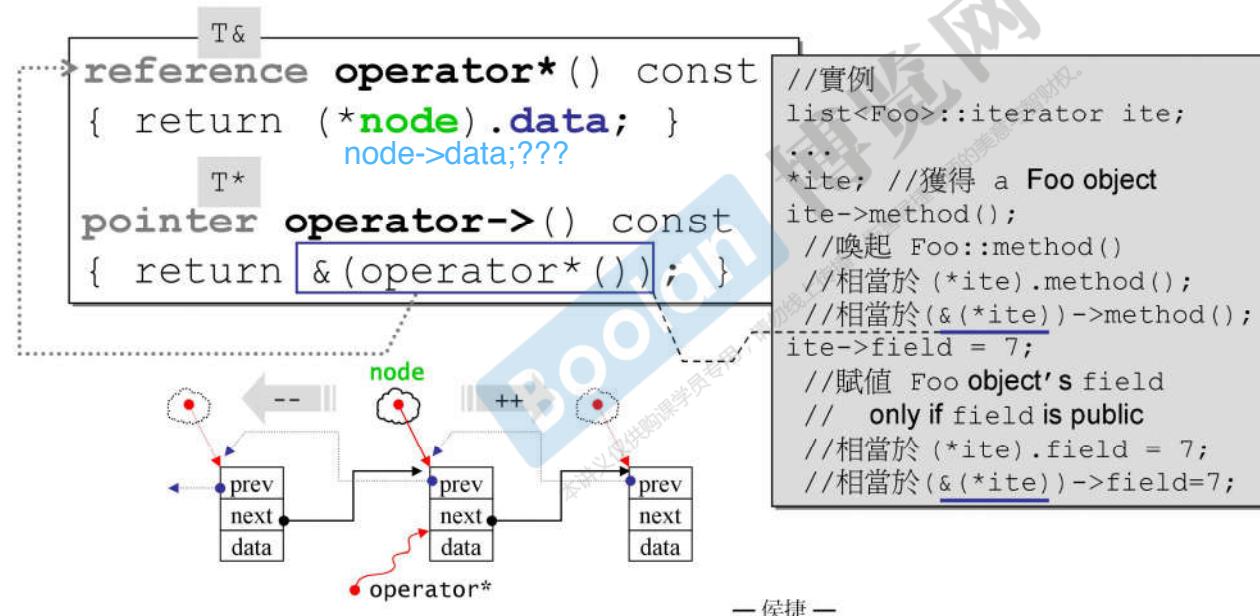
i++, 规定：c++不允许int后++两次

所以postfix form为了阻止两次后++，返回值为self

list的iterator有很多操作符重载

## 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 —

所有容器的iterator都分为两部分：一堆typedef(五个必须有，因为traits需要)，一堆operator overload

## 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 {
    typedef 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
{
    typedef _Tp* pointer; // (3)
    typedef _Tp& reference; // (4)
    ...
};
```

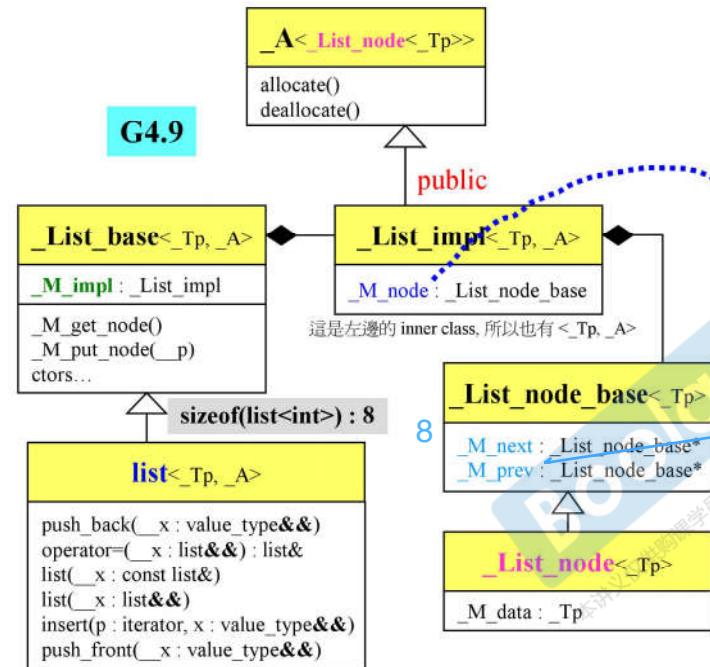
```
struct __List_node_base
{
    __List_node_base* __M_next;
    __List_node_base* __M_prev;
};
```

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



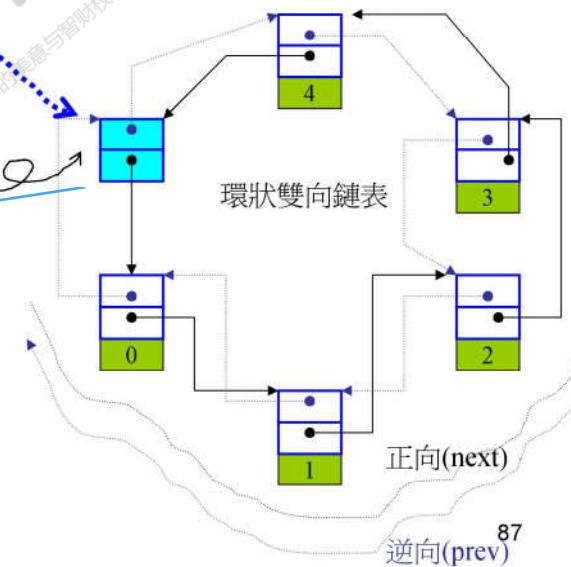
```

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 —

list模板设计不如G2.9的地方

虛元素G4.9和G2.9版本list的大小8和4

## 15.iterator的设计原则与iterator traits的设计作用

traits: 特征，提取容器的一些特征

标准库有多种traits, iterator traits, type traits, character traits, pointer traits...

这里讲iterator traits是要萃取iterator的特性，那iterator有哪些特性(原则)

algo通过iterator操作containers中元素时， algo可能需要知道iterator有哪些性质，以便做最佳操作：

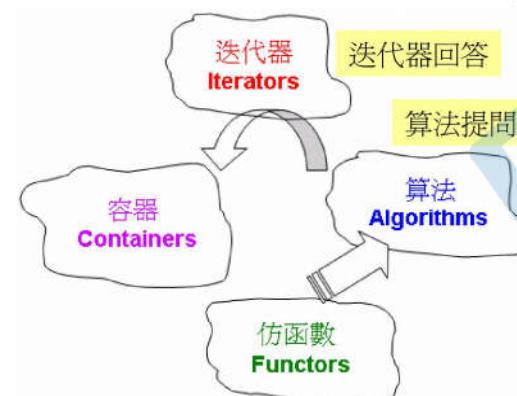
eg:rotate()想知道iterator哪些性质？

### Iterator 需要遵循的原则

```
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)
            }
        }
    }
}
```

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。

iterator\_category(): 分类，想萃取出iterator的分类，分类指iterator的移动性质，如：有的只能++/--，有的可+n

rotate()知道iterator后可能会采取最佳的操作方式

value\_type: iterator所指元素的类型

difference\_type: 两个iterator间的距离的类型，如:可用unsigned integer，即容器可存储0~2^32-1满足使用，则可设计difference\_type为unsigned integer

没有algo向iterator的reference和pointer

此5种叫iterator的associate type (相关类型)， iterator必须能回答

`ptrdiff_t`: 大部分`difference_type`都是此类型，定义在c++标准库某个header中

`bidirectional_iterator`: 双向，标记此容器是list

## Iterator 必須提供的 5 種 associated types

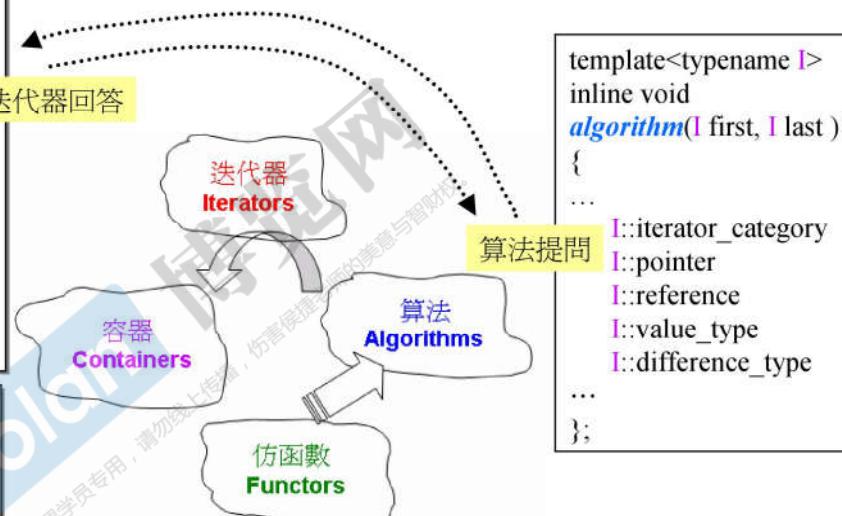
```
template<class T, class Ref, class Ptr>
struct __list_iterator
{
    typedef bidirectional_iterator_tag iterator_category; // (1)
    typedef T      value_type;           // (2)
    typedef Ptr   pointer;             // (3)
    typedef Ref   reference;           // (4)
    typedef ptrdiff_t difference_type;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

G2.9

迭代器回答

```
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



```
template<typename I>
inline void
algorithm(I first, I last )
{
    ...
I::iterator_category
I::pointer
I::reference
I::value_type
I::difference_type
...
};
```

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

只有class才能做typedef, native pointer:指针

指针是退化的iterator, iterator是泛化的指针

调用algo时传入pointer, pointer回答不了algo的问题, 此时需要traits

traits: 能区分所收到的iterator是class的iterator, 还是一般的iterator

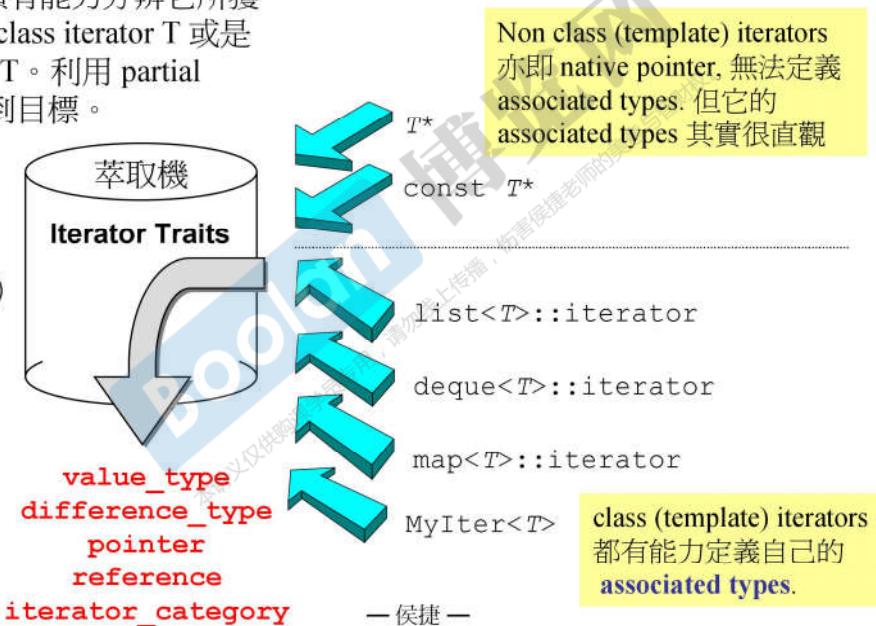
中间层: silver bullet, 尚方宝剑, 银弹打狼人心脏, 狼人即死

## ■■■ Traits 特性, 特徵, 特質

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

這個 traits 機器必須有能力分辨它所獲得的 iterator 是 (1) class iterator T 或是 (2) native pointer to T。利用 partial specialization 可達到目標。

解決計算機問題的尚方寶劍：加一個中介層



— 侯捷 —

大I是iterator对象

## ■■■ 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 :
template <class T>
struct iterator_traits<T*> {
  typedef T value_type; //如果 I 是 pointer to T 進入這裡
};

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
};
```

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.

## |||| 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>

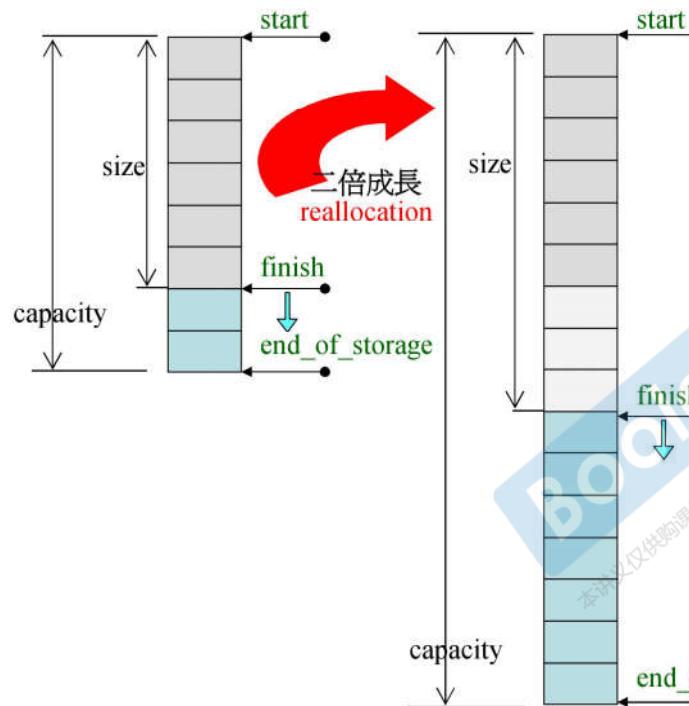
gnu c/bc/vc都是2倍成长

sizeof(vector)==三根指针大小

inline函数没有调用开销

带有连续空间的容器，都支持[], deque也有[], 但不是连续的，号称连续

## 容器 vector



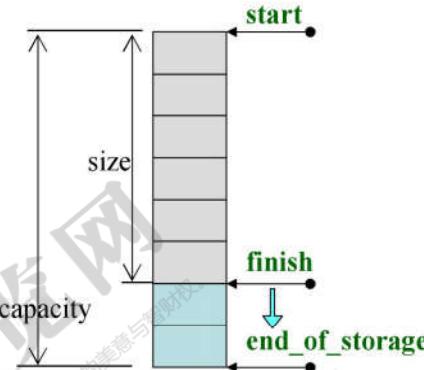
```
template <class T, class Alloc = alloc>
class vector {
public:
    typedef T           value_type;
    typedef value_type* iterator; //T*
    typedef value_type& reference;
    typedef size_t      size_type;
protected:
    iterator start;
    iterator finish;
    iterator end_of_storage;
public:
    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); }
};
```

两倍成长

insert\_aux()再次检查finish != end\_of\_storage, 即：除了push\_back(), 还有其他函数调用

### 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

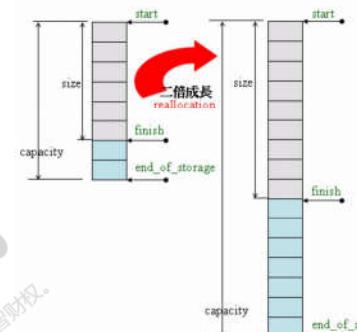
```
else { //已無備用空間
    const size_type old_size = size();
    const size_type len = old_size != 0 ? [2 * old_size] : 1;
    //以上分配原則：如果原大小為0，則分配 1 (個元素大小)；  

    //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;
}
```

vector成长时，大量调用cp ctor和dtor



list的iterator需单独设计一个class

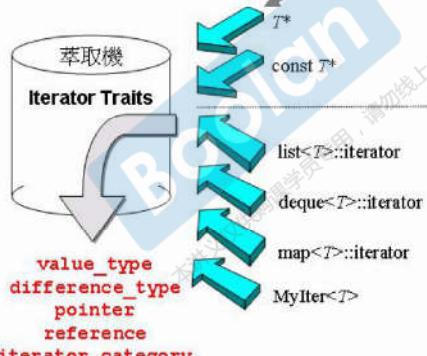
vector是连续空间，其iterator可不必设计太复杂

## vector's iterator

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

```
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



G2.9

```
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;
};

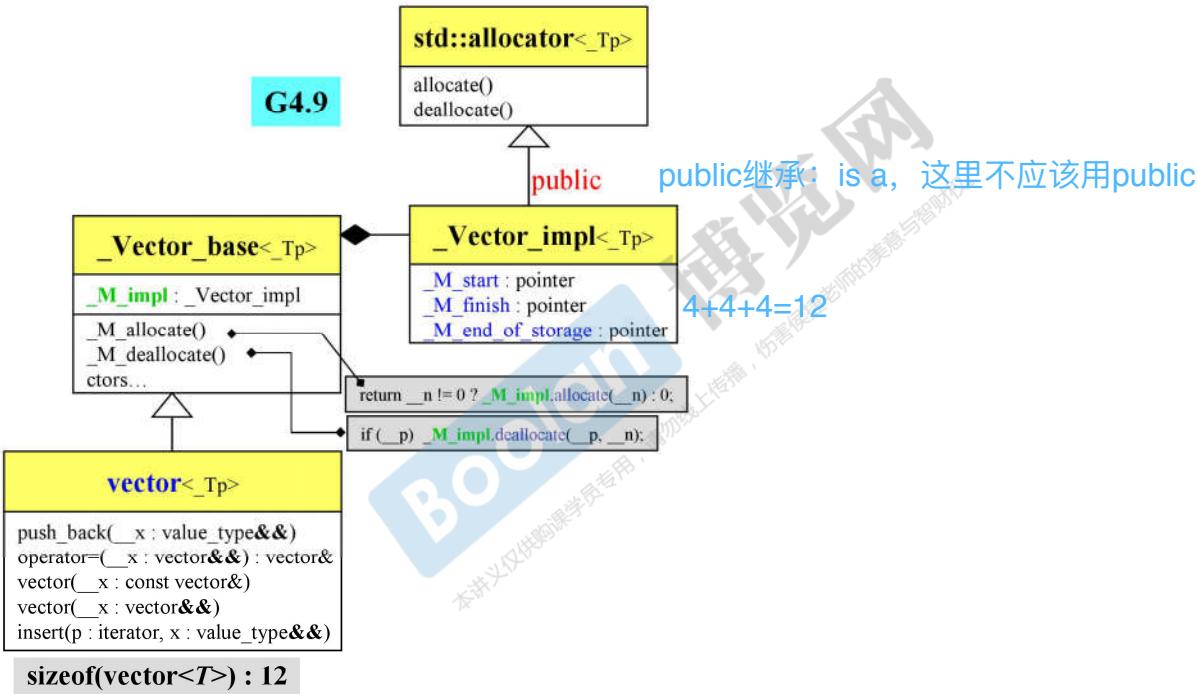
//partial specialization for regular pointers
template <class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T      value_type;
    typedef ptrdiff_t difference_type;
    typedef T*    pointer;
    typedef T&    reference;
};

//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;
};
```

## 容器 vector

0

G4.9



— 侯捷 —

98

## 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**;

**vector**<\_Tp>::**iterator**

**M\_current** : **\_Base**::**pointer**

```
    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;  
  
    _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**<\_Tp>::**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.  
};
```

## vector's iterator

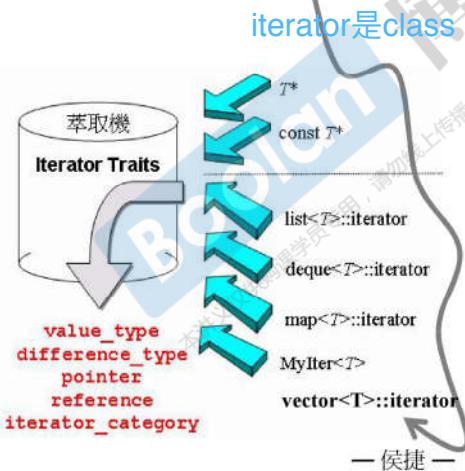
```
template<typename _Tp, typename _Alloc = std::allocator<_Tp>>
class vector : protected _Vector_base<_Tp, _Alloc>
{
...
typedef _Vector_base<_Tp, _Alloc> _Base;
typedef typename _Base::pointer pointer;
typedef __gnu_cxx::__normal_iterator<pointer, vector> iterator;
```

G4.9

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<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
```



`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 !!

## 17. 深度探索array & forward\_list

把[]包装成容器array，就可遵循容器规则，提供iterator(含5种associate type)，能和algo更好的交互  
由vector可知，连续空间的容器，其iterator可不用设计成class，直接用pointer

### 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

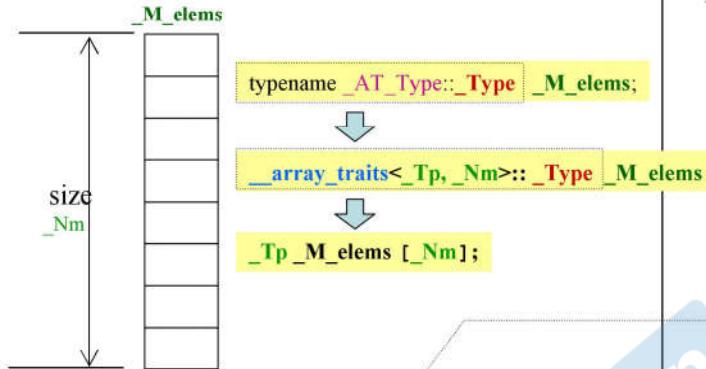
— 侯捷 —

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

`T*`  
`const T*`  
  
`list<T>::iterator`  
`deque<T>::iterator`  
`map<T>::iterator`  
`MyIter<T>`

`value_type`  
`difference_type`  
`pointer`  
`reference`  
`iterator_category`

## 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
```

语言本身规定

// 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)); }
```

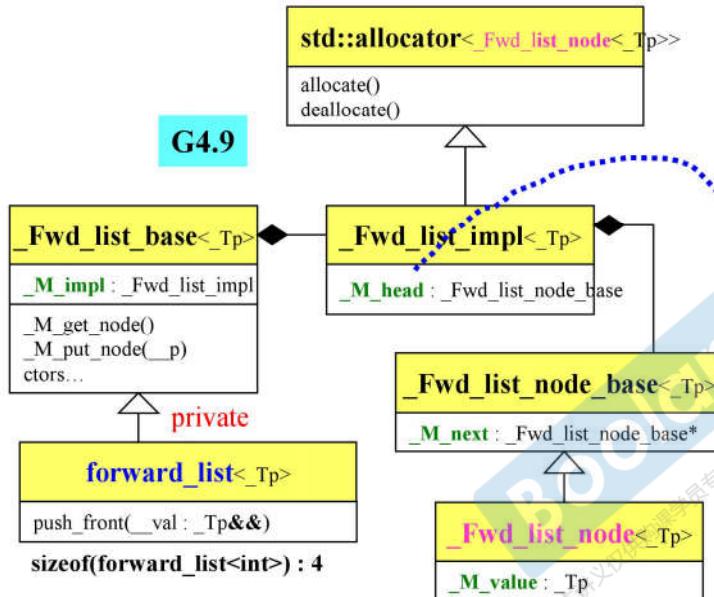
```
...
```

```
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



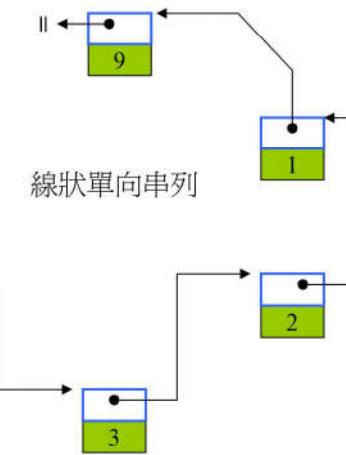
```

iterator
begin() noexcept
{ return iterator(this->_M_impl._M_head, _M_next); }

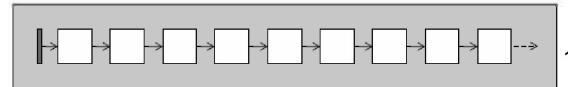
iterator
end() noexcept
{ return iterator(0); }

```

不供應 operator--(),  
所以不可能 --end();  
所以 end() 的定義ok



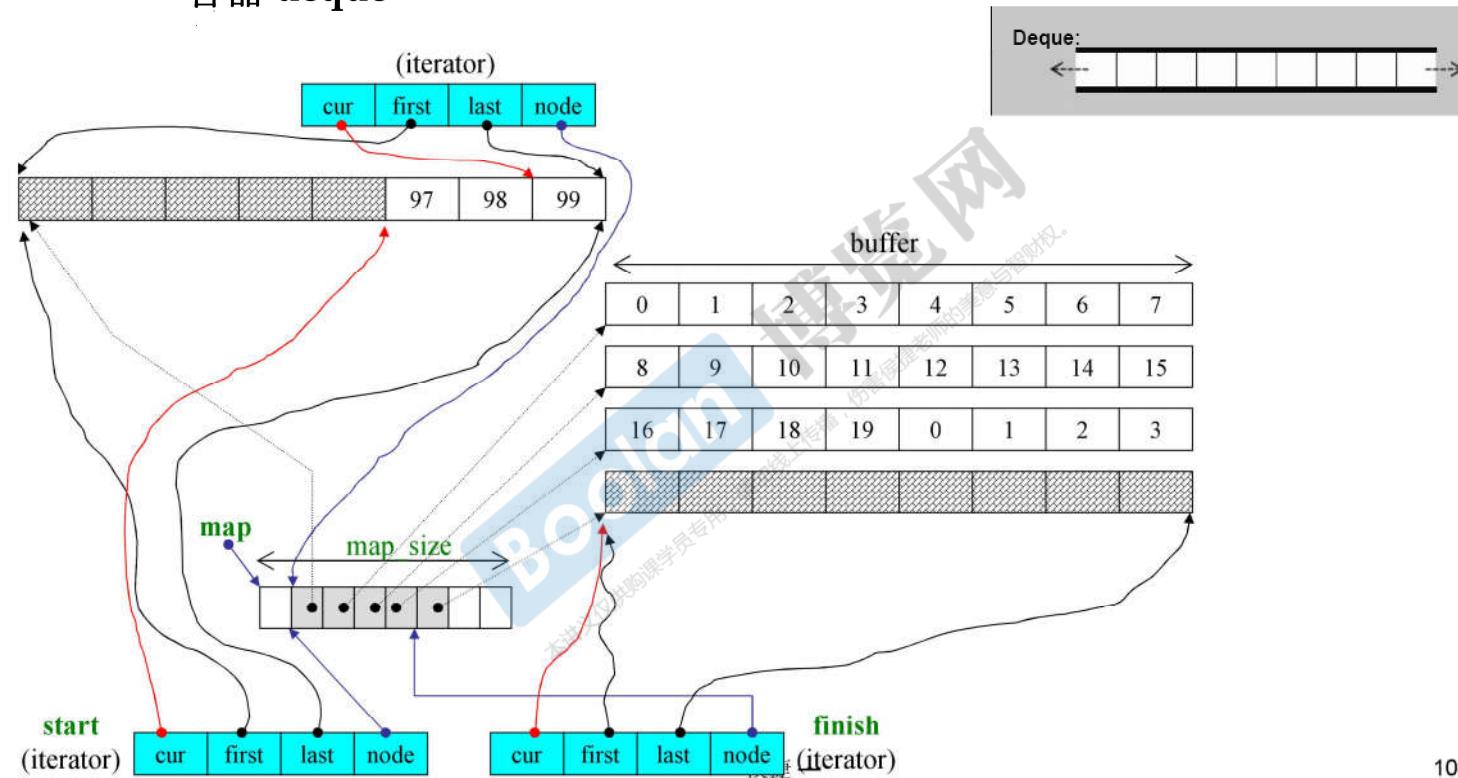
— 侯捷 —



103

只能單向遍歷

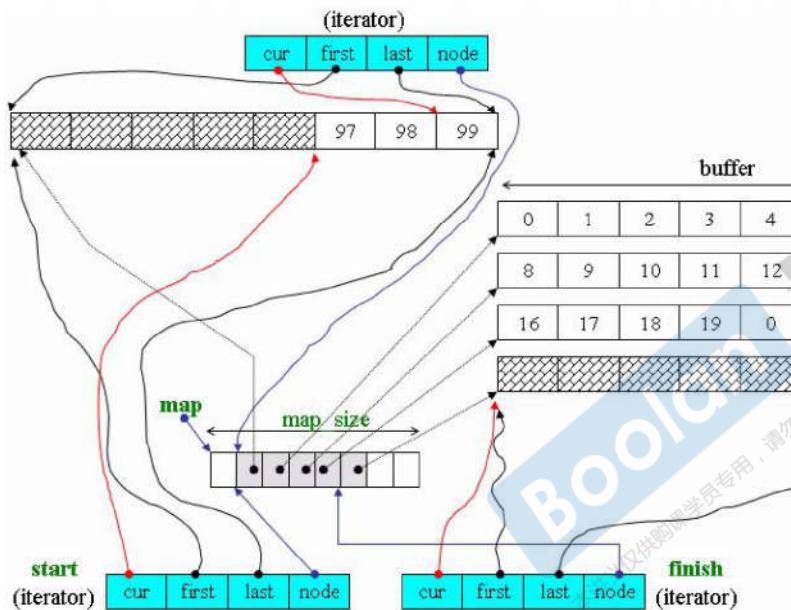
## 容器 deque



104

分段并串接buffer, 让使用者觉得是连续的  
vector存储指针, 前后扩充时, 先分配buffer, 再存放指针到vector  
deque的iterator是class, 有4个成员  
node指向vector, iterator++时可在不同buffer间跳转  
first & last指向node所指buffer的头尾, 和node结合使用能表现出deque的连续性  
deque中begin()&end()的返回值是start&finish

## 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:  
    typedef T value_type;  
    typedef __deque_iterator<T&, T*, BufSiz> iterator;  
protected:  
    typedef 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; }  
    ...  
};
```

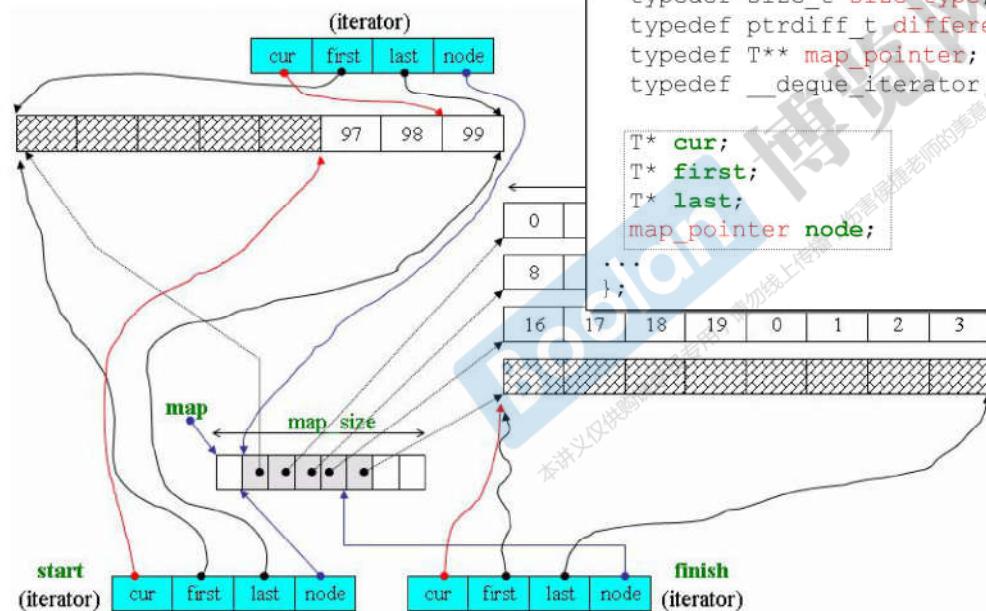
所謂 buffer size 是指每個 buffer 容納的元素個數

总是先看数据  
size\_type可能是unsigned long/unsigned int

`sizeof(deque)=2*16+4+4`

`template<>`: 2.9中3个模板参数, BufSiz可指定buffer size, 4.9以后不是  
`__deque_buf_size()`:计算一个buffer可容纳多少数据

## ///// 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;
};

T* cur;
T* first;
T* last;
map_pointer node;
...
};
```

106

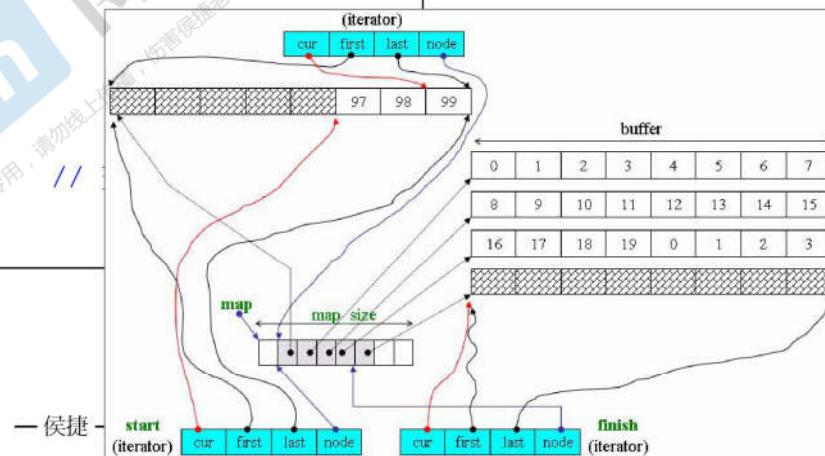
容器肯定都有自己专属的iterator, deque的iterator的分类是random\_access\_xxx, deque可以iterator+n操作  
分段是真相, 连续是假象, 假象靠iterator解释, 即iterator\_category

因为deque的insert会造成元素移动，所以会先计算距离头尾哪个近，每次推都调用ctor&dtor

## ■■■■ deque<T>::insert()

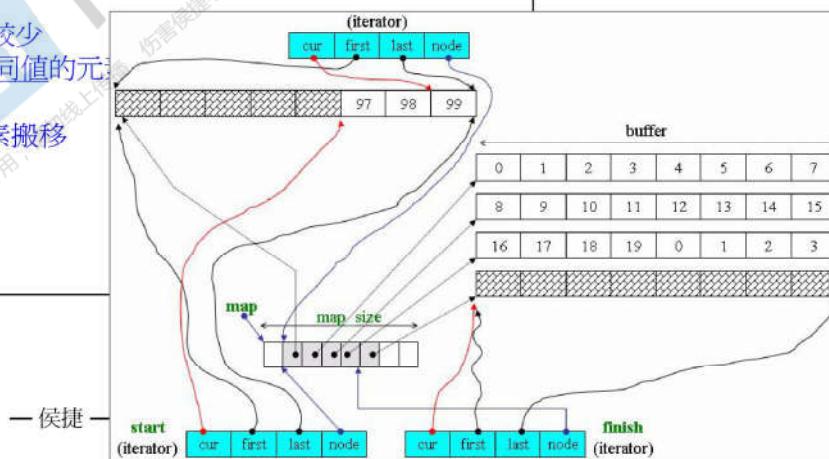
```
// 在position 處安插一個元素，其值為 x
iterator insert(iterator position, const value_type& x) {
    if (position.cur == start.cur) { //如果安插點是deque最前端
        push_front(x);           //交給push_front()做
        return start;
    }
    else if (position.cur == finish.cur) { //如果安插點是deque最尾端
        push_back(x);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

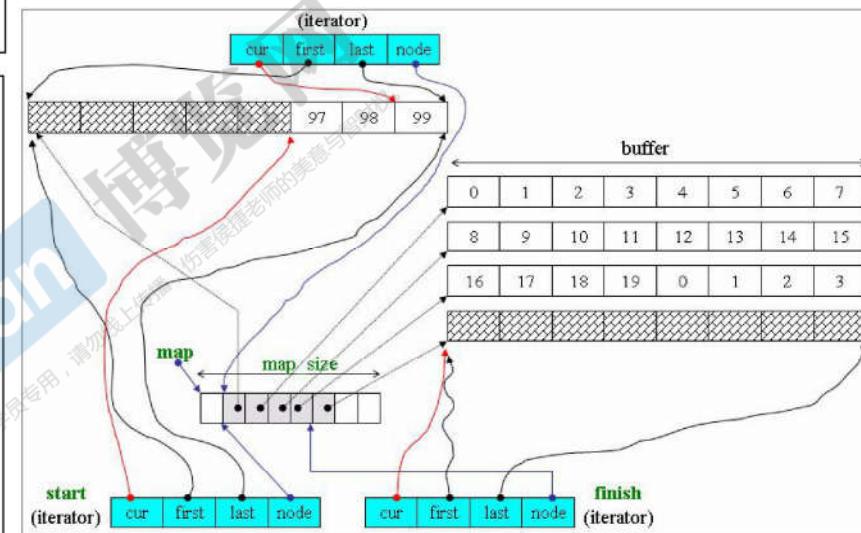
reference front()
{ return *start; }

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; }

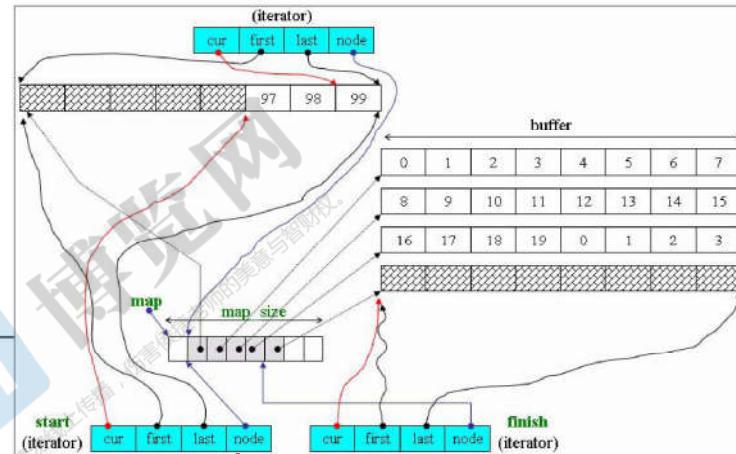
pointer operator->() const
{ return &(operator*()); }
```

```
//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 —

110

buffer\_size()是前面的\_\_deque\_buf\_size()

## 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;
}

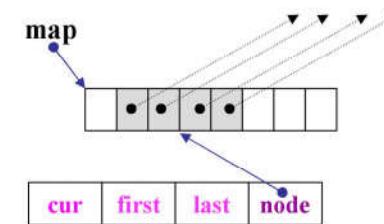
self operator++(int) {
    self tmp = *this;
    ++*this;
    return tmp;
}

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; //往前移一元素(此即最末元素)
    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());
}

```



用后++调用前++, 后--调用前--  
图中++cur和--cur的位置不同, 和[ ]有关



## 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;
}
```



## deque 如何模擬連續空間

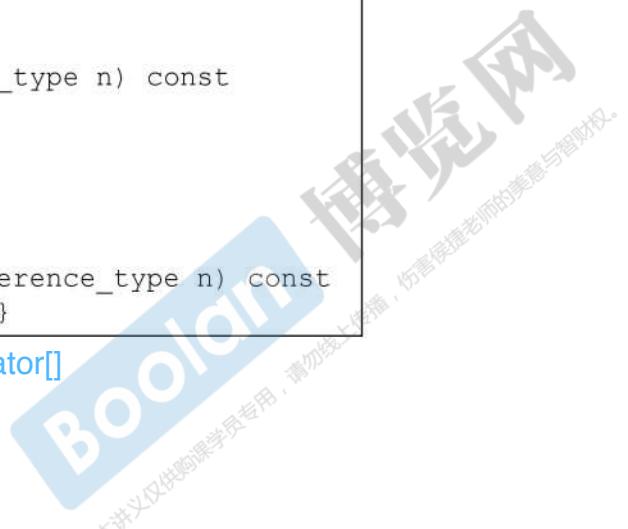
全都是 deque iterators 的功勞

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

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

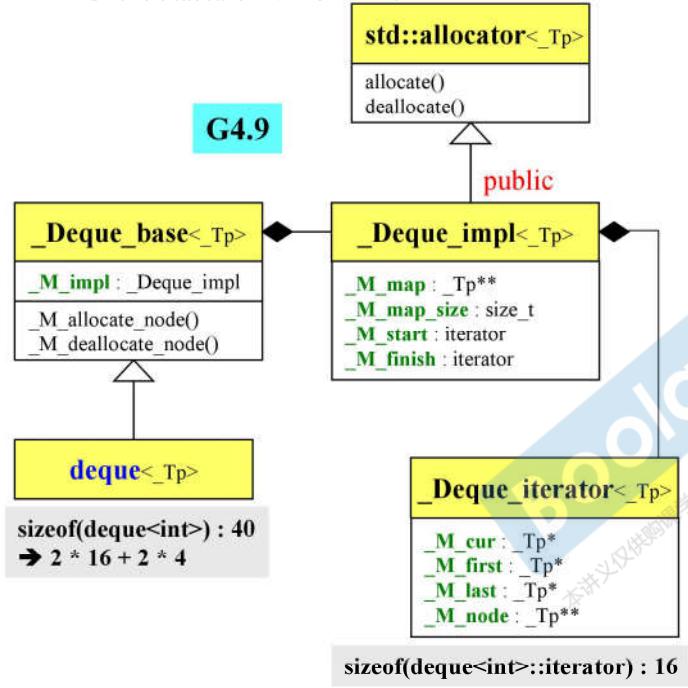
reference operator[](difference_type n) const
{ return *( *this + n); }
```

连续空间，iterator都提供operator[]



## deque

G4.9 每个容器都被设计成4个class



```

template<typename _Tp, typename _Alloc = std::allocator<_Tp>>
class deque : protected _Deque_base<_Tp, _Alloc>
{
    ...
    static size_t _S_buffer_size()
    { return __deque_buf_size(sizeof(_Tp)); }
};
  
```

G2.91 允許指派 buffer size,  
G4.53 沒了

#define \_GLIBCXX\_DEQUE\_BUF\_SIZE 512 默认512/元素大小=元素个数

```

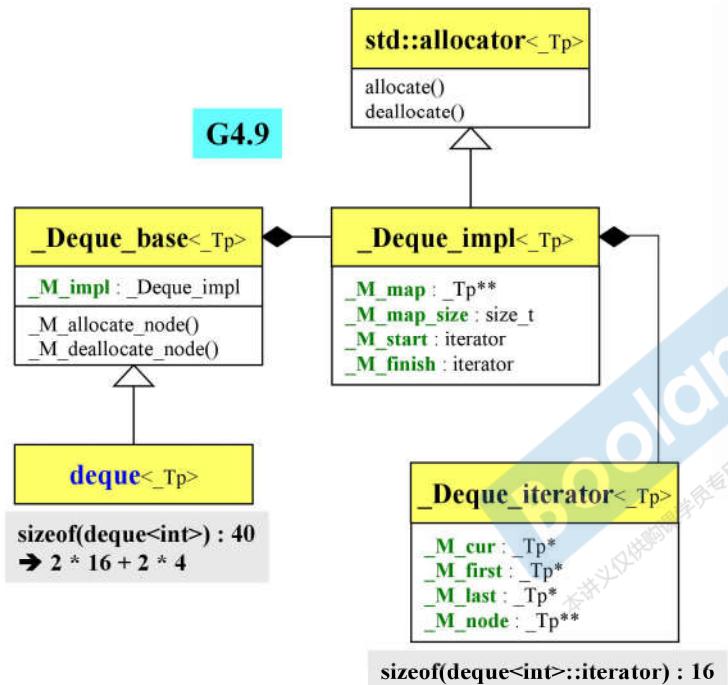
inline size_t
__deque_buf_size(size_t __size)
{ return (__size < _GLIBCXX_DEQUE_BUF_SIZE
? size_t(_GLIBCXX_DEQUE_BUF_SIZE / __size)
: size_t(1));
}
  
```

如果 node 是個 int, 則 buffer 內含的 node 數量是  $512 / \text{sizeof(int)} = 128$

如果 node 是個 double, 則 buffer 內含的 node 數量是  $512 / \text{sizeof(double)} = 64$

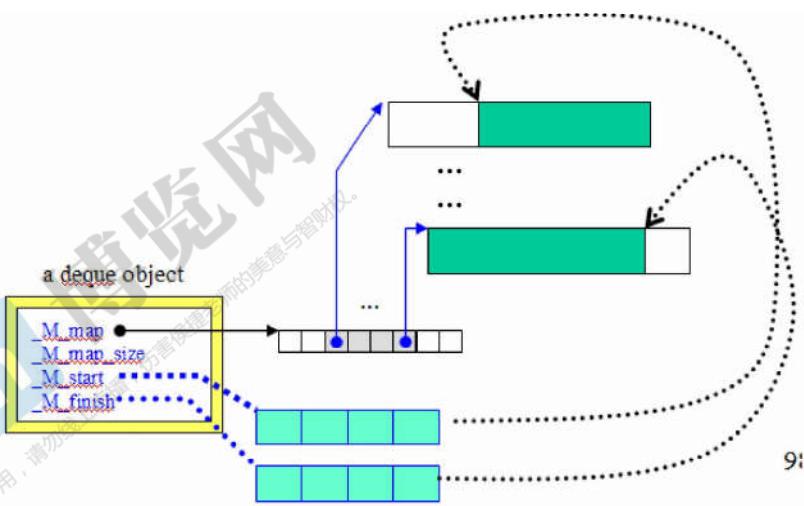
如果 node 是個 class/struct/array 而其大小  $\geq 512$ , 則 buffer 內含的 node 數量是 1

## ///// deque



`_M_map_size`记录vector容量

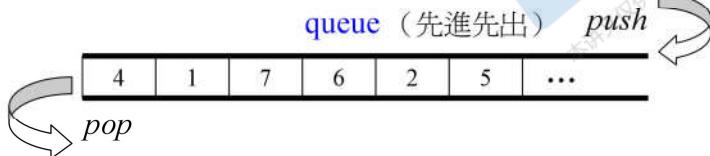
vector成长时需cp, 如8成长为16, 则9个元素cp到16里的中间部分, 这样可等量的成长



— 侯捷 —

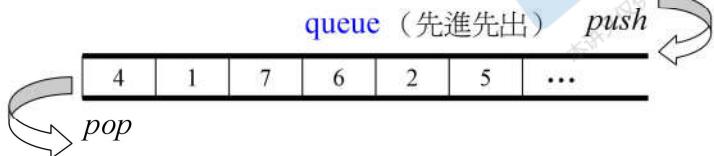
115

## 容器 queue



```
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(); }
};
```

— 侯捷 —

117

## queue 和 stack, 關於其 iterator 和底層結構

stack 或 queue 都不允許遍歷，前面讲的容器和后面的set/map等标准容器，都可遍历和insert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 链栈和顺序栈  
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;
```

一候

```
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;
```

## 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;
```

使用template时，编译器不会做全面检查，用时再检查，如用了c.pop()时才知道vector里没pop\_front()

## 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... kv
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。

之前都是sequence container，现在开始是associative container

associative container查找快，insert也快，类似于小型数据库

标准库里，associative container底层用rb\_tree和hash\_table做支持

理解rb\_tree的原理及标准库里的实现

rb\_tree: 二分树，rb表示此节点状态

平衡：某条手臂数据分布太长，导致查找慢，希望tree可自动平衡，自平衡树有rb\_tree&avl\_tree

容器维护一个begin()和end()

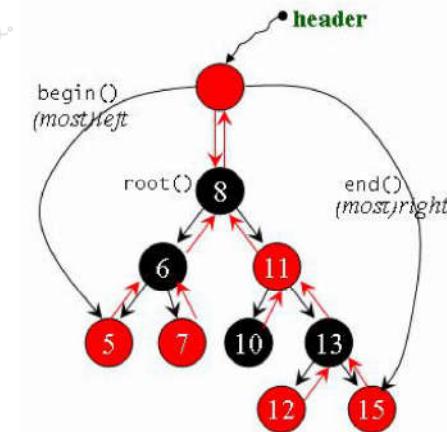
insert时已排序，++ite时是按照此序在走，如图：从最左边开始，5->6->7->8->10->11->12->13->15

“正确”：若rb-tree做map，key不能改v可改

insert\_equal(6)，则6一定和原有的6相邻，tree可能调整为了保持高度平衡

— 侯捷 —

121



value=key+data

keyofvalue: value中的key怎么获取

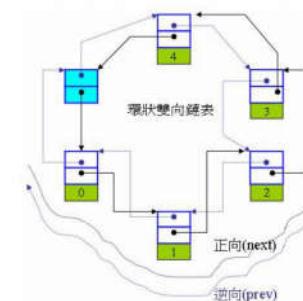
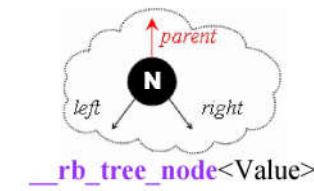
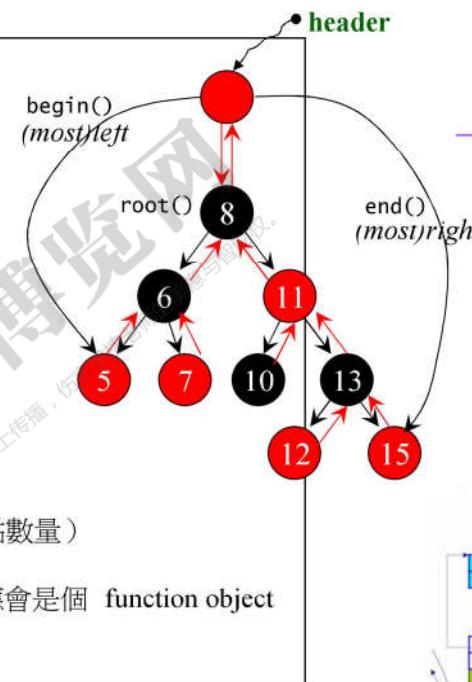
compare: key obj比大小func

node\_count: node即元素，元素个数

## 容器 rb\_tree

```
template <class Key,
          class Value,
          class KeyOfValue,
          > class Compare,
          class Alloc = alloc>
class rb_tree {
protected:
    typedef __rb_tree_node<Value> rb_tree_node;
...
public:
    typedef rb_tree_node* link_type;
...
protected:
    // RB-tree 只以三筆資料表現它自己
    size_type node_count;           //rb_tree的大小(節點數量)
    link_type header;              //key的大小比較準則；應會是個 function object
    Compare key_compare;
...
};
```

大小: 9 → 12



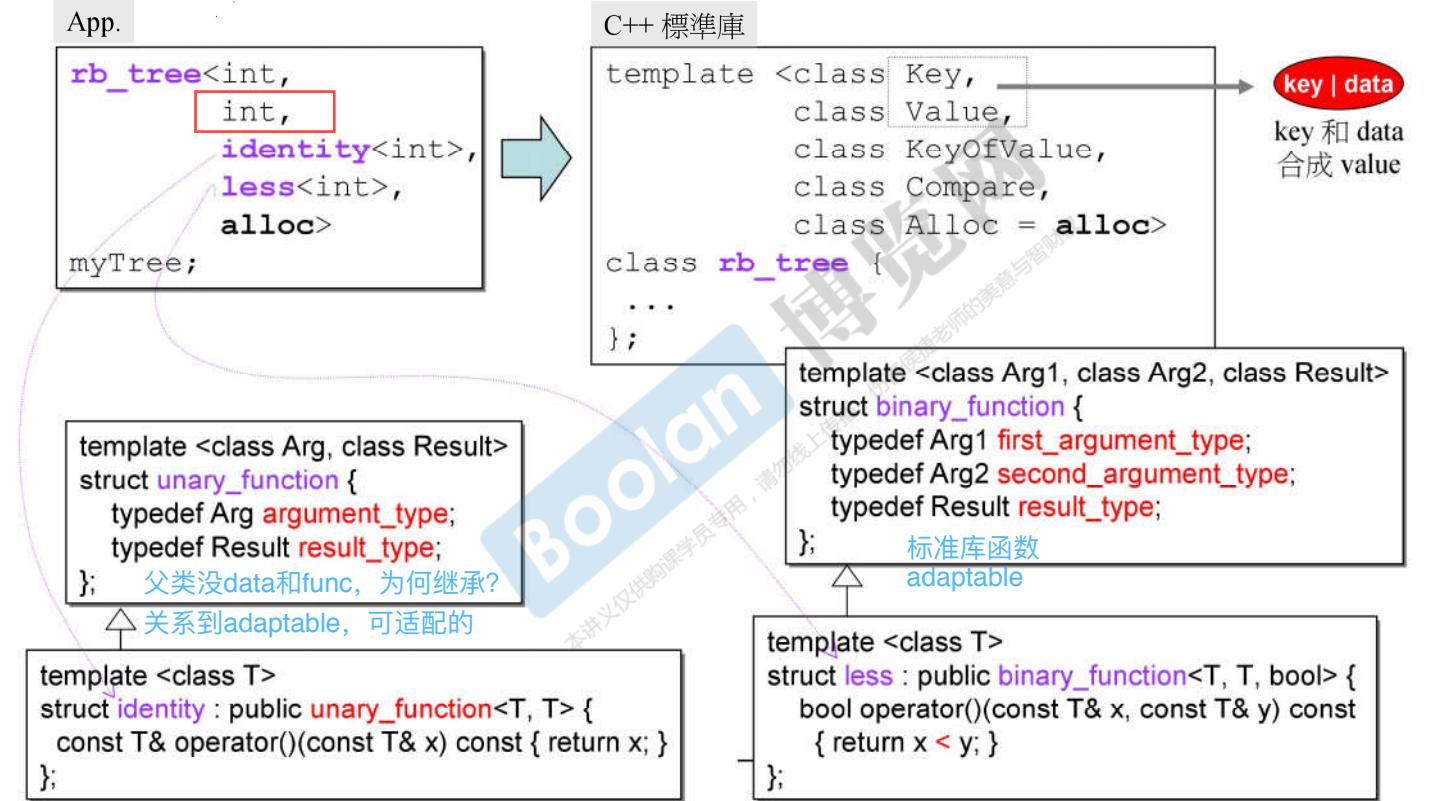
sizeof(rb\_tree)=4+4+9, 默认4对齐

— 侯捷 —

仿函数大小=0, 没数据, 但任何compiler对于大小=0的class, sizeof(obj)==1

list的虚空节点是为了做[]区间, 非真正元素, rb\_tree为了方便实现, 也有虚空节点header

## 容器 rb\_tree



Value==int?, 即没data部分

keyofvalue: identity<int>, gnuc特有, 标准库没有(vc自己设计), identity<>可叫仿函数&函数对象, 因为没数据成员, 函数对象sizeof=0, 实际=1  
identity”同一个东西“, 通过此仿函数告诉rb\_tree, 输入value=x, 输出key也=x, key和value是同一个东西

## 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 元素5有几个

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

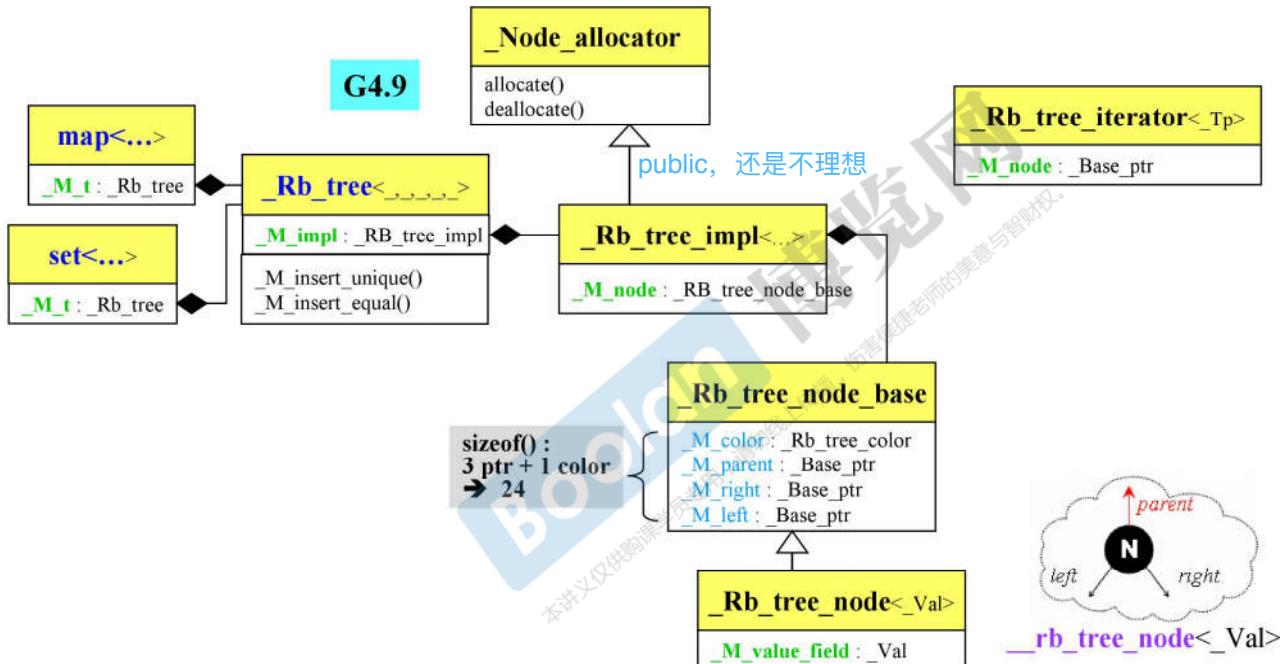
```
_Rb_tree<int, int, _Identity<int>, less<int>> itree;
cout << itree.empty() << endl; //1
cout << itree.size() << endl; //0

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

4.9比2.9复杂的设计，oo为了某个目标而很乐意这样实现，在一个class里有个指针来表现其实现impl，而class没太多设计，此手法叫HandleAndBody  
`_Rb_tree_color`: 是enum

## 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()`。

排序状态：指已经有序的容器，遍历它可按序输出

## 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;
...
};
```

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。

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不能用iterator改变key, const\_iterator不允许改内容, 常量iterator指向常量  
set和queue & stack一路货色

## 容器 set, in VC6

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

inner class

```
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;
};

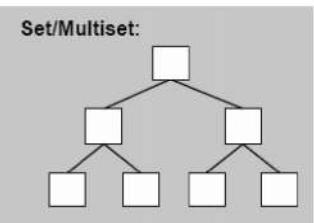
template <class T>
struct identity : public unary_function<T, T> {
    const T& operator()(const T& x) const { return x; }
};
```

— 侯捷 —

130



## 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()); 0~32767
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)。

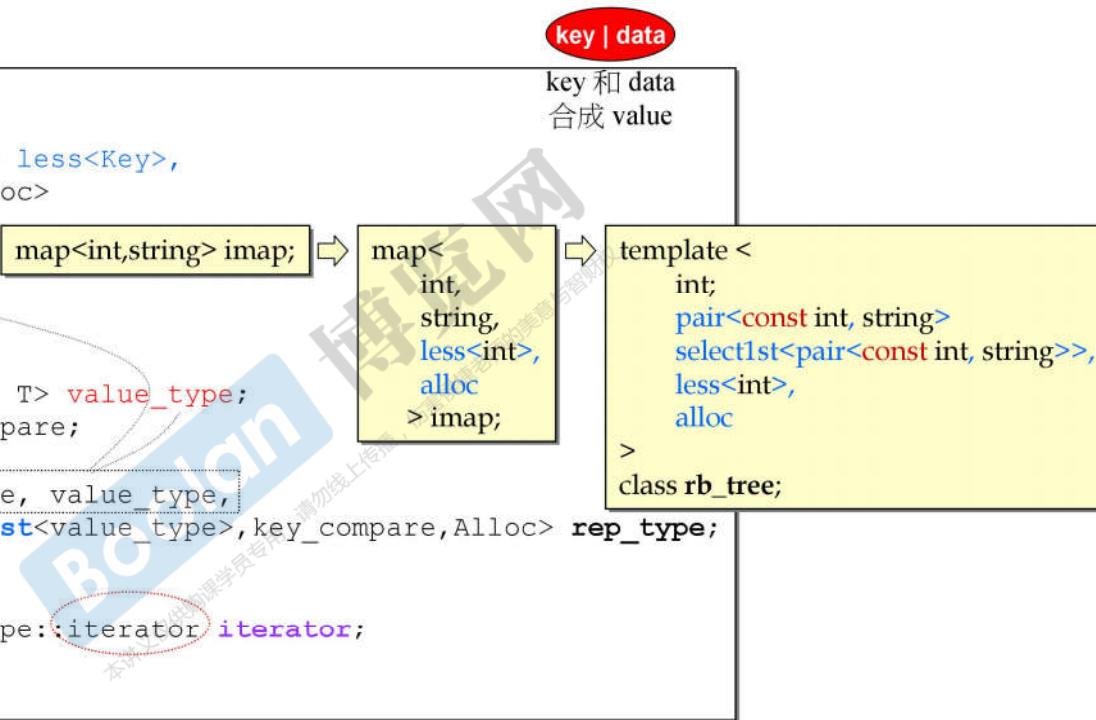
我們無法使用 map/multimap 的 iterators 改變元素的 key  
(因為 key 有其嚴謹排列規則)，但可以用它來改變元素的 data。因此 map/multimap 內部自動將 user 指定的 key type 設為 const，如此便能禁止 user 對元素的 key 賦值。

```
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;
...
};
```



select1st<>从一堆中取出第一个，而value的第一个总是key, gnuc特有  
iterator就是rb\_tree的iterator，虽然map<int, string>, 但pair<const Key, T>才是rb\_tree的真实value

— 侯捷 —

## 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; }  
};
```

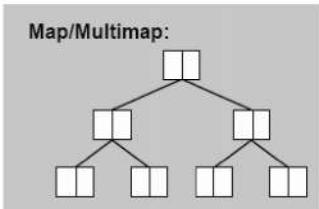
— 侯捷 —

134

pair<>是标准库中的struct, first & second

所以, identity和select1st是gnuc多提供的, 不是vc少提供的, 因为标准库没要求

## 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 }
```

虽然选择multimap, 但这里key不重复, data会重复

需要自己pair

map独有(标准规定): []下标操作符: 可指定index拿值(index可是int, string等), 也可做insert

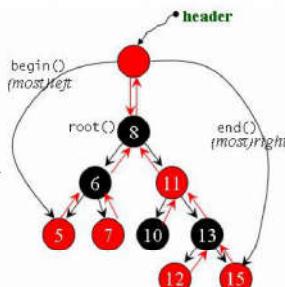
lower\_bound(): 标准库algo, 70多个

第一个元素: 若有重复的, 如7个"hello"(肯定排在一起)

## 容器 map, 独特的 operator[]

G4.9

```
mapped_type&
operator[](const key_type& __k)
{
    // concept requirements
    __glibcxx_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.
 */
```

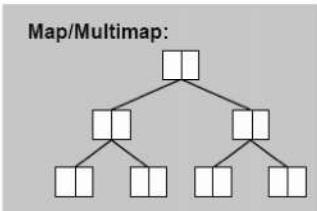
- lower\_bound 是二分搜尋 (binary search) 的一種版本, 試圖在 sorted [first,last) 中尋找元素 value。若 [first,last) 擁有與 value 相等的元素(s), 便返回一個 iterator 指向其中第一個元素。如果沒有這樣的元素存在, 便返回「假設該元素存在時應該出現的位置」。也就是說它會返回 iterator 指向第一個「不小於 value」的元素。如果 value 大於 [first,last) 內的任何元素, 將返回 last。換句話說 lower\_bound 返回的是「不破壞排序得以安插 value 的第一個適當位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

— 侯捷 —

if(\_\_i == end() || key\_comp)(\_\_k, (\*\_\_i).first) //如果没找到, 就insert(), 速度不如直接insert()



## 使用容器 map



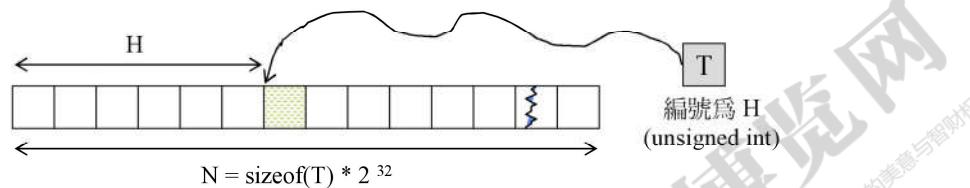
```
ex D:\handout\c++11-test-DevC++\Test-STI
select: 14
how many elements: 1000000

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

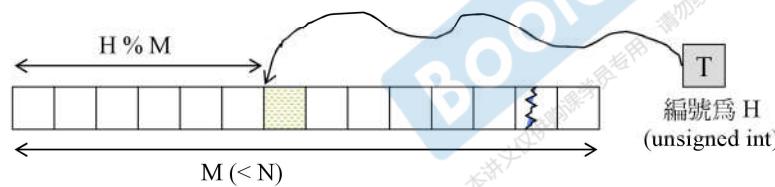
```
696 namespace jj14
697 {
698     void test_map(long& value)
699     {
700         cout << "\ntest_map()..... \n";
701
702         map<long, string> c;
703         char buf[10];
704
705         clock_t timeStart = clock();
706         for(long i=0; i< value; ++i)
707         {
708             try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

## 容器 hashtable

空間足夠時：



空間不足時：



同样是set/map/multiset/multimap，但是hashtable支持的

`rb_tree`中，红黑元素怎么跑，红黑怎么转换，高度平衡实现

hashtable没太多数学，却有很多经验值在其中

`obj`元素可以通过func折合成数值(unsigned int)，则数值变化介于0~ $2^{32}$ ，若放下所有不重复变化，需要N个mem，不可能准备这么多

每个元素有0~ $2^{32}$ 多变化，但一般不会有这么多个元素，所以无需准备N个mem

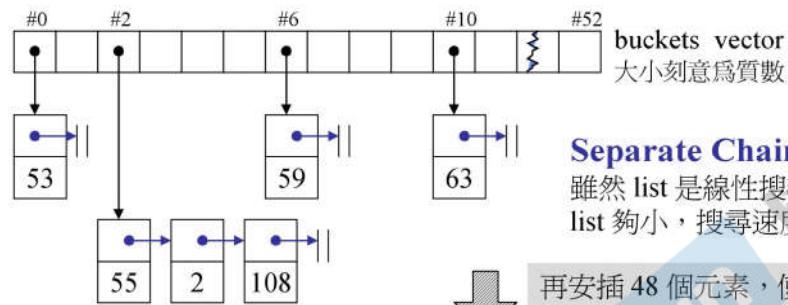
用M个mem存放T，若T编号H=32767，则H=32767，H%M时，容易发生碰撞

— 侯捷 —

138

## 容器 hashtable

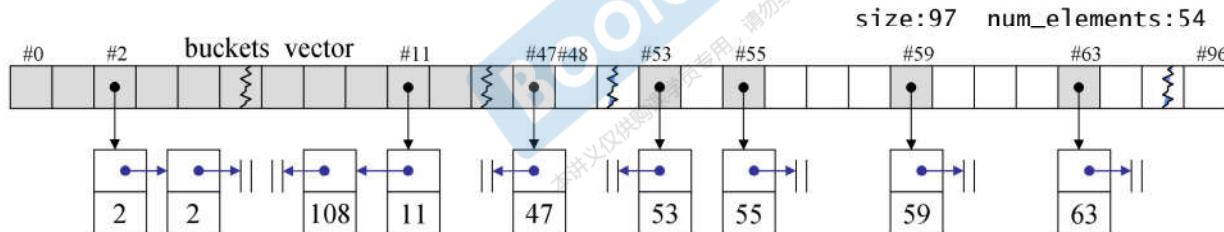
`insert_unique()`, 次序: 59, 63, 108, 2, 53, 55 num\_elements: 6



### 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



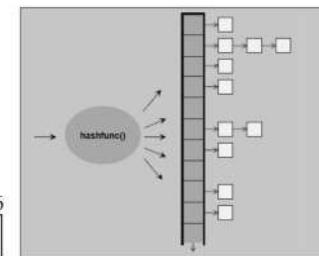
我們可以使用 hashtable iterators 改變元素的 data，但不能改變元素的 key  
(因為 hashtable 根據 key 實現嚴謹的元素排列)。

G2.9

x G4.9

`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 };
```



139

链表长时，查找会很慢，所以链表太长则需打散

打散：bucket，当buckets数量 < 元素个数，就打散(buckets\*2)，但一般都用buckets\*2附近的质数(素数)，gnuc初值为53，质数在`_stl_prime_list`  
 $53 \times 2 = 106$ ，附近质数97

## 容器 hashtable

```

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;

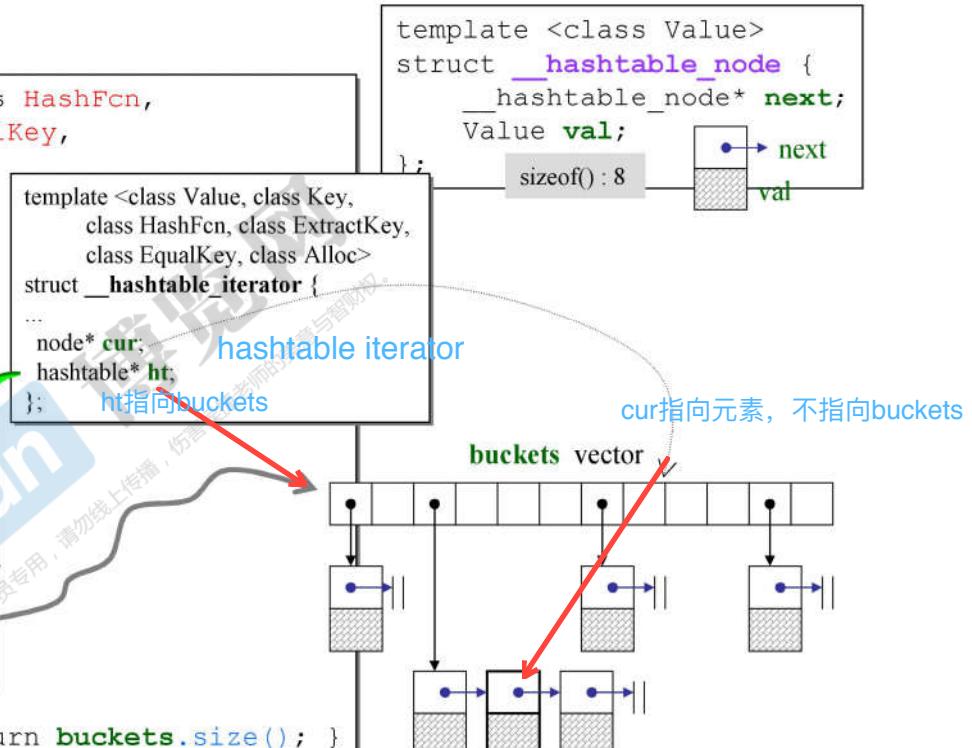
private:
    hasher hash;
    key_equal equals;
    ExtractKey get_key;
    sizeof() : 19 → 20

    typedef __hashtable_node<Value> node;
    vector<node*, Alloc> buckets;
    size_type num_elements; 元素总数

public:
    size_type bucket_count() const { return buckets.size(); }
    ...
};

```

三个函数对象



140

key & value同rb\_tree

HashFcn, 把每个obj折合成编号, 仿函数/函数/函数对象, 算出的编号叫hashcode

ExtractKey, 同rb\_tree中keyofvalue

EqualKey, obj如何比大小

三个函数对象理论=0, 但没法实现, 只能=1, sizeof=3\*1 + vector的3根指针 + 4 = 19, 对齐=20

node: 元素节点

这里是单项链表, vc是双向链表

hashtable iterator指向某个元素, 当移动到某个链表尾部时, 可回到下一个bucket

## 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; }
};
```

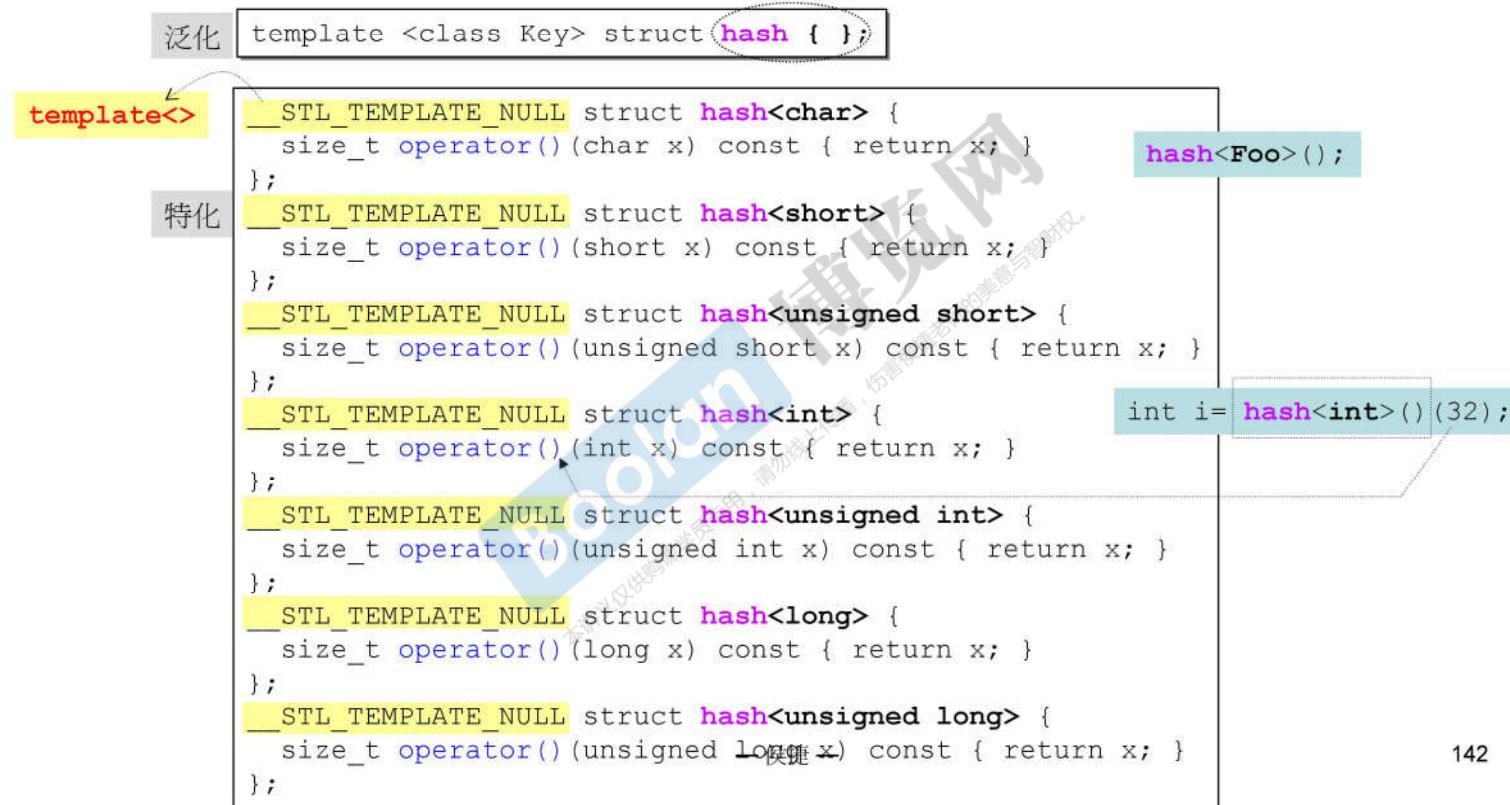
C++ 標準庫

```
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



142

自定义hash func也要如上特化

## 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); }
};
```

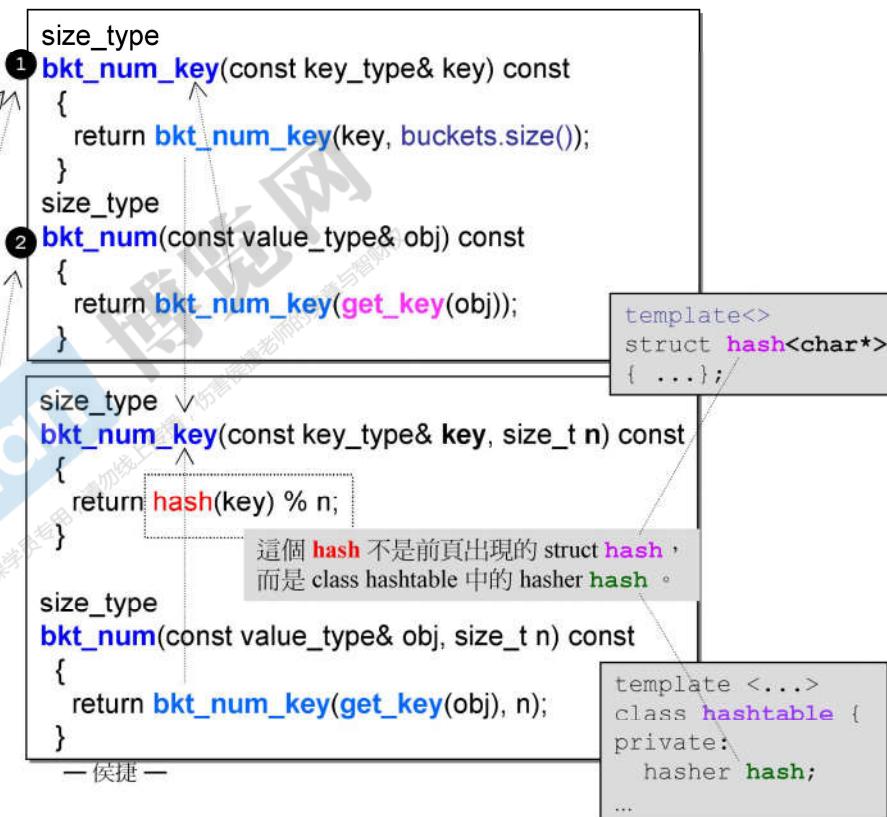
注意，標準庫 **G2.9** 沒有提供現成的  
`hash<std::string>` 自己寫 hash func  
**G4.9** 有提供，在  
...\\4.9.2\\include\\c++\\bits\\basic\_string.h

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);
...
}
```

hash table中一些函数，都是关于计算元素落在哪个buckets



— 侯捷 —

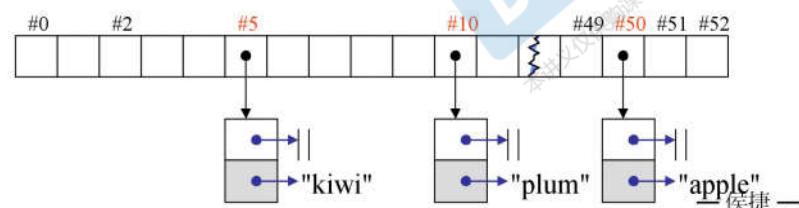
## hash-function, hash-code

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");

```



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

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

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

↑

```

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             _Selectist< 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;
```



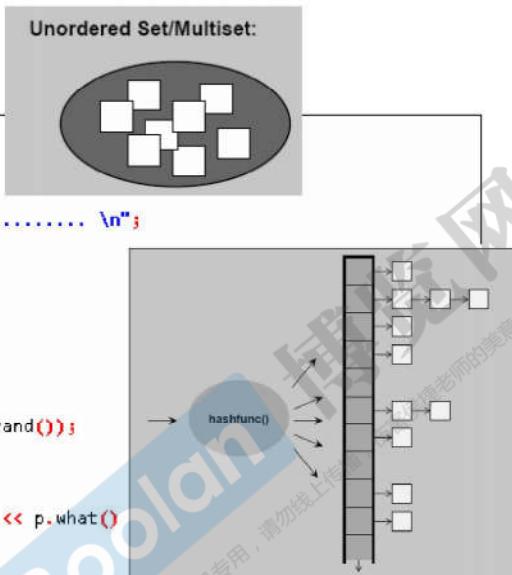
容器 `unordered_set`, `unordered_multiset`

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     }
```



第i个bucket中元素个数

```
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
```

buckets永远 > 元素总数

bucket\_count() == 62233, 4.9版的不再是事先定义好的\_\_stl\_prime\_list[] = 一堆质数

associative container包括rb\_tree和hash\_table

## 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



27.算法形式

28.迭代器分类category

# C++ 標準庫

## 體系結構與內核分析

(C++ Standard Library — architecture & sources)

第三講



侯捷

Boolan

本讲义仅供购课学员专用，请勿线上传播，伤害侯捷老师的美意与知识产权。

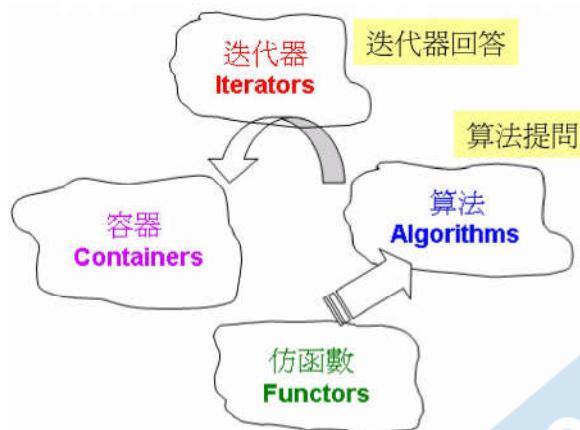
— 侯捷 —

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 —

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

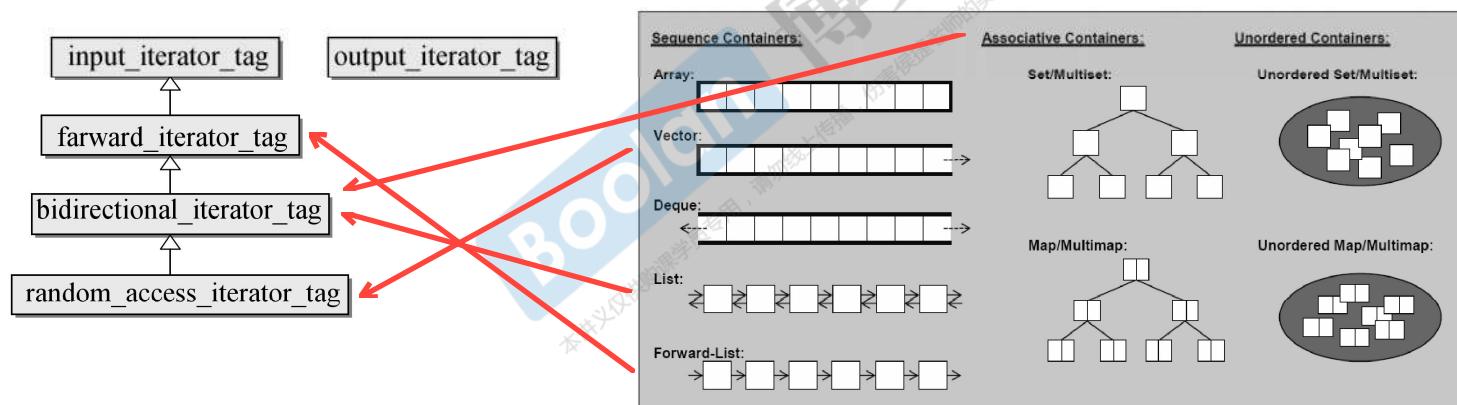
```
template<typename Iterator>
Algorithm(Iterator itr1, Iterator itr2)
{
    ...
}
```

```
template<typename Iterator, typename Cmp>
Algorithm(Iterator itr1, Iterator itr2, Cmp comp)
{
    ...
}          多数algo允许传criterion, 即Cmp, cmp是仿函数
```

iterator能回答，那algo虽然看不到container，但也可通过iterator知道container属性

## ■■■ 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 {};
```



— 侯捷 —

157

hashtable 需要根据 buckets 的链表是单向还是双向  
category: iterator 移动方式的分类

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



```
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,""));  
  
void _display_category(random_access_iterator  
{ cout << "random_access_iterator" << endl; }  
void _display_category(bidirectional_iterator  
{ 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);  
}
```

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 —

istream\_iterator <> 和 ostream\_iterator <> 技术上是 adapter

## ■■■ 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);           typeid(itr)是obj

    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`

istream\_iterator & ostream\_iterator 是标准库规定的，各个版本实现不同，但接口相同  
4.9 版的继承，就是为了少打几个字

## 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;
```

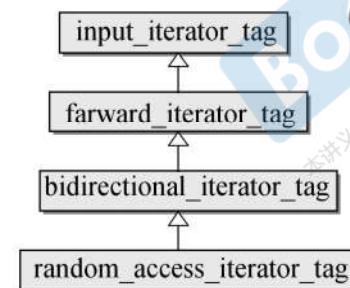
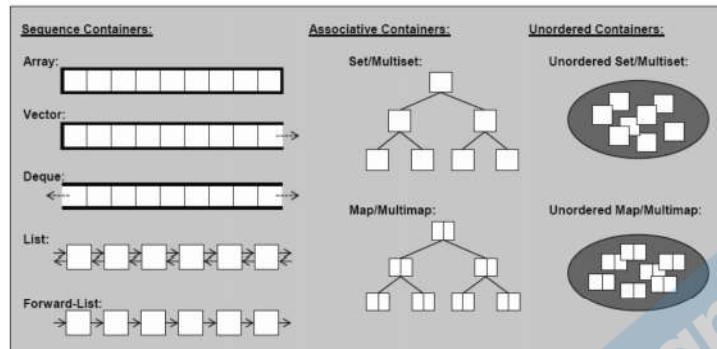
G3.3

```
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



```

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;
}

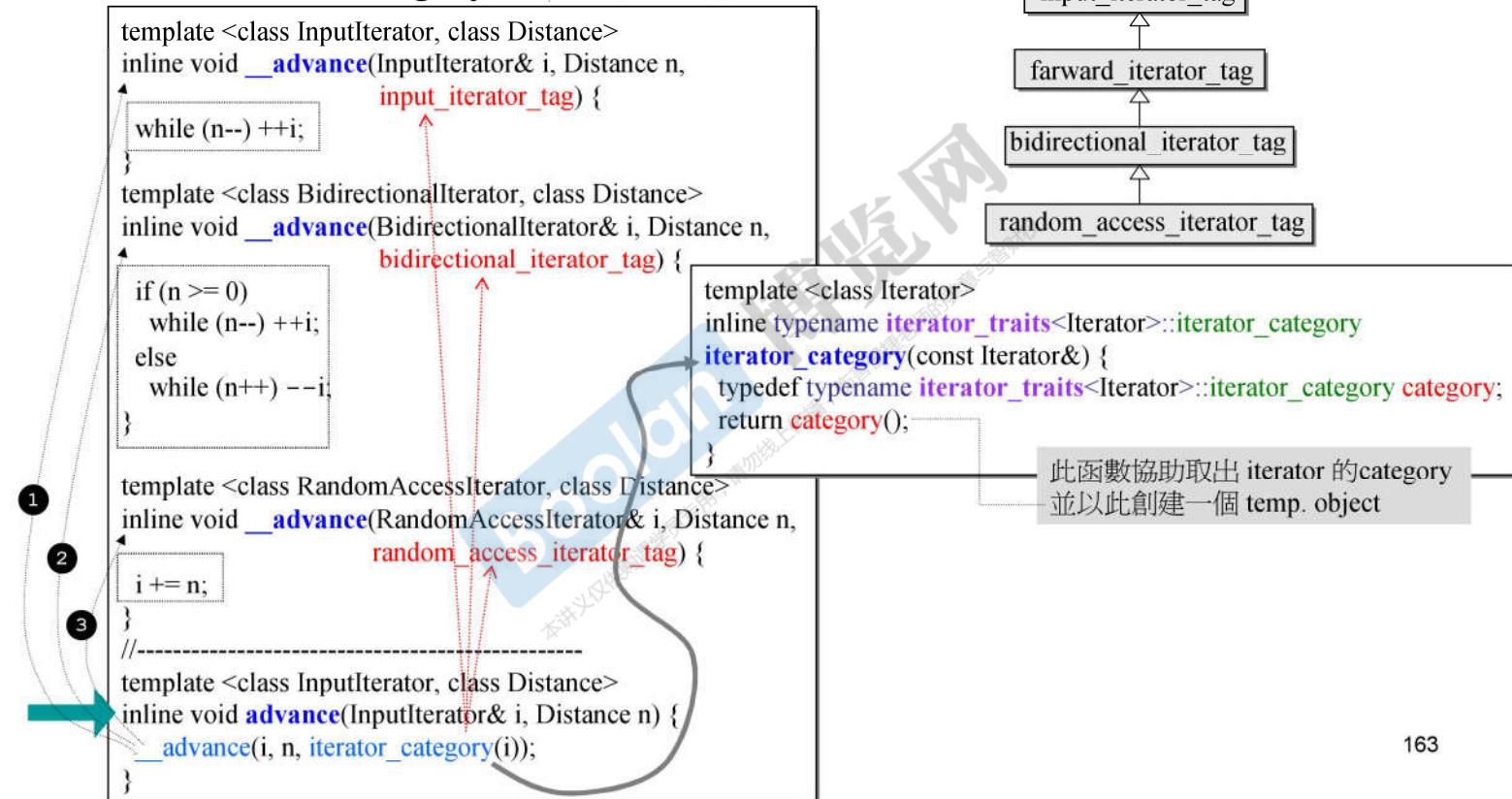
template <class RandomAccessIterator>
inline iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first, RandomAccessIterator last,
           random_access_iterator_tag) {
    return last - first;
}
// -----
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type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

A red arrow points from the `random_access_iterator_tag` box to the `random_access_iterator_tag` parameter in the second `__distance` template. A green arrow points from the `input_iterator_tag` box to the `input_iterator_tag` parameter in the first `__distance` template. A red dotted line connects the `random_access_iterator_tag` parameter to the `random_access_iterator_tag` in the third template definition.

distance()非一线algorithms, 两个指针距离, 作为其他algorithms的基础



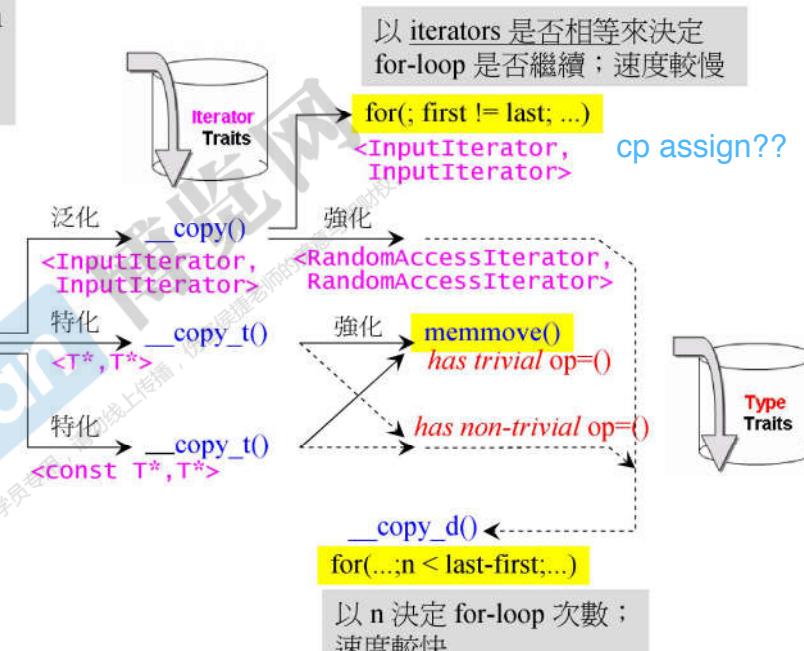
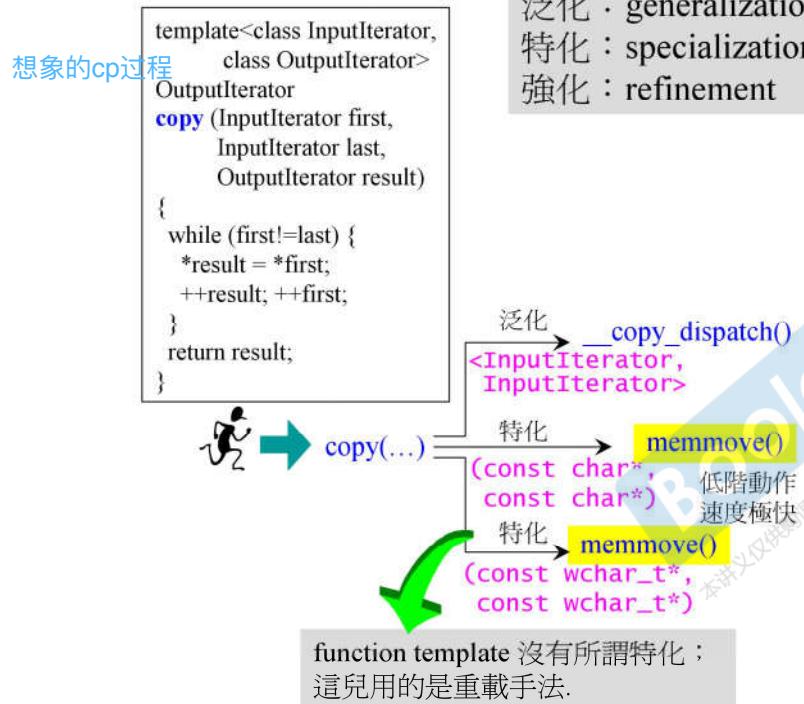
## iterator\_category 對算法的影響



`advance()`也是基础

因为category间的关系是继承，所以只实现3个或2个函数即可

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



— 侯捷 —

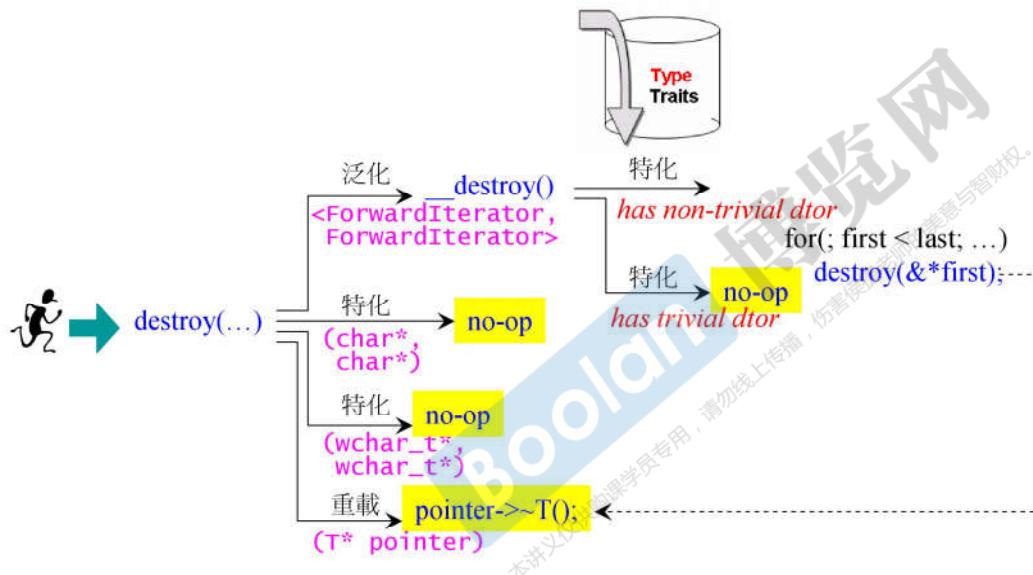
164

copy一直在检查category，来确定是否做特殊动作

type traits问type问题：copy assign是否重要，eg:复数类，不需要写5大，用默认的，而默认的不重要

type traits不属于stl(6大部件)，属于标准库

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

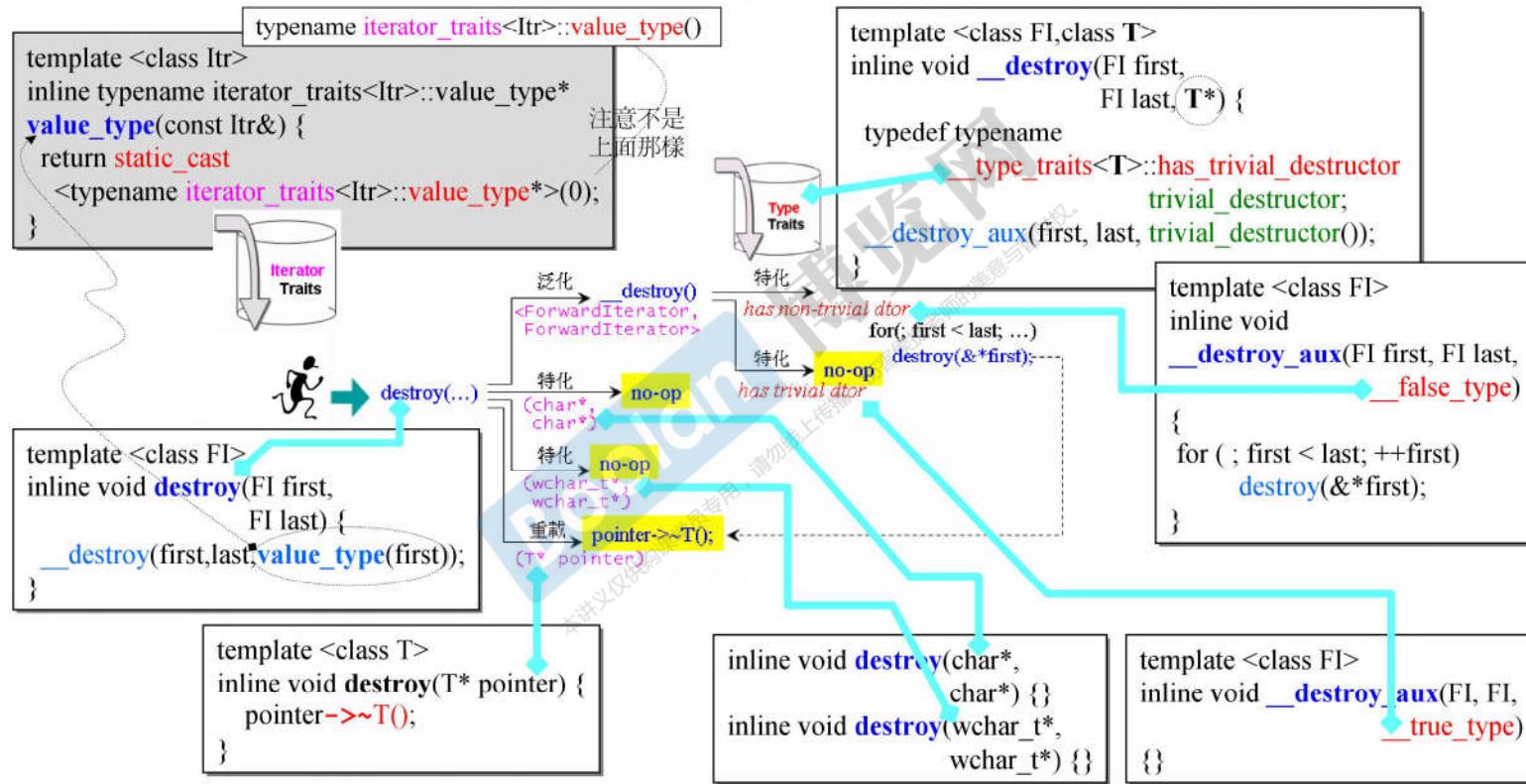


— 侯捷 —

165

释放mem动作是其他函数

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



可见：algo效率和它能否判断iterator的category有关系

## 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。

```
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()`

vs.

```
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;
}
```

**unique\_copy:** 重复的不cp

右边result传进**output\_iterator\_tag**, 则不能\*result操作, 所以才有左边的特化版本

## //// 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);
    }
}
```

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

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

168

sort是qsort, qsort的运行需要用iterator的直接跳跃, random\_access, 但sort主函数中没有限制只接受random\_access的iterator, 只是暗示了iterator的类型为RandomAccessIterator, 这样, 主函数编译过了, 但真正到跳跃的地方就失败

## 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 }
```

functor

很多算法都有两个版本

BinaryOperation: 两个操作数的

## //// 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 }
```

initializer\_list: 也是个container  
for(int i : container)

## ■■■ 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;
}
```

predicate: 暗示是判断式，即能判断真假

## //// 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  
都是通过key找data

## ■■■ 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

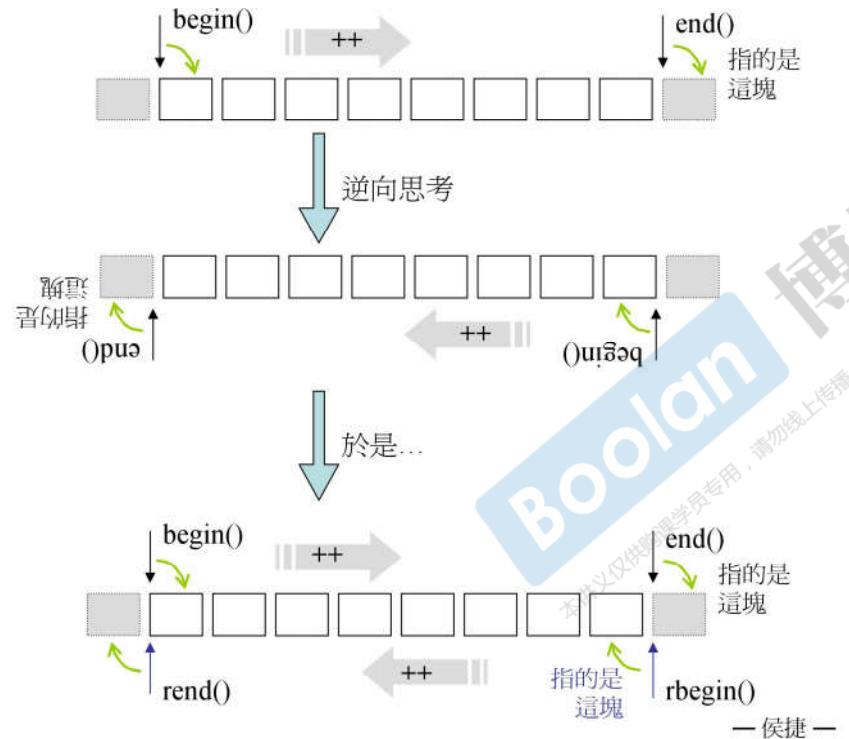
遍歷自然形成 sorted 狀態

容器帶有成員函數 sort() :

list, forward\_list

std::sort要求是random\_access, 可跳跃的, 而list等不能跳跃, 否则会出错

## 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



## 算法 binary\_search

Test if value exists in **sorted** sequence

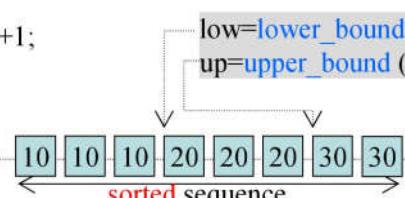
```
template <class ForwardIterator, class T>
bool binary_search (ForwardIterator first,
                    ForwardIterator last,
                    const T& val)
{
    first = std::lower_bound(first, last, val);
    return (first!=last && !(val < *first));
}
```

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。

```
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;
}
```



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.

查找插入20的low/up位置

## 仿函数 functors

```
//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——
```

仿函数：只为算法服务

图中是标准库提供的三大分类的functor

Logical: bit级别运算

这里列出5个functor, 大约有24个

标准库提供的functor都有继承关系, 如: `binary_function<T,T,T>`

没有继承unary / binary 的functor, 表示没融入stl体系结构, 可能没满足stl对functor的各种要求, 给algo用并运行下去, 可能出问题

## //// 仿函數 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>());
```

這就沒有融入 STL

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

//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); }  
};

## 仿函數 functors 的可適配 (adaptable) 條件

```
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;
};
```

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;
```

— 侯捷 —

181

一个是unary\_function，一个是binary\_function

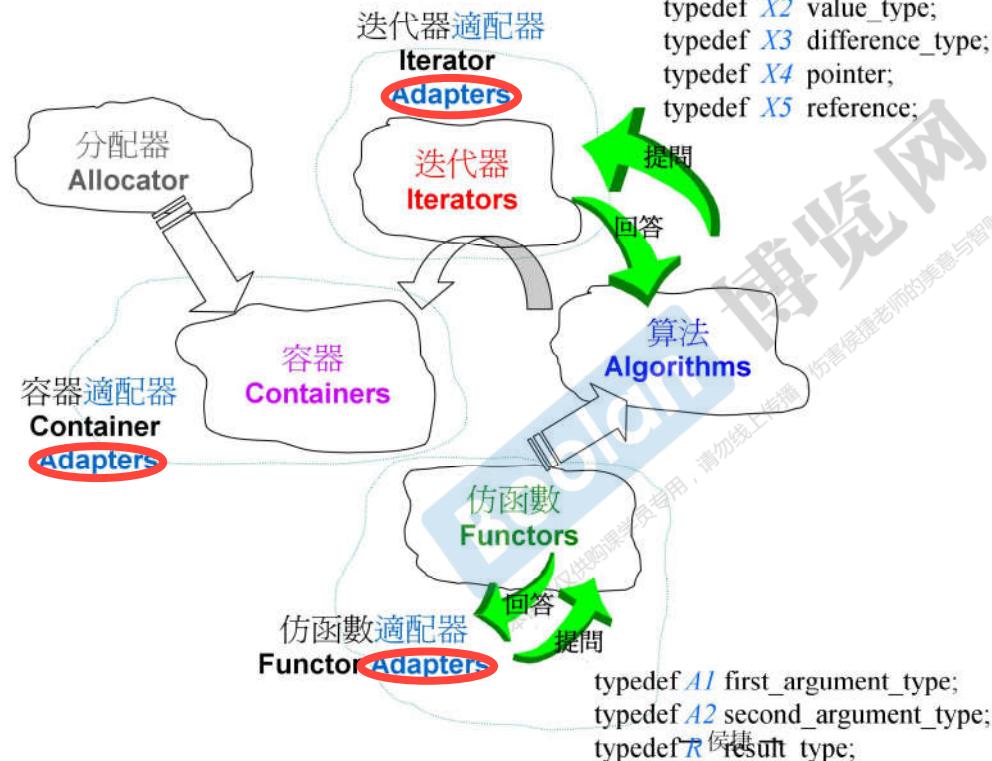
sizeof都是0，但编译器实现问题，实际是1，但有子类时，则unary\_function/binary\_function都是0

继承：可继承父类数据，方法，typedef

adaptable：此functor可否被修改，被适配，涉及到部件adapter，若希望自己的functor被适当调整，则选择一个，适当继承

因为当functor被adapter改造时，adapter可能问typedef相关的问题

## 存在多種 Adapters



182

23个设计模式里，有一个是adapter

adapter：改造既有的东西，如三个参数变两个参数，改函数名等

三种adapter(主要是functor adapter)有共性：关键是，A改造了B，以后用A，看不到B

三种adapter修饰之后，都会表现的像个被修饰的样子

A取到了B的功能：1>继承，2>composition，而adapter都是用了composition

functor adapter包装functor后再给algo用，即algo也需要知道一些functor的问题，需要问adapter一些问题

## 容器适配器 : 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(); }
};
```

适配deque这个容器：改函数名，如：push\_back()改为push()  
改造成stack, queue的味道

## 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));
}
```

```
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;
}
```

binder2nd先记录下来，用时才开始绑定value, op(x, value)

binder2nd是主角，直接用时模板参数不好确定，eg: less<int>()的类型不好确定，这里类型是less<int>

使用less<int>说明比int，所以40应该是int，否则编译报错，图中灰色部分做检查

arg2\_type(x), 40是否是 / 可转换成问出来的类型

binder2nd和binder2nd都是adapter

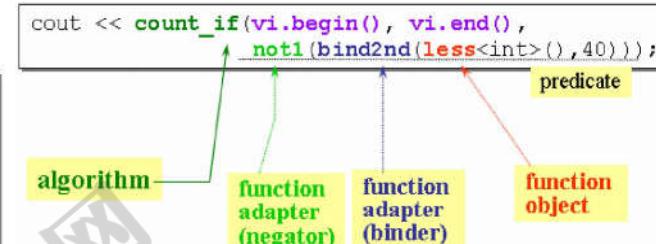
functor必须能回答三个问题，才能和binder2nd这样的adapters搭配

如果functor能回答，就叫adaptable functor

灰色都加了typename，有些编译器(每个编译器带一套标准库)在x::y前没加typename，希望在此类编译器也能编译过

编译器在编译到Operation::second\_argument\_type时(还没被调用)，还不知道Operation

把x<y换成x<40，即bind2nd修饰了比大小动作  
less<int>() : obj



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

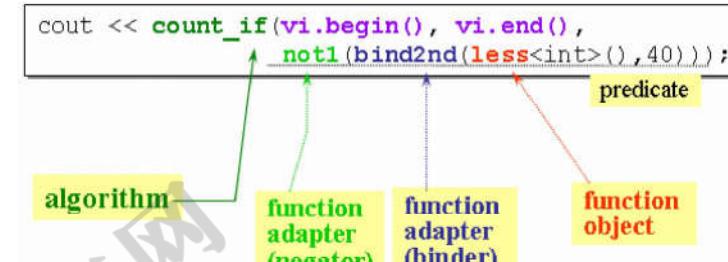
```
template <class Operation>
class binder2nd
: public unary_function<typename Operation::first_argument_type,
    typename Operation::result_type> {
protected:
    Operation op; // 內部成員，分別用以記錄算式和第二實參 Operation是靠bind2nd辅助函数推的
    typename Operation::second_argument_type value;
public:
    // constructor
    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 { adapter表现的像个functor
        return op(x, value); // 實際呼叫算式並取 value 為第二實參
    }
};
```

binder2nd修饰less, binder2nd继承unary\_function为了自己可adaptable  
同样有bind1st

## 函数适配器：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)
    }
};
```

—侯捷—

185

条件是: not1(xxx)不小于40

还有个not2

gnu c独有adapter: compose1和compose2, 非标准, 如复杂的条件: 容器中元素\*3+2/100, 可用compose去组合条件

## 新型适配器, bind

...\\include\\c++\\backward\\backward\_warning.h backward 向后兼容

```
/*
```

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

Use: 2.0

<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: 1.0

<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

— 侯捷 —



## 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);
2785 cout << fn_five() << '\n'; // 5
2786
2787 auto fn_half = bind (my_divide,_1,_2);
2788 cout << fn_half(10) << '\n'; // 5
2789
2790 auto fn_invert = bind (my_divide,_2,_1);
2791 cout << fn_invert(10,2) << '\n'; // 0.2
2792
2793 auto fn_rounding = bind<int> (my_divide,_1,_2);
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() _1 for this
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
```

\_1代表实际调用时的第一实参位置

bind<int>(my\_divide, \_1, \_2) : bind函数模板只有一个模板参数用来表示返回类型, 不指定<int>时返回类型是my\_divide的返回类型

MyPair ten\_two{10,2} : obj{x,y}赋初值

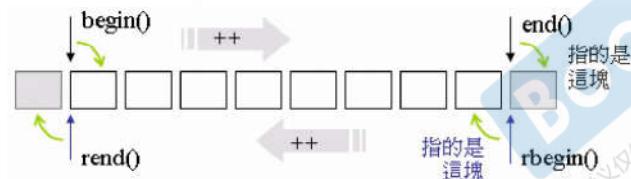
multiply()无参, 但bind时要给this占位

cbegin()表示const begin(), 不能改内容

## 迭代器适配器： 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。

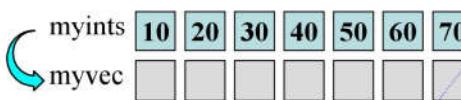
    // 前進變成後退，後退變成前進
    self& operator++() { --current; return *this; }
    self& operator--() { ++current; return *this; }
    self operator+(difference_type n) const { return self(current - n); }
    self operator-(difference_type n) const { return self(current + n); }
};
```

iterator做adapter后，形成reverse\_iterator

## 迭代器适配器： inserter

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

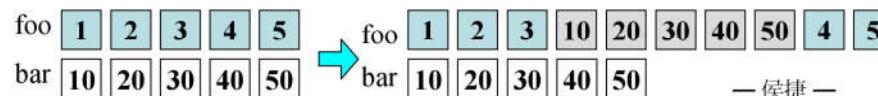
copy(myints, myints+7, myvec.begin());
```



```
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_iterator {
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_iterator<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\_iterator<Container>
inserter(Container& x, Iterator i) {
 typedef typename Container::iterator iter;
 return inserter\_iterator<Container>(x, iter(i));
}

copy中是拷贝赋值，赋值前应有足够空间，而inserter是先做出空间在assign

advance(it, 3)，it前进3个位置，不能it+3

copy中已经是赋值拷贝了，如何改变赋值拷贝为insert？用inserter把foo的iterator改成inserter

重点：operator=()

## 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; &cout
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 adapter, 未知的adapter, 不属于iterator,container,functor的adapter, 因为存储并改造ostream, 所以可叫ostream adapter

out\_it adapter绑定到std::cout, ", "作为delimiter

copy已经是写好的, 重点操作符重载operator=()

作用在result上的操作符有: \*, =, ++, return后的拷贝构造, 想个彼此不矛盾的规则, 达到这些操作符形成的元素跑到cout里

## 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

eos: 无参代表end of stream

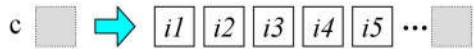
(\*in\_stream&gt;&gt;value): cin等待输入

只要对iit做++, 就是在读

```
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;  
}
```

希望 copy 不断从 cin 读 data 并放到 c 容器的某个位置

copy 搭配不同的 adapter 有不同表现

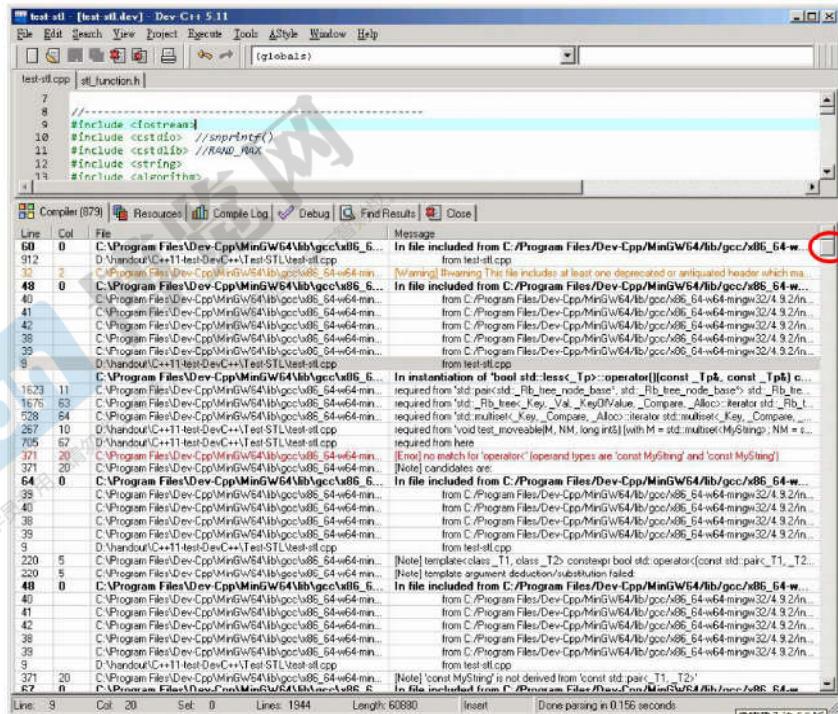
iit 初始化时，就已经在读

```
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

— 侯捷 —

195

第三讲是stl

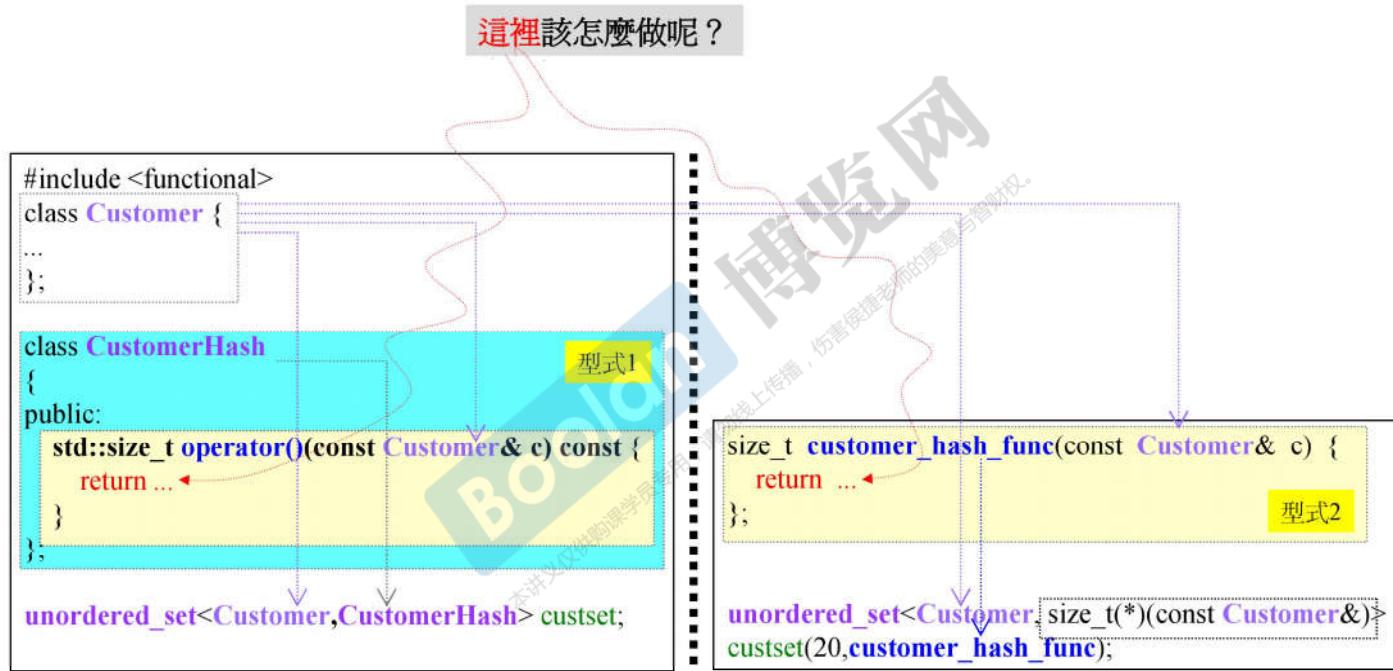
第四讲是标准库中stl的周围



# 勿在浮沙築高台



## ■■■■ 一個萬用的 Hash Function



— 侯捷 —

197

必须为要放入hash table的元素写hash func

## 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);  
    } 碰撞多  
};
```

```
#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);  
}  
  
last  
③ // auxiliary generic functions  
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);  
    };
```

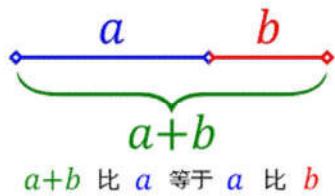
```
② template <typename T, typename... Types>  
    inline void hash_val(size_t& seed,  
    recursive const T& val, const Types&... args) {  
        hash_combine(seed, val);  
        hash_val(seed, args...);  
    } 逐一取 val  
    改變 seed  
    (pass by reference)  
  
① // auxiliary generic function  
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 —

tr1版本提供了`hash_val`, 比CustomerHash好, 现在放在cpp2.0

## ■■■ 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 }
```

↓

```
1 //auxiliary generic function
2 template <typename... Types>
3 inline size_t hash_val (const Types&... args)
4 {
5     size_t seed = 0;
6     hash_val(seed, args...);
7     return seed;
8 }
9
10 class CustomerHash {
11 public:
12     size_t operator()(const Customer& c) const {
13         return hash_val(c.fname, c.lname, c.no);
14     }
15 }
```

— 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.
```

先算出hash\_code，看落在哪个篮子



```
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
```

— 侯捷 —

201

以前初始bucket\_count()==57, 4.9改了  
bucket\_count() < 元素总数, 做rehash

## 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

形式3

## 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()); }  
};
```

标准库在g2.9中没给string写hash\_func, g4.9中写了

— 侯捷 —

203

## 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);

```

— 侯捷 —

```

// 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 fl = 1.0; // yields float
typedef tuple_element<1,TupleType>::type T;

```

tr1后，标准库才有tuple

**t1 < t2**, 逐个成员比较, 成员个数不同? google  
**cout << t1**:需为tuple设计operator<<  
**tie(i1, f1, s1) = t3**; //tuple t3要绑定i1,f1,s1  
**tuple\_size**里肯定有**typedef xxx value**;

## tuple 元之組合, 數之組合

### G4.8 節錄並簡化

```

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

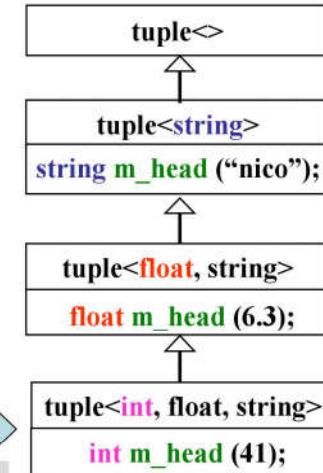
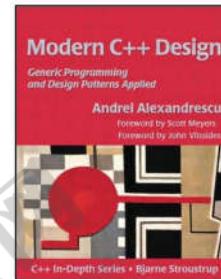
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
: private tuple<Tail...>
{
    typedef tuple<Tail...> inherited;
public:
    tuple() { }
    tuple(Head v, Tail... vtail) {
        : m_head(v), inherited(vtail...) { }
    }
    typename Head::type head() { return m_head; }
    inherited&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

```

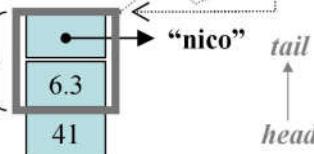
呼叫 base ctor 並予參數  
(不是創建 temp object)

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()) →`



205

modern c++ design早就提出tuple觀念，但是c++沒有語法可支持此觀念(variadic template, 可實現自動遞歸)

**type traits**

**G2.9**

<b>泛化</b> <pre>template &lt;class type&gt; struct __type_traits {     typedef __true_type     typedef __false_type     typedef __false_type     typedef __false_type     typedef __false_type     typedef __false_type };</pre>	<b>Plain Old Data</b>	<pre>struct __true_type { }; struct __false_type { };  this_dummy_member_must_be_first; has_trivial_default_constructor; has_trivial_copy_constructor; has_trivial_assignment_operator; has_trivial_destructor; is_POD_type;</pre>
		pod: 平淡类型, 即c类型struct, 只有data
<b>特化</b> <pre>template&lt;&gt; struct __type_traits&lt;int&gt; {     typedef __true_type     typedef __true_type     typedef __true_type     typedef __true_type     typedef __true_type };</pre>	<code>__type_traits&lt;Foo&gt;::has_trivial_destructor</code>	
<b>特化</b> <pre>template&lt;&gt; struct __type_traits&lt;double&gt; {     typedef __true_type     typedef __true_type     typedef __true_type     typedef __true_type     typedef __true_type };</pre>		

206

2.9版的type traits实现很简单

默认的回答`__false_type`是重要的, 默认所有type都重要

若自定义class已确定这些都不重要, 可对此class特化

eg: 自定义complex时, dtor / cp ctor / cp assign用编译器默认的即可, 所以不重要

一般只有带着指针成员的class, 需要考虑特化; 或者需要在dtor里关闭资源的class

目的: 为algo服务

# 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 )

## Type traits

### 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 )

cpp2.0 type traits更庞大且无需对自定义class写偏特化

只需传自定义class到如下函数即可

如何知道自定义class里是否有virtual func, dtor是否重要?

## 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; ret: trur/false
    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>::valu
    cout << "is_rvalue_reference\t" << is_rvalue_reference<T>::valu
    cout << "is_pointer\t" << is_pointer<T>::value << endl;
    cout << "is_member_pointer\t" << is_member_pointer<T>::value <<
    cout << "is_member_object_pointer\t" << is_member_object_pointe
    cout << "is_member_function_pointer\t" << is_member_function_po
    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

oo思想：当类想把自己当成base时，才需要virtual dtor(), 否则不必写virtual dtor()

若一个类被当成base才可能会有virtual func

但没virtual func的类被当成base，他是不会拒绝的，java可以拒绝

## //// 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 —

=default; 和不写一样

## //// 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 —

以下都是利用template操作type，而不是操作变量

## //// type traits, 實現, is\_void

<pre> 1423 // remove_const 1424 template&lt;typename _Tp&gt; 1425     struct remove_const 1426     { <b>typedef</b> _Tp type; }; 1427 1428 template&lt;typename _Tp&gt; 1429     struct remove_const&lt;_Tp const&gt; 1430     { <b>typedef</b> _Tp type; }; 1431 1432 // remove_volatile 1433 template&lt;typename _Tp&gt; 1434     struct remove_volatile 1435     { <b>typedef</b> _Tp type; }; 1436 1437 template&lt;typename _Tp&gt; 1438     struct remove_volatile&lt;_Tp volatile&gt; 1439     { <b>typedef</b> _Tp type; }; 1440 1441 // remove_cv 1442 template&lt;typename _Tp&gt; 1443     struct remove_cv 1444     { 1445         <b>typedef</b> typename 1446             remove_const&lt;typename remove_volatile&lt;_Tp&gt;::type&gt;::type 1447             type; 1448     }; 1449 1450 // add_const 1451 template&lt;typename _Tp&gt; 1452     struct add_const 1453     { <b>typedef</b> _Tp const type; }; </pre>	<pre> 165 template&lt;typename&gt; 166     struct __is_void_helper 167     : public false_type { }; 168 169 template&lt;&gt; 170     struct __is_void_helper&lt;void&gt; 171     : public true_type { }; 172 173 /// is_void 174 template&lt;typename _Tp&gt; 175     struct is_void 176     : public __is_void_helper&lt;typename remove_cv&lt;_Tp&gt;::type&gt;::type 177     { }; </pre>
--	---

216

remove\_cv<>: c is const, v is volatile, 去掉干扰

remove\_const: 总是类似于这种，先写泛化，再写特化

## 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

猜: \_\_is\_xxx 是編譯器在處理, 編譯器編譯時當然知道 class 有無 virtual func / cp ctor / move ctor

雖然寫出 \_\_is\_xxx 這些函數, 但不以 c++ 源碼形式出現

編譯器暗中幫助了標準庫

— 侯捷 —

218

## ■■■■ 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

220

extern xxx cout; //cout可被此文件之外使用，不是说cout来自外界

## 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

标准库中的operator<<  
typedef basic\_string string;  
thread::id / regular express属于stl之外

## 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

```

CCtor=0 MCtor=7194303 CAsgn=0 MAsgn=0 Dtor=7194309 Ctor=3000006 DCtor=0  
 CCtor=7194303 MCtor=0 CAsgn=0 MAsgn=0 Dtor=7194307 Ctor=3000004 DCtor=0

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

```

M c1;
...
M c11(c1);
M c12(std::move(c1));
c11.swap(c12);

```

— 侯捷 —

222

8547和14235根据当前mem状态有关

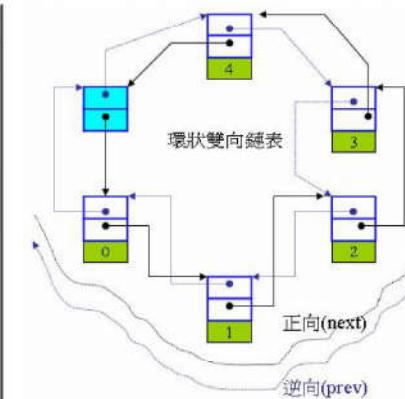
insert(it, val); 刚好rb\_tree和hash\_table都有指定it的版本，但it只起提示作用，不能决定具体存放位置

## 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

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
```



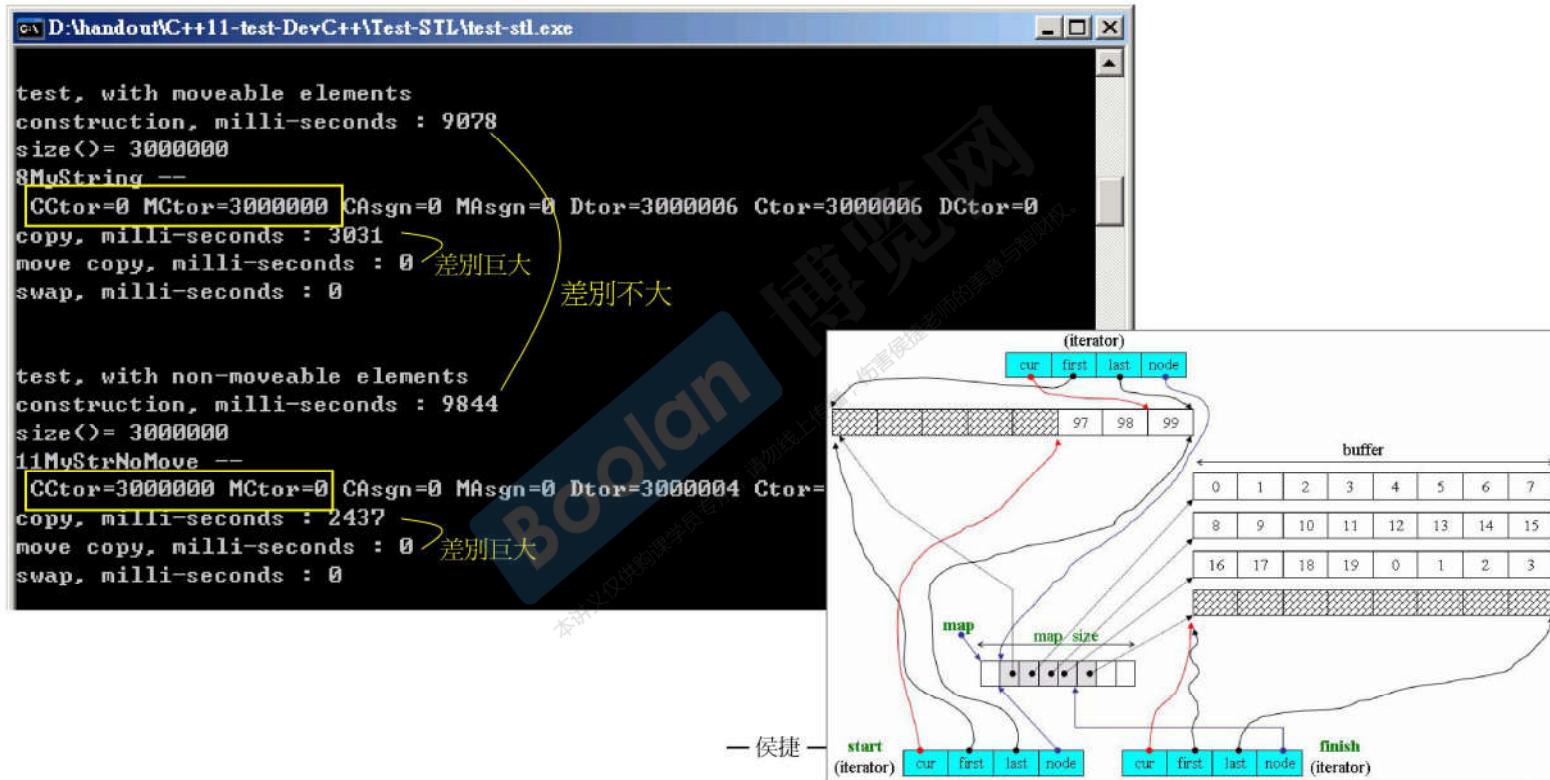
— 侯捷 —

223

这里10765和11016感觉差别不大，那是不是可在容器选择上去避免写带有move的元素？

这里只对容器的insert做了比较，容器其他操作也受move影响

## moveable 元素對於 deque 速度效能的影響

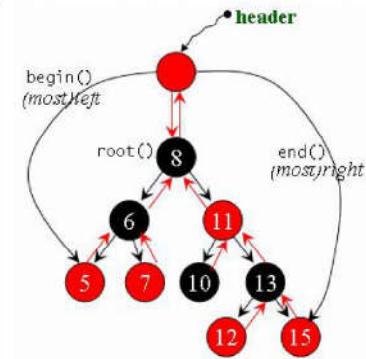


## moveable 元素對於 multiset 速度效能的影響

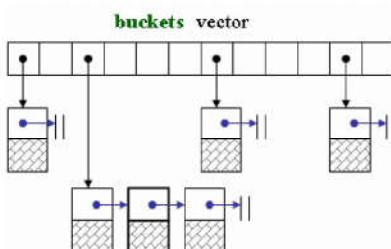
```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

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);
    }
}

```

*(本講義仅供課堂學習用，請勿上傳至網上)*

```

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

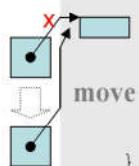
//move ctor, with "noexcept"
MyString(MyString&& str) noexcept
: _data(str._data), _len(str._len) {
    ++MCtor;
    str._len = 0;
    str._data = NULL; //避免 delete (in dtor)
}

//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)
};
```

delete \_data; //cpp沒明說double delete \_data會如何

## 測試函數

```
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**

typedef typename iterator\_traits<typename **M**::iterator>::value\_type **V1type**;

clock\_t timeStart = clock();

for(long i=0; i< value; ++i) {

snprintf(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);

**2 //測試 non-moveable**

...

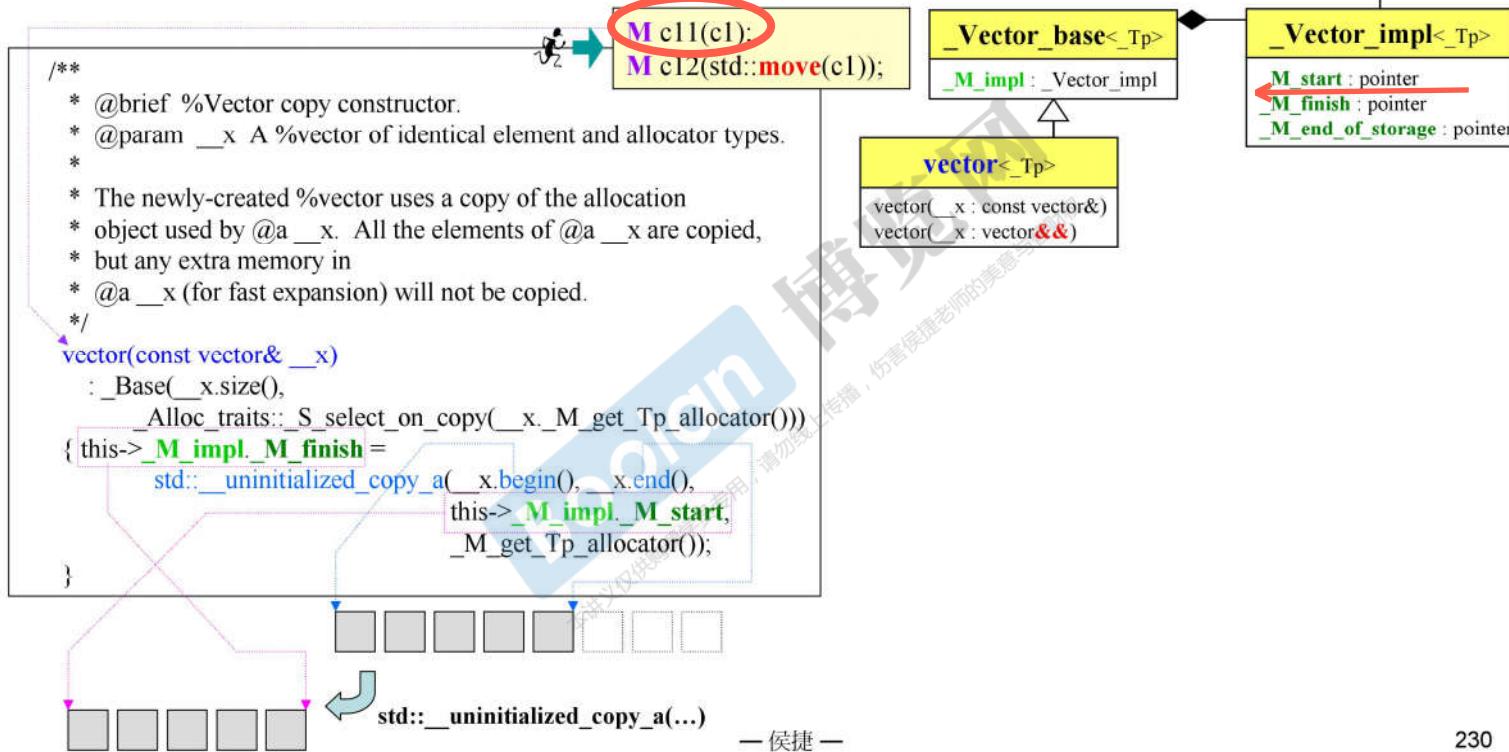
}

调用**test\_moveable(obj)**时，传入的是obj，可否传入**template?** 可以用模板的模板，这里用**iterator\_traits**

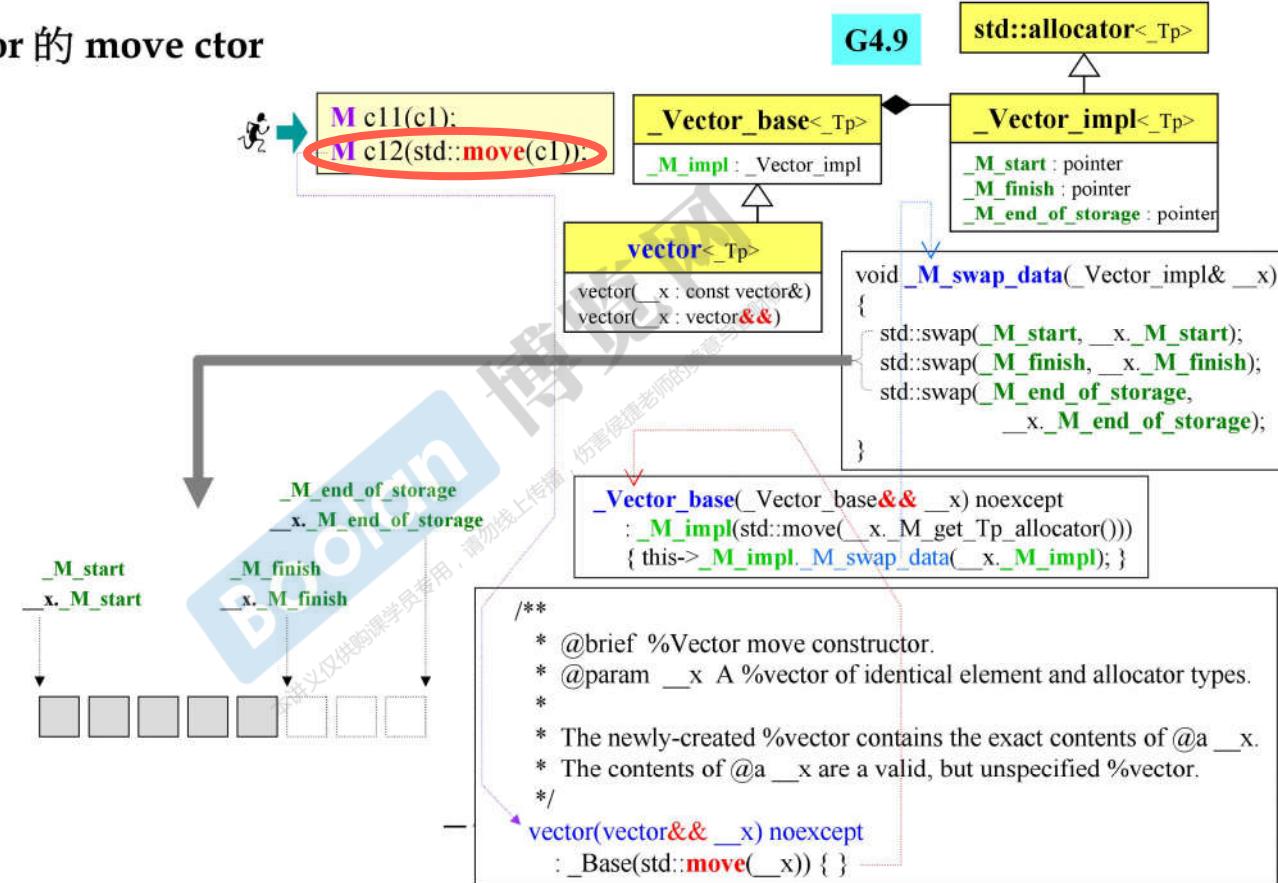
**std::move(c1)**之后，不再使用c1，**V1type(buf)**是临时对象，是rvalue，编译器看到rvalue自动调用**move ctor**(如果有)

**M** c11(c1); //c1是lvalue，编译器不会调用**move ctor**

## vector 的 copy ctor



## 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 }

```

549	/**
550	* @brief Assign the value of @a str to this string.
551	* @param __str Source string.
552	*/
553	basic_string&
554	operator=(const basic_string& __str)
555	{ return this->assign(__str); }

456	/**
457	* @brief Construct string with copy of value of @a str.
458	* @param __str Source string.
459	*/
460	basic_string(const basic_string& __str);

168	template<typename _CharT, typename _Traits, typename _Alloc>
169	basic_string(_CharT, _Traits, _Alloc)::
170	basic_string(const basic_string& __str)
171	: _M_dataplus(__str._M_rep()>_M_grab(_Alloc(__str.get_allocator()),
172	__str._get_allocator(),
173	__str._get_allocator())
174	{ }



The End

