C++標準庫
體系結構與內核分析

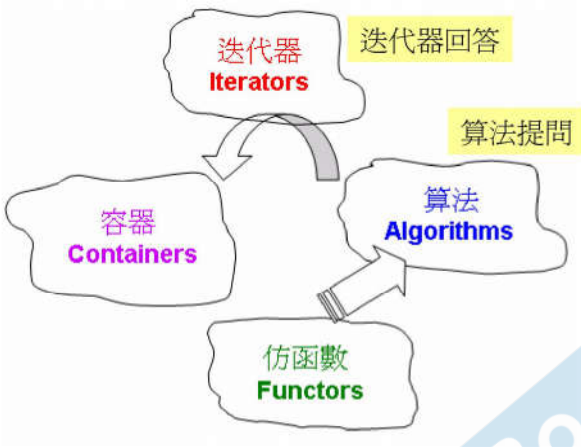
(C++ Standard Library — architecure & sources)

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)
{
  …
}
```

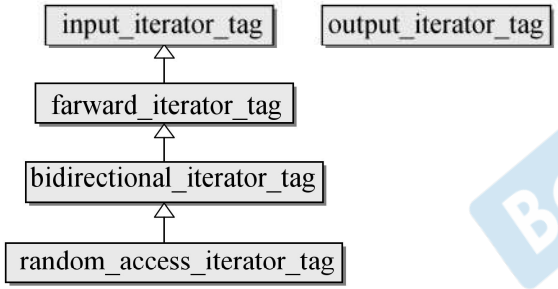
```
template<typename Iterator, typename Cmp>
Algorithm(Iterator itr1, Iterator itr2, Cmp comp)
{
  …
}
```

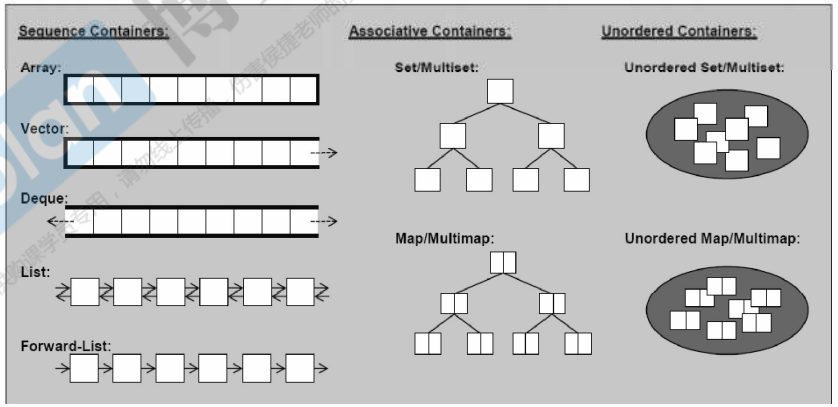
Algorithms 看不見 Containers，對其一無所知；所以，它所需要的一切信息都必須從 Iteratrors 取得，而 Iterators (由 Containers 供應) 必須能夠回答 Algorithm 的所有提問，才能搭配該 Algorithm 的所有操作。

— 侯捷 —

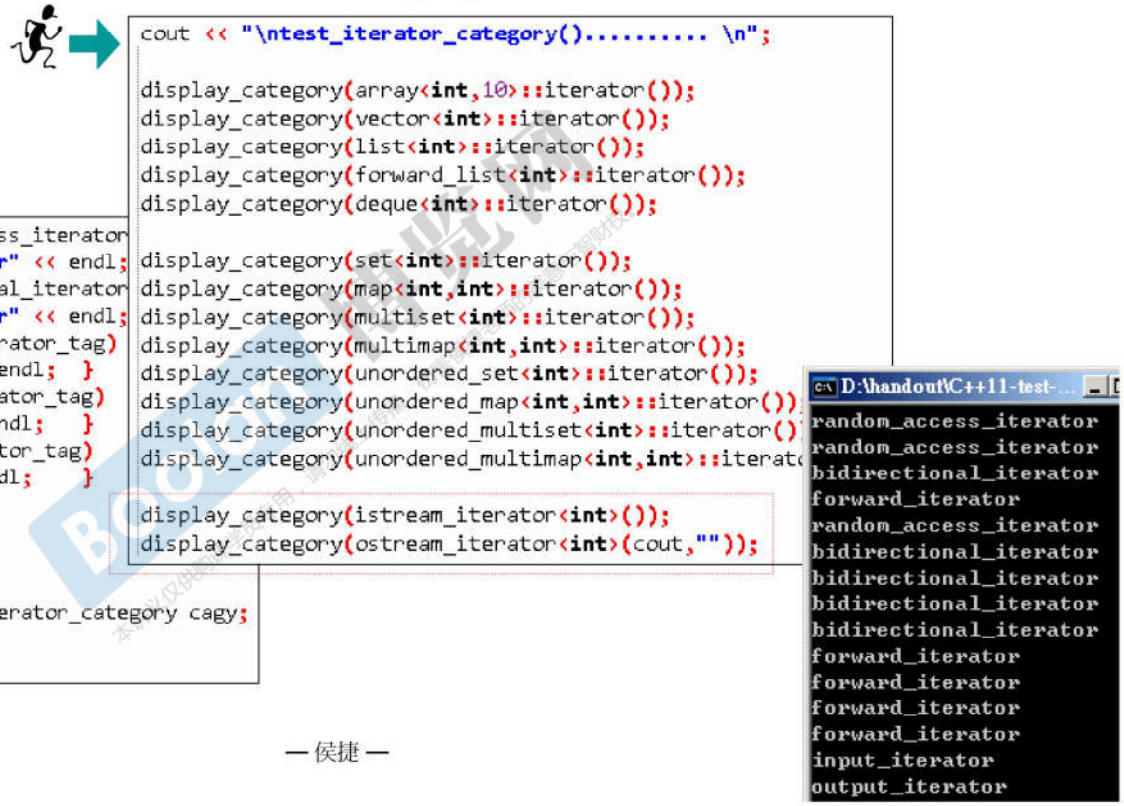
各種容器的 iterators 的 iterator_category

```
// 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;
…
```

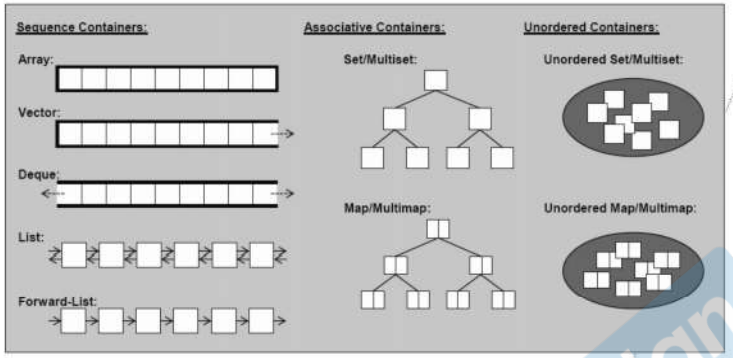
G4.9

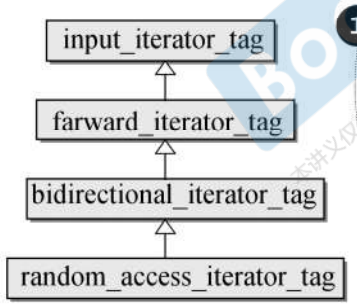
```
template<typename _Tp, typename _CharT = char,
         typename _Traits = char_traits<_CharT> >
 class ostream_iterator
 : public iterator<output_iterator_tag, void, void, void, void>
 {
```

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





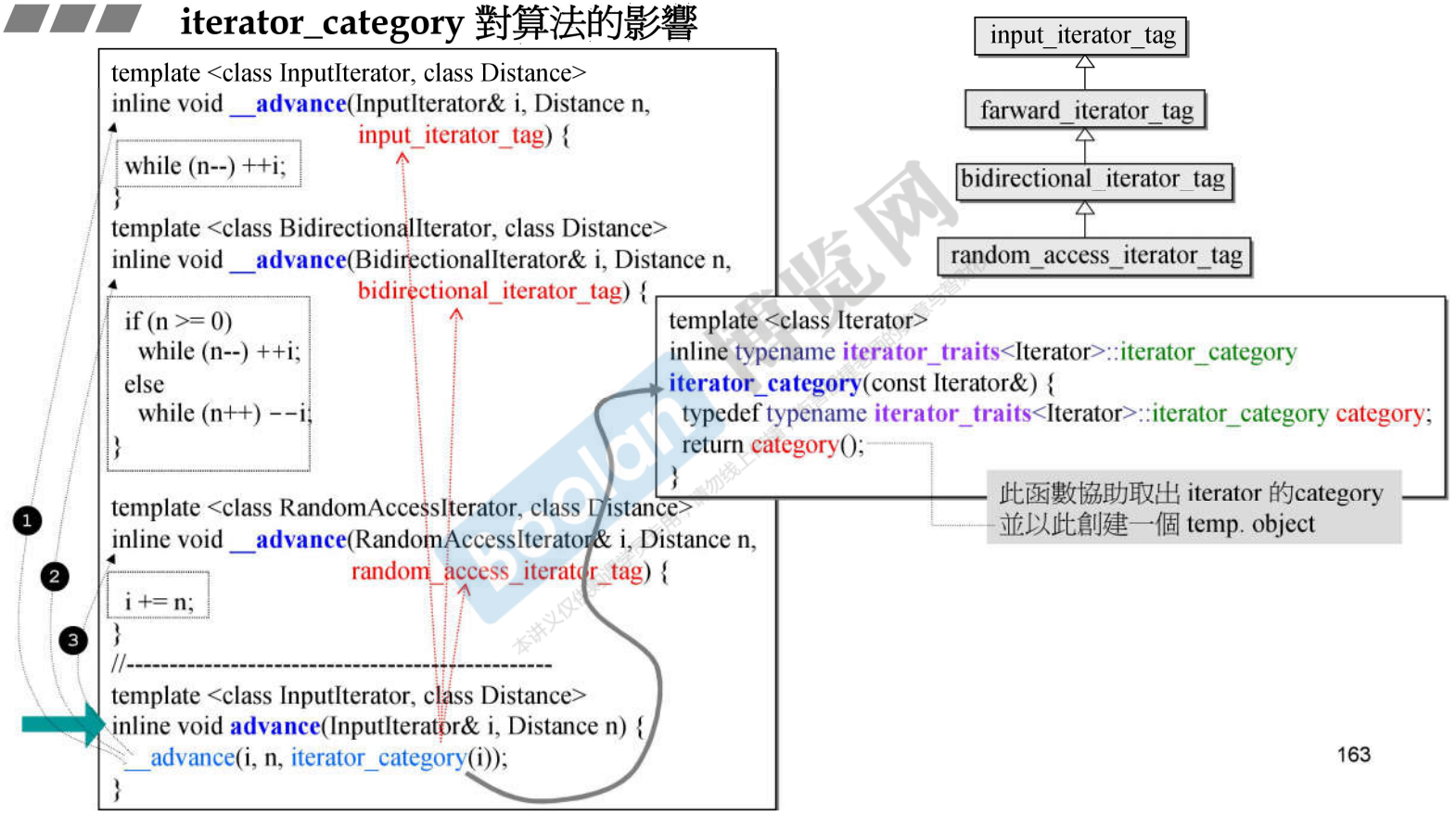
1

2

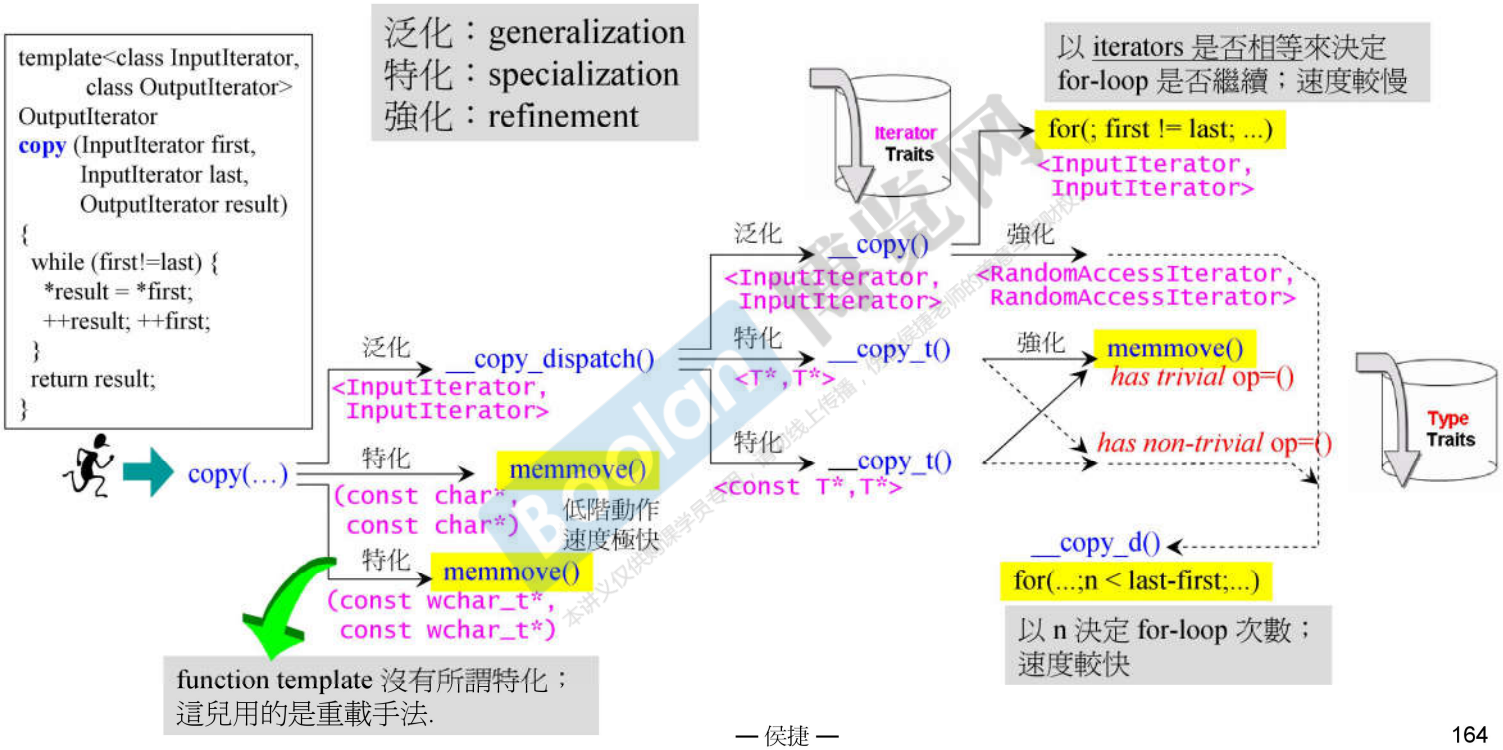
```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last,
           input_iterator_tag) {
  iterator_traits<InputIterator>::difference_type n = 0;
  while (first != last) {
    ++first; ++n;
  }
  return n;
}
template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

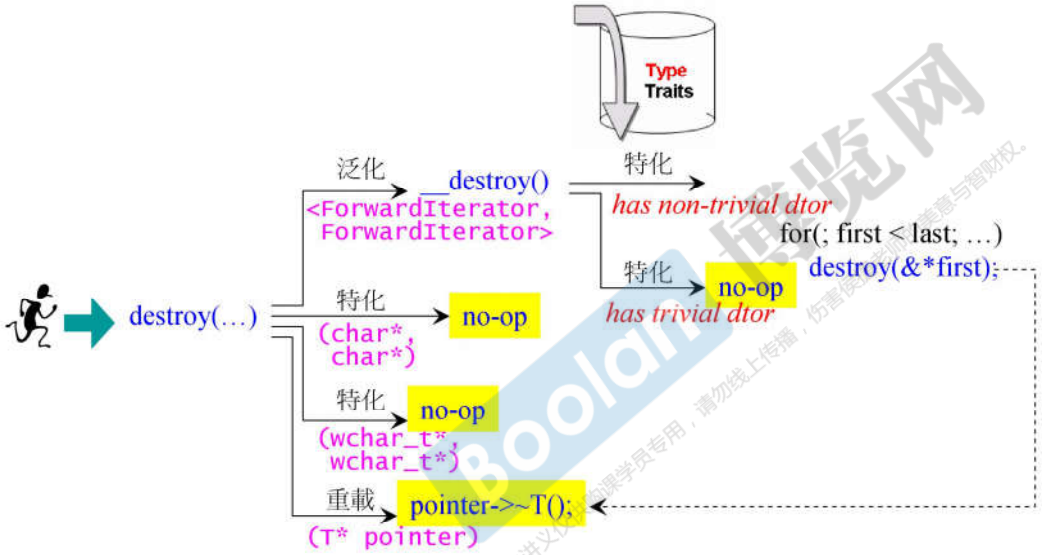
iterator_category 對算法的影響



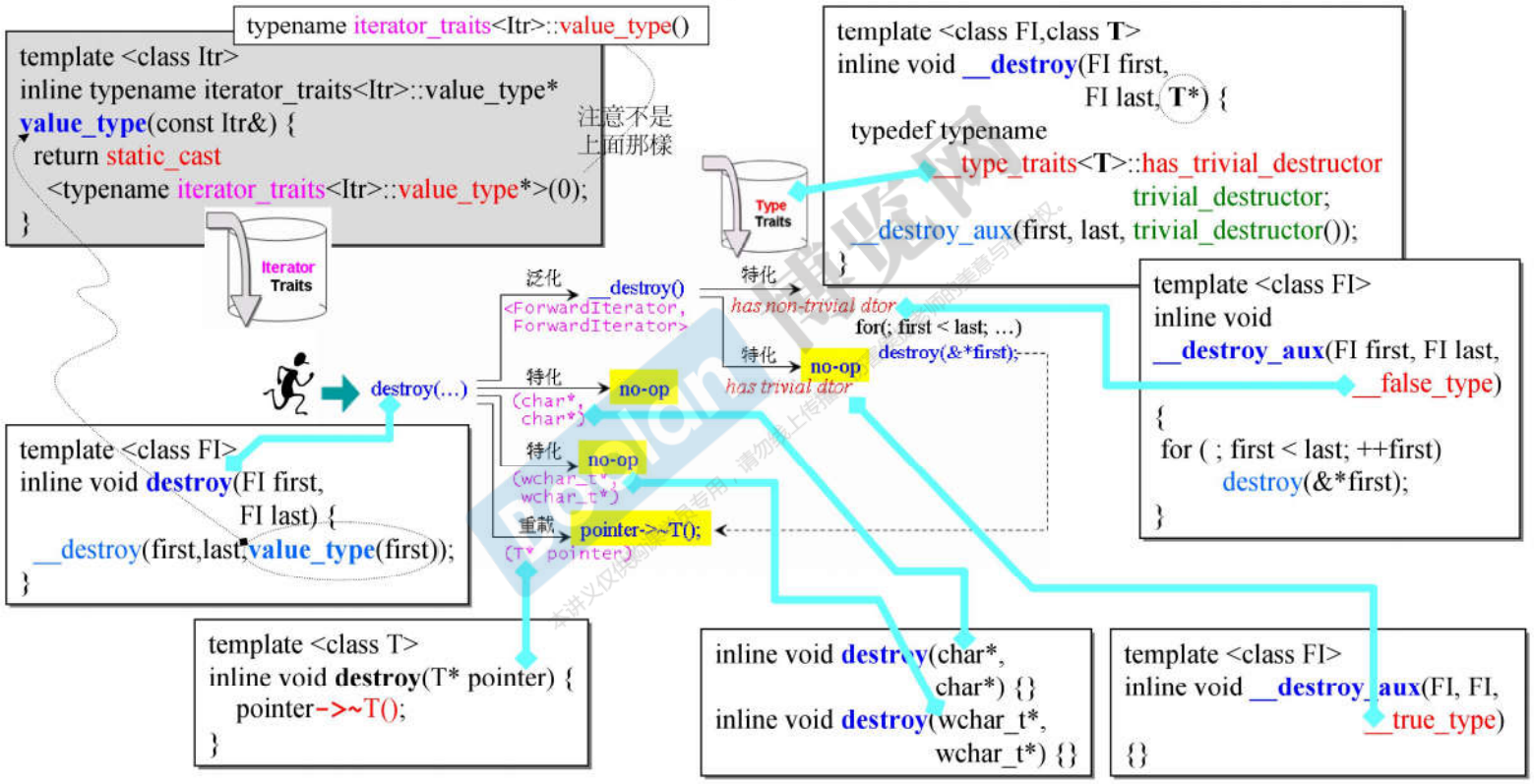
iterator_category 和 type traits 對算法的影響



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

算法 sort

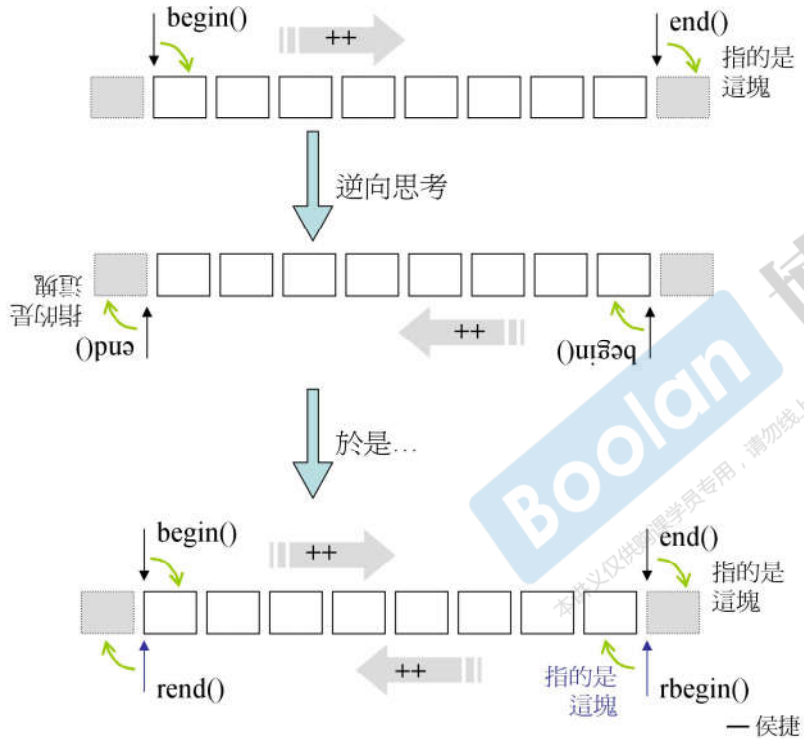
```cpp
bool myfunc (int i,int j) { return (i<j); }

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

bool test_sort()
{
  int myints[] = {32,71,12,45,26,80,53,33};
  vector<int> myvec(myints, myints+8);          // 32 71 12 45 26 80 53 33

  // using default comparison (operator <):
  sort(myvec.begin(), myvec.begin()+4);         //(12 32 45 71)26 80 53 33

  // using function as comp
  sort(myvec.begin()+4, myvec.end(), myfunc);   // 12 32 45 71(26 33 53 80)

  // using object as comp
  sort(myvec.begin(), myvec.end(), myobj);      //(12 26 32 33 45 53 71 80)

  // print out content:
  cout << "myvec contains:";
  for (auto elem : myvec)       //C++11 range-based for statement
      cout << ' ' << elem ;     //output: 12 26 32 33 45 53 71 80

  // using reverse iterators and default comparison (operator <):
  sort(myvec.rbegin(), myvec.rend());

  // print out content:
  cout << "myvec contains:";
  for (auto elem : myvec)       //C++11 range-based for statement
      cout << ' ' << elem ;     //output: 80 71 53 45 33 32 26 12
}
```

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));
}
```

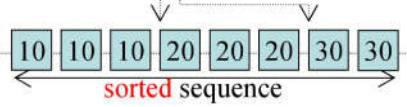
獲得的 iterator 所指的位置
既非 end, 目標值 val 亦不小於首元素 (sorted range 之首元素值最小)

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) {}
};
```

G4.9

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

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

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

...\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> {};
```

仿函數 functors

```
// using default comparison (operator <):
sort(myvec.begin(), myvec.end());

// using function as comp
sort(myvec.begin(), myvec.end(), myfunc);

// using object as comp
sort(myvec.begin(), myvec.end(), myobj);

// using explicitly default comparison (operator <):
sort(myvec.begin(), myvec.end(), less<int>());

// using another comparision criteria (operator >):
sort(myvec.begin(), myvec.end(), greater<int>());
```

這就沒有融入 STL

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

bool myfunc (int i, int j) { return (i < j); }
```

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

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

仿函數 functors 的可適配 (adaptable) 條件

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

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

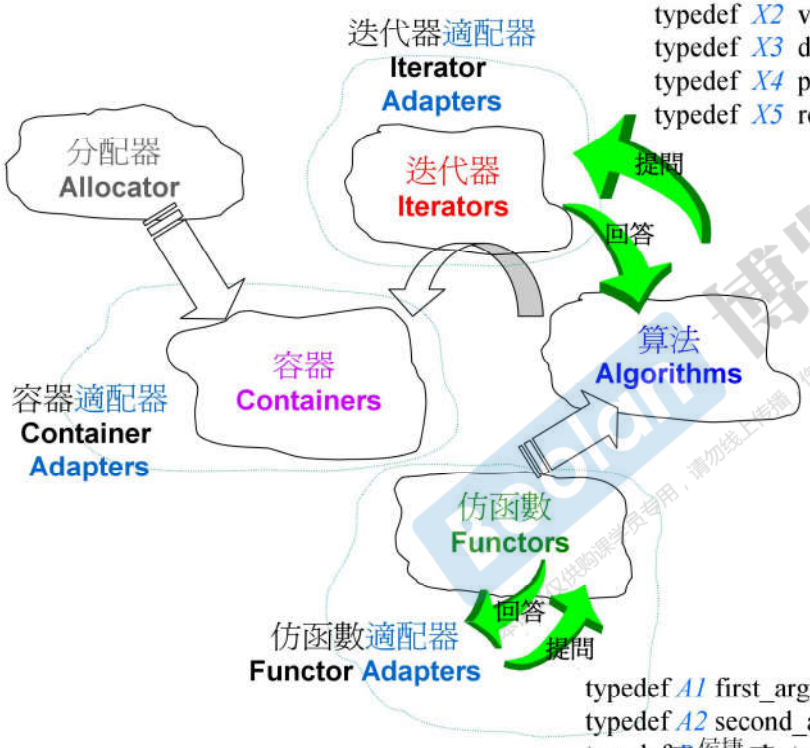
STL 規定每個 Adaptable Function 都應挑選適當者繼承之 (因爲 Function Adapter 將會提問)。例如：

```
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;
```

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

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(); }
};
```

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
  }
};
```

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：

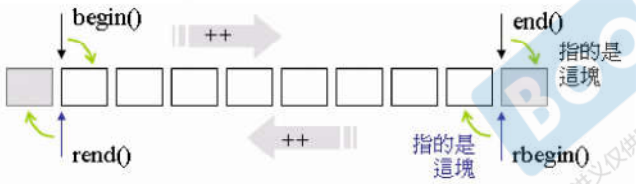
1. functions
2. function objects
3. member functions, _1 必須是某個 object 地址.
4. data members, _1 必須是某個 object 地址.

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

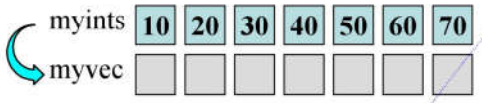
```
reverse_iterator
rbegin()
{ return reverse_iterator(end()); }

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



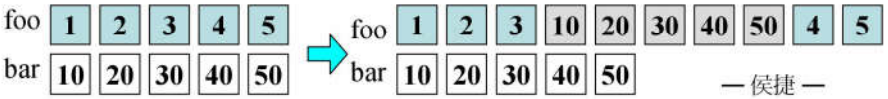

```
template <class Iterator>
class reverse_iterator
{
protected:
  Iterator current;   // 對應之正向迭代器
public:
  // 逆向迭代器的5種 associated types 都和其對應之正向迭代器相同
  typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
  typedef typename iterator_traits<Iterator>::value_type  value_type;
  ...
  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;
}
```

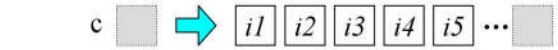
這就相當於 `cin >> value;`

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



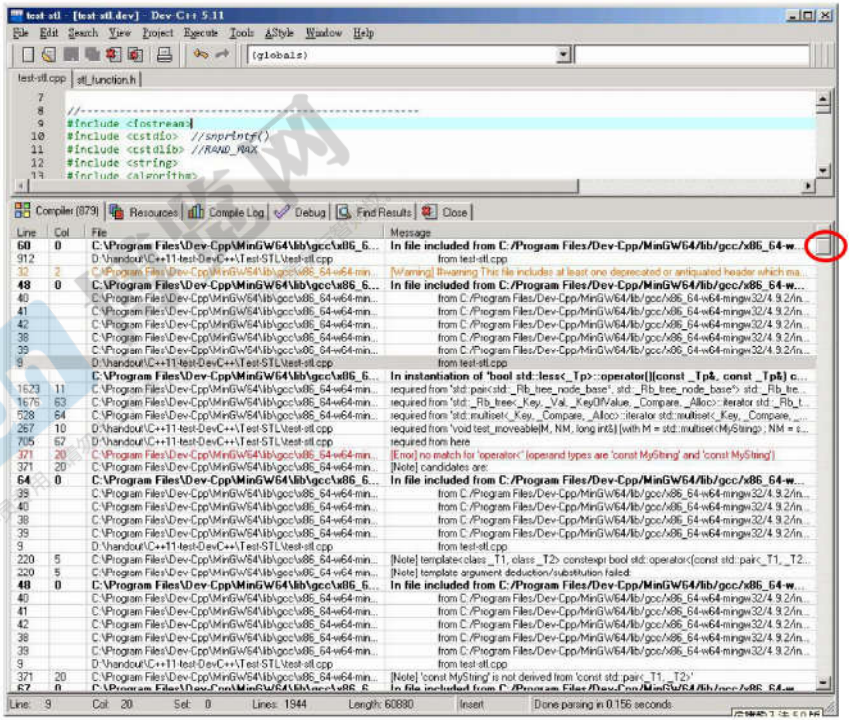

```
istream_iterator<int> iit(cin), eos;          例 2

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！



The End

C++標準庫
體系結構與內核分析

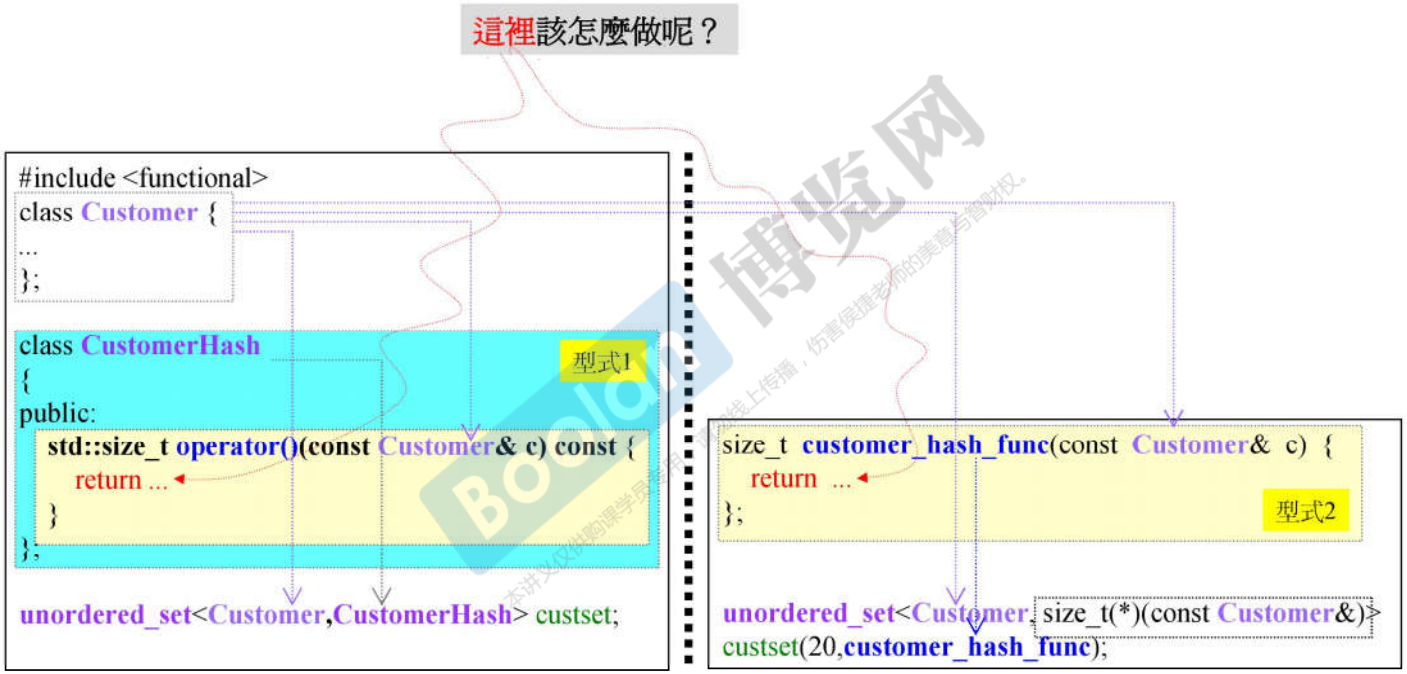
(C++ Standard Library — architecure & sources)

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

```
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;
}
```

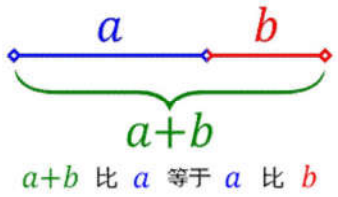
seed 最終就被視爲 hash code

— 侯捷 —

一個萬用的 Hash Function

黃金比例，又稱**黃金比**，是一種數學上的比例關係。黃金分割具有嚴格的比例性、藝術性、和諧性，蘊藏著豐富的美學價值。應用時一般取0.618或1.618 ，就像圓周率在應用時取3.14一樣。黃金分割早存在於大自然中，呈現於不少動物和植物外觀。現今很多工業產品、電子產品、建築物或藝術品均普遍應用黃金分割，呈現其功能性與美觀性。常用希臘字母 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);
```

— 侯捷 —

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

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

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

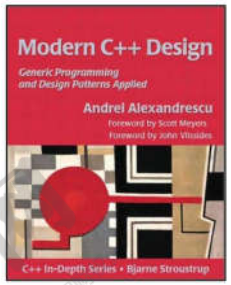
```
template<typename... Values> class tuple;
template<> class tuple<> { };
```

```
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;
};
```

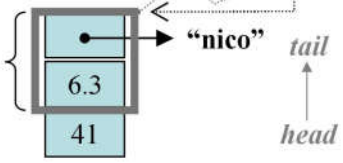
呼叫 base ctor 並予參數
(不是創建 temp object)

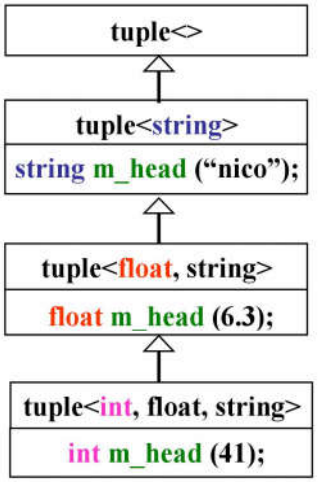
注意這是 initialization list

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       { };
```

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)
  { … }
```

moveable 元素對於 vector 速度效能的影響

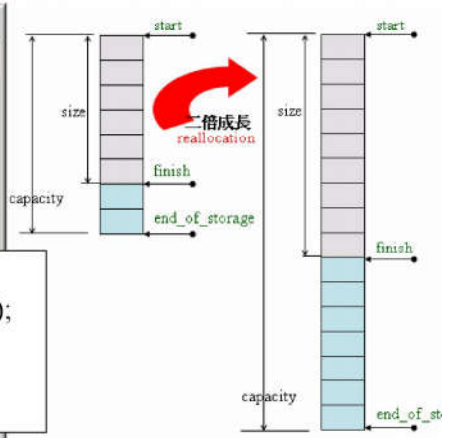
```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

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


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

差別巨大

差別很大，有時候更是巨大得多

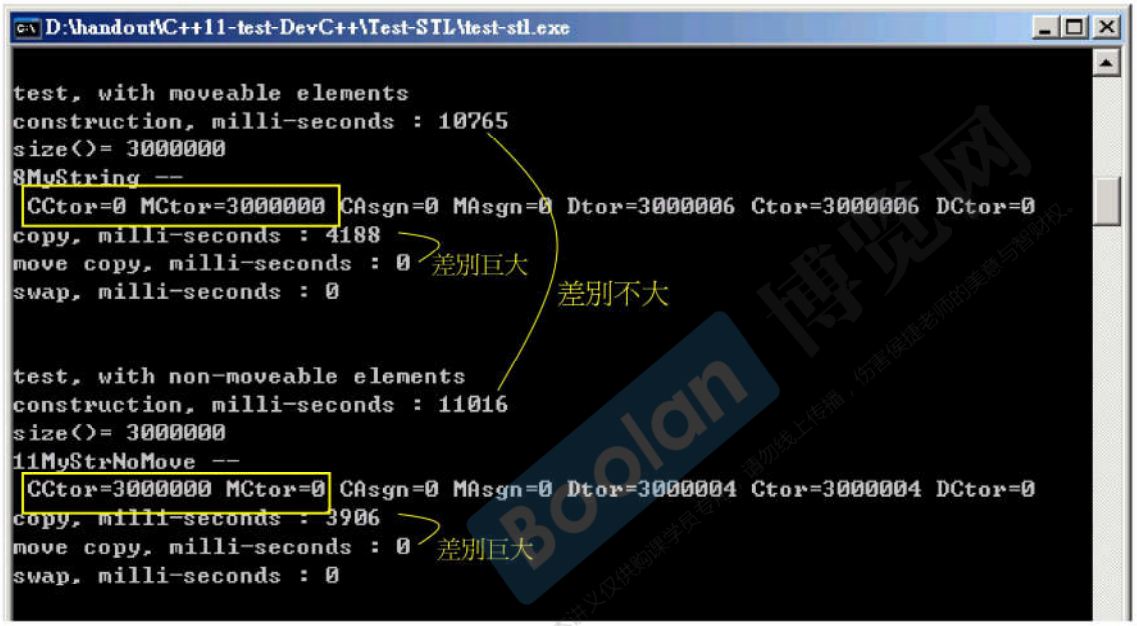
差別巨大

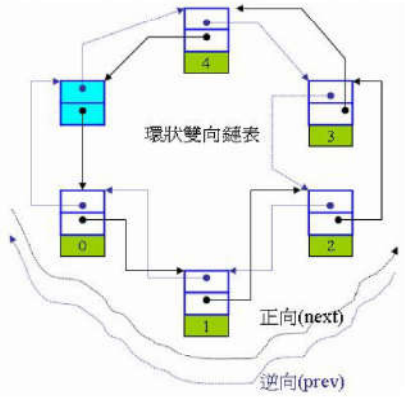
```
for(long i=0; i< value; ++i) {
    snprintf(buf, 10, "%d", rand());
    auto ite = c1.end();
    c1.insert(ite, V1type(buf));
}
```




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

moveable 元素對於 list 速度效能的影響





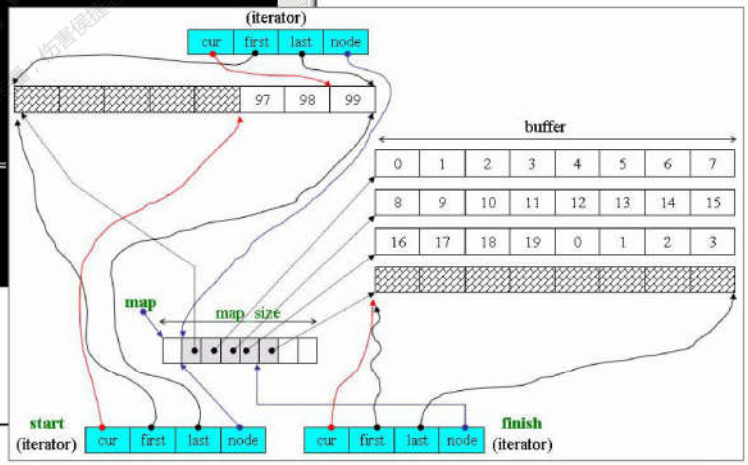
moveable 元素對於 deque 速度效能的影響

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

test, with moveable elements
construction, milli-seconds : 9078
size()= 3000000
8MyString --
 CCtor=0 MCtor=3000000 CAsgn=0 MAsgn=0 Dtor=3000006 Ctor=3000006 DCtor=0
copy, milli-seconds : 3031
move copy, milli-seconds : 0
swap, milli-seconds : 0


test, with non-moveable elements
construction, milli-seconds : 9844
size()= 3000000
11MyStrNoMove --
 CCtor=3000000 MCtor=0 CAsgn=0 MAsgn=0 Dtor=3000004 Ctor=
copy, milli-seconds : 2437
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

差別巨大

差別不大

差別巨大



— 侯捷 —

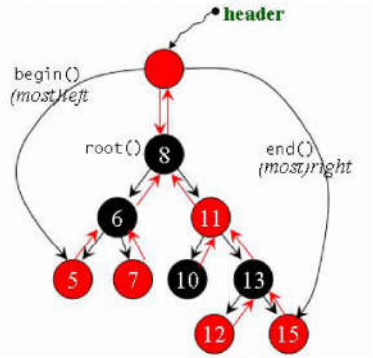
moveable 元素對於 multiset 速度效能的影響

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

test, with moveable elements
construction, milli-seconds : 74125
size()= 3000000
8MyString --
 CCtor=0 MCtor=3000000 CAsgn=0 MAsgn=0 Dtor=3000006 Ctor=3000006 DCtor=0
copy, milli-seconds : 5438
move copy, milli-seconds : 0
swap, milli-seconds : 0


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

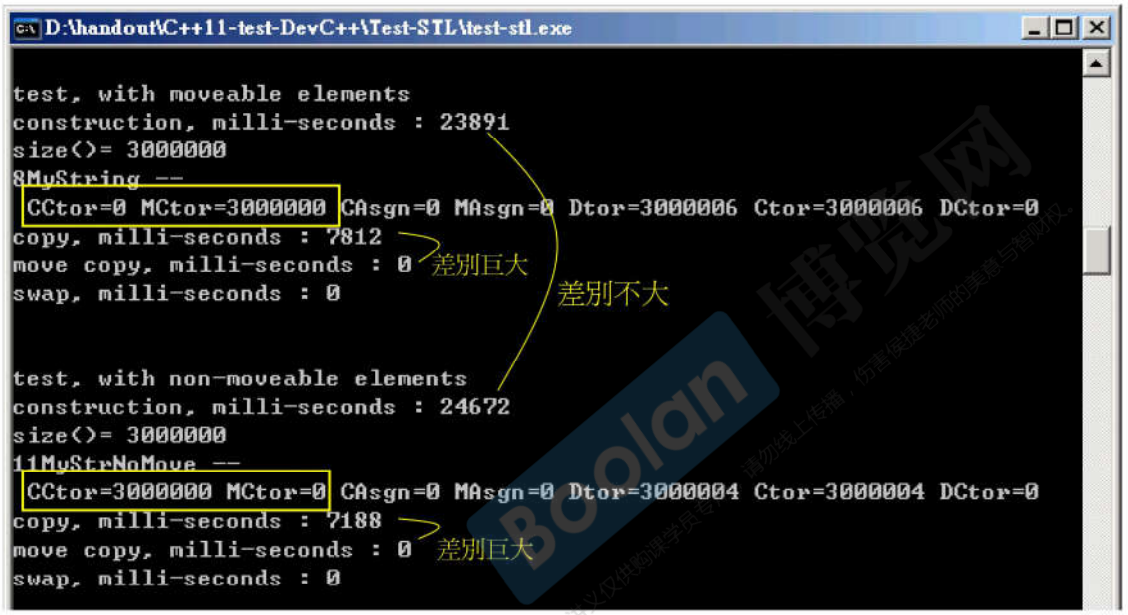
差別巨大

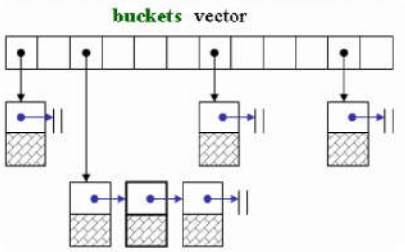
差別不大

差別巨大



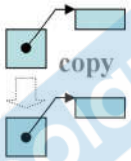
moveable 元素對於 unordered_multiset 速度效能的影響



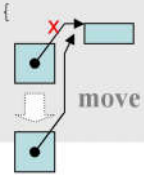


寫一個 moveable class

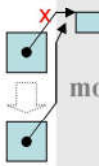
```
class MyString {
public:
    static size_t DCtor;   //累計 default-ctor 呼叫次數
    static size_t Ctor;    //累計 ctor 呼叫次數
    static size_t CCtor;   //累計 copy-ctor 呼叫次數
    static size_t CAsgn;   //累計 copy-asgn 呼叫次數
    static size_t MCtor;   //累計 move-ctor 呼叫次數
    static size_t MAsgn;   //累計 move-asgn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

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




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

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

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



寫一個 moveable class




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

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

char* get() const { return _data; }
};
size_t MyString::DCtor=0;
size_t MyString::Ctor=0;
size_t MyString::CCtor=0;
size_t MyString::CAsgn=0;
size_t MyString::MCtor=0;
size_t MyString::MAsgn=0;
size_t MyString::Dtor=0;

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

測試函數

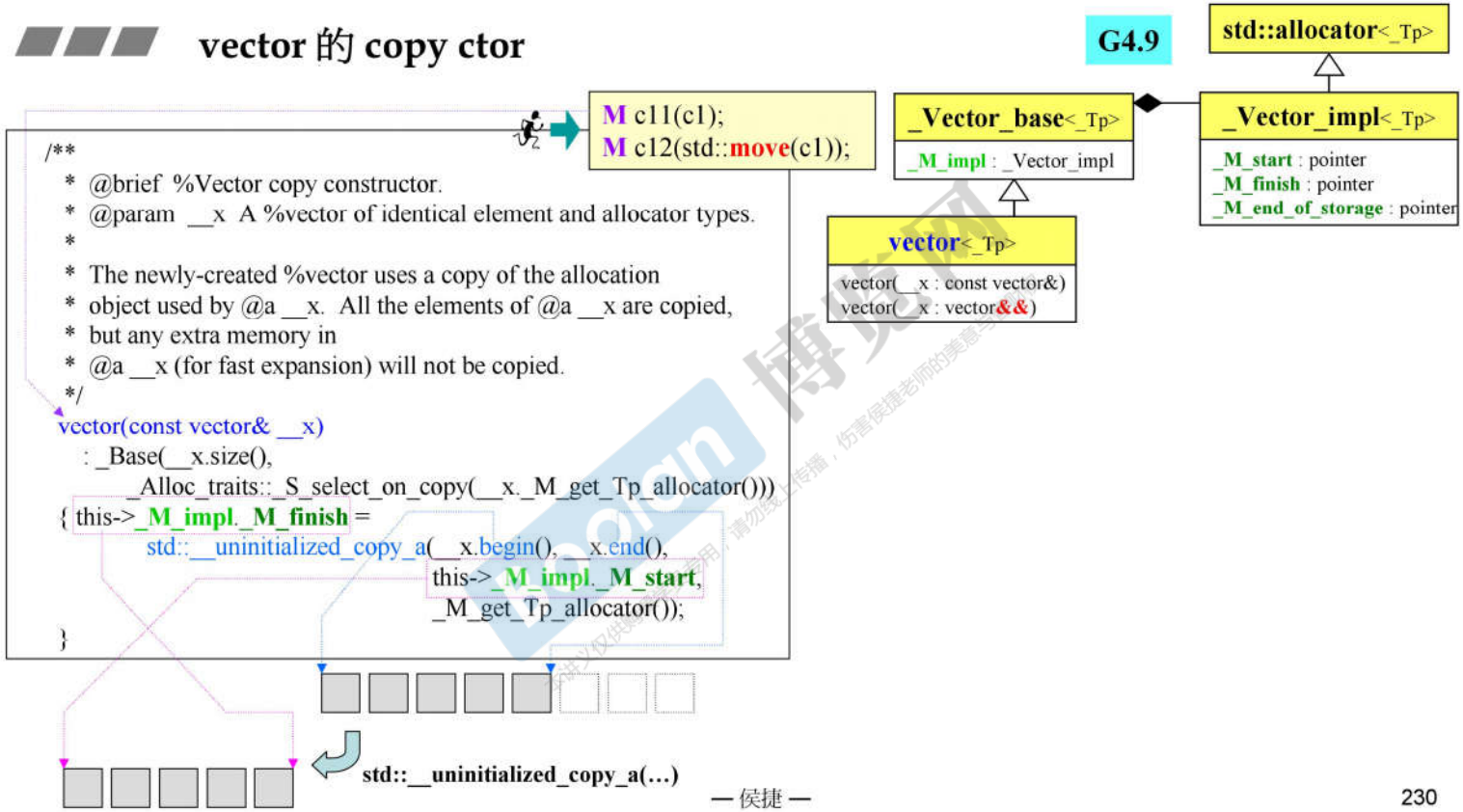
```
test, with moveable elements
construction, milli-seconds : 8547
size()= 3000000
8MyString --
 CCtor=0 MCtor=7194303 CAsgn=0 MAsgn=
copy, milli-seconds : 3500
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

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

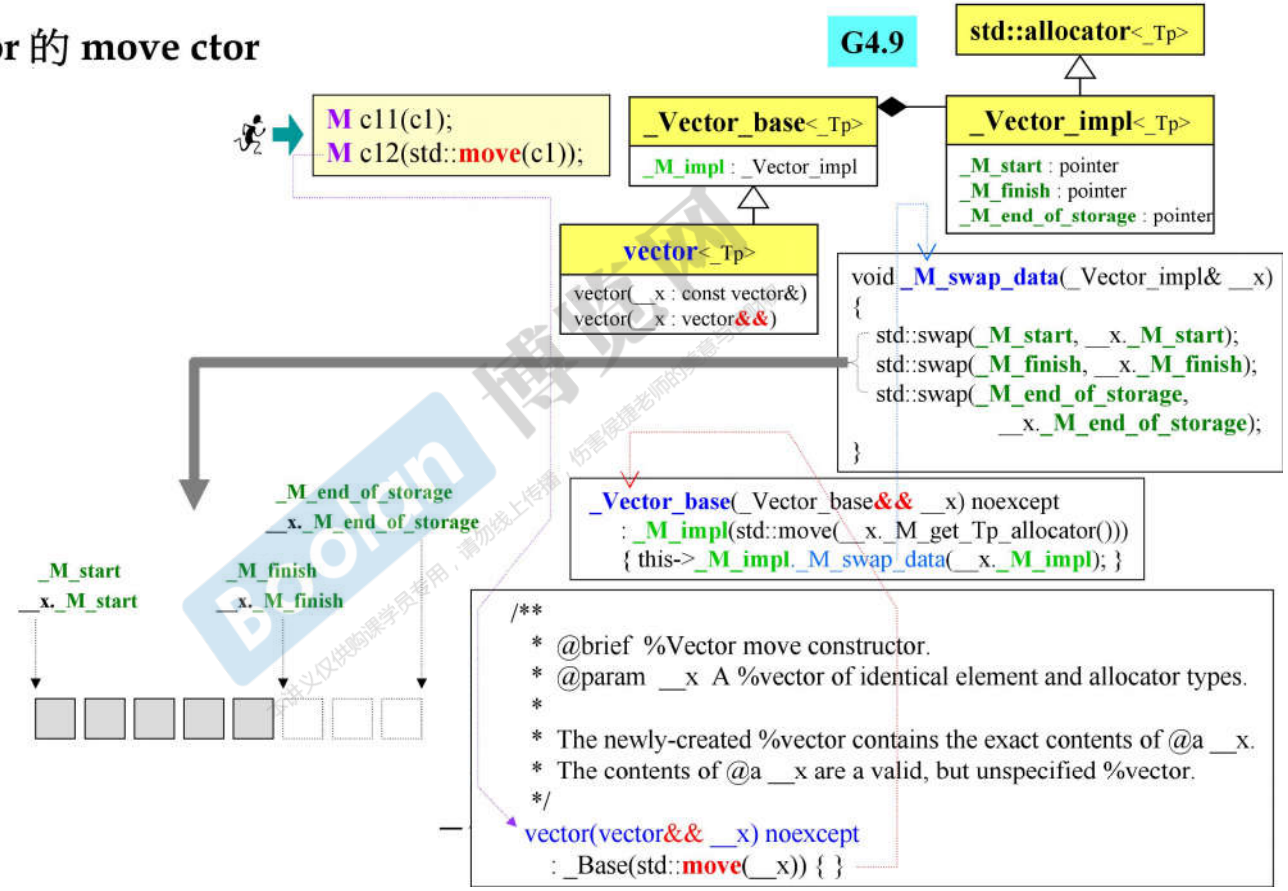
```
template<typename M, typename NM>
void test_moveable(M c1, NM c2, long& value)
{
char buf[10];

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

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

2 //測試 non-moveable
…
}
```

```
test_moveable(vector<MyString>(),
              vector<MyStrNoMove>(),
              value);
```

vector 的 copy ctor



vector 的 move ctor



std::string 是否 moveable？

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

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

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

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

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

The End