C++ 標準庫

體系結構與內核分析

(C++ Standard Library — architecure & sources)

第一講

侯捷

使用一個東西，
卻不明白它的道理，
不高明！

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

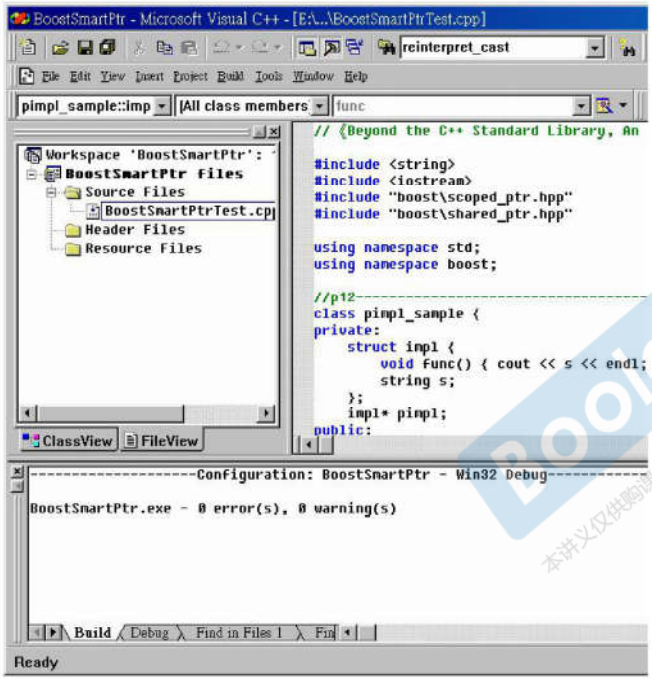
- C++ 標準庫的 header files 不帶副檔名 (.h)，例如 #include <vector>
- 新式 C header files 不帶副檔名 .h，例如 #include <cstdio>
- 舊式 C header files (帶有副檔名 .h) 仍然可用，例如 #include <stdio.h>

- 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"

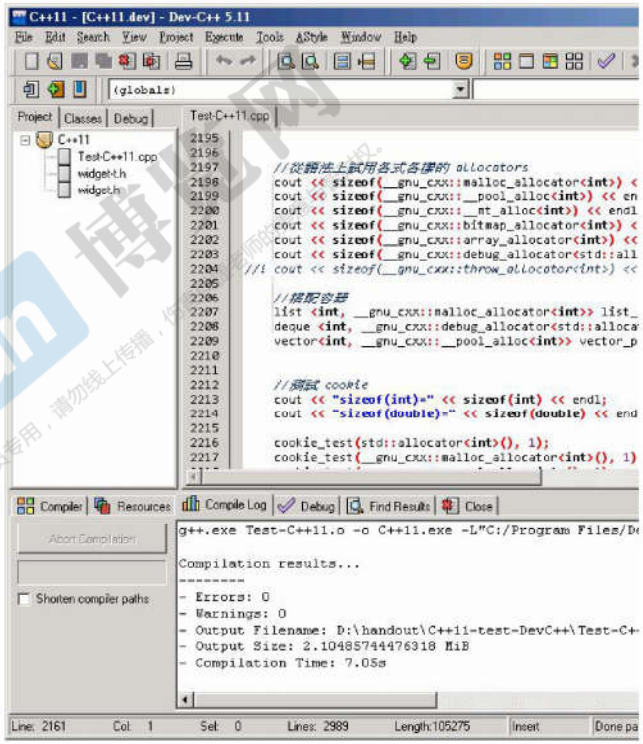
```
#include <string>
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
```

C++ 標準庫, 版本

(Visual C++ 6.0 畫面)



(Dev-C++ 5.11 畫面; with GNU 4.9.2)



重要網頁, CPlusPlus.com

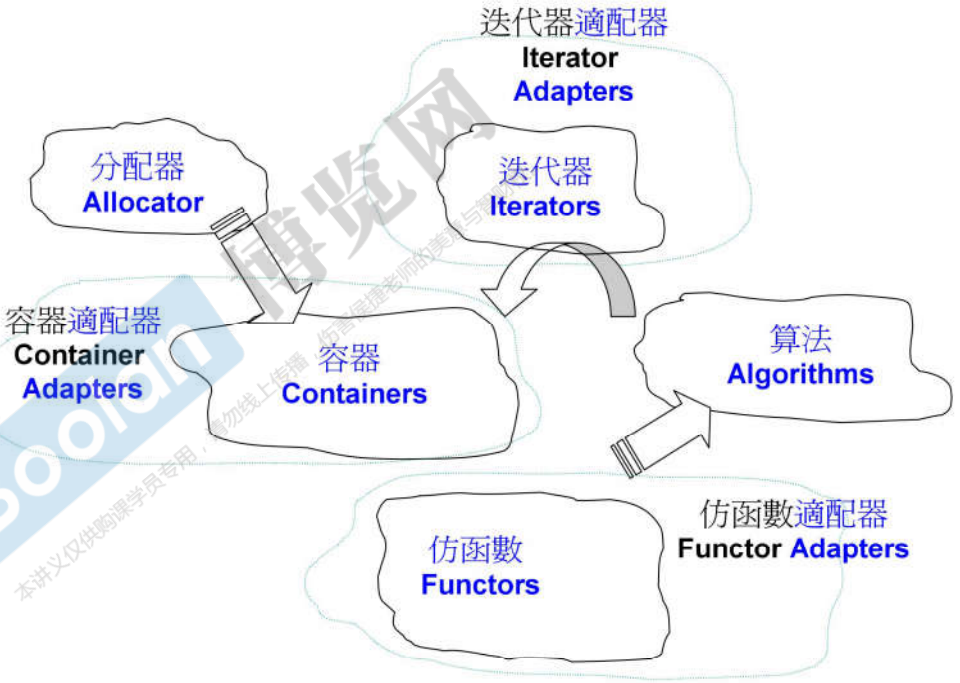
重要網頁, CppReference.com

重要網頁, gcc.gnu.org

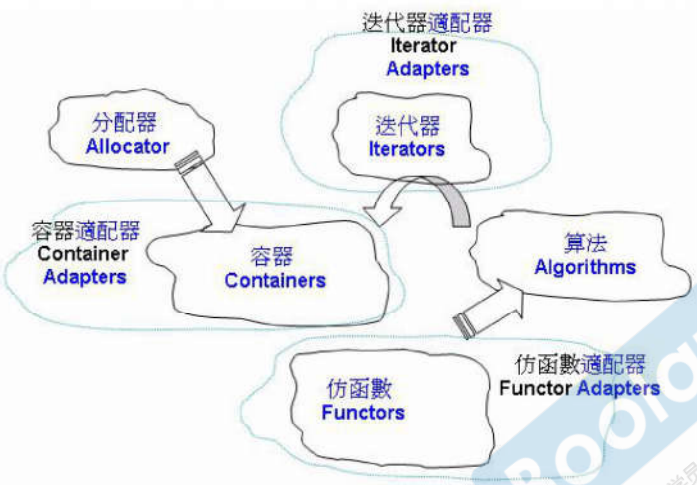
STL 六大部件

STL 六大部件 (Components)：

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



STL 六大部件關係




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

allocator
iterator
container
predicate
algorithm
function adapter (negator)
function adapter (binder)
function object

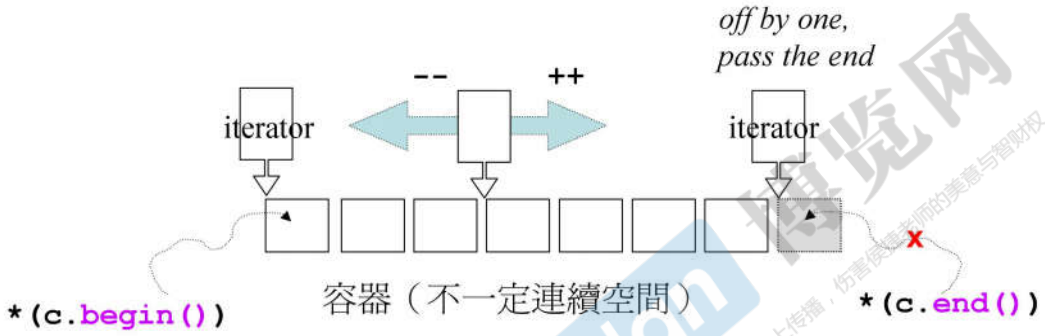
複雜度, Complexity, Big-oh

目前常見的 Big-oh 有下列幾種情形：

1. $O(1)$或$O(c)$：稱爲常數時間(constant time)
2. $O(n)$：稱爲線性時間(linear time)
3. $O(\log_2 n)$：稱爲次線性時間(sub-linear time)
4. $O(n^2)$：稱爲平方時間(quadratic time)
5. $O(n^3)$：稱爲立方時間(cubic time)
6. $O(2^n)$：稱爲指數時間(exponential time)
7. $O(n\log_2 n)$：介於線性及二次方成長的中間之行爲模式。

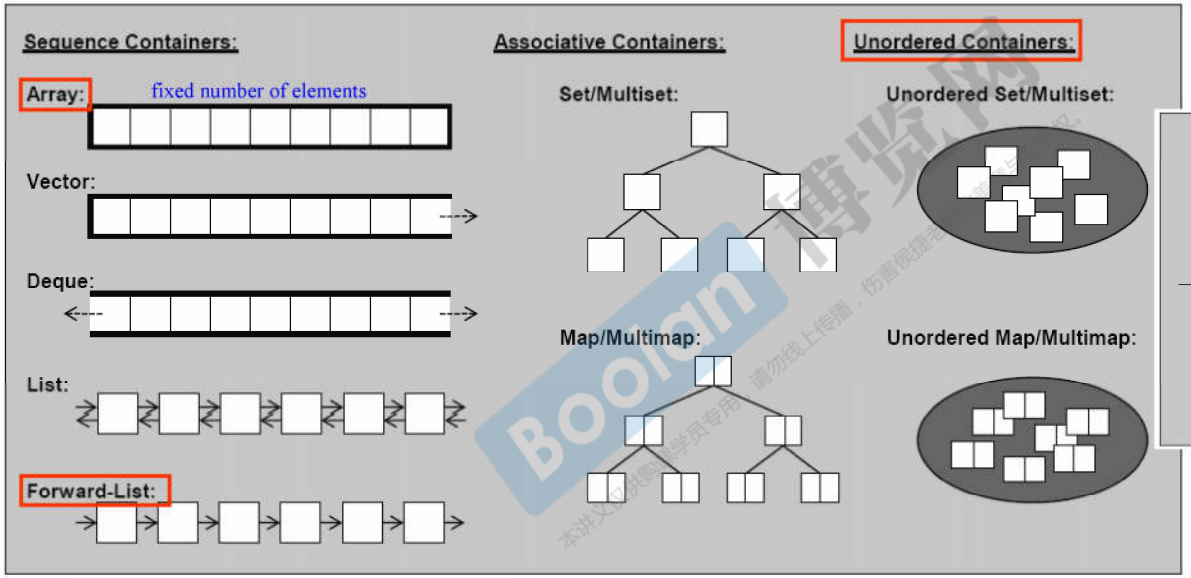
“前閉後開”區間

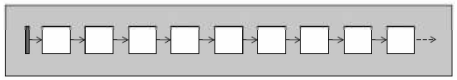
[ ] ( ) [ )

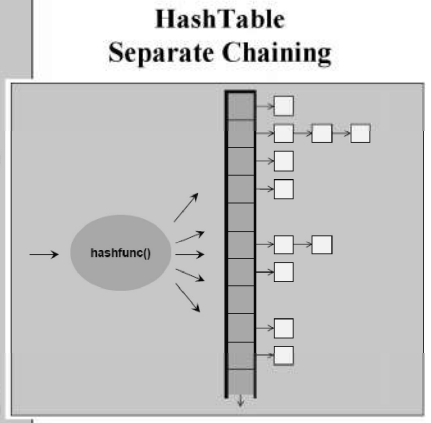



```
Container<T> c;
...
Container<T>::iterator ite = c.begin();
for (; ite != c.end(); ++ite)
      ...
```

容器 – 結構與分類





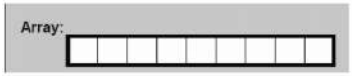


以下測試程序之 輔助函數

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

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

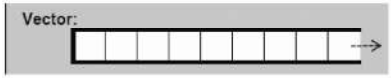
使用容器 array




```cpp
#include <array>
#include <iostream>
#include <ctime>
#include <cstdlib> //qsort, bsearch, NULL
namespace jj01
{
void test_array()
{
    cout << "\ntest_array()..... \n";

array<long,ASIZE> c;

clock_t timeStart = clock();
    for(long i=0; i< ASIZE; ++i) {
        c[i] = rand();
    }
    cout << "milli-seconds : " << (clock()-timeStart) << endl;  //
    cout << "array.size()= " << c.size() << endl;
    cout << "array.front()= " << c.front() << endl;
    cout << "array.back()= " << c.back() << endl;
    cout << "array.data()= " << c.data() << endl;

long target = get_a_target_long();

    timeStart = clock();
    qsort(c.data(), ASIZE, sizeof(long), compareLongs);
long* pItem = (long*)bsearch(&target, (c.data()), ASIZE, sizeof(long), compareLongs);
    cout << "qsort()+bsearch(), milli-seconds : " << (clock()-timeStart) << endl;   //
    if (pItem != NULL)
        cout << "found, " << *pItem << endl;
    else
        cout << "not found! " << endl;
}
}
```

```
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 (0~32767): 20000
qsort()+bsearch(), milli-seconds : 187
found, 20000
```

使用容器 vector




```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 2
how many elements: 1000000

test_vector()..........
milli-seconds : 3063
vector.size()= 1000000
vector.front()= 4047
vector.back()= 2877
vector.data()= 0x2880020
vector.capacity()= 1048576
target (0~32767): 23456
::find(), milli-seconds : 0
found, 23456
sort()+bsearch(), milli-seconds : 2765
found, 23456
```

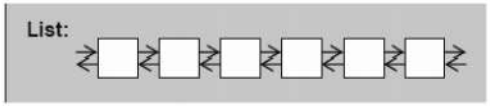
```cpp
#include <vector>
#include <stdexcept>
#include <string>
#include <cstdlib> //abort()
#include <cstdio>  //snprintf()
#include <iostream>
#include <ctime>
#include <algorithm>    //sort()
namespace jj02
{
void test_vector(long& value)
{
    cout << "\ntest_vector()..........\n";

vector<string> c;
char buf[10];

clock_t timeStart = clock();
    for(long i=0; i< value; ++i)
    {
        try {
            snprintf(buf, 10, "%d", rand());
            c.push_back(string(buf));
        }
        catch(exception& p) {
            cout << "i=" << i << " " << p.what() << endl;
                //曾經最高 i=58389486 then std::bad_alloc
            abort();
        }
    }
    cout << "milli-seconds : " << (clock()-timeStart) << endl;
    cout << "vector.size()= " << c.size() << endl;
    cout << "vector.front()= " << c.front() << endl;
    cout << "vector.back()= " << c.back() << endl;
    cout << "vector.data()= " << c.data() << endl;
    cout << "vector.capacity()= " << c.capacity() << endl;
```

使用容器 vector

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

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 2
how many elements: 1000000

test_vector()..........
milli-seconds : 3063
vector.size()= 1000000
vector.front()= 4047
vector.back()= 2877
vector.data()= 0x2880020
vector.capacity()= 1048576
target (0~32767): 23456
::find(), milli-seconds : 0
found, 23456
sort()+bsearch(), milli-seconds : 2765
found, 23456
```

使用容器 list




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

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

```
159 namespace jj03
160 {
161 void test_list(long& value)
162 {
163     cout << "\ntest_list()......... \n";
164
165 list<string> c;
166 char buf[10];
167
168 clock_t timeStart = clock();
169     for(long i=0; i< value; ++i)
170     {
171         try {
172             snprintf(buf, 10, "%d", rand());
173             c.push_back(string(buf));
174         }
175         catch(exception& p) {
176             cout << "i=" << i << " " << p.what() << endl;   //
177             abort();
178         }
179     }
180     cout << "milli-seconds : " << (clock()-timeStart) << endl;
181     cout << "list.size()= " << c.size() << endl;
182     cout << "list.max_size()= " << c.max_size() << endl;
183     cout << "list.front()= " << c.front() << endl;
184     cout << "list.back()= " << c.back() << endl;
185
186 string target = get_a_target_string();
187     timeStart = clock();
188 auto pItem = ::find(c.begin(), c.end(), target);
189     cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
190
191     if (pItem != c.end())
192         cout << "found, " << *pItem << endl;
193     else
194         cout << "not found! " << endl;
195
196     timeStart = clock();
197     c.sort();
198     cout << "c.sort(), milli-seconds : " << (clock()-timeStart) << endl;
199
200 }   //又一陣子才離開函數; 它在 destroy...
201 }
```

```
/** 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(); }
```

std::allocator<_Tp>，亦即 new_allocator，內有：

```
size_type
max_size() const _GLIBCXX_USE_NOEXCEPT
{ return size_t(-1) / sizeof(_Tp); }
```

C++11

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             snprintf(buf, 10, "%d", rand());
224             c.push_front(string(buf));
225         }
226         catch(exception& p) {
227             cout << "i=" << i << " " << p.what() << endl;   //
228             abort();
229         }
230     }
231     cout << "milli-seconds : " << (clock()-timeStart) << endl;  //
232     cout << "forward_list.max_size()= " << c.max_size() << endl;
233     cout << "forward_list.front()= " << c.front() << endl;
234     //! cout << "forward_list.back()= " << c.back() << endl;     //no such member function
235     //! cout << "forward_list.size()= " << c.size() << endl;     //no such member function
236
237
238 string target = get_a_target_string();
239     timeStart = clock();
240 auto pItem = ::find(c.begin(), c.end(), target);
241     cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;    //
242
243     if (pItem != c.end())
244         cout << "found, " << *pItem << endl;
245     else
246         cout << "not found! " << endl;
247
248     timeStart = clock();
249     c.sort();
250     cout << "c.sort(), milli-seconds : " << (clock()-timeStart) << endl;    //
251 }
252 }
```

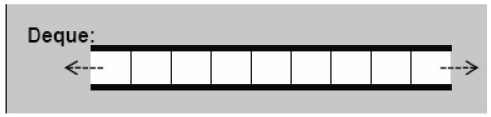
GNU C++

使用容器 slist

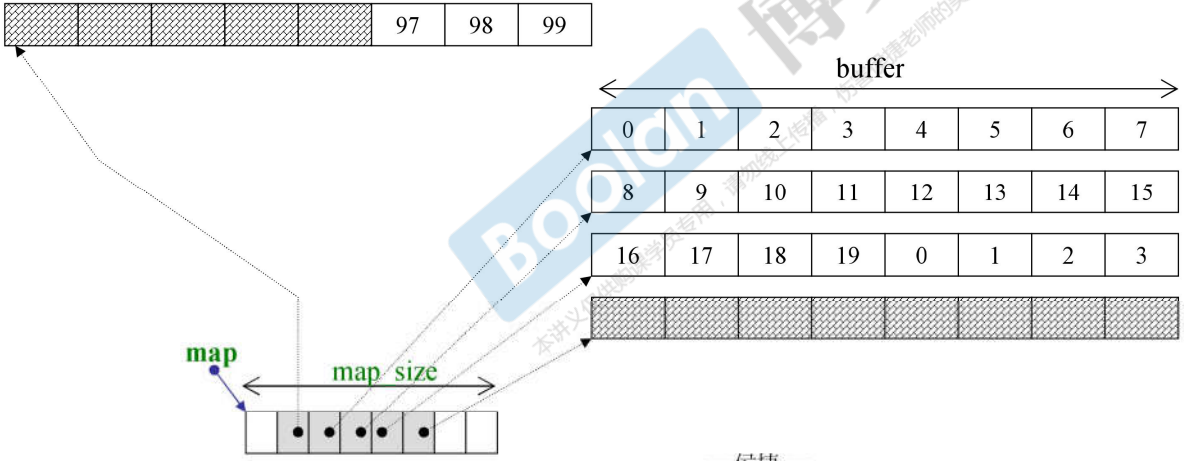
```
D:\handout\C++11-test-DevC++\Test-S
select: 10
how many elements: 1000000

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              snprintf(buf, 10, "%d", rand());
529              c.push_front(string(buf));
530          }
531          catch(exception& p) {
532              cout << "i=" << i << " " << p.what() << endl;   //
533              abort();
534          }
535      }
536      cout << "milli-seconds : " << (clock()-timeStart) << endl;
537
538  }                                                          //
539  }
```

使用容器 deque



分段连续



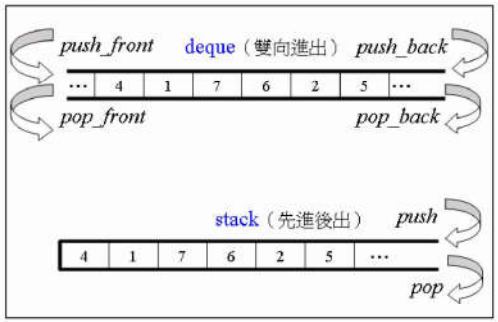
使用容器 deque


```
D:\handout\C++11-test-DevC++\Test-STL\tes
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             snprintf(buf, 10, "%d", rand());
275             c.push_back(string(buf));
276         }
277         catch(exception& p) {
278             cout << "i=" << i << " " << p.what() << endl;   //
279             abort();
280         }
281     }
282     cout << "milli-seconds : " << (clock()-timeStart) << endl;  //
283     cout << "deque.size()= " << c.size() << endl;
284     cout << "deque.front()= " << c.front() << endl;
285     cout << "deque.back()= " << c.back() << endl;
286     cout << "deque.max_size()= " << c.max_size() << endl;
287
288 string target = get_a_target_string();
289     timeStart = clock();
290 auto pItem = ::find(c.begin(), c.end(), target);
291     cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
292
293     if (pItem != c.end())
294         cout << "found, " << *pItem << endl;
295     else
296         cout << "not found! " << endl;
297
298     timeStart = clock();
299     ::sort(c.begin(), c.end());
300     cout << "::sort(), milli-seconds : " << (clock()-timeStart) << endl;
301 }
302 }
```

Container Adapter
不提供 iterator

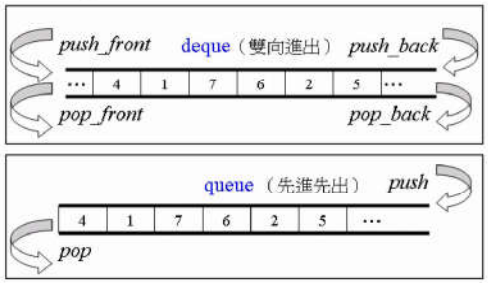
使用容器 stack




```
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 }
```

使用容器 queue

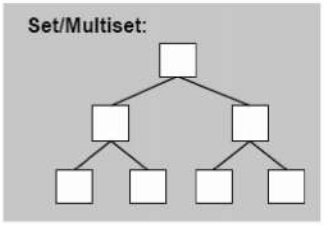



```
D:\handout\C++11-test-DevC++\Test-
select: 18
how many elements: 300000

test_queue().........
milli-seconds : 890
queue.size()= 300000
queue.front()= 12074
queue.back()= 14585
queue.size()= 299999
queue.front()= 6888
queue.back()= 14585
```

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

使用容器 multiset

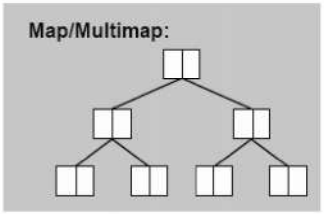
红黑树




```
D:\handout\C++11-test-DevC++\Test-STL\t
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             snprintf(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();
347 auto pItem = c.find(target);        //比 ::find(...) 快很多
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 }
```

使用容器 multimap

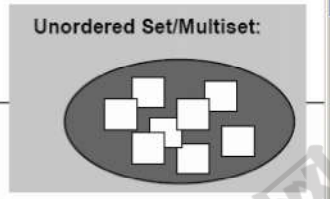



```
D:\handout\C++11-test-DevC++\Test-STL\t
select: 7
how many elements: 1000000

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

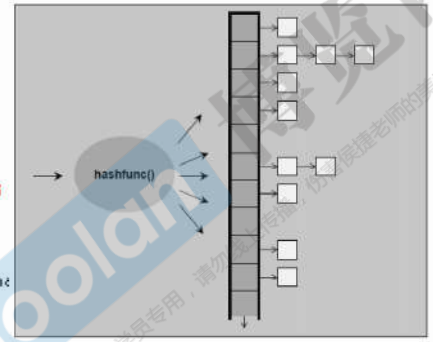
```
364  namespace jj07
365  {
366  void test_multimap(long& value)
367  {
368      cout << "\ntest_multimap()......... \n";
369
370  multimap<long, string> c;
371  char buf[10];
372
373  clock_t timeStart = clock();
374      for(long i=0; i< value; ++i)
375      {
376          try {
377              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  }
```

Hashable

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.wha
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      }
```




```
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
```

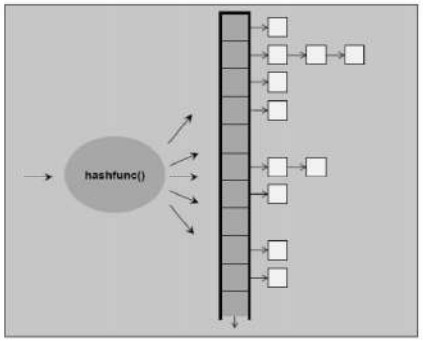
bucker一定比元素多

使用容器 unordered_multiset

```
439
440  string target = get_a_target_string();
441      {
442      timeStart = clock();
443  auto pItem = ::find(c.begin(), c.end(), target);     //比 c.find(...) 慢很多
444      cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
445      if (pItem != c.end())
446          cout << "found, " << *pItem << endl;
447      else
448          cout << "not found! " << endl;
449      }
450
451      {
452      timeStart = clock();
453  auto pItem = c.find(target);         //比 ::find(...) 快很多
454      cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
455      if (pItem != c.end())
456          cout << "found, " << *pItem << endl;
457      else
458          cout << "not found! " << endl;
459      }
460  }
461  }
```

```
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
```

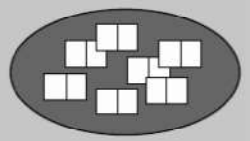
使用容器 unordered_multimap




```
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              snprintf(buf, 10, "%d", rand());
484              //multimap 不可使用 [] 進行 insertion
485              c.insert(pair<long,string>(i,buf));
486          }
487          catch(exception& p) {
488              cout << "i=" << i << " " << p.what() << endl;   //
489              abort();
490          }
491      }
492      cout << "milli-seconds : " << (clock()-timeStart) << endl;  //
493      cout << "unordered_multimap.size()= " << c.size() << endl;
494      cout << "unordered_multimap.max_size()= " << c.max_size() << endl;
495
496
497  long target = get_a_target_long();
498      timeStart = clock();
499  auto pItem = c.find(target);
500      cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
501      if (pItem != c.end())
502          cout << "found, value=" << (*pItem).second << endl;
503      else
504          cout << "not found! " << endl;
505  }
506  }
```

Unordered Map/Multimap:

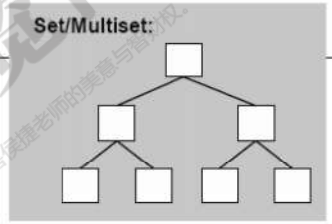


key必须不重复

使用容器 set

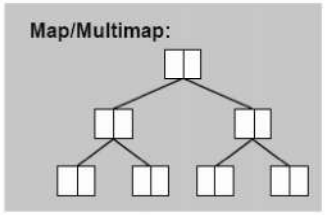
```
641  namespace jj13
642  {
643  void test_set(long& value)
644  {
645      cout << "\ntest_set()..... \n";
646
647  set<string> c;
648  char buf[10];
649
650  clock_t timeStart = clock();
651      for(long i=0; i< value; ++i)
652      {
653          try {
654              snprintf(buf, 10, "%d",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

```
676
677      {
678      timeStart = clock();
679  auto pItem = c.find(target);        //比 ::find(...) 快很多
680      cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
681      if (pItem != c.end())
682          cout << "found, " << *pItem << endl;
683      else
684          cout << "not found! " << endl;
685      }
686  }
687  }
```




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

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

使用容器 map

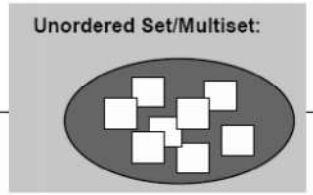



```
D:\handout\C++11-test-DevC++\Test-ST
select: 14
how many elements: 1000000

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

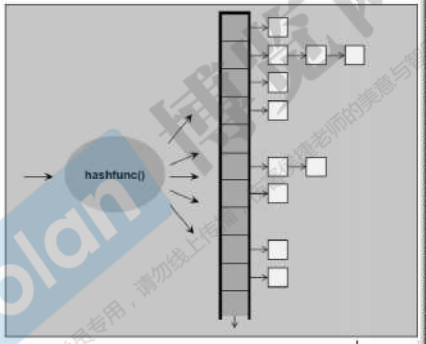
```
696  namespace jj14
697  {
698  void test_map(long& value)
699  {
700      cout << "\ntest_map()..... \n";
701
702  map<long, string> c;
703  char buf[10];
704
705  clock_t timeStart = clock();
706      for(long i=0; i< value; ++i)
707      {
708          try {
709              snprintf(buf, 10,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

value重复 key不重复

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              snprintf(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      }
```



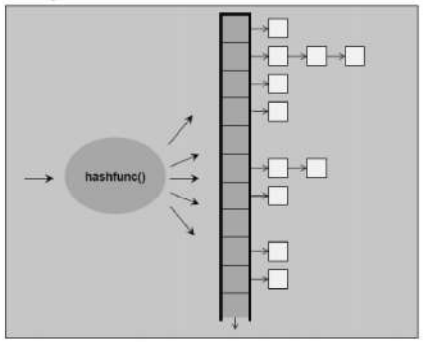

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 15
how many elements: 1000000

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

使用容器 unordered_set

```
770
771   string target = get_a_target_string();
772       {
773       timeStart = clock();
774   auto pItem = ::find(c.begin(), c.end(), target);     //比 c.find(...) 慢很多
775       cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
776       if (pItem != c.end())
777           cout << "found, " << *pItem << endl;
778       else
779           cout << "not found! " << endl;
780       }
781
782       {
783       timeStart = clock();
784   auto pItem = c.find(target);         //比 ::find(...) 快很多
785       cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
786       if (pItem != c.end())
787           cout << "found, " << *pItem << endl;
788       else
789           cout << "not found! " << endl;
790       }
791   }
792   }
```

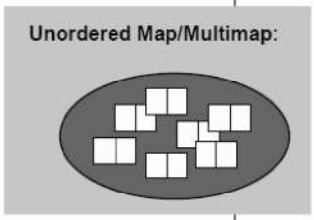
```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 15
how many elements: 1000000

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

使用容器 unordered_map




```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.e
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              snprintf(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  //1 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  }
```



使用容器 **hash_set**
hash_map
hash_multiset
hash_multimap

使用分配器 allocator

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

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

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

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

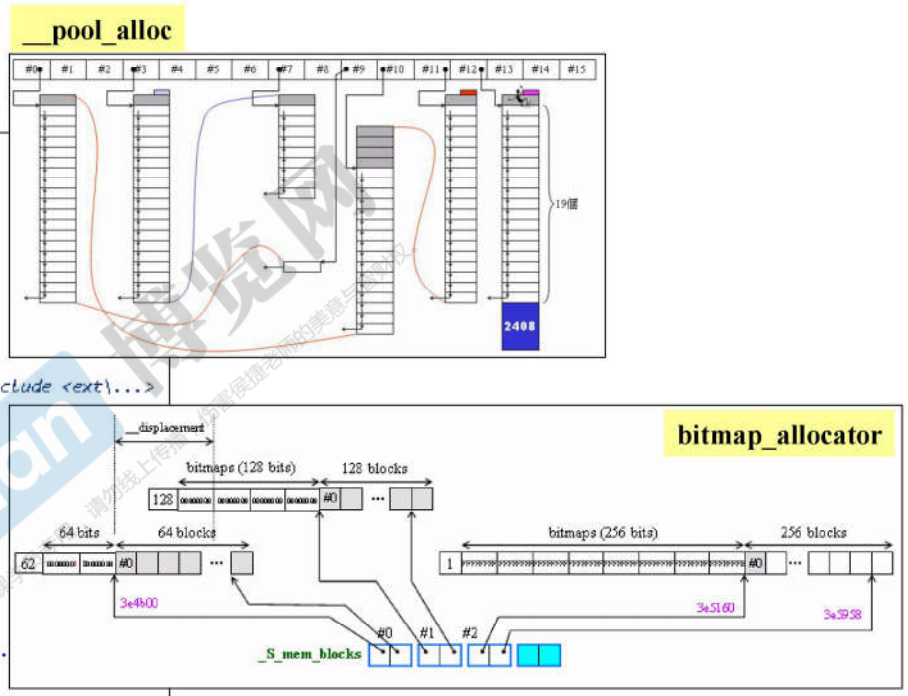
```
template <typename _Key, typename _Tp, typename _Compare = std::less<_Key>,
      typename _Alloc = std::allocator<std::pair<const _Key, _Tp> > >
  class map
```

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

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

使用分配器
allocator

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



使用分配器 allocator

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

```
989
990
991      //test all allocators' allocate() & deallocate();
992      int* p;
993      allocator<int> alloc1;
994      p = alloc1.allocate(1);
995      alloc1.deallocate(p,1);
996
997      __gnu_cxx::malloc_allocator<int> alloc2;
998      p = alloc2.allocate(1);
999      alloc2.deallocate(p,1);
1000
1001     __gnu_cxx::new_allocator<int> alloc3;
1002     p = alloc3.allocate(1);
1003     alloc3.deallocate(p,1);
1004
1005     __gnu_cxx::__pool_alloc<int> alloc4;
1006     p = alloc4.allocate(2);
1007     alloc4.deallocate(p,2);
1008
1009     __gnu_cxx::__mt_alloc<int> alloc5;
1010     p = alloc5.allocate(1);
1011     alloc5.deallocate(p,1);
1012
1013     __gnu_cxx::bitmap_allocator<int> alloc6;
1014     p = alloc6.allocate(3);
1015     alloc6.deallocate(p,3);
1016 }
1017 }
```


不建议直接使用allocate
需要记住分配内存的大小

C++標準庫

體系結構與內核分析

(C++ Standard Library — architecure & sources)

第二講

侯捷

源碼之前
了無秘密

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);
  }
}
```

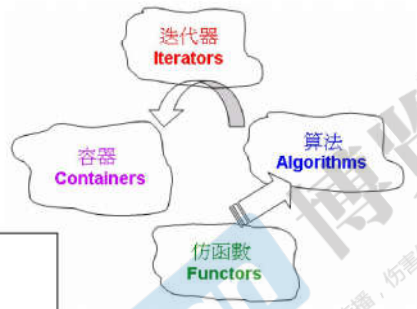
```
template <class RandomAccessIterator, class T, class Size>
void __introsort_loop(RandomAccessIterator first,
                      RandomAccessIterator last,
                      T*,
                      Size depth_limit) {
...
  RandomAccessIterator cut = __unguarded_partition
    (first, last,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

只有 RandomAccessIterator 才能如此操作

标准库的sort需要的iterator list不能满足
所以list不用标准库的sort

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

GP 卻是將 datas 和 methods 分開來



Data Structures (Containers)

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

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

這兩個容器
都提供
RandomAccessIterator

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

Algorithms

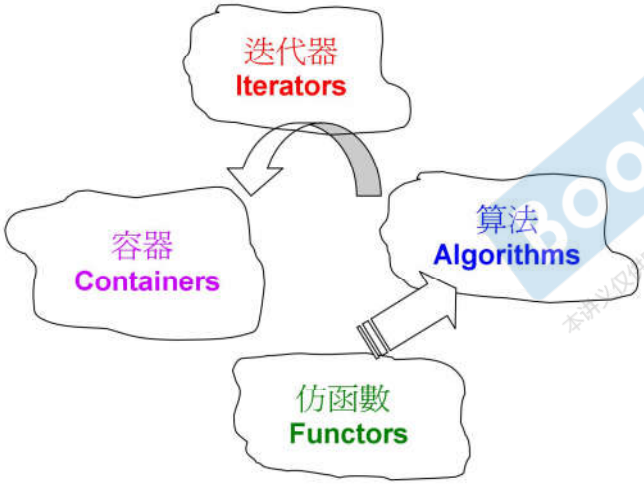
```
template<typename _RandomAccessIterator>
inline void
sort(_RandomAccessIterator __first,
     _RandomAccessIterator __last)
{
  ...
}

template<typename _RandomAccessIterator,
         typename _Compare>
inline void
sort(_RandomAccessIterator __first,
     _RandomAccessIterator __last,
     _Compare __comp)
{
  ...
}
```

— 侯捷 —

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。



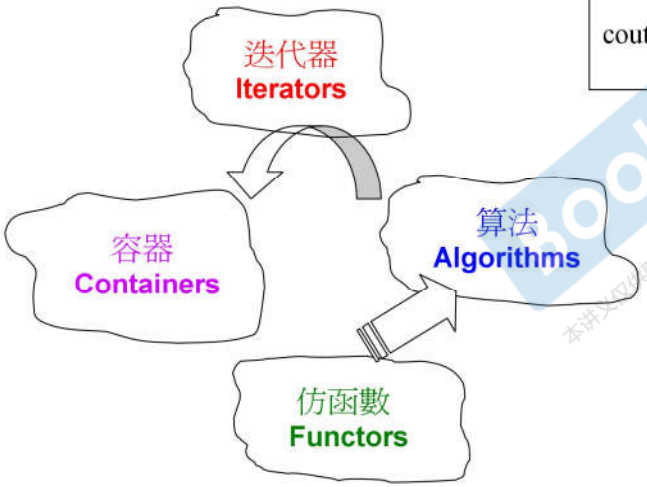
— 侯捷 —

```
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 —

分配器 allocators

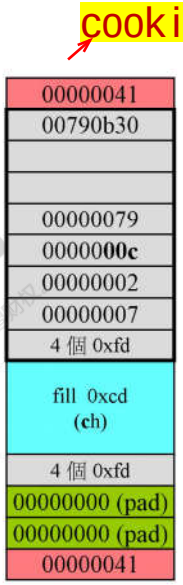
先談 operator new() 和 malloc()

…\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带来额外开销

分配器 allocators

VC6 STL 對 allocator 的使用

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

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

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

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

分配器 allocators

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

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

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

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

其中用到的 _Allocate() 定義如下：

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

```
//分配 512 ints.
int* p = allocator<int>().allocate(512, (int*)0);
allocator<int>().deallocate(p, 512);
```

临时对象

分配器 allocators

BC5 STL 對 allocator 的使用

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

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

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

define _RWSTD_COMPLEX_DEFAULT(a) = a <stdcomp.h>

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

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

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

分配器 allocators

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

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

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

<new.h> of CB5

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

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

分配器 allocators

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

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

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

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

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

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 {
  ...
};
```

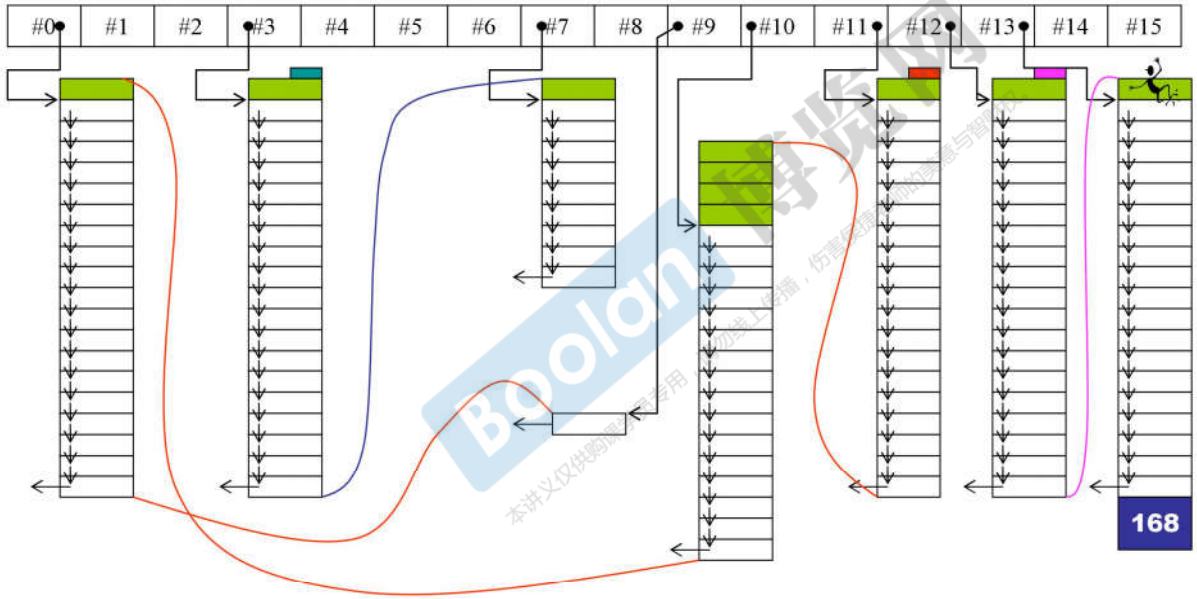
分配器 allocators

GNU C++ 独特

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

不带cookie 容器中无需记录大小 少用很多空间

8 16 32 ..



分配器 allocators

G4.9 STL 對 allocator 的使用

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

分配器 allocators

不用alloc

G4.9 所附的標準庫，其 allocator 實現如下

<bits/allocator.h>

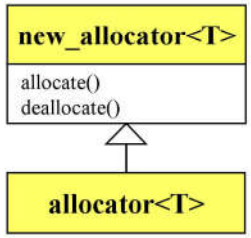
```
template<typename _Tp>
class allocator: public __allocator_base<_Tp>
{
  ...
};
```

<bits/c++allocator.h>

```
# define __allocator_base  __gnu_cxx::new_allocator
```

<bits/new_allocator.h>

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

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



分配器 allocators

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

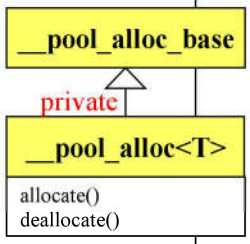
用例：
vector<string, __gnu_cxx::__pool_alloc<string>> vec;

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

```
class __pool_alloc_base
{
protected:
    enum { _S_align = 8 };
    enum { _S_max_bytes = 128 };
    enum { _S_free_list_size = (size_t)_S_max_bytes / (size_t)_S_align };

    union _Obj
    {
      union _Obj* _M_free_list_link;
      char        _M_client_data[1];    // The client sees this.
    };

    static _Obj* volatile  _S_free_list[_S_free_list_size];
    ...
```

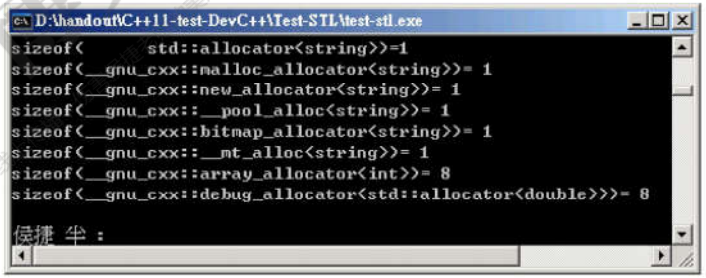
這段有趣的注解仍然沒變.

// Try to make do with what we have. That can't hurt. We
// do not try smaller requests, since that tends to result
// in disaster on multi-process machines.



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
                                                    //==> 因爲它有一個 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>>) << er
```



空类的大小为1字节
一个类中，虚函数本身、成员函数（包括静态与非静态）和静态数据成员都是不占用类对象的存储空间。
对于包含虚函数的类，不管有多少个虚函数，只有一个虚指针,vptr的大小。
普通继承，派生类继承了所有基类的函数与成员，要按照字节对齐来计算大小
虚函数继承，不管是单继承还是多继承，都是继承了基类的vptr。(32位操作系统4字节，64位操作系统 8字节)！
虚继承,继承基类的vptr。

和容器所包含的大小无关，指的是为表示所含内容所需的大小即容器本身的大小

容器和其迭代器, sizeof()

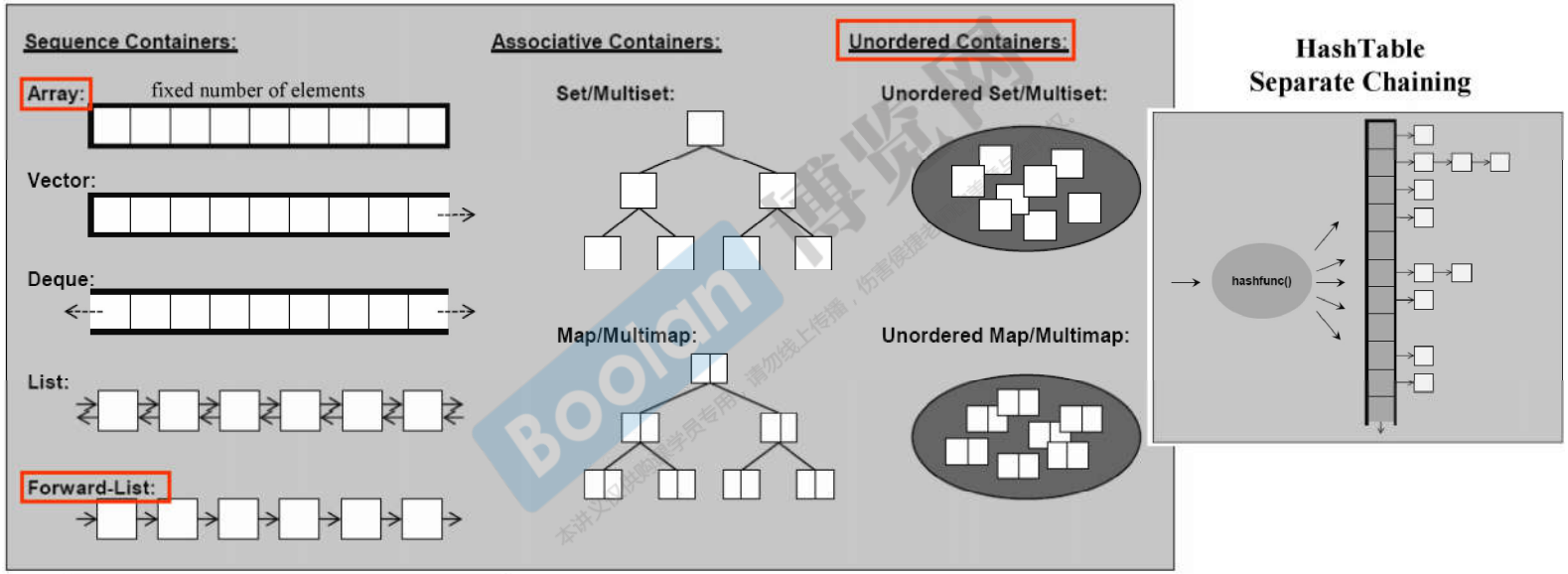
```
cout << "sizeof(array<int,100>)= " << sizeof(array<int,100>) << endl;       //400
cout << "sizeof(vector<int>)= " << sizeof(vector<int>) << endl;             //12
cout << "sizeof(list<int>)= " << sizeof(list<int>) << endl;                 //8
cout << "sizeof(forward_list<int>)= " << sizeof(forward_list<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>) << en
```

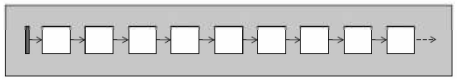
```
cout << "sizeof(array<int,100>::iterator)= " << sizeof(array<int,100>::iterator) << endl;
cout << "sizeof(vector<int>::iterator)= " << sizeof(vector<int>::iterator) << endl;
cout << "sizeof(list<int>::iterator)= " << sizeof(list<int>::iterator) << endl;
cout << "sizeof(forward_list<int>::iterator)= " << sizeof(forward_list<int>::iterator) << enc
cout << "sizeof(deque<int>::iterator)= " << sizeof(deque<int>::iterator) << endl;
//! 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(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
侵捷 半 :
```

G4.9

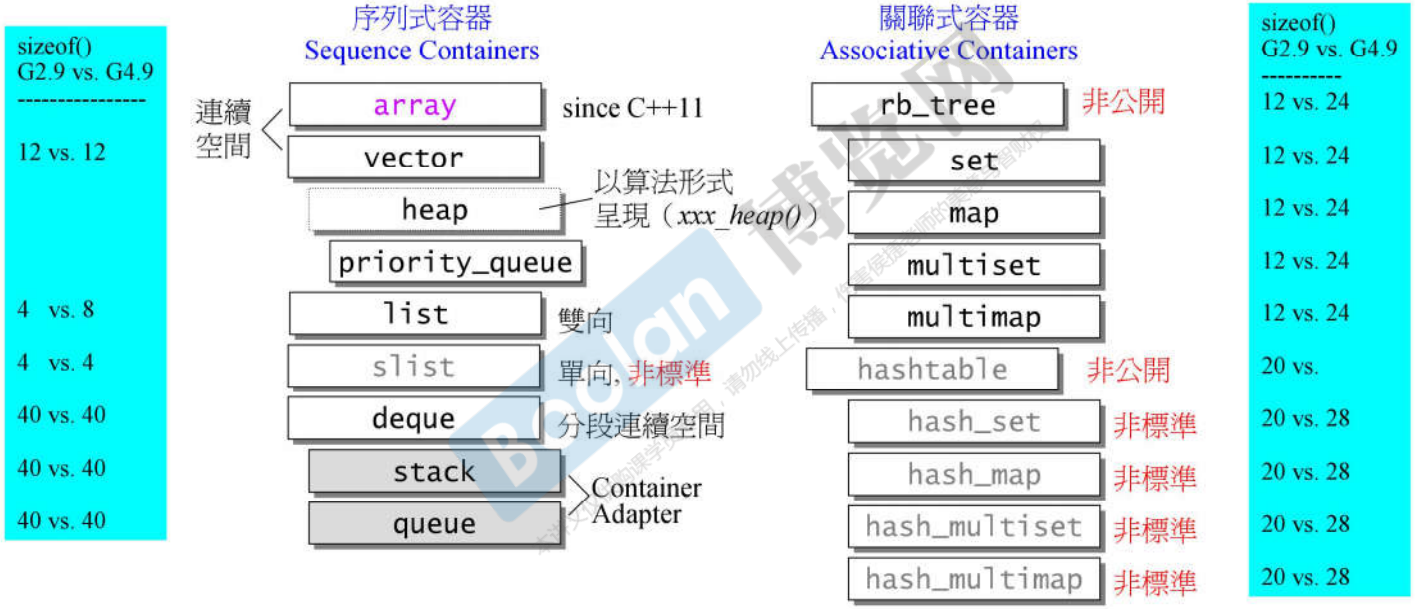
容器 – 結構與分類





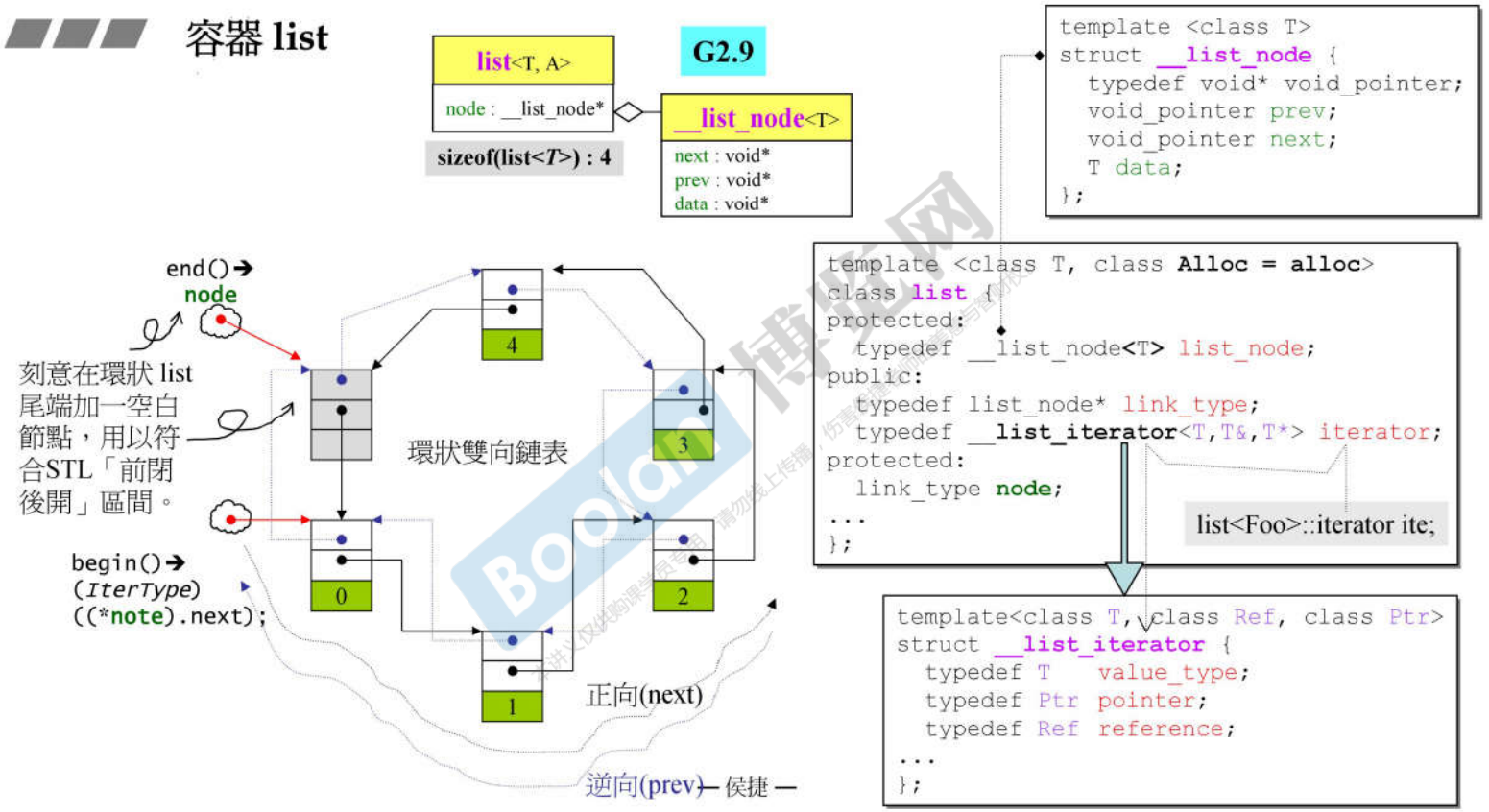
容器, 結構與分類

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）。

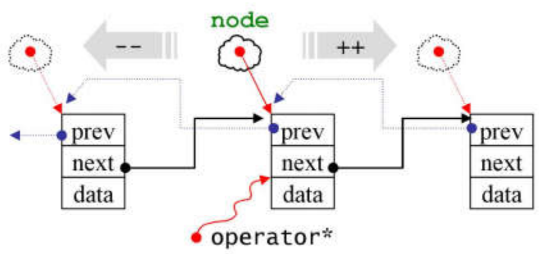


在 C++11 中，slist 名爲 **forward_list**，hash_set, hash_map 名爲 **unordered**_set, **unordered**_map，
hash_multiset, hash_multimap 名爲 **unordered**_multiset, **unordered**_multimap；且新添 **array**.

容器 list


list<T, A>
node : __list_node*
sizeof(list<T>) : 4
G2.9
__list_node<T>
next : void*
prev : void*
data : void*
template <class T>
struct __list_node {
typedef void* void_pointer;
void_pointer prev;
void_pointer next;
T data;
};
template <class T, class Alloc = alloc>
class list {
protected:
typedef __list_node<T> list_node;
public:
typedef list_node* link_type;
typedef __list_iterator<T,T&,T*> iterator;
protected:
link_type node;
...
};
list<Foo>::iterator ite;
template<class T, class Ref, class Ptr>
struct __list_iterator {
typedef T value_type;
typedef Ptr pointer;
typedef Ref reference;
...
};
end()
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。
begin()
(IterType)
((*note).next);
環狀雙向鏈表
0
1
2
3
4
正向(next)
逆向(prev)
侯捷

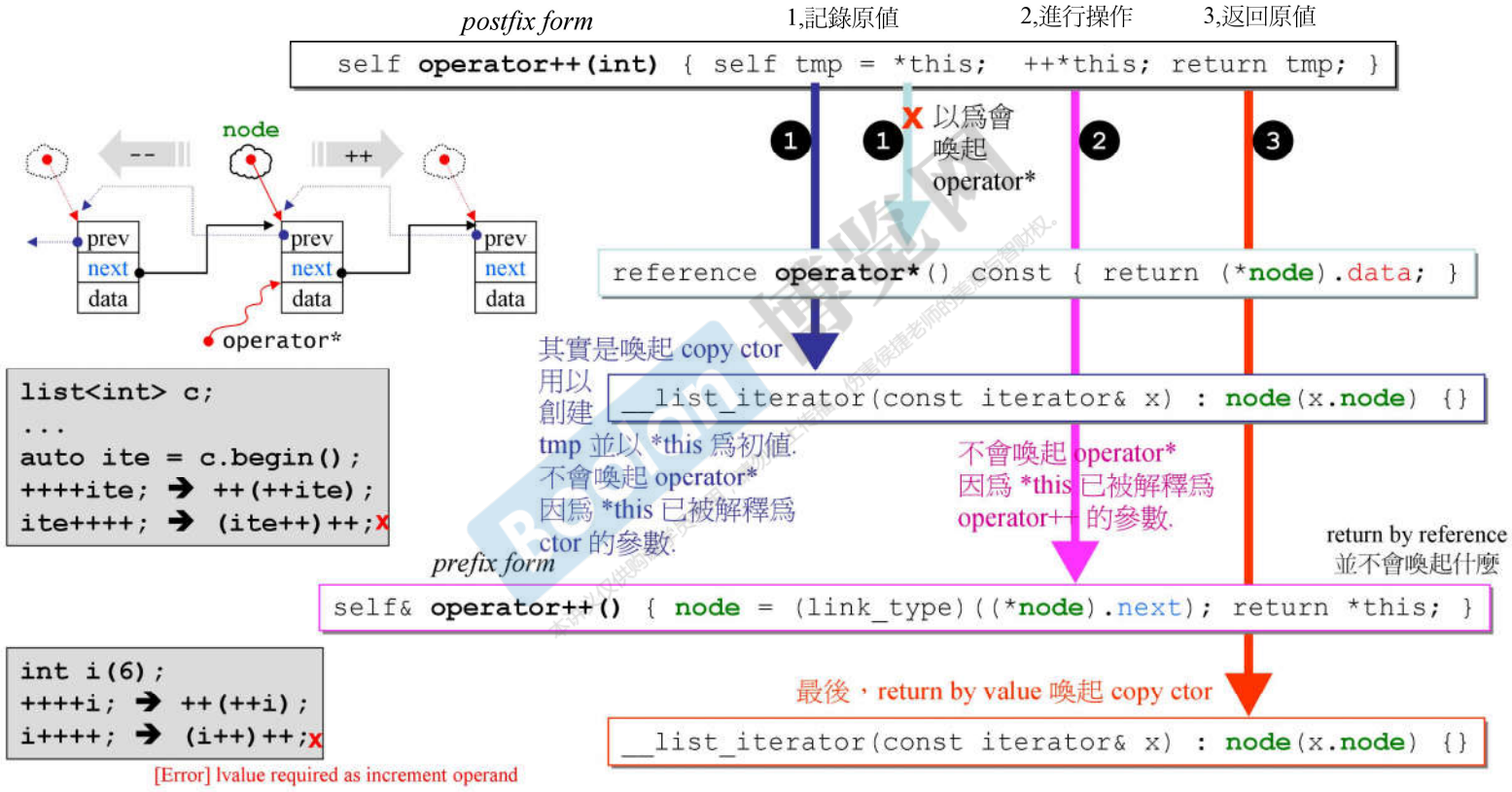
list's iterator




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

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

  link_type node;

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

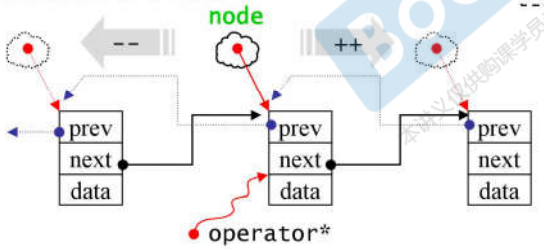
list's iterator



list's iterator

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

```
T&
reference operator*() const
{ return (*node).data; }

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




```
//實例
list<Foo>::iterator ite;
...
*ite; //獲得 a Foo object
ite->method();
 //喚起 Foo::method()
 //相當於 (*ite).method();
 //相當於(&(*ite))->method();
ite->field = 7;
 //賦值 Foo object's field
 //   only if field is public
 //相當於 (*ite).field = 7;
 //相當於(&(*ite))->field=7;
```

list's iterator

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

```
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;
};
```

G2.9

G4.9

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

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

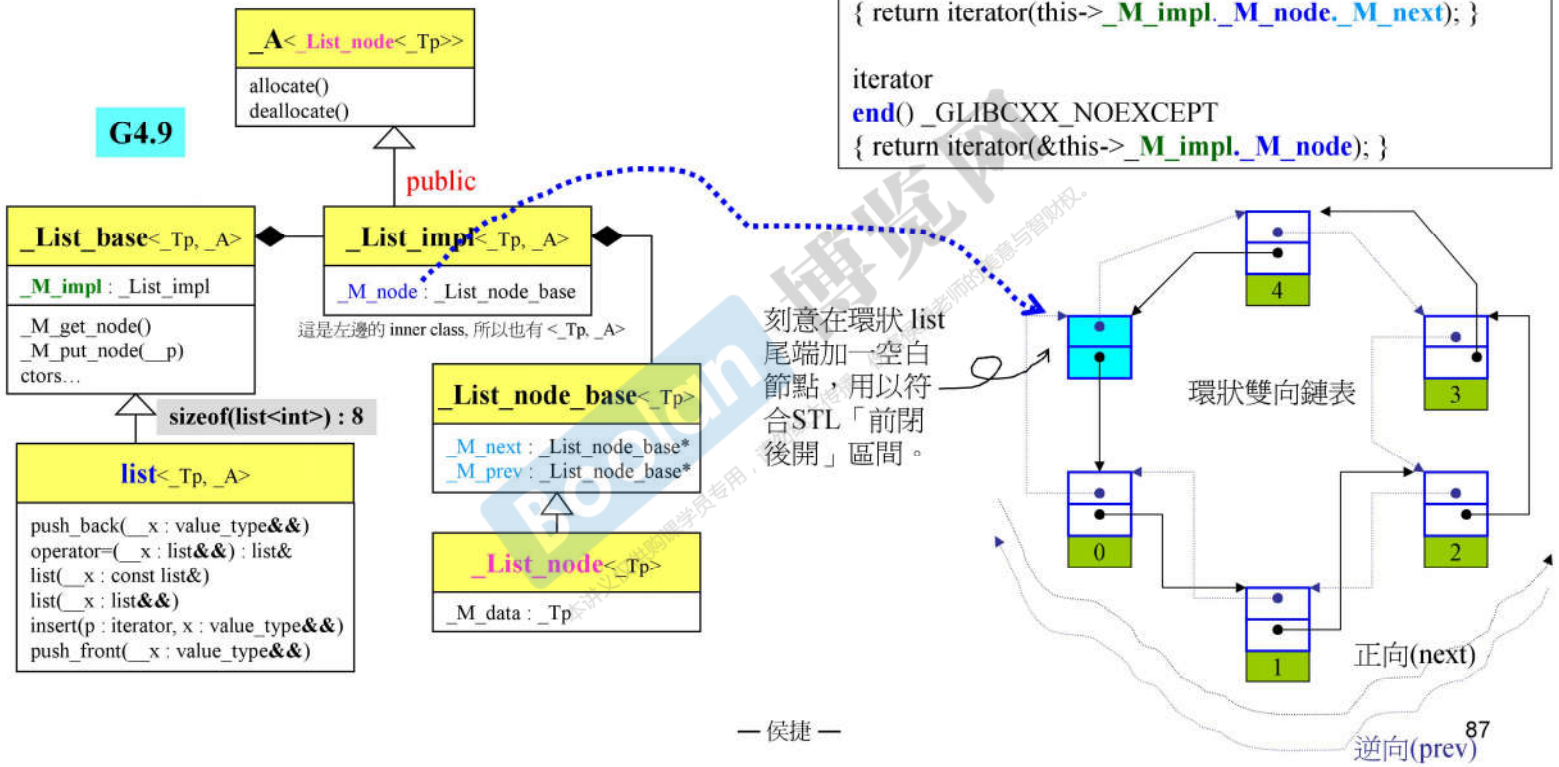
```
struct _List_node_base
{
  _List_node_base* _M_next;
  _List_node_base* _M_prev;
};
```

```
template<typename _Tp>
struct _List_node
  : public _List_node_base
{
  _Tp _M_data;
};
```

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

容器 list


G4.9
_A<_List_node<_Tp>>
allocate()
deallocate()
public
_List_base<_Tp, _A>
_M_impl : _List_impl
_M_get_node()
_M_put_node(__p)
ctors…
_List_impl<_Tp, _A>
_M_node : _List_node_base
這是左邊的 inner class, 所以也有 <_Tp, _A>
sizeof(list<int>) : 8
list<_Tp, _A>
push_back(__x : value_type&&)
operator=(__x : list&&) : list&
list(__x : const list&)
list(__x : list&&)
insert(p : iterator, x : value_type&&)
push_front(__x : value_type&&)
_List_node_base<_Tp>
_M_next : _List_node_base*
_M_prev : _List_node_base*
_List_node<_Tp>
_M_data : _Tp
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。
環狀雙向鏈表
4
3
0
2
1
正向(next)
逆向(prev)

Iterator设计出来以回答算法的提问

Iterator 需要遵循的原則

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

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

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

❶

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

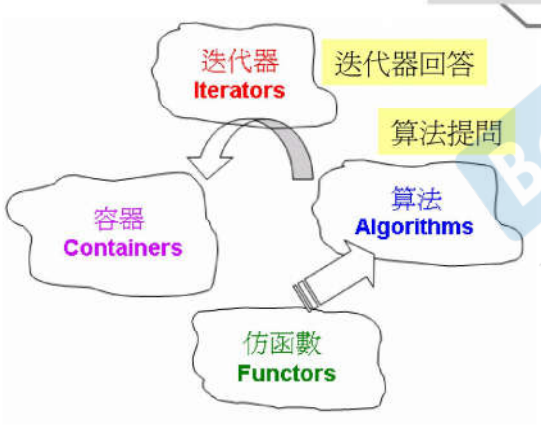
如果是 RAI

iterators 必須有能力回答 algorithms 的提問

❷

❸

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**。

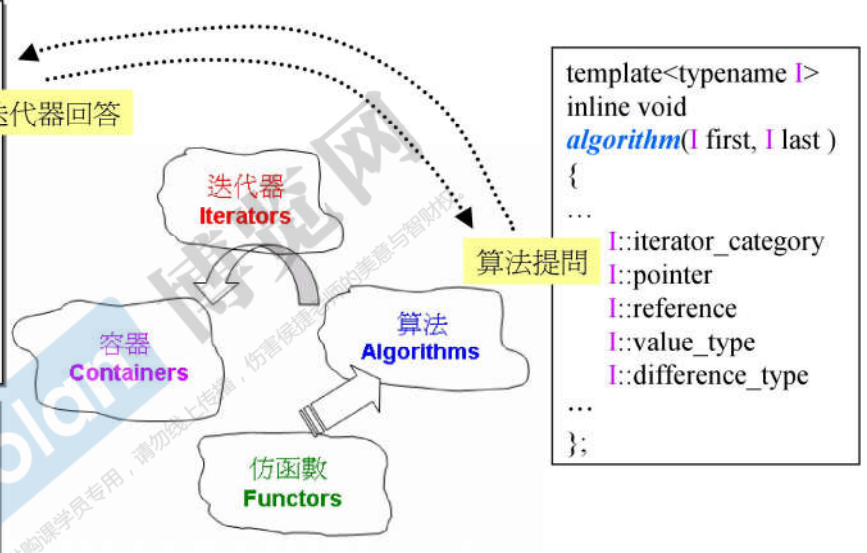


Iterator 必須提供的 5 種 associated types

```
template<class T, class Ref, class Ptr>
struct __list_iterator                          G2.9
{
  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)
...
};
```

距离

```
template<typename _Tp>
struct _List_iterator                           G4.9
{
  typedef std::bidirectional_iterator_tag
                    iterator_category;
  typedef _Tp       value_type;
  typedef _Tp*      pointer;
  typedef _Tp&      reference;
  typedef ptrdiff_t difference_type;
...
};
```

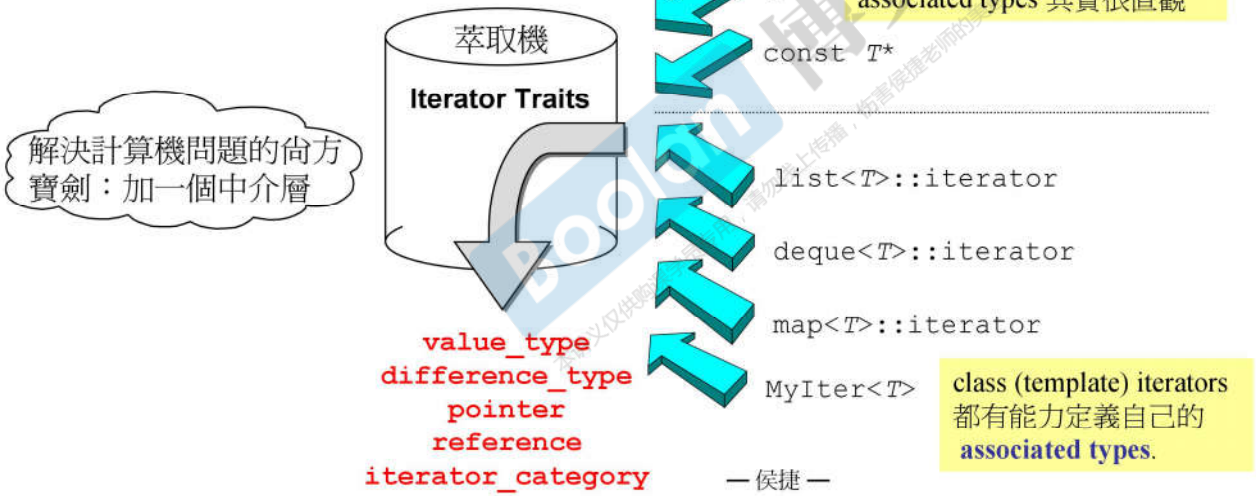


但，如果 **iterator** 並不是個 **class** 呢？
例如 **native pointer**
(它被視爲一種退化的 iterator)。

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。



Non class (template) iterators 亦即 native pointer, 無法定義 associated types. 但它的 associated types 其實很直觀

class (template) iterators 都有能力定義自己的 **associated types**.

— 侯捷 —

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;
   };

2  //兩個 partial specialization：
   template <class T>
   struct iterator_traits<T*> {
     typedef T value_type;
   };
3  template <class T>
   struct iterator_traits<const T*> {
     typedef T value_type;    //注意是 T 而不是 const T
   };
```

如果 I 是 class iterator 進入這裡

如果 I 是 pointer to T 進入這裡

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;
}
```

—侯捷—

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>

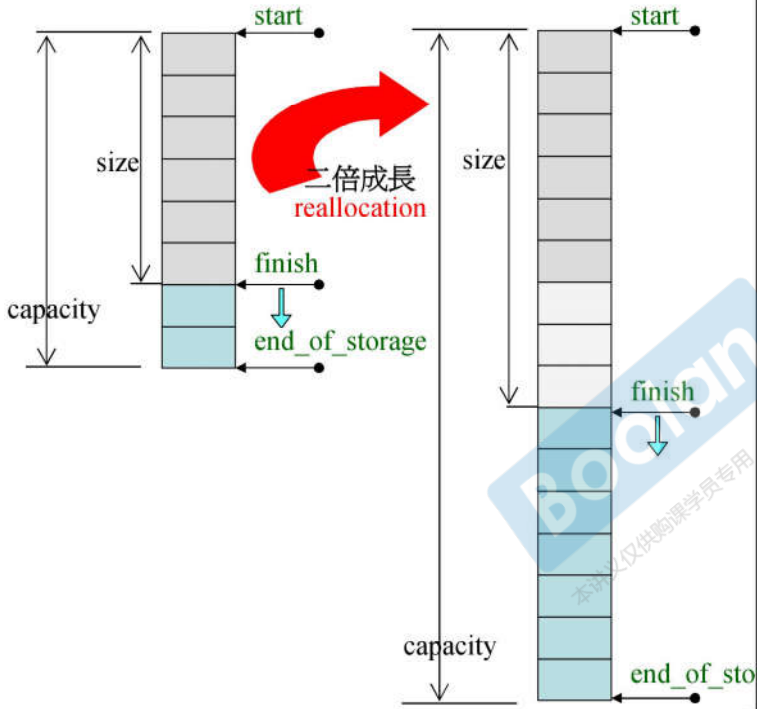
```
template <class I>
struct iterator_traits {
  typedef typename I::iterator_category  iterator_category;
  typedef typename I::value_type         value_type;
  typedef typename I::difference_type    difference_type;
  typedef typename I::pointer            pointer;
  typedef typename I::reference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

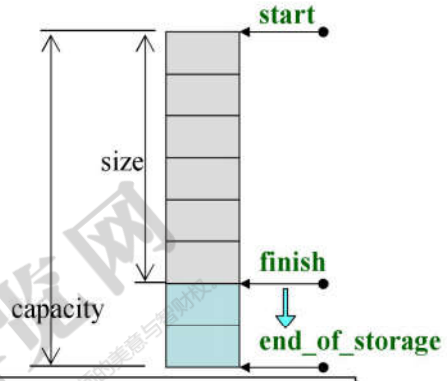
容器 vector




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

容器 vector

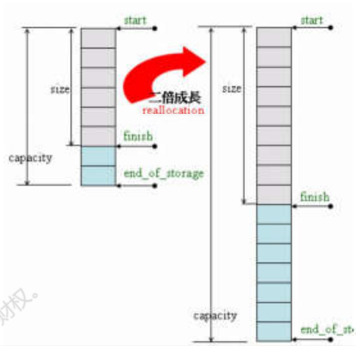
```
void push_back(const T& x) {
    if (finish != end_of_storage) { //尚有備用空間
      construct(finish, x);          //全局函數
      ++finish;                      //調整水位高度
    }
    else  //已無備用空間
      insert_aux(end(), x);
}
```




```
template <class T, class Alloc>
void vector<T,Alloc>::insert_aux(iterator position, const T& x) {
  if (finish != end_of_storage) {  //尚有備用空間
    //在備用空間起始處建構一個元素，並以vector最後一個元素值爲其初值。
    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
    續下頁
  }
}
```

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：如果原大小爲0，則分配 1（個元素大小）；
  //如果原大小不爲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;
}
```

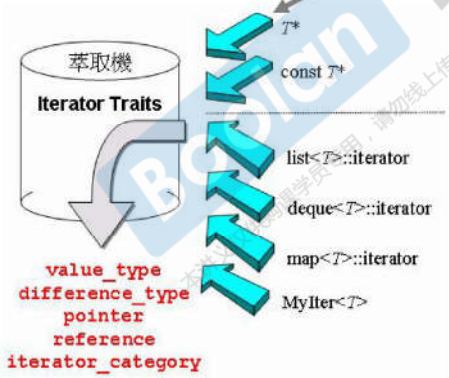



```
try {
  //將原vector的內容拷貝到新vector
  new_finish = uninitialized_copy(start,
                       position, new_start);
  construct(new_finish, x); //爲新元素設初値x
  ++new_finish; //調整水位。
  //拷貝安插點後的原內容（因它也可能被insert(p,x)呼叫）
  new_finish = uninitialized_copy(position,
                           finish, new_finish);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

vector's iterator

```
template <class T, class Alloc = alloc>
class vector {                          G2.9
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
```

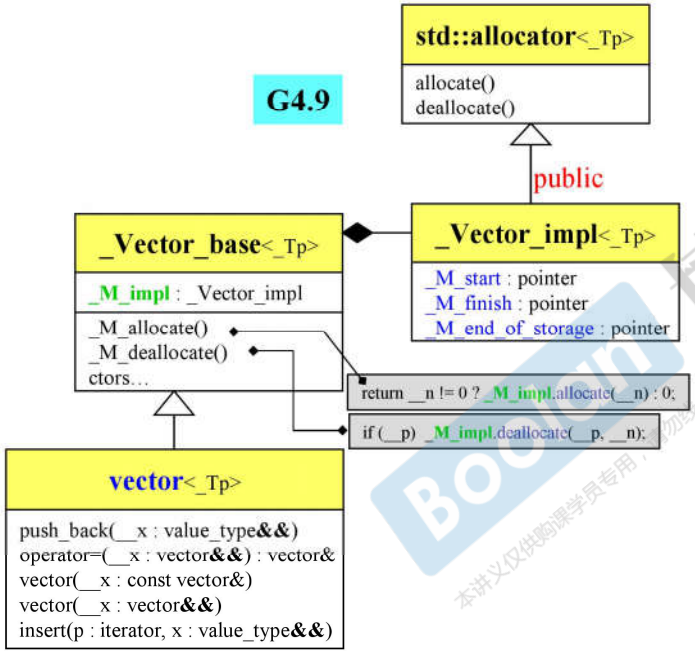



```
template <class I>
struct iterator_traits {
  typedef typename I::iterator_category  iterator_category;
  typedef typename I::value_type         value_type;
  typedef typename I::difference_type    difference_type;
  typedef typename I::pointer            pointer;
  typedef typename I::reference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

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

— 侯捷 —

容器 vector



sizeof(vector<*T*>) : 12

vector's iterator

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 = 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;
  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()) { }
```

vector<T>::iterator

_M_current : _Base::pointer

vector<T>::iterator

_M_current : _Tp*

```
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 pointe
```

```
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

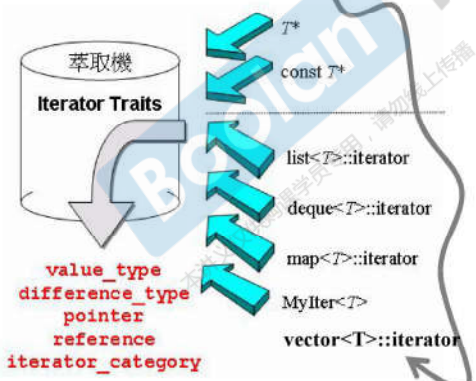
本頁追蹤 **vector::iterator**，發現它就是 _Tp* 外覆一個 iterator adapter，使能支持 5 associated types

vector<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_categor
    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;
...
```

```
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
```



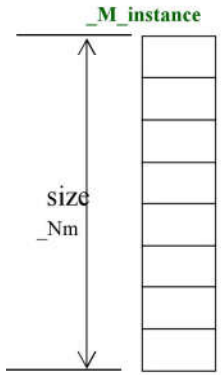
— 侯捷 —

亂七八糟，捨近求遠，何必如此 !!

容器 array

TR1

array<_Tp, _Nm>
_M_instance : _Tp[_Nm]

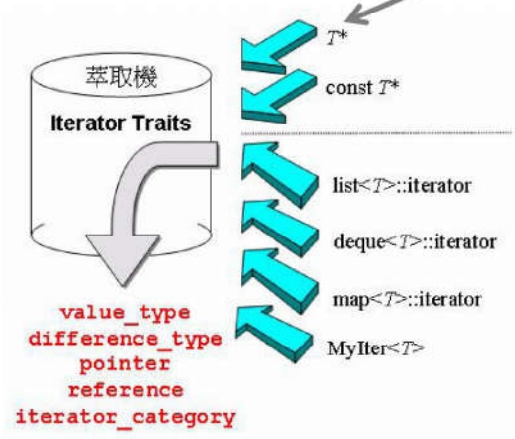



```
template<typename _Tp, std::size_t _Nm>
  struct array
  {
    typedef _Tp          value_type;
    typedef _Tp*         pointer;
    typedef value_type*  iterator;

    // 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

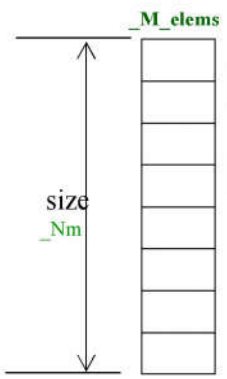
```
array<int, 10> myArray;
auto ite = myArray.begin();
   // array<int, 10>::iterator ite = …
ite += 3;
cout << *ite;
```

其 iterator 是 native pointer,
(G2.9 vector 也是如此)



容器 array

G4.9



typename _AT_Type::_Type _M_elems;

⇩

__array_traits<_Tp, _Nm>:: _Type _M_elems

⇩

_Tp _M_elems [_Nm];

```
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]); }
  };
```

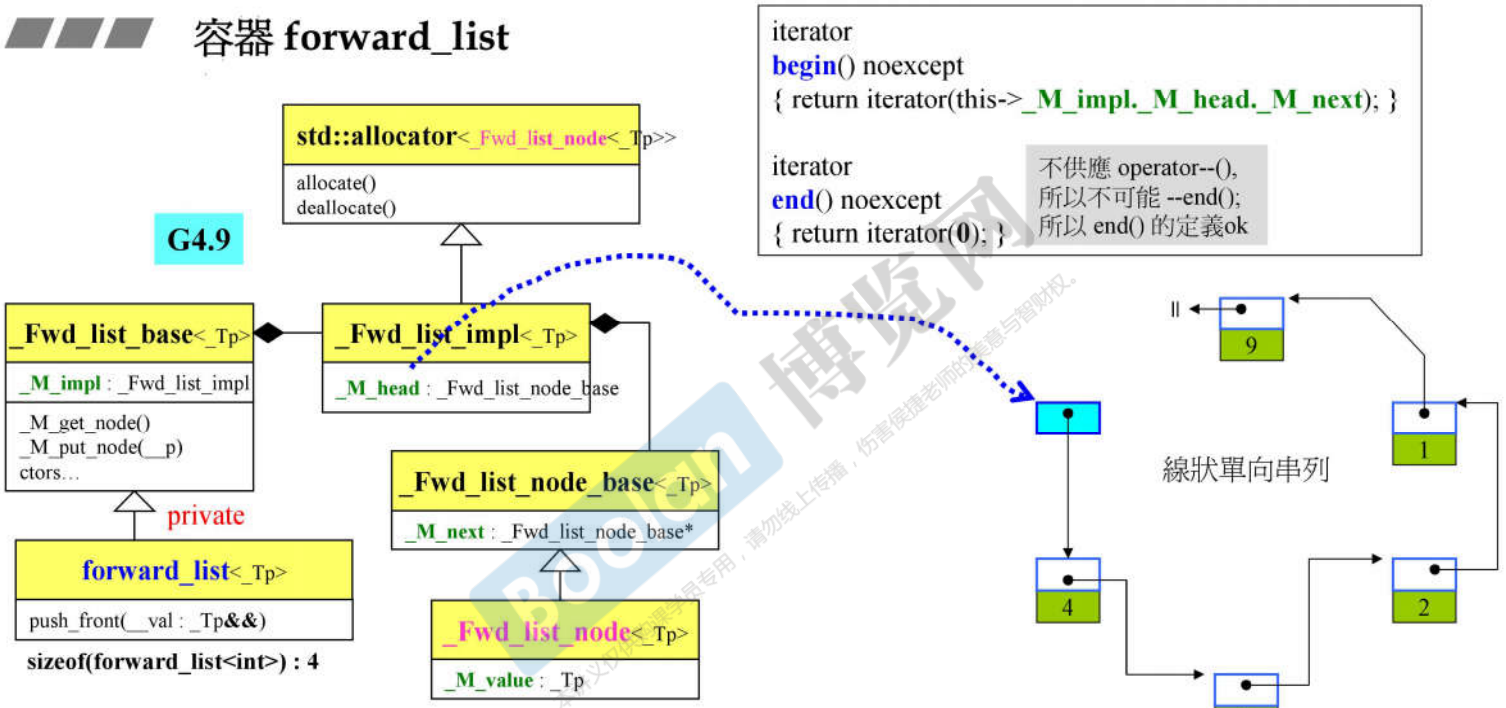
```
template<typename _Tp, std::size_t _Nm>
  struct array
  {
    typedef _Tp             value_type;
    typedef value_type*     pointer;
    typedef value_type&     reference;
    typedef value_type*     iterator;
    typedef std::size_t     size_type;

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

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

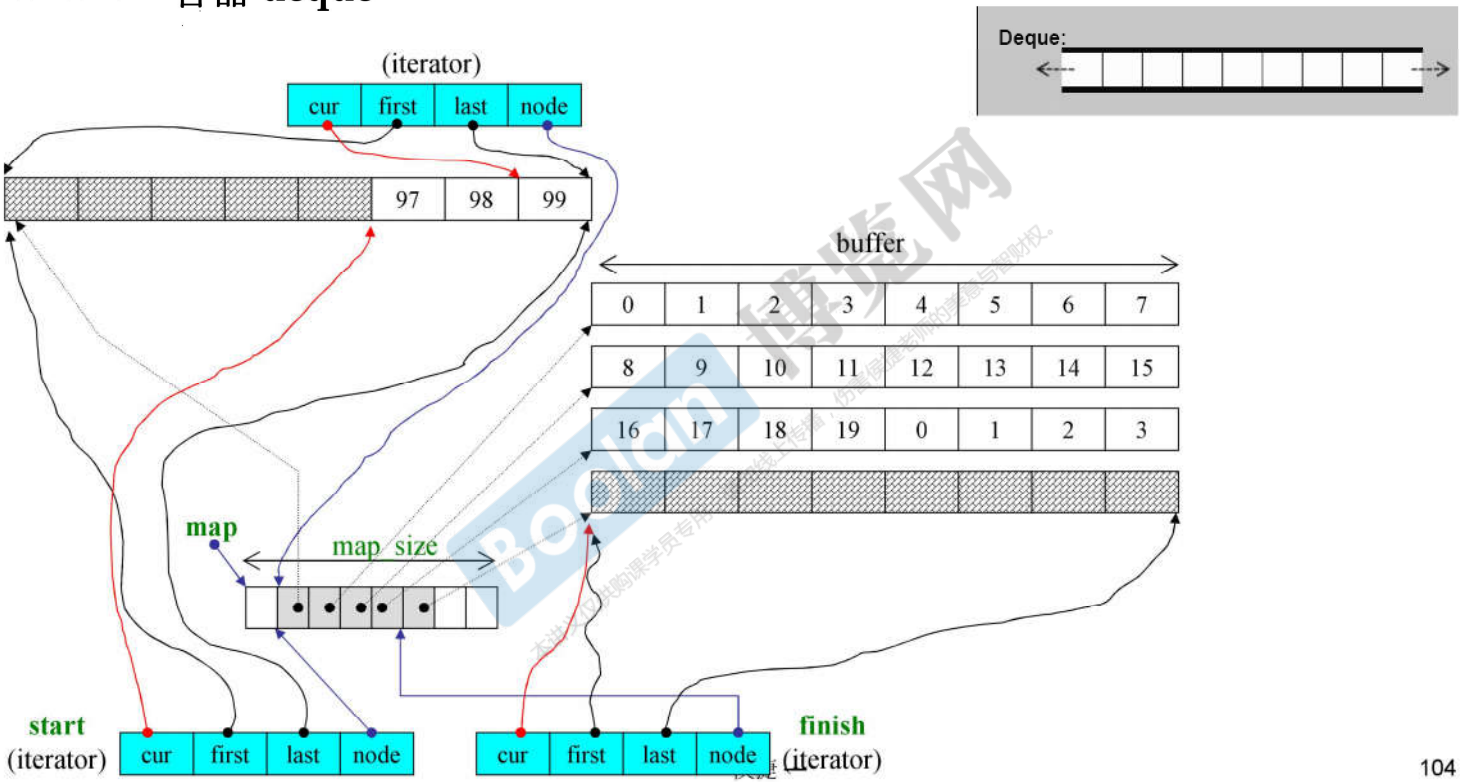
没有邊檢

有邊檢

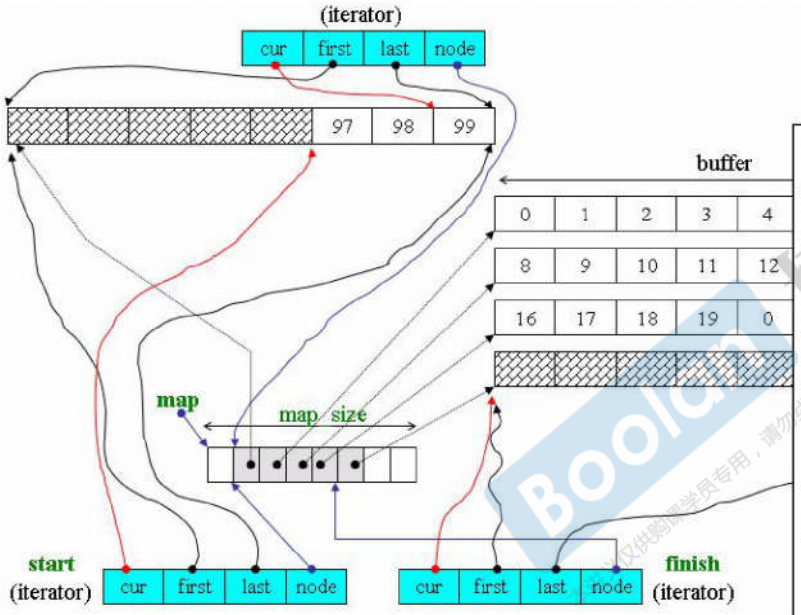
容器 forward_list



容器 deque


Deque:
(iterator)
cur
first
last
node
97
98
99
buffer
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
0
1
2
3
map
map_size
start
(iterator)
cur
first
last
node
cur
first
last
node
finish
(iterator)

容器 deque



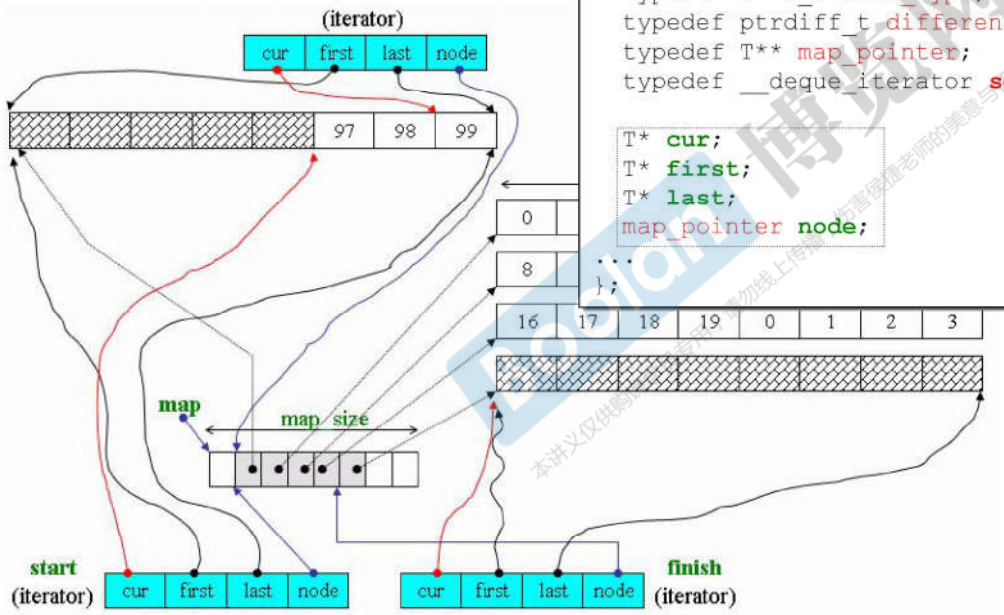

```
// 如果 n 不爲 0，傳回 n，表示 buffer size 由使用者自定。
// 如果 n 爲 0，表示 buffer size 使用預設值，那麼
//    如果 sz（sizeof(value_type)）小於 512，傳回 512/sz，
//    如果 sz 不小於 512，傳回 1。
inline size_t __deque_buf_size(size_t n, size_t sz)
{
  return n != 0 ? n : (sz < 512 ? size_t(512 / sz) : size_t(1));
}
```

```
template <class T, class Alloc=alloc,
          size_t BufSiz=0>
class deque {
public:
  typedef T value_type;
  typedef __deque_iterator<T,T&,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

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;
  ...
};
```

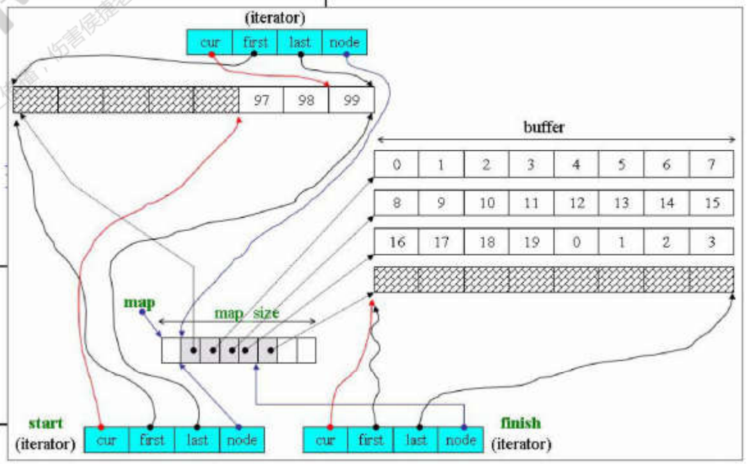


deque<T>::insert()

```
// 在position 處安插一個元素，其值爲 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);       //
    }
  }
```

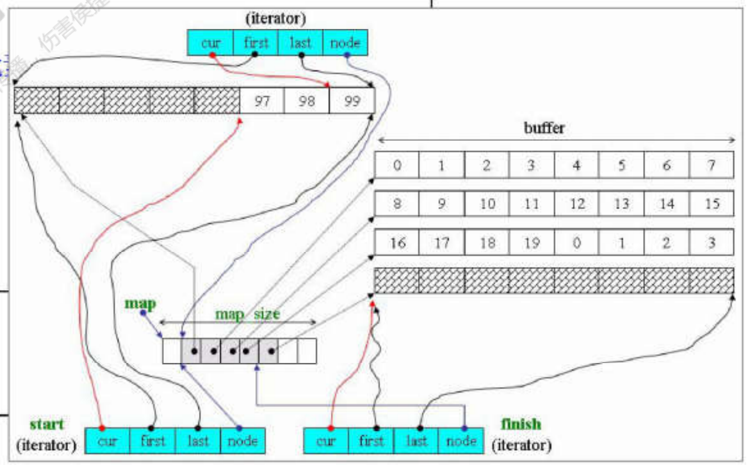
下頁

— 侯捷 —



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
    ...
    copy_backward(pos, back2, back1); //元素搬移
  }
  *pos = x_copy;    // 在安插點上設定新值
  return pos;
}
```



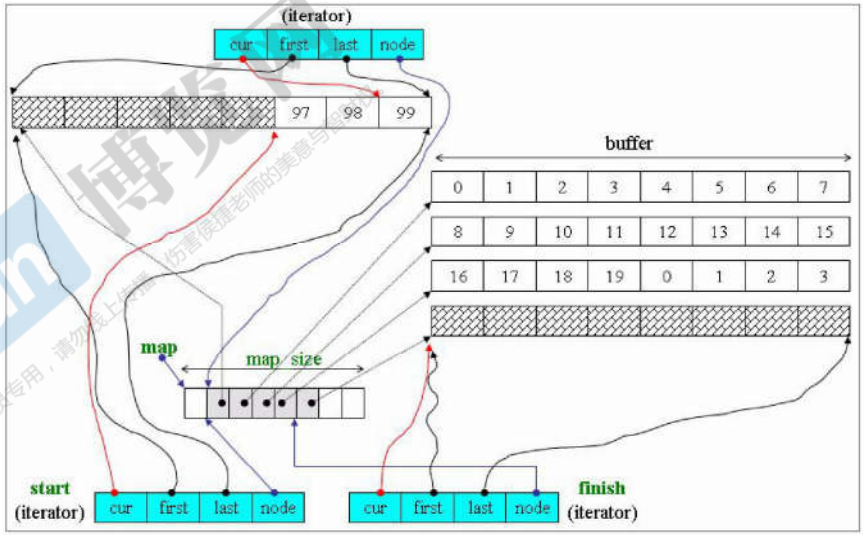
— 侯捷 —

deque 如何模擬連續空間

全都是 deque iterators 的功勞

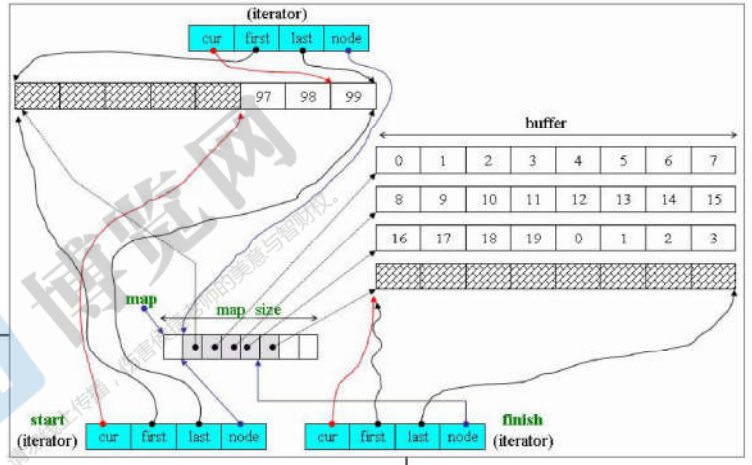
```
reference operator[](size_type n)
{
    return start[difference_type(n)];
}
```

```
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);
}
```

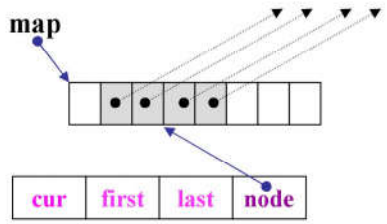
首尾 buffers 之間的 buffers 數量

末尾 (當前) buffer 的元素量

起始 buffer 的元素量

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());
}
```

deque 如何模擬連續空間

全都是 deque iterators 的功勞

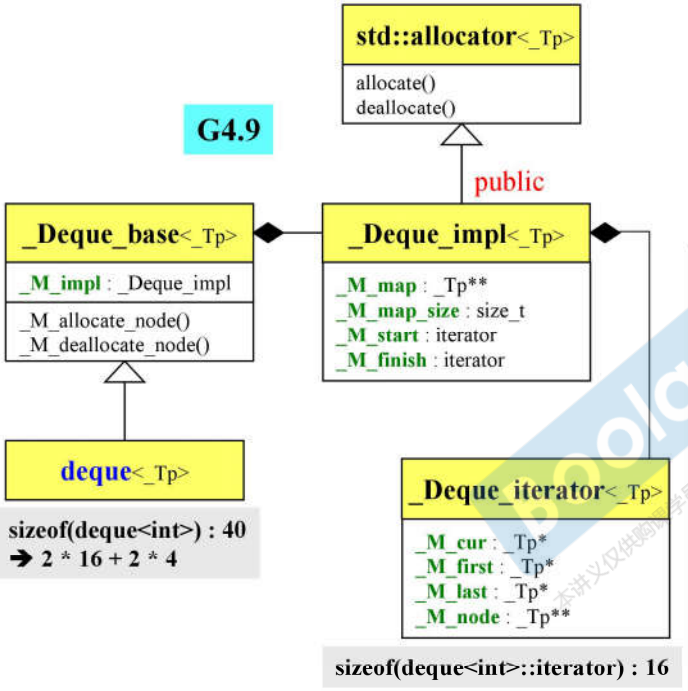
```
self& operator+=(difference_type n) {
    difference_type offset = n + (cur - first);
    if (offset >= 0 && offset < difference_type(buffer_size()))
      //目標位置在同一緩衝區內
      cur += n;
    else {
      //目標位置不在同一緩衝區內
      difference_type node_offset =
        offset > 0 ? offset / difference_type(buffer_size())
                   : -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); }
```

deque




```
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

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));
}
```

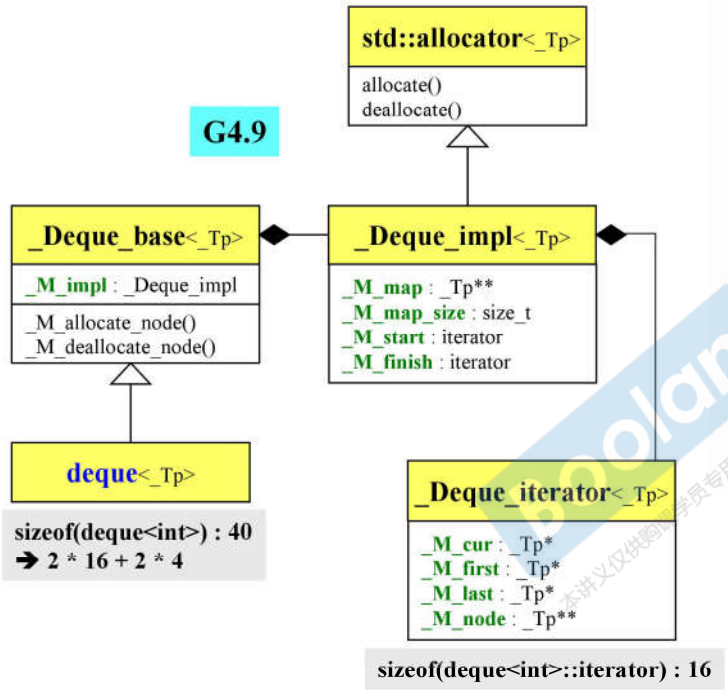
```
template<typename _Tp,
         typename _Alloc>
class _Deque_base
{
  protected:
    struct _Deque_impl
      : public _Tp_alloc_type {
        _Tp** _M_map;
        size_t _M_map_size;
        iterator _M_start;
        iterator _M_finish;
        …
    };
  …
};
```

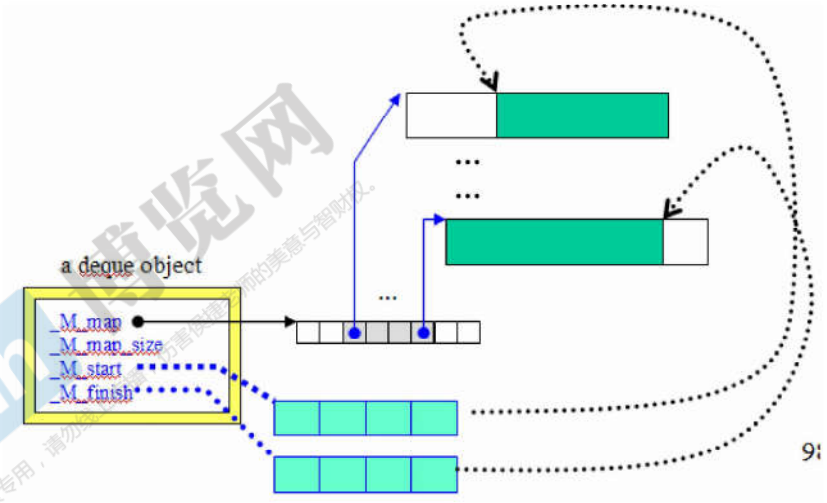
如果 node 是個 int, 則 buffer 內含的 node 數量是 512 / sizeof(int) = 128

如果 node 是個 double, 則 buffer 內含的 node 數量是 512 / sizeof(double) = 64

如果 node 是個 class/struct/array 而其大小 >= 512, 則 buffer 內含的 node 數量是 1

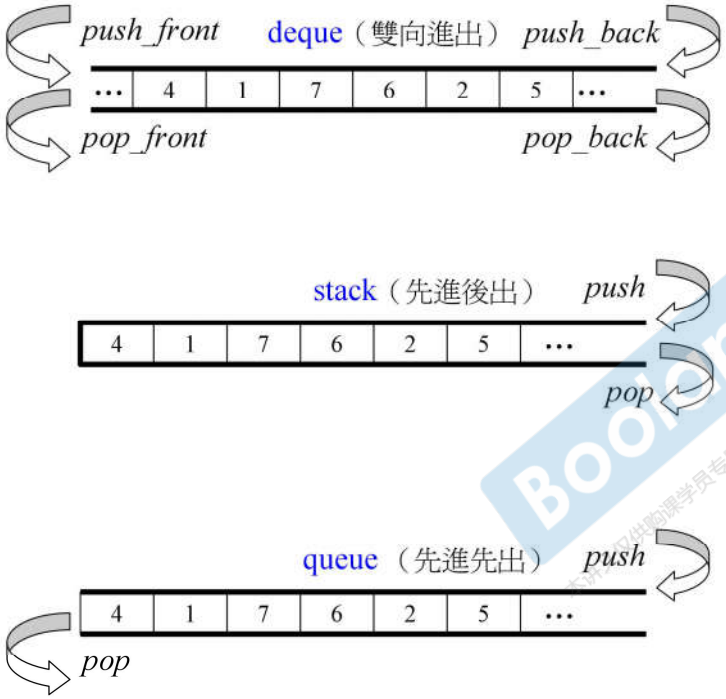
deque





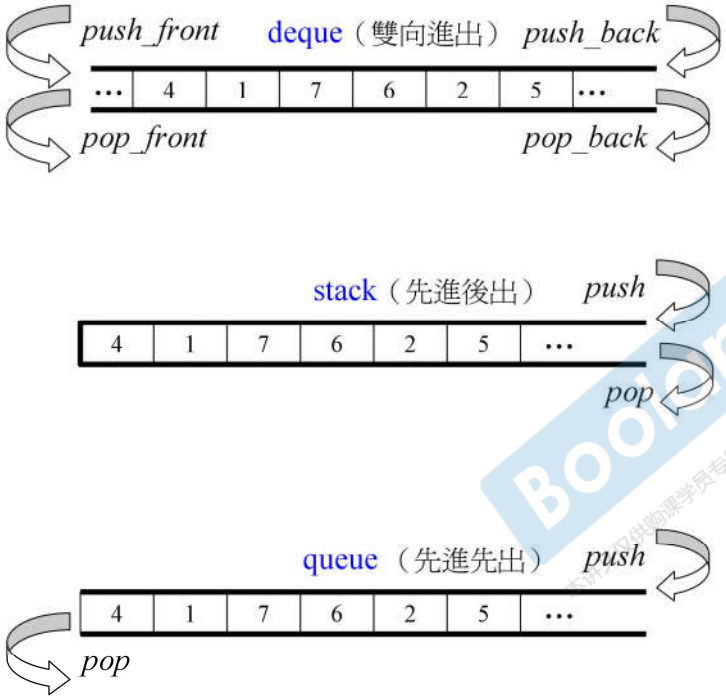
中段

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(); }
};
```

queue 和 stack, 關於其 iterator 和底層結構

stack 或 queue 都不允許遍歷，
也不提供 iterator。

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

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

stack 和 queue 都可選擇 list 或 deque
做爲底層結構。

```
stack<string, list<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;
```

```
queue<string, list<string>> c;
    for(long i=0; i< 10; ++i) {
            snprintf(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

stack 可選擇 vector 做爲底層結構

```
stack<string, vector<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;
```

queue 不可選擇 vector 做爲底層結構

```
queue<string, vector<string>> c;
    for(long i=0; i< 10; ++i) {
            snprintf(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;
```

不会全面检查

queue 和 stack, 關於其 iterator 和底層結構

stack 和 queue 都不可選擇 set 或 map 做爲底層結構

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

編譯器 無異議

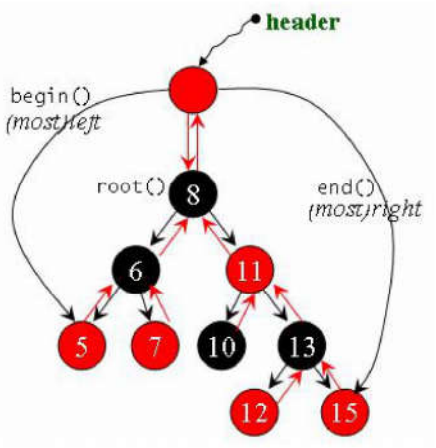
[Error] 'class std::set<std::basic_string<char> >' has no member named 'push_back'
[Error] 'class std::set<std::basic_string<char> >' has no member named 'back'
[Error] 'class std::set<std::basic_string<char> >' has no member named 'pop_back'

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

容器 rb_tree

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

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



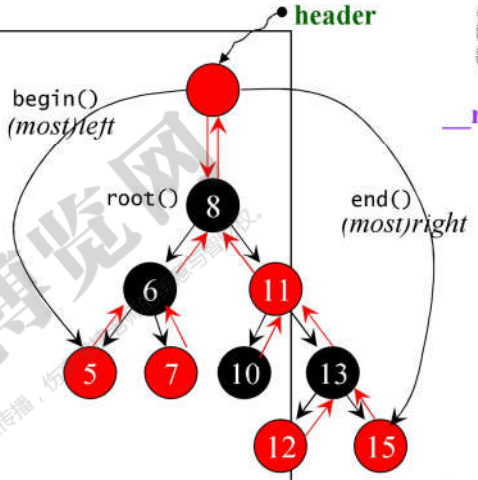
我們不應使用 rb_tree 的 iterators 改變元素值（因爲元素有其嚴謹排列規則）。編程層面 (programming leve) 並未阻絕此事。如此設計是正確的，因爲 rb_tree 即將爲 set 和 map 服務（做爲其底部支持），而 map 允許元素的 data 被改變，只有元素的 key 才是不可被改變的。

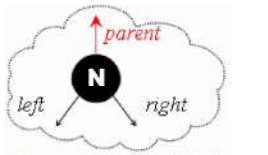
rb_tree 提供兩種 insertion 操作：**insert_unique()** 和 **insert_equal()**。
前者表示節點的 key 一定在整個 tree 中獨一無二，否則安插失敗；
後者表示節點的 key 可重複。

容器 rb_tree

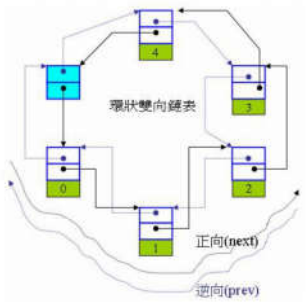
```
template <class Key,
          class Value,
          class KeyOfValue,
          class Compare,
          class Alloc = alloc>
class rb_tree {
protected:
  typedef __rb_tree_node<Value> rb_tree_node;
  ...
public:
  typedef rb_tree_node* link_type;
  ...
protected:
  // RB-tree 只以三筆資料表現它自己
  size_type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

大小：9 ➔ 12



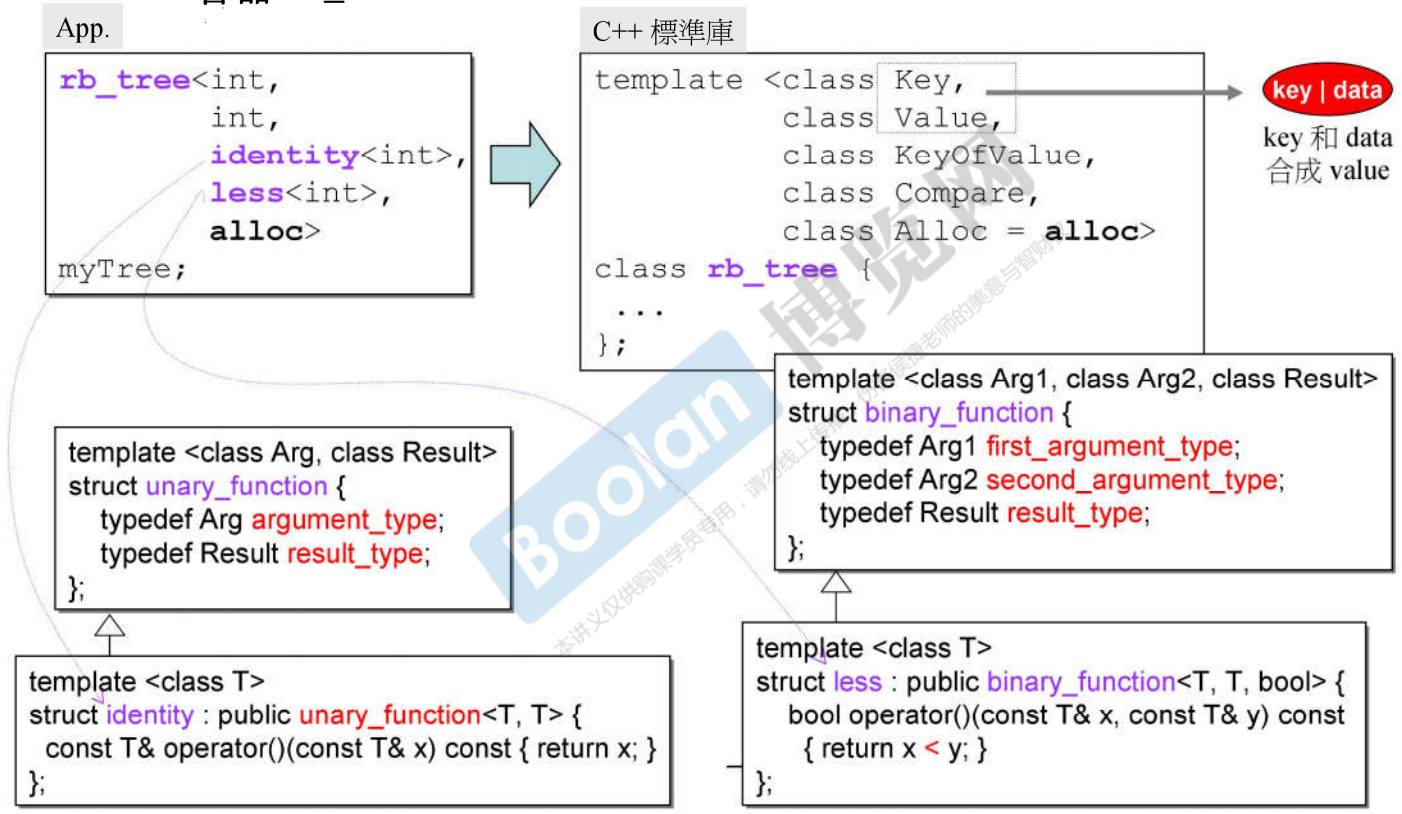


__rb_tree_node<Value>



仿函数 理论不占大小
实际占用1字节

容器 rb_tree


App.
rb_tree<int,
int,
identity<int>,
less<int>,
alloc>
myTree;
C++ 標準庫
template <class Key,
class Value,
class KeyOfValue,
class Compare,
class Alloc = alloc>
class rb_tree {
...
};
key | data
key 和 data
合成 value
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;
};
template <class T>
struct identity : public unary_function<T, T> {
const T& operator()(const T& x) const { return x; }
};
template <class T>
struct less : public binary_function<T, T, bool> {
bool operator()(const T& x, const T& y) const
{ return x < y; }
};

容器 rb_tree, 用例

G2.9

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

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

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

容器 _Rb_tree, 用例

G4.9

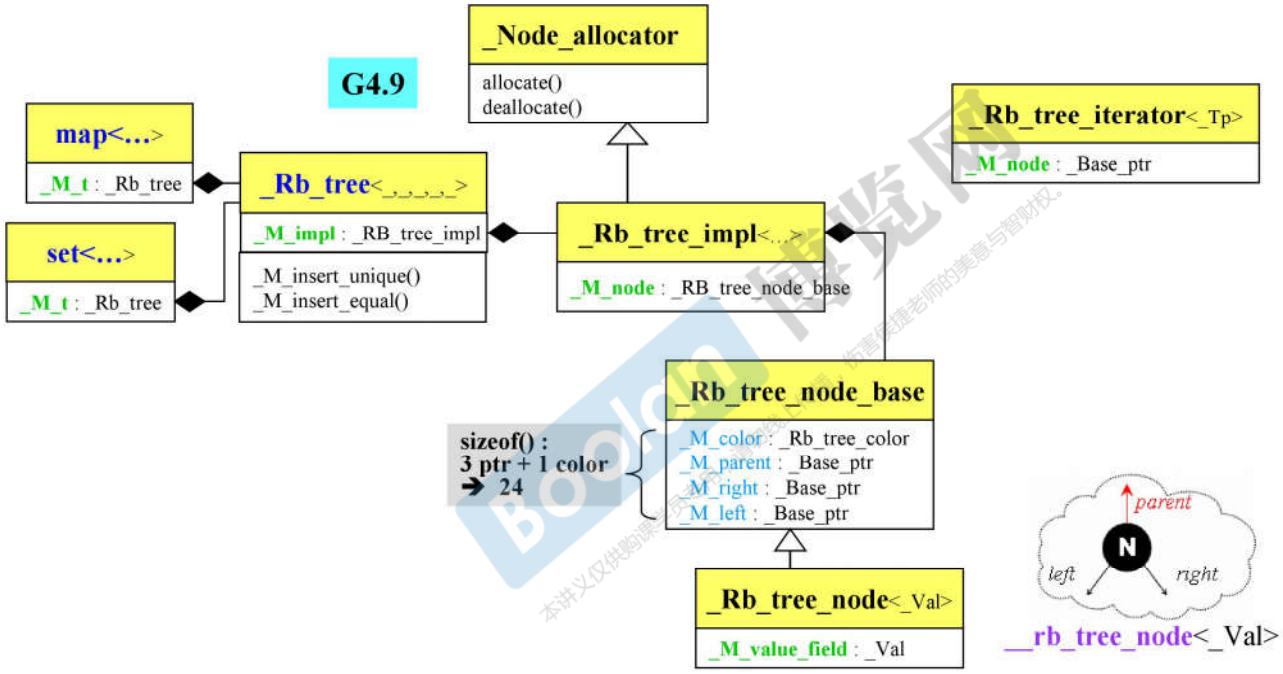
```
//測試 rb_tree
_Rb_tree<int, int, _Identity<int>, less<int>> itree;
cout << itree.empty() << endl;  //1
cout << itree.size() << endl;   //0

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

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

容器 _Rb_tree, 用例

G4.9

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

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

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

容器 _Rb_tree


G4.9
_Node_allocator
allocate()
deallocate()
map<...>
_M_t : _Rb_tree
_Rb_tree<_,_,_,_,_>
_M_impl : _RB_tree_impl
_M_insert_unique()
_M_insert_equal()
set<...>
_M_t : _Rb_tree
_Rb_tree_impl<...>
_M_node : _RB_tree_node_base
_Rb_tree_iterator<_Tp>
_M_node : _Base_ptr
_Rb_tree_node_base
sizeof() :
3 ptr + 1 color
➔ 24
_M_color : _Rb_tree_color
_M_parent : _Base_ptr
_M_right : _Base_ptr
_M_left : _Base_ptr
_Rb_tree_node<_Val>
_M_value_field : _Val
parent
N
left
right
__rb_tree_node<_Val>

容器 set, multiset

key 和 data
合成 value

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

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（因爲 key 有其嚴謹排列規則）。set/multiset 的 iterator 是其底部的 RB tree 的 const-iterator，就是爲了禁止 user 對元素賦值。

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

容器 set

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

...
};
```

key | data

key 和 data
合成 value

```
set<int> iset;
```

⇨

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

⇨

```
template <
            int,
            int,
            identity<int>,
            less<int>,
            alloc
          >
class rb_tree;
```

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

容器 set, in VC6

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

VC6

```
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;
};
```

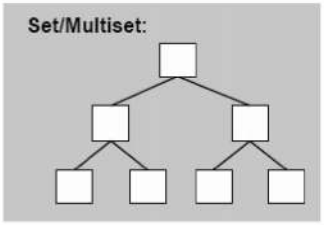
這相當 G2.9 的 identity()
或 G4.9 的 _Identity()

G2.9

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

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

使用容器 multiset




```
D:\handout\C++11-test-DevC++\Test-STL\t
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             snprintf(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();
347 auto pItem = c.find(target);        //比 ::find(...) 快很多
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 爲底層結構，因此有「元素自動排序」特性。
排序的依據是 key。

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（因爲 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 設爲 **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

key | data

key 和 data 合成 value

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

```
map<int,string> imap;
```

⇨

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

⇨

```
template <
      int;
      pair<const int, string>
      select1st<pair<const int, string>>,
      less<int>,
      alloc
>
class rb_tree;
```

容器 map, in VC6

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

VC6

```
template<class _K, class _Ty, class _Pr = less<_K>,
         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;
};
```

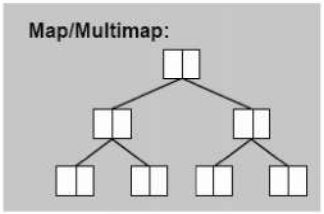
這相當 G2.9 的 select1st()
或 G4.9 的 _Select1st()

G2.9

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

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

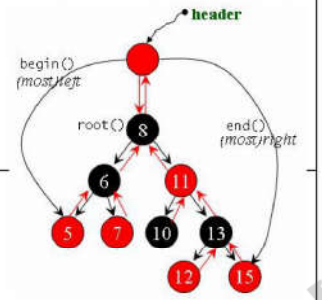
使用容器 multimap




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

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

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

容器 map, 獨特的 operator[ ]



G4.9

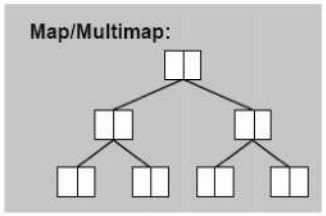
```
mapped_type&
    operator[](const key_type& __k)
    {
        // concept requirements
        __glibcxx_function_requires(_DefaultConstructibleCon

        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_typ
                                            std::tuple<>());
#else
          __i = insert(__i, value_type(__k, mapped_type()));
#endif
        return (*__i).second;
    }
```

— 侯捷 —

```
// [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」。

使用容器 map

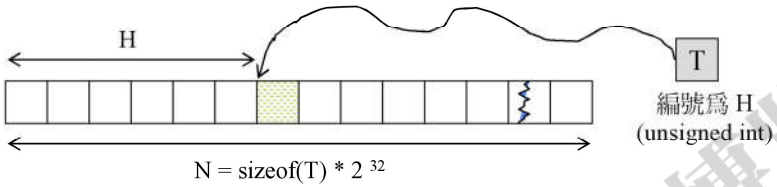



```
D:\handout\C++11-test-DevC++\Test-ST
select: 14
how many elements: 1000000

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

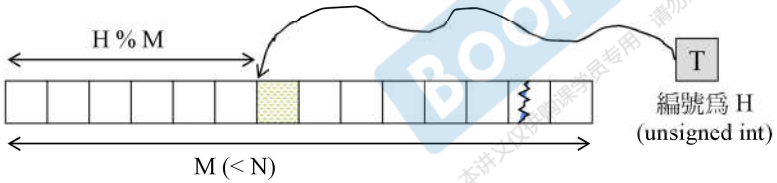
```
696  namespace jj14
697  {
698  void test_map(long& value)
699  {
700      cout << "\ntest_map()..... \n";
701
702  map<long, string> c;
703  char buf[10];
704
705  clock_t timeStart = clock();
706      for(long i=0; i< value; ++i)
707      {
708          try {
709              snprintf(buf, 10,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

容器 hashtable

空間足夠時：



空間不足時：



选2倍附近的素数

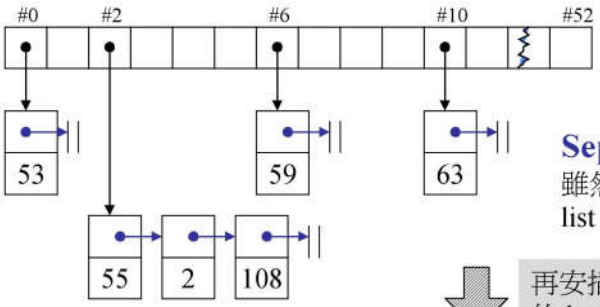
容器 hashtable

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

G2.9

×G4.9

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

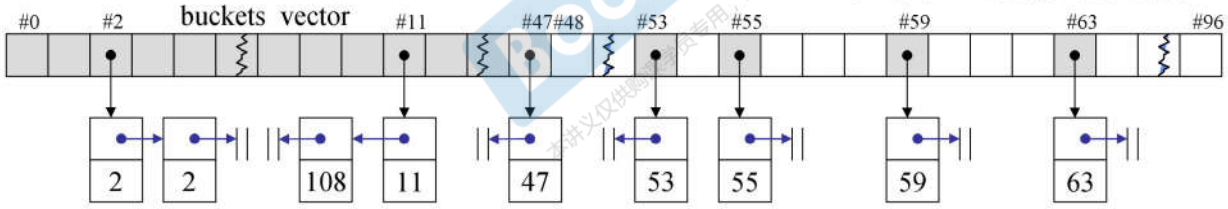


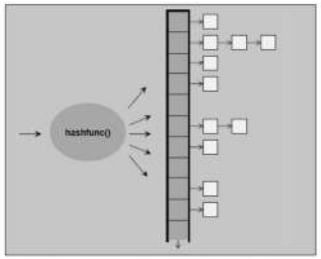
Separate Chaining

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**

size:97 num_elements:54





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

容器 hashtable

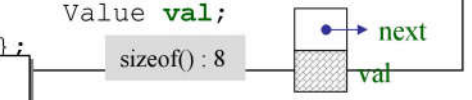
```
template <class Value, class Key, class HashFcn,
          class ExtractKey, class EqualKey,
          class Alloc=alloc>
class hashtable {
public:
  typedef HashFcn  hasher;
  typedef EqualKey key_equal;
  typedef size_t   size_type;

private:
  hasher hash;
  key_equal equals;
  ExtractKey get_key;

  typedef __hashtable_node<Value> node;

  vector<node*,Alloc> buckets;
  size_type num_elements;
public:
  size_type bucket_count() const { return buckets.size(); }
...
};
```

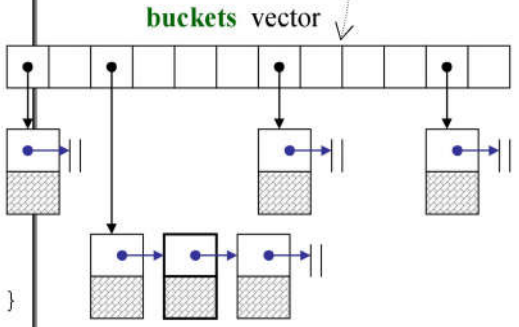
sizeof() : 19 → 20

```
template <class Value, class Key,
          class HashFcn, class ExtractKey,
          class EqualKey, class Alloc>
struct __hashtable_iterator {
...
  node* cur;
  hashtable* ht;
};
```

```
template <class Value>
struct __hashtable_node {
    __hashtable_node* next;
    Value val;
};
```

sizeof() : 8





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

hash-function, hash-code

泛化

```
template <class Key> struct hash { };
```

特化

```
template<>
__STL_TEMPLATE_NULL struct hash<char> {
  size_t operator()(char x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<short> {
  size_t operator()(short x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned short> {
  size_t operator()(unsigned short x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<int> {
  size_t operator()(int x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned int> {
  size_t operator()(unsigned int x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<long> {
  size_t operator()(long x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned long> {
  size_t operator()(unsigned long x) const { return x; }
};
```

```
hash<Foo>();
```

```
int i= hash<int>()(32);
```

hash-function, hash-code

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

  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); }
};
```

本例若 s 指向 "abc"，計算得 h 爲
5 * (5 * 'a' + 'b') + 'c'

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

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

modulus 運算

```
iterator find(const key_type& key)
{ size_type n = bkt_num_key(key); //花落何家
  ...                                  ❶

size_type count(const key_type& key) const
{ const size_type n = bkt_num_key(key);
  ...                                  ❶

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);
...                                    ❷
}
```

```
❶ size_type
bkt_num_key(const key_type& key) const
 {
   return bkt_num_key(key, buckets.size());
 }
❷ size_type
bkt_num(const value_type& obj) const
 {
   return bkt_num_key(get_key(obj));
 }
```

```
size_type
bkt_num_key(const key_type& key, size_t n) const
 {
   return hash(key) % n;
 }
size_type
bkt_num(const value_type& obj, size_t n) const
 {
   return bkt_num_key(get_key(obj), n);
 }
```

這個 **hash** 不是前頁出現的 struct **hash**，
而是 class hashtable 中的 hasher **hash** 。

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

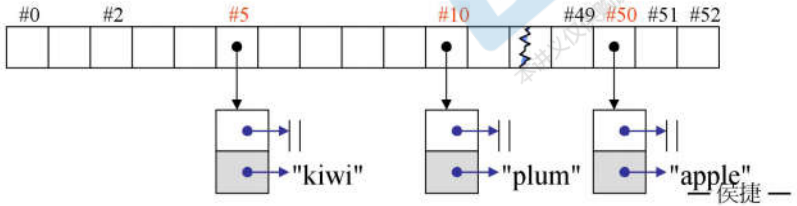
```
template <...>
class hashtable {
private:
  hasher hash;
...
```

— 侯捷 —

hash-function, hash-code

App.

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

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



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             select1st< 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

```cpp
_Hashtable< string,
            string,
            hash<string>,
            _Identity<string>,
            equal_to<string>,
          >     //[Error] wrong number of template arguments (6, shou
sht(50, hash<string>(), equal_to<string>());

cout<< sht.size() << endl;              // 0
cout<< sht.bucket_count() << endl;      // 53

// my goal!
_Hashtable< pair<const string,int>,
            string,
            hash<string>,
            _Select1st< pair<const string,int> >,
            equal_to<string>,
          >     //[Error] wrong number of template arguments (6, should be 10)
siht(100, hash<string>(), equal_to<string>());

cout<< siht.size() << endl;              // 0
cout<< siht.bucket_count() << endl;      // 193

siht.insert_unique( make_pair(string("jjhou"),95) );
siht.insert_unique( make_pair(string("sabrina"),90) );
siht.insert_unique( make_pair(string("mjchen"),85) );
cout<< siht.size() << endl;              // 3
cout<< siht.bucket_count() << endl;      // 193
cout << siht.find(string("sabrina"))->second << endl;   //90
cout << siht.find(string("jjhou"))->second << endl;     //95
cout << siht.find(string("mjchen"))->second << endl;    //85
```

…\4.9.2\include\c++\bits\basic_string.h

```cpp
/// std::hash specialization for string.
template<>
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
{ };
```

unordered 容器

Before C++11

- **hash_set**
- **hash_multiset**
- **hash_map**
- **hash_multimap**

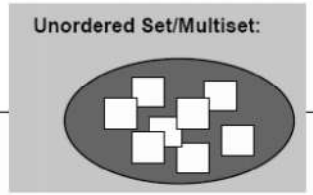
Since C++11

- **unordered_set**
- **unordered_multiset**
- **unordered_map**
- **unordered_multimap**

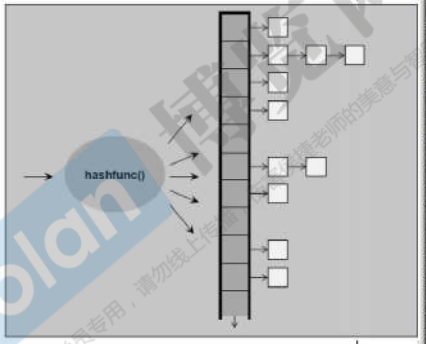
G4.9

```
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;
```

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              snprintf(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      }
```




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

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

使用容器 unordered_set

```
770
771   string target = get_a_target_string();
772       {
773       timeStart = clock();
774   auto pItem = ::find(c.begin(), c.end(), target);     //比 c.find(...) 慢很多
775       cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
776       if (pItem != c.end())
777           cout << "found, " << *pItem << endl;
778       else
779           cout << "not found! " << endl;
780       }
781
782       {
783       timeStart = clock();
784   auto pItem = c.find(target);         //比 ::find(...) 快很多
785       cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
786       if (pItem != c.end())
787           cout << "found, " << *pItem << endl;
788       else
789           cout << "not found! " << endl;
790       }
791   }
792   }
```

```
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
```

C++標準庫
體系結構與內核分析

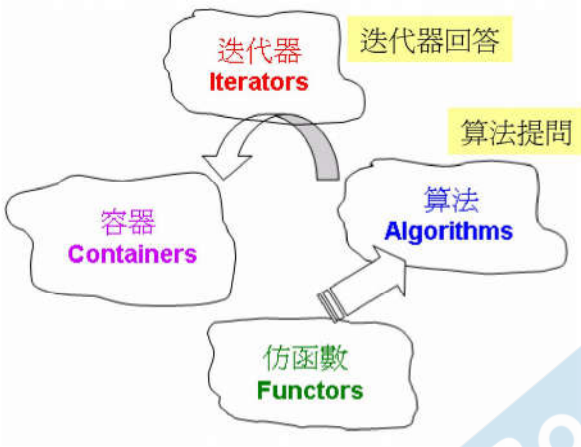
(C++ Standard Library — architecure & sources)

第三講

侯捷

源碼之前
了無秘密

C++標準庫的算法, 是什麼東西？



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)
{
  …
}
```

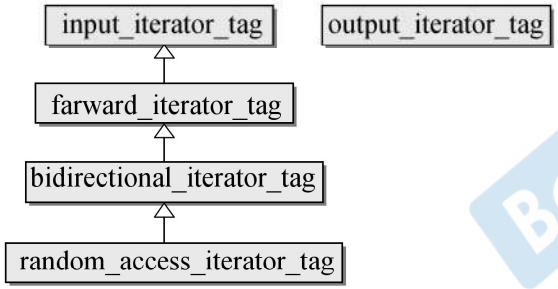
```
template<typename Iterator, typename Cmp>
Algorithm(Iterator itr1, Iterator itr2, Cmp comp)
{
  …
}
```

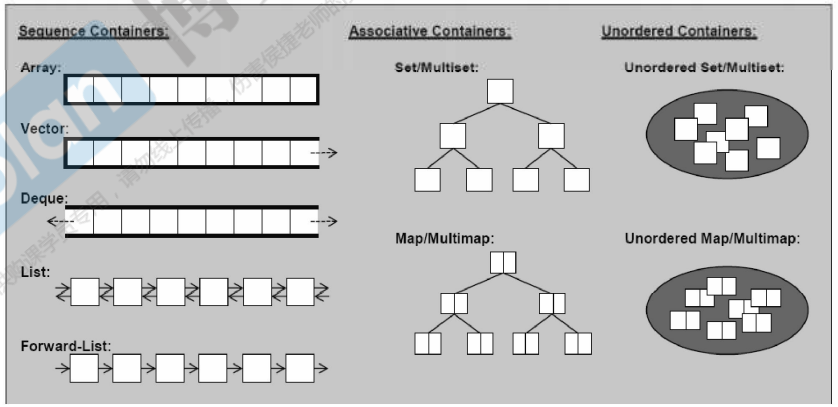
Algorithms 看不見 Containers，對其一無所知；所以，它所需要的一切信息都必須從 Iteratrors 取得，而 Iterators (由 Containers 供應) 必須能夠回答 Algorithm 的所有提問，才能搭配該 Algorithm 的所有操作。

— 侯捷 —

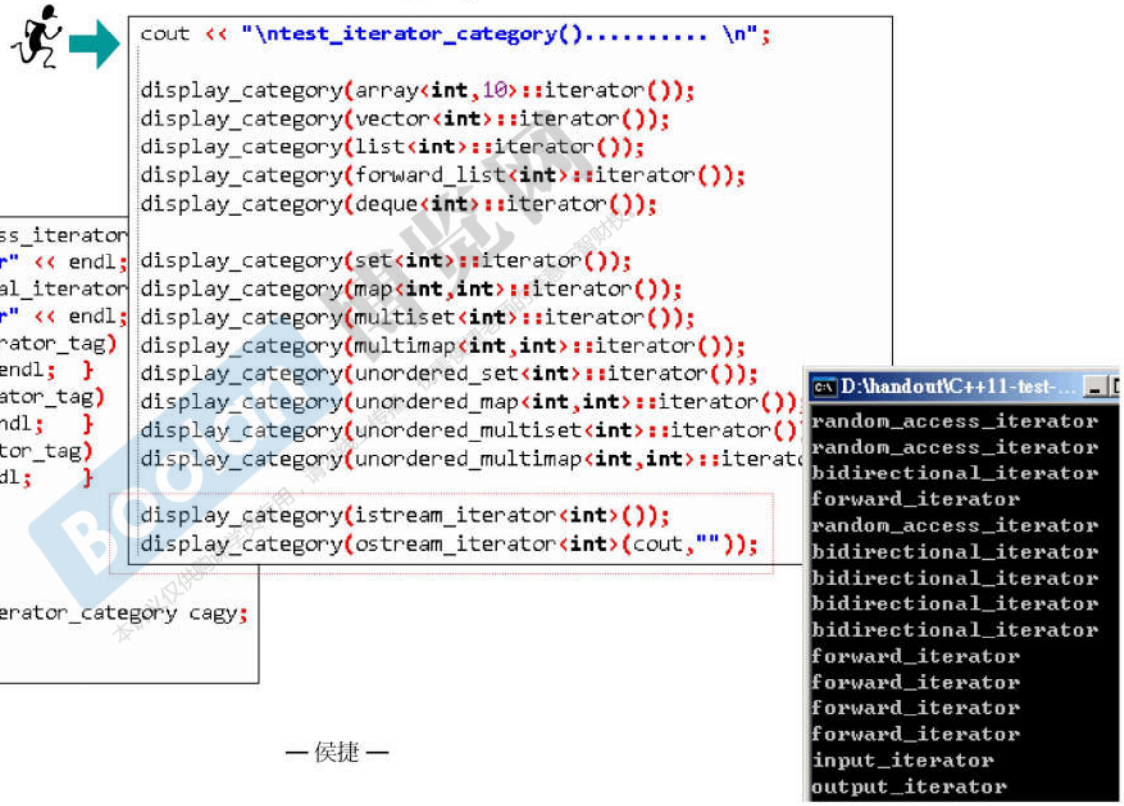
各種容器的 iterators 的 iterator_category

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





各種容器的 iterators 的 iterator_category



```
void _display_category(random_access_iterator
{   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);
}
```

```
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>::iterat

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

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

各種容器的 iterators 的 iterator_category 的 typeid

#include <typeinfo> // typeid

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

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

   cout << "typeid(itr).name()= " << typeid(itr).name() << endl << en
      //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.
}
```

— 侯捷 —

```
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__detail14_Node_iteratorIiLb1ELb0EEE

forward_iterator
typeid(itr).name()= NSt8__detail14_Node_iteratorISt4pairIKiiELb0ELb0EEE

forward_iterator
typeid(itr).name()= NSt8__detail14_Node_iteratorIiLb1ELb0EEE

forward_iterator
typeid(itr).name()= NSt8__detail14_Node_iteratorISt4pairIKiiELb0ELb0EEE

input_iterator
typeid(itr).name()= St16istream_iteratorIicSt11char_traitsIcEiE

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

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;
  };
```

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;
...
```

G4.9

```
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

```
D:\handout\C++11-test-...
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
```

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;
 };
```

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;
…
```

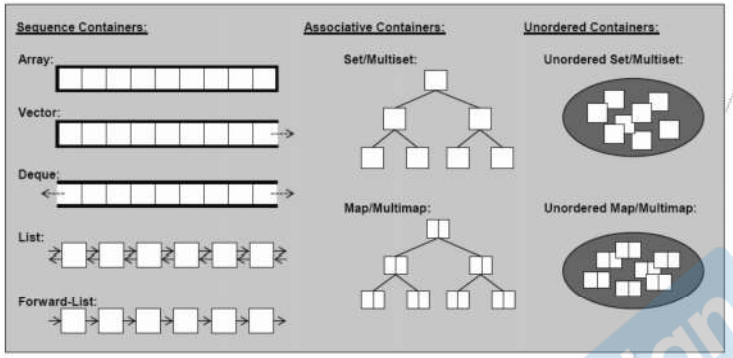
G4.9

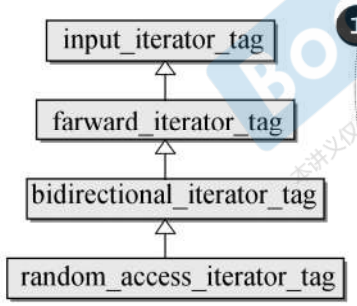
```
template<typename _Tp, typename _CharT = char,
         typename _Traits = char_traits<_CharT> >
 class ostream_iterator
 : public iterator<output_iterator_tag, void, void, void, void>
 {
```

❶ ❷ ❸ ❹ ❺

```
D:\handout\C++11-test-...
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





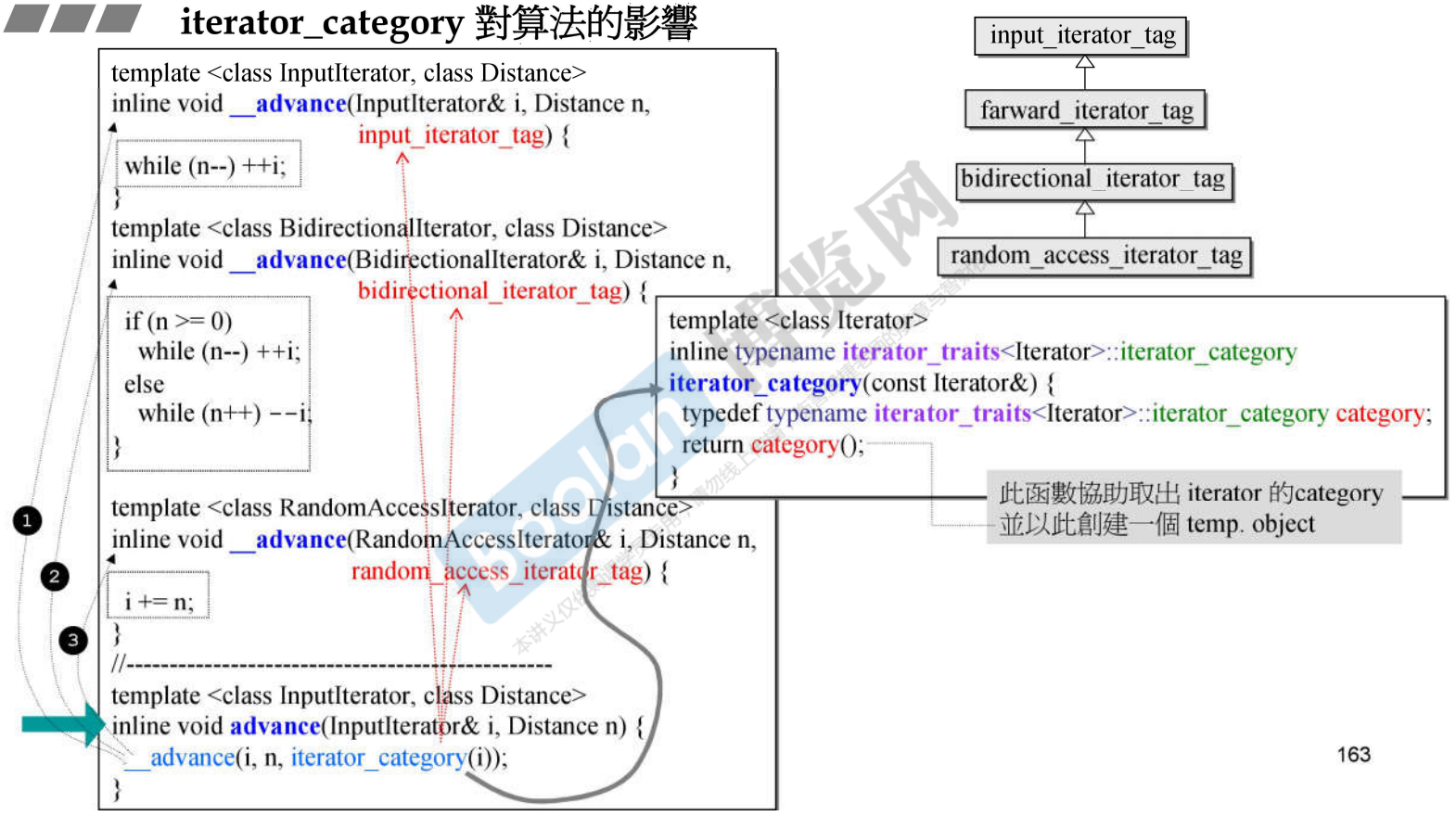
1

2

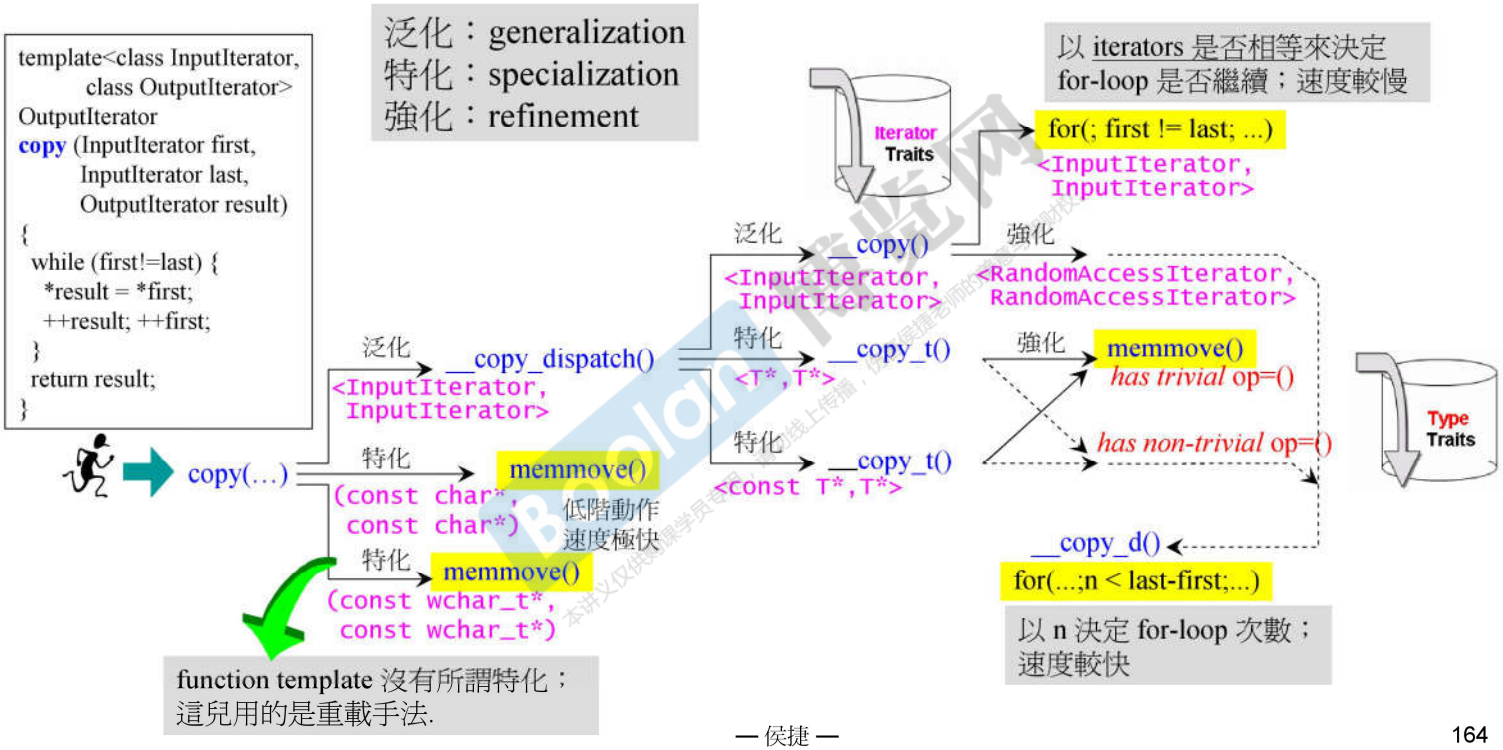
```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last,
           input_iterator_tag) {
  iterator_traits<InputIterator>::difference_type n = 0;
  while (first != last) {
    ++first; ++n;
  }
  return n;
}
template <class RandomAccessIterator>
inline iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first, RandomAccessIterator last,
           random_access_iterator_tag) {
  return last - first;
}
// ------------------------------------------------------------
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last) {
  typedef typename
    iterator_traits<InputIterator>::iterator_category category;
  return __distance(first, last, category());
}
```

temp. object

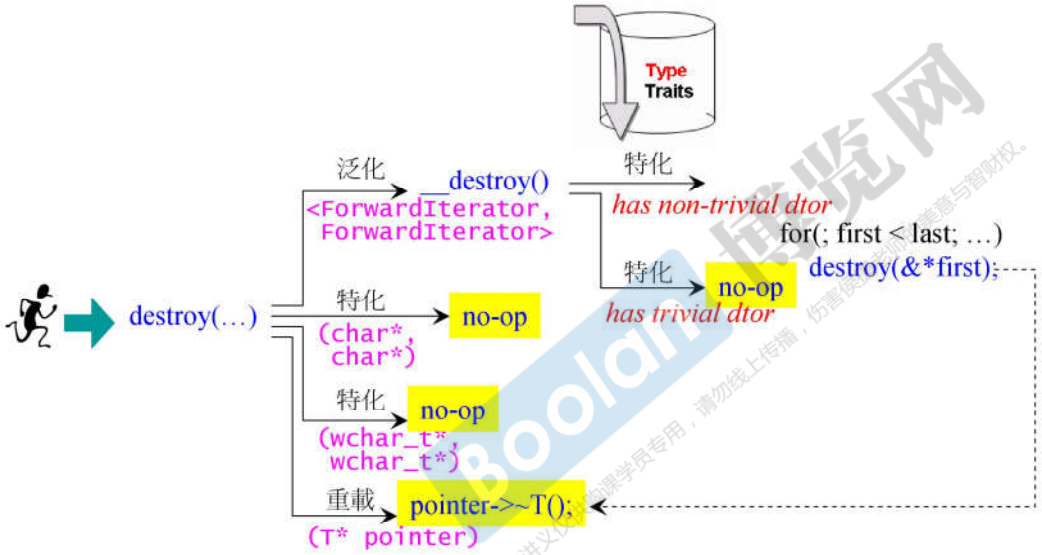
iterator_category 對算法的影響



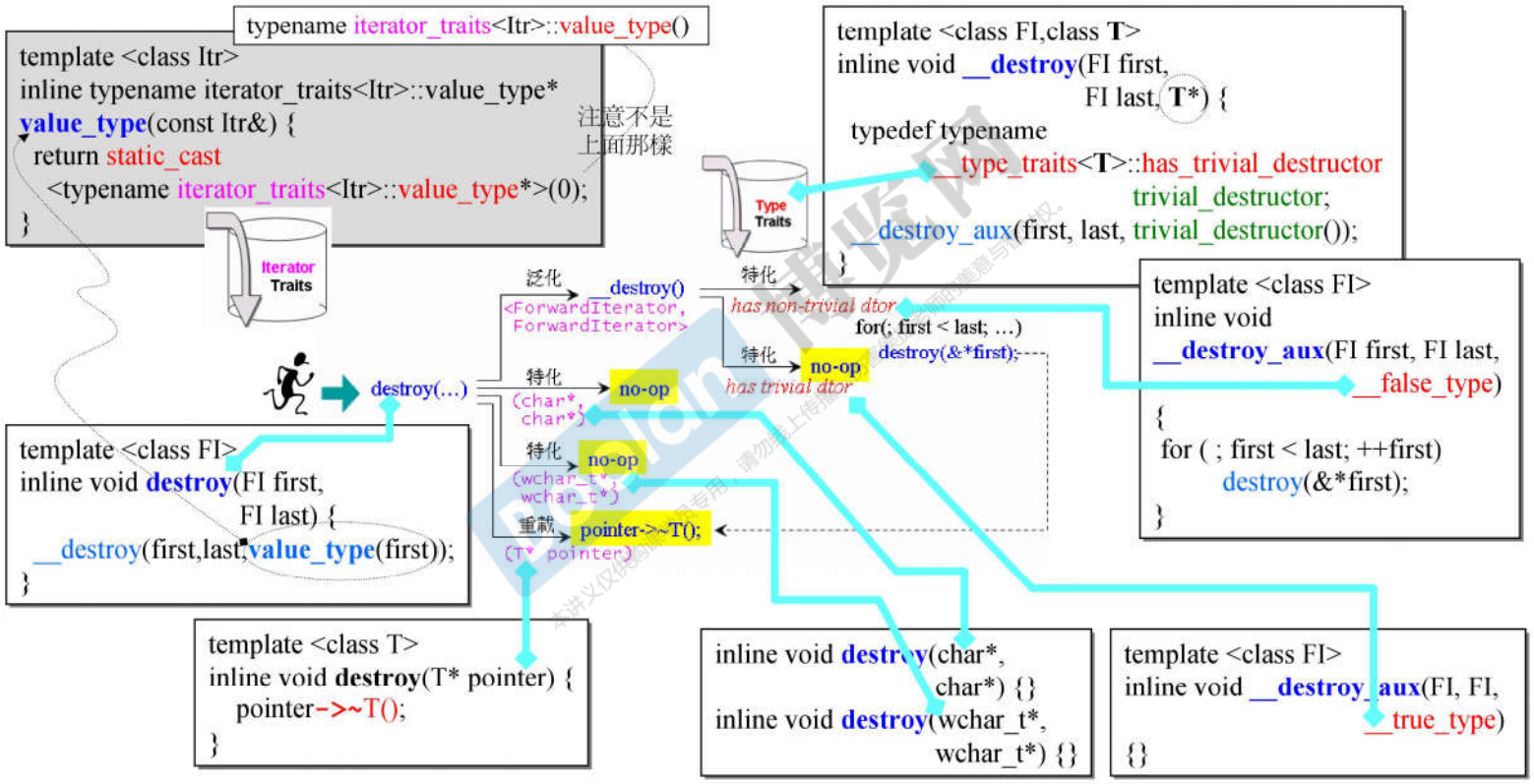
iterator_category 和 type traits 對算法的影響



iterator traits 和 type traits 對算法的影響



iterator traits 和 type traits 對算法的影響



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

注意不是下面這樣

```
typename iterator_traits<Itr>::value_type()
```

```
template <class InputIterator, class 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;
}
```

vs.

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 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;
}
```

這是 **read**-op

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

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

先前示例中出現的算法

```
qsort(c.data(), ASIZE, sizeof(long), compareLongs);
```

這是 C 函數.

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

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  }
```

算法 for_each

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

range-based for statement

(since C++11)

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

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

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

算法 replace, replace_if, replace_copy

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

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

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

算法 count, count_if

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last,
      const T& value) {
 //以下定義一個初值爲 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) {
 //以下定義一個初值爲 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 的結果爲 true
    ++n;                         //計數器累加1
 return n;
}
```

容器不帶成員函數 count()：
array, vector, list, forward_list, deque,

容器帶有成員函數 count()：
set / multiset,
map / multimap,
unordered_set / unordered_multiset
unordered_map / unordered_multimap

算法 find, find_if

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

循序式 搜尋/查找

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

循序式 搜尋/查找

容器不帶成員函數 find()：
array, vector, list, forward_list, deque,

容器帶有成員函數 find()：
set / multiset,
map / multimap
unordered_set / unordered_multiset
unordered_map / unordered_multimap

要求random access iterator

算法 sort

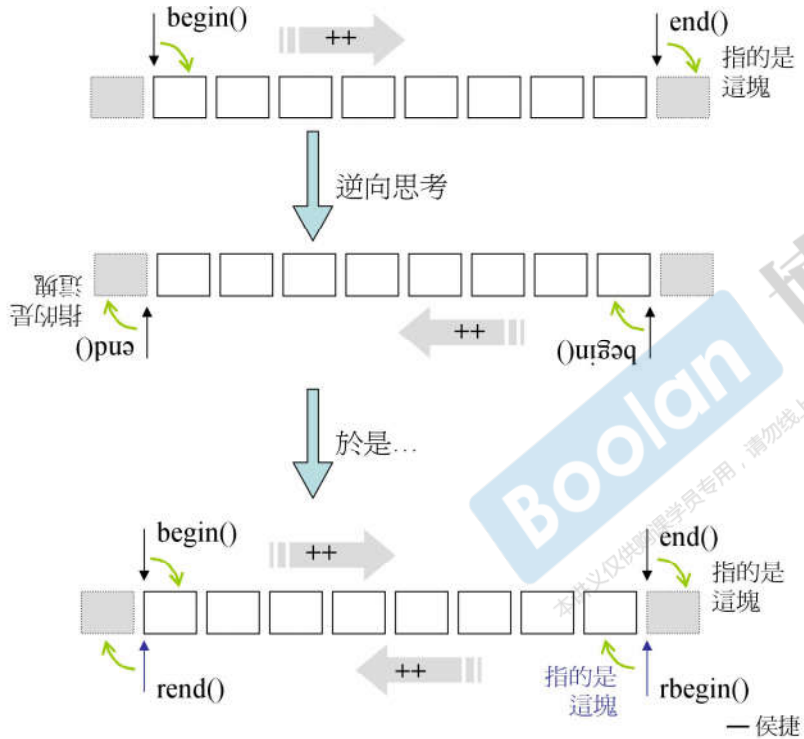
```
1791  bool myfunc (int i,int j) { return (i<j); }
1792
1793  struct myclass {
1794    bool operator() (int i,int j) { return (i<j);}
1795  } myobj;
1796
1797  bool test_sort()
1798  {
1799    int myints[] = {32,71,12,45,26,80,53,33};
1800    vector<int> myvec(myints, myints+8);          // 32 71 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

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));
}
```

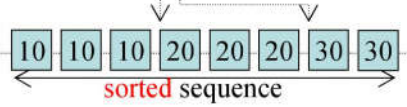
獲得的 iterator 所指的位置
既非 end, 目標值 val 亦不小於首元素 (sorted range 之首元素值最小)

侯捷檢討：先判斷 !(val < *first) 然後才調用 lower_bound()，將獲得較佳效率。因爲進入 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;
}
```

二分查找

low=lower_bound (v.begin(), v.end(), 20);
up=upper_bound (v.begin(), v.end(), 20);



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

仿函數 functors

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

```
//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 —

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

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

G4.9

```
template <class _Tp>
  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

没有改造的机会

```
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 (因爲 Function Adapter 將會提問)。例如：

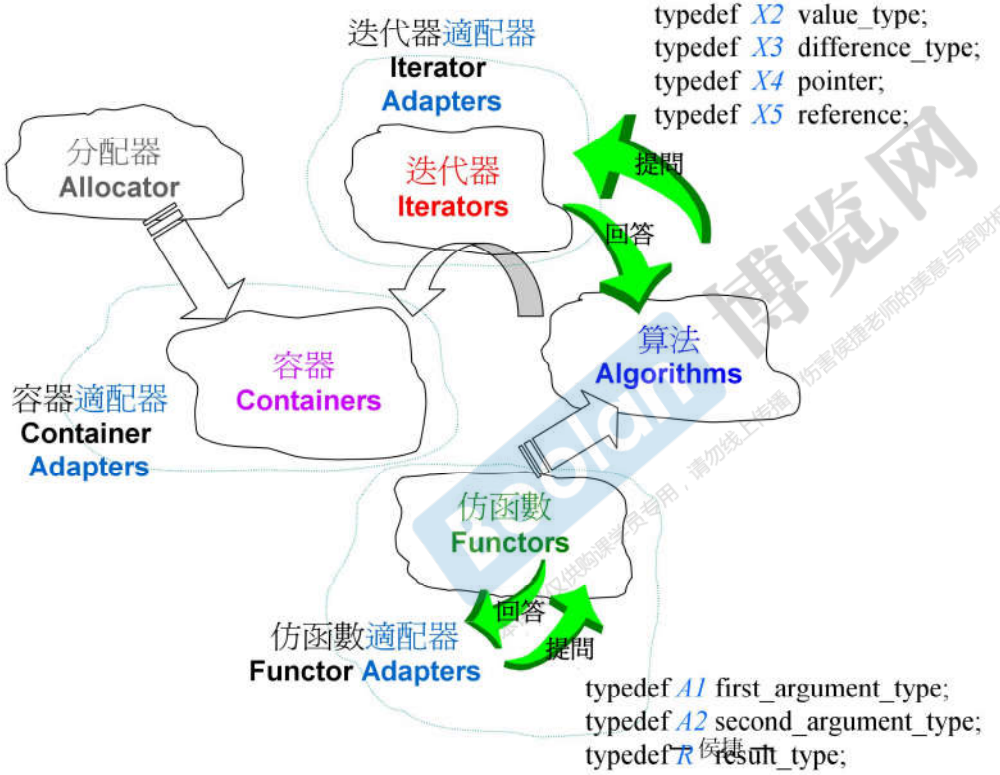
```
template <class T>
struct less : public binary_function<T, T, bool> {
   bool operator()(const T& x, const T& y) const
   { return x < y; }
};
```

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;
```

adapter询问这些
functor回答
所以functor要继承

存在多種 Adapters




```
typedef X1 iterator_category;
typedef X2 value_type;
typedef X3 difference_type;
typedef X4 pointer;
typedef X5 reference;
```

```
typedef A1 first_argument_type;
typedef A2 second_argument_type;
typedef R result_type;
```

侯捷

用复合/内含的方式

容器適配器 : stack, queue

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

```
template <class T, class Sequence=deque<T>>
class queue {
...
public:
  typedef typename Sequence::value_type value_type;
  typedef typename Sequence::size_type size_type;
  typedef typename Sequence::reference reference;
  typedef typename Sequence::const_reference const_reference;
protected:
  Sequence c;  //底層容器
public:
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference front() { return c.front(); }
  const_reference front() const { return c.front(); }
  reference back() { return c.back(); }
  const_reference back() const { return c.back(); }
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_front(); }
};
```

內含+改造

函數適配器：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) {
 //以下定義一個初值爲 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 的結果爲 true
    ++n;            //計數器累加1
 return n;
}
```

```
cout << count_if(vi.begin(), vi.end(),
                 not1(bind2nd(less<int>(),40)));
```

predicate

algorithm

function adapter (negator)

function adapter (binder)

function object

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

先前只是记下参数
调用时再绑定

函數適配器：not1

```
cout << count_if(vi.begin(), vi.end(),
                 not1(bind2nd(less<int>(),40)));
```

predicate

algorithm

function adapter (negator)

function adapter (binder)

function object

```
// 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) {
 //以下定義一個初值爲 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 的結果爲 true
     ++n;             //計數器累加1
 return n;
}
```

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

新型適配器, bind

...\include\c++\backward\backward_warning.h

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

 Use:                                      Instead of:
 <sstream>, basic_stringbuf                <strstream>, strstreambuf
 <sstream>, basic_istringstream            <strstream>, istrstream
 <sstream>, basic_ostringstream            <strstream>, ostrstream
 <sstream>, basic_stringstream             <strstream>, strstream
 <unordered_set>, unordered_set            <ext/hash_set>, hash_set
 <unordered_set>, unordered_multiset       <ext/hash_set>, hash_multiset
 <unordered_map>, unordered_map            <ext/hash_map>, hash_map
 <unordered_map>, unordered_multimap       <ext/hash_map>, hash_multimap
 <functional>, bind                        <functional>, binder1st
 <functional>, bind                        <functional>, binder2nd
 <functional>, bind                        <functional>, bind1st
 <functional>, bind                        <functional>, bind2nd
 <memory>, unique_ptr                      <memory>, auto_ptr
*/
```

新型適配器, bind

Since C++11

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

#include <functional> // std::bind

```
// a function: (also works with function object:
//              std::divides<double> my_divide;)
double my_divide (double x, double y)
    { return x / y; }

struct MyPair {
  double a,b;
  double multiply() { return a * b; }
    //member function 其實有個 argument: this
};
```

std::**bind** 可以綁定：

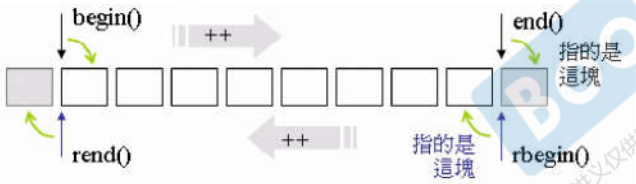
1. functions
2. function objects
3. member functions, _1 必須是某個 object 地址.
4. data members, _1 必須是某個 object 地址.

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.

```
using namespace std::placeholders;    // adds visibility of _1, _2, _3,...

// binding functions:
auto fn_five = bind (my_divide,10,2);               // returns 10/2
cout << fn_five() << '\n';                          // 5

auto fn_half = bind (my_divide,_1,2);               // returns x/2
cout << fn_half(10) << '\n';                        // 5

auto fn_invert = bind (my_divide,_2,_1);            // returns y/x
cout << fn_invert(10,2) << '\n';                    // 0.2

auto fn_rounding = bind<int> (my_divide,_1,_2);     // returns int(x/y)
cout << fn_rounding(10,3) << '\n';                  // 3


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

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

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


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

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

迭代器適配器：reverse_iterator

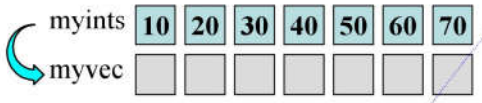
```
reverse_iterator
rbegin()
{ return reverse_iterator(end()); }

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



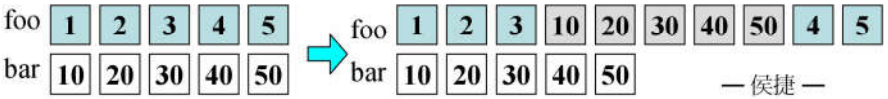

```
template <class Iterator>
class reverse_iterator
{
protected:
  Iterator current;   // 對應之正向迭代器
public:
  // 逆向迭代器的5種 associated types 都和其對應之正向迭代器相同
  typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
  typedef typename iterator_traits<Iterator>::value_type  value_type;
  ...
  typedef Iterator iterator_type;                    // 代表正向迭代器
  typedef reverse_iterator<Iterator> self;           // 代表逆向迭代器
public:
  explicit reverse_iterator(iterator_type x) : current(x) {}
           reverse_iterator(const self& x) : current(x.current) {}

  iterator_type base() const { return current; } // 取出對應的正向迭代器
  reference operator*() const {    Iterator tmp = current;    return *--tmp; }
  // 以上爲關鍵所在。對逆向迭代器取值，就是
  //   將「對應之正向迭代器」退一位取值。
  pointer operator->() const { return &(operator*()); } // 意義同上。

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

迭代器適配器：inserter

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

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




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

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

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



— 侯捷 —

```
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) 操作改變爲
// 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」的行爲。
template <class Container>
class insert_iterator {
protected:
  Container* container; // 底層容器
  typename Container::iterator iter;
public:
  typedef output_iterator_tag iterator_category;   // 注意類型

  insert_iterator(Container& x, typename Container::iterator i)
    : container(&x), iter(i) {}

  insert_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 insert_iterator。
template <class Container, class Iterator>
inline insert_iterator<Container>
inserter(Container& x, Iterator i) {
  typedef typename Container::iterator iter;
  return insert_iterator<Container>(x, iter(i));
}
```

X 適配器 : ostream_iterator

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

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

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

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

這就相當於

```
cout << *first;
cout << ", ";
```

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

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

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

X 適配器 : istream_iterator

例 1

```
// 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;
}
```

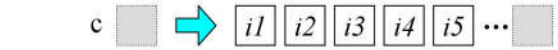
這就相當於 `cin >> value;`

這就相當於 `return value;`

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

一旦創建立刻 **read**, 可能令 user 吃驚

X 適配器 : istream_iterator



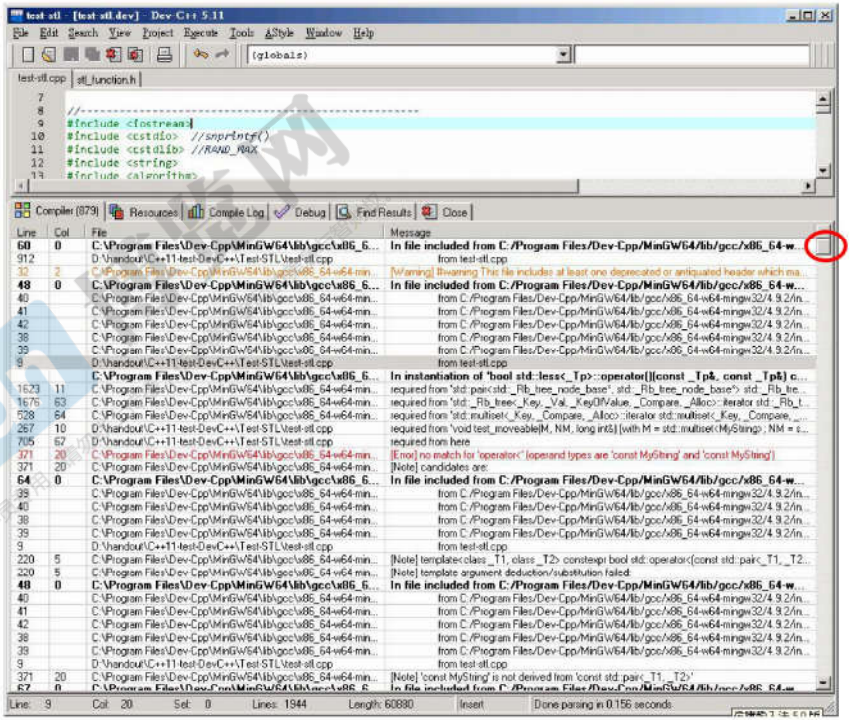

```
istream_iterator<int> iit(cin), eos;          例 2

copy(iit, eos, inserter(c, c.begin()));
```

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

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

一旦創建立刻 **read**，可能令 user 吃驚

使用一個東西,
卻不明白它的道理,
不高明！



C++標準庫
體系結構與內核分析

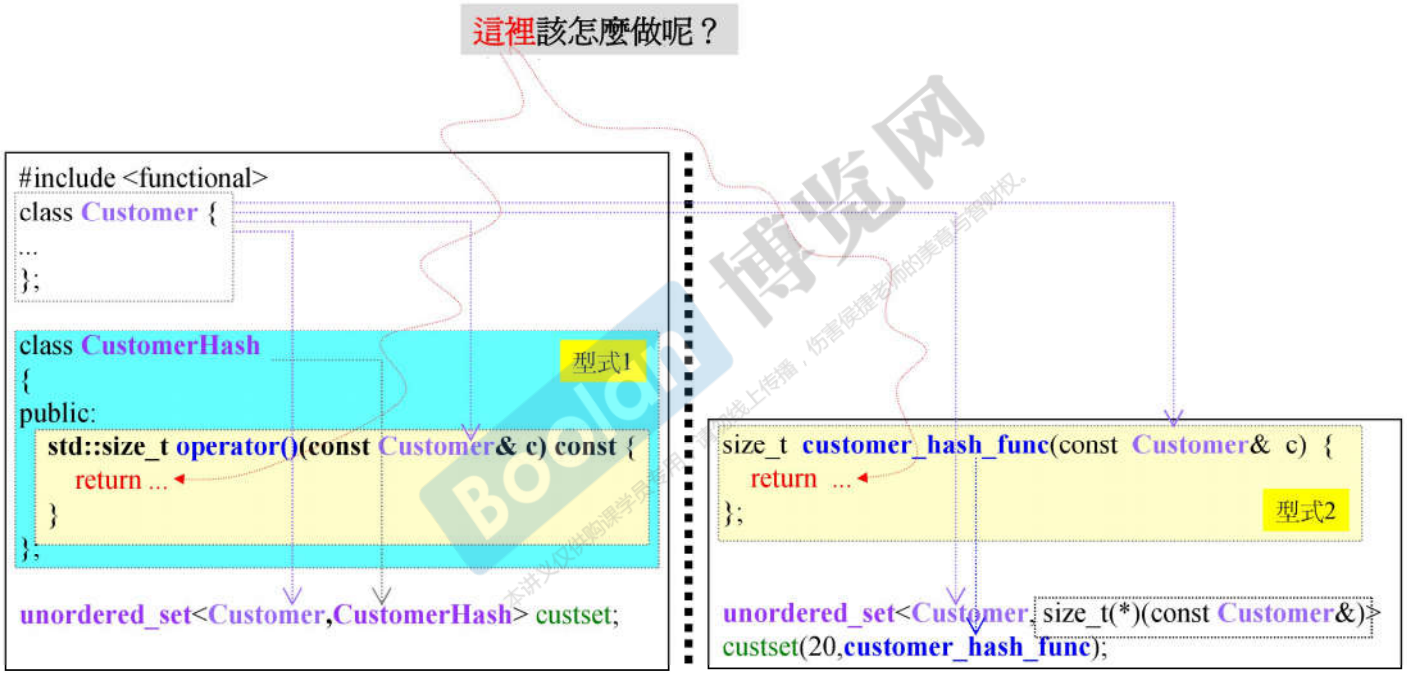
(C++ Standard Library — architecure & sources)

第四講

侯捷

勿在浮沙築高台

一個萬用的 Hash Function



一個萬用的 Hash Function

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

太過天真 ✖

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);
    }
};
```

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

```
// auxiliary generic functions
template <typename T>
inline void hash_val(size_t& seed, const T& val) {
    hash_combine(seed, val);
}
```

last

— 侯捷 —

```
template <typename T, typename... Types>
inline void hash_val (size_t& seed,
                      const T& val, const Types&... args) {
    hash_combine(seed, val);
    hash_val(seed, args...);
}
```

recursive

逐一取 val
改變 **seed**
(pass by reference)

```
// auxiliary generic function
template <typename... Types>
inline size_t hash_val(const Types&... args) {
    size_t seed = 0;
    hash_val (seed, args...);
    return seed;
}
```

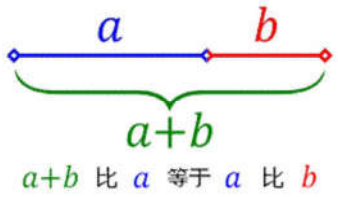
seed 最終就被視爲 hash code

n
↓
1 n-1
↓
1 n-2
↓
. . .

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 φ 表示黃金比值，用代數式表達就是：

$$\frac{a+b}{a} = \frac{a}{b} \equiv \varphi$$



$a+b$ 比 a 等于 a 比 b

0x9e3779b9 / 0x7FFFFFFF = 0.618033.....
http://en.wikipedia.org/wiki/Golden_ratio

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

一個萬用的 Hash Function

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

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

— 侯捷 —

一個萬用的 Hash Function

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

for (unsigned i=0; i<set3.bucket_count(); ++i) {
    cout << "bucket #" << i << " has " << set3.bucket_size(i) << " elements.\n";
}
//bucket #0 has 0 elements.
//bucket #1 has 0 elements.
//bucket #2 has 3 elements.
//bucket #3 has 0 elements.
//bucket #4 has 1 elements.
//bucket #5 has 0 elements.
//bucket #6 has 1 elements.
//bucket #6 has 0 elements.
//bucket #6 has 0 elements.
//bucket #6 has 1 elements.
//bucket #6 has 1 elements.
```

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

以 struct hash 偏特化形式 實現 Hash Function

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;
```

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>   //爲了 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<>                                  …\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()); }
  };
```

tuple, 用例

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

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

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

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

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

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

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

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

tuple 元之組合, 數之組合

G4.8 節錄並簡化

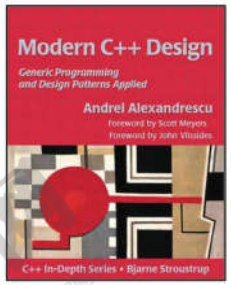
```
template<typename... Values> class tuple;
template<> class tuple<> { };
```

```
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;
};
```

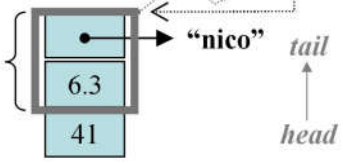
呼叫 base ctor 並予參數
(不是創建 temp object)

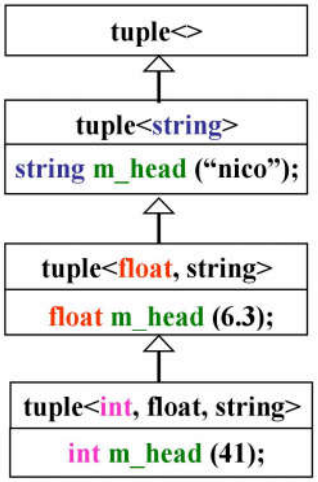
注意這是 initialization list

return 後
轉型爲 inherited,
獲得的是
— 侯捷 —



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





type traits

G2.9

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

泛化

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

Plain Old Data

特化

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

__type_traits<Foo>::has_trivial_destructor

特化

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

type traits

Since C++11

http://www.cplusplus.com/reference/type_traits/?kw=type_traits

Type properties

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

Type traits

Primary type categories

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

Composite type categories

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

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

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

type traits, 測試

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

    cout << "is_void\t" << is_void<T>::value << endl;
    cout << "is_integral\t" << is_integral<T>::value << endl;
    cout << "is_floating_point\t" << is_floating_point<T>::value <<
    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     {
```

```
460       basic_string(const basic_string& __str);
```

```
512       basic_string(basic_string&& __str)
513   #if _GLIBCXX_FULLY_DYNAMIC_STRING == 0
514       noexcept // FIXME C++11: should always be noexcept.
515   #endif
516       : _M_dataplus(__str._M_dataplus)
517       {
```

```
546       ~basic_string() _GLIBCXX_NOEXCEPT
547       { _M_rep()->_M_dispose(this->get_allocator()); }
```

```
553       basic_string&
554       operator=(const basic_string& __str)
555       { return this->assign(__str); }
```

```
589       operator=(basic_string&& __str)
590       {
591     // NB: DR 1204.
592     this->swap(__str);
593     return *this;
594       }
```

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

type traits, 測試

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

type_traits_output(Foo());
```

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

```
has_virtual_destructor  0
is_default_constructible 1
is_copy_constructible   1
is_move_constructible   1
is_copy_assignable      1
is_move_assignable      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_copy_assignable      1
is_nothrow_move_assignable      1
is_nothrow_destructible 1
```

— 侯捷 —

type traits, 測試

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

type_traits_output(Goo());
```

```
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
```

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

— 侯捷 —

type traits, 測試

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

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

```
type traits for type : N4jj263ZooE
is_void 0
is_integral     0
is_floating_point       0
is_arithmetic   0
is_signed       0
is_unsigned     0
is_const        0
is_volatile     0
is_class        1
is_function     0
is_reference    0
is_lvalue_reference     0
is_rvalue_reference     0
is_pointer      0
is_member_pointer       0
is_member_object_pointer        0
is_member_function_pointer      0
is_fundamental  0
is_scalar       0
is_object       1
is_compound     1
is_standard_layout      0
is_pod  0
is_literal_type 0
is_empty        0
is_polymorphic  1
is_abstract     0
```

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

— 侯捷 —

type traits, 測試

type_traits_output(complex<float>());

```
type traits for type : St7complexIfE
is_void    0
is_integral             0
is_floating_point       0
is_arithmetic           0
is_signed               0
is_unsigned             0
is_const                0
is_volatile             0
is_class                1
is_function             0
is_reference            0
is_lvalue_reference          0
is_rvalue_reference          0
is_pointer              0
is_member_pointer       0
is_member_object_pointer     0
is_member_function_pointer   0
is_fundamental          0
is_scalar               0
is_object               1
is_compound             1
is_standard_layout      1
is_pod                  0
is_literal_type         1
is_empty                0
is_polymorphic          0
is_abstract             0
```

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

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
```

type traits, 實現, is_void

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

```
165     template<typename>
166       struct __is_void_helper
167       : public false_type { };
168
169     template<>
170       struct __is_void_helper<void>
171       : public true_type { };
172
173     /// is_void
174     template<typename _Tp>
175       struct is_void
176       : public __is_void_helper<typename remove_cv<_Tp>::type>::type
177       { };
```

泛化 + 偏特化

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   template<>
222     struct __is_integral_helper<int>
223     : public true_type { };
224
225   template<>
226     struct __is_integral_helper<unsigned int>
227     : public true_type { };
228
229   template<>
230     struct __is_integral_helper<long>
231     : public true_type { };
232
233   template<>
234     struct __is_integral_helper<unsigned long>
235     : public true_type { };
236
237   template<>
238     struct __is_integral_helper<long long>
239     : public true_type { };
240
241   template<>
242     struct __is_integral_helper<unsigned long long>
243     : public true_type { };
```

```
255   /// is_integral
256   template<typename _Tp>
257     struct is_integral
258     : public __is_integral_helper<typename remove_cv<_Tp>::type>::type
259     { };
```

type traits, 實現 is_class, is_union, is_enum, is_pod

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

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

藍色這些 _is_*xxx* 未曾出現於 C++ 標準庫源代碼

type traits, 實現 is_move_assignable

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

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

cout

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

G2.9
iostream.h

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

extern _IO_ostream_withassign cout;
```

G2.9
iostream.h

cout

G4.9

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

```
template <class _CharT, class _Traits, size_t _Nb>
  std::basic_ostream<_CharT, _Traits>&
  operator<<(std::basic_ostream<_CharT, _Traits>& __os,
             const bitset<_Nb>& __x)
  { … }
```

```
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>& __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)
  { … }
```

The End