STL算法：用来处理一个或多个iterator区间。

for\_each()：允许以不同的方式访问、处理、修改每一个元素。

例：程序stl\_test64

*vector*<int> coll;

InsertElements(coll, 1, 9);

*for\_each*(coll.*begin*(), coll.*end*(), [](int& elem) {

elem += 10;

});

PrintElements(coll);

*for\_each*(coll.*begin*(), coll.*end*(), [=](int& elem) {

elem += \*coll.*begin*();

});

PrintElements(coll);

输出为：

11 12 13 14 15 16 17 18 19

22 23 24 25 26 27 28 29 30

非更易型算法：不改变元素值，也不改动元素次序。

1. 元素计数count()：线性复杂度

difference\_type

count(Iterator beg, Iterator end, const T& value)

计算[beg, end)中“元素值等于value”的元素个数。

difference\_type

count\_if(Iterator beg, Iterator end, UnaryPredicate op)

计算[beg, end)中“令unary predicate op(elem)结果为true”的元素个数。

difference\_type表示iterator间距的类型

例：程序stl\_test65

// 元素计数count

*vector*<int> coll;

InsertElements(coll, 1, 9);

PrintElements(coll, "coll: ");

int number = *count*(coll.*cbegin*(), coll.*cend*(), 4);

*cout* << "number of elements equal to 4: " << number << *endl*;

number = *count\_if*(coll.*cbegin*(), coll.*cend*(), [](int elem) {

return elem % 2 == 0;

});

*cout* << "number of elements with even value: " << number << *endl*;

number = *count\_if*(coll.*cbegin*(), coll.*cend*(), [](int elem) {

return elem > 4;

});

*cout* << "number of elements greater than 4: " << number << *endl*;

输出为：

coll: 1 2 3 4 5 6 7 8 9

number of elements equal to 4: 1

number of elements with even value: 4

number of elements greater than 4: 5

1. 最小值和最大值：min\_element和max\_element，线性复杂度

Iterator min\_element(Iterator beg, Iterator end)

Iterator min\_element(Iterator beg, Iterator end, ComFunc op)

Iterator max\_element(Iterator beg, Iterator end)

Iterator max\_element(Iterator beg, Iterator end, ComFunc op)

pair<Iterator, Iterator> minmax\_element(Iterator beg, Iterator end)

pair<Iterator, Iterator> minmax\_element(Iterator beg, Iterator end, ComFunc op)

注意：如果存在多个最小值或最大值，min\_element()和max\_element()返回其所找到的第1个目标元素。minmax\_element()返回第1个最小元素和最后1个最大元素。所以此时max\_element()和minmax\_element()返回的最大元素不是同一个。

例：程序stl\_test65

bool AbsLess(int elem1, int elem2)

{

return *abs*(elem1) < *abs*(elem2);

}

// 最小值和最大值

*deque*<int> coll1;

InsertElements(coll1, 2, 6);

InsertElements(coll1, -3, 6);

*cout* << "min and max: " << *endl*;

PrintElements(coll1, "coll1: ");

*cout* << "minimum: "

<< \**min\_element*(coll1.*cbegin*(), coll1.*cend*()) << *endl*;

*cout* << "maximum: "

<< \**max\_element*(coll1.*cbegin*(), coll1.*cend*()) << *endl*;

auto mm = *minmax\_element*(coll1.*cbegin*(), coll1.*cend*());

// 第1个最小元素

*cout* << "min: " << \*(mm.*first*) << *endl*;

// 最后一个最大元素

*cout* << "max: " << \*(mm.*second*) << *endl*;

*cout* << "distance between min and max: " << *distance*(mm.*first*, mm.*second*) << *endl*;

*cout* << "minimun of absolute values: "

<< \**min\_element*(coll1.*cbegin*(), coll1.*cend*(), AbsLess) << *endl*;

*cout* << "maximum of absolute values: "

<< \**max\_element*(coll1.*cbegin*(), coll1.*cend*(), AbsLess) << *endl*;

输出为：

min and max:

coll1: 2 3 4 5 6 -3 -2 -1 0 1 2 3 4 5 6

minimum: -3

maximum: 6

min: -3

max: 6

distance between min and max: 9

minimun of absolute values: 0

maximum of absolute values: 6

1. 查找元素：find

Iterator find(Iterator beg, InputIterator end, const T& value)

Iterator find\_if(Iterator beg, Iterator end, UnaryPredicate op)

Iterator find\_if\_not(Iterator beg, Iterator end, UnaryPredicate op)

返回[beg, end)区间中第1个“造成unary predicate op(element)结果为false”的元素。

如果没有结果，返回end。

例：程序stl\_test65

// 查找元素find

*cout* << "find " << *endl*;

*list*<int> coll2;

InsertElements(coll2, 1, 9);

InsertElements(coll2, 1, 9);

PrintElements(coll2, "coll2: ");

//find first element with value 4

*list*<int>::*iterator* pos1 = *find*(coll2.*begin*(), coll2.*end*(), 4);

//find second element with value 4

*list*<int>::*iterator* pos2;

if (pos1 != coll2.*cend*())

{

pos2 = *find*(*next*(pos1), coll2.*end*(), 4);

}

// print all elements from first to second 4

if (pos1 != coll2.*end*() && pos2 != coll2.*end*())

{

*copy*(pos1, *next*(pos2), *ostream\_iterator*<int>(*cout*, " "));

*cout* << *endl*;

}

*cout* << *endl*;

*vector*<int> coll3 = {1, 2, 3, 4, 5, 6, 7, 8, 9};

PrintElements(coll3, "coll3: ");

// find first element greater than 3

*vector*<int>::*iterator* pos = *find\_if*(coll3.*begin*(), coll3.*end*(),

*bind*(*greater*<int>(), *\_1*, 3));

*cout* << "the "

<< *distance*(coll3.*begin*(), pos) + 1

<< ". element is the first greater than 3" << *endl*;

// find first element divisible by 3

pos = *find\_if*(coll3.*begin*(), coll3.*end*(), [](int elem) {

return elem % 3 == 0;

});

*cout* << "the "

<< *distance*(coll3.*begin*(), pos) + 1

<< ". element is the first divisible by 3" << *endl*;

// find first element not < 5

pos = *find\_if\_not*(coll3.*begin*(), coll3.*end*(), *bind*(*less*<int>(), *\_1*, 5));

*cout* << "first value >= 5: " << \*pos << *endl*;

输出为：

find

coll2: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

4 5 6 7 8 9 1 2 3 4

coll3: 1 2 3 4 5 6 7 8 9

the 4. element is the first greater than 3

the 3. element is the first divisible by 3

first value >= 5: 5

1. 查找前n个连续匹配值：search\_n，线性复杂度

Iterator search\_n(Iterator beg, Iterator end, Size count, const T& value)

返回[beg, end)内“连续count个元素都等于value”中的第1个元素位置。

Iterator search\_n(Iterator beg, Iterator end, Size count, const T& value, BinaryPredicate op)

返回[beg, end)内“连续count个元素造成以下binary predicate结果为true”中第1个元素位置：

op(elem, value)

例：stl\_test66

//查找前n个连续匹配值search\_n

*deque*<int> coll = {1, 2, 7, 7, 6, 3, 9, 5, 7, 7, 7, 3, 6};

*deque*<int>::*iterator* pos = *search\_n*(coll.*begin*(), coll.*end*(), 3, 7);

if (pos != coll.*end*())

{

*cout* << "first three consecutive elements with value 7 "

<< "start with " << *distance*(coll.*begin*(), pos) + 1

<< ". element" << *endl*;

}

else {

*cout* << "no three consecutive elements with value 7 for found" << *endl*;

}

// find four consecutive odd elements

// 自定义的Lambda必须接收两个参数

// search\_n()接收binary predicate op

// 而不是unary predicate

pos = *search\_n*(coll.*begin*(), coll.*end*(),

4,

0,

[](int elem, int value) {

return elem % 2 == 1;

});

if (pos != coll.*end*())

{

*cout* << "first four consecutive odd elements are: ";

for (int i = 0; i < 4; ++i, ++pos)

{

*cout* << \*pos << ' ';

}

}

else

{

*cout* << "no four consecutive odd elements found";

}

输出为：

first three consecutive elements with value 7 start with 9. element

first four consecutive odd elements are: 3 9 5 7

1. 查找第1个子区间：search，线性复杂度

Iterator1 search(Iterator1 beg, Iterator1 end,

Iterator2 searchBeg, Iterator2 searchEnd)

Iterator1 search(Iterator1 beg, Iterator1 end,

Iterator2 searchBeg, Iterator2 searchEnd, BinaryPredicate op)

返回[beg, end)区间内“与[searchBeg, searchEnd)区间完全吻合”的第1个子区间的第1个元素的位置。

第一形式中，子区间内的元素必须完全等于[searchBeg, searchEnd)的元素。

第二形式中，子区间内的元素和[searchBeg, searchEnd)的对应元素必须造成以下binary predicate的结果为true：

op(elem, searchElem)

查找不到，返回end。

例：程序stl\_test66

// 查找第1个子区间search

*deque*<int> coll1;

*list*<int> sub\_coll;

InsertElements(coll1, 1, 7);

InsertElements(coll1, 1, 7);

InsertElements(sub\_coll, 3, 6);

PrintElements(coll1, "coll1: ");

PrintElements(sub\_coll, "sub coll: ");

// search first occurrence

*deque*<int>::*iterator* pos1 = *search*(coll1.*begin*(), coll1.*end*(), // range

sub\_coll.*begin*(), sub\_coll.*end*()); // subrange

while (pos1 != coll1.*end*())

{

*cout* << "subcoll found starting with element "

<< *distance*(coll1.*begin*(), pos1) + 1

<< *endl*;

// search next occurrence of subcoll

++pos1;

pos1 = *search*(pos1, coll1.*end*(),

sub\_coll.*begin*(), sub\_coll.*end*());

}

输出为：

coll1: 1 2 3 4 5 6 7 1 2 3 4 5 6 7

sub coll: 3 4 5 6

subcoll found starting with element 3

subcoll found starting with element 10

例：程序stl\_test66

*vector*<int> coll2;

InsertElements(coll2, 1, 9);

PrintElements(coll2, "coll2: ");

// check for "even odd even"

bool check\_even\_args[3] = {true, false, true};

// CheckEven(\*coll2::iterator, check\_even\_args[i])

// 返回true才是要查找的元素

// check\_even\_args对应的end为check\_even\_args + 3

*vector*<int>::*iterator* pos2 = *search*(coll2.*begin*(), coll2.*end*(),

check\_even\_args, check\_even\_args + 3, CheckEven);

while (pos2 != coll2.*end*())

{

*cout* << "subrange found starting with element "

<< *distance*(coll2.*begin*(), pos2) + 1

<< *endl*;

++pos2;

pos2 = *search*(pos2, coll2.*end*(),

check\_even\_args, check\_even\_args + 3,

CheckEven);

}

输出为：

coll2: 1 2 3 4 5 6 7 8 9

subrange found starting with element 2

subrange found starting with element 4

subrange found starting with element 6

1. 查找最后一个子区间：find\_end()，线性复杂度。

Iterator1 find\_end(Iterator1 beg, Iterator1 end,

Iterator2 searchBeg, Iterator2 searchEnd)

Iterator1 find\_end(Iterator1 beg, Iterator1 end,

Iterator2 searchBeg, Iterator2 searchEnd,

BinaryPredicate op)

返回[beg, end)区间中“和区间[searchBeg, searchEnd)完全吻合”的最后一个子区间内的第1元素位置。

第一形式中，子区间内的元素必须完全等于[searchBeg, searchEnd)的元素。

第二形式中，子区间内的元素和[searchBeg, searchEnd)的对应元素必须造成以下binary predicate的结果为true：

op(elem, searchElem)

查找不到，返回end。

1. 检验相等性：equal()，线性复杂度

bool equal(Iterator1 beg, Iterator1 end, Iterator2 cmpBeg)

bool equal(Iterator1 beg, Iterator1 end, Iterator2 cmpBeg, BinaryPredicate op)

第1形式判断[beg, end)区间内的元素是否都和“以cmpBeg开头的区间”内的元素相等。

第2形式判断[beg, end)区间内的元素和“以cmpBeg开头的区间”内的对应元素是否都能够造成以下binary predicate为true：

op(elem, cmpElem)

注意：必须确保“以cmpBeg开头”的区间内含足够元素（如果元素不足够貌似在Visual Studio上也没什么影响）。

例：程序stl\_test67

bool BothEvenOrOdd(int elem1, int elem2)

{

return elem1 % 2 == elem2 % 2;

}

// equal

*vector*<int> coll1;

*list*<int> coll2;

InsertElements(coll1, 1, 7);

InsertElements(coll2, 3, 9);

PrintElements(coll1, "coll1: ");

PrintElements(coll2, "coll2: ");

if (*equal*(coll1.*begin*(), coll1.*end*(), coll2.*begin*()))

{

*cout* << "coll1 == coll2" << *endl*;

}

else

{

*cout* << "coll1 != coll2" << *endl*;

}

if (*equal*(coll1.*begin*(), coll1.*end*(), coll2.*begin*(), BothEvenOrOdd))

{

*cout* << "even and odd elements correspond" << *endl*;

}

else

{

*cout* << "even and odd elements do not correspond" << *endl*;

}

coll1: 1 2 3 4 5 6 7

coll2: 3 4 5 6 7 8 9

coll1 != coll2

even and odd elements correspond

1. 测试不定序之相等性：is\_permuation，最坏情况是二次的。

bool is\_permutation(Iterator1 beg1, Iterator1 end1, Iterator2 beg2)

bool is\_permutation(Iterator1 beg1, Iterator1 end1, Iterator2 beg2, CompFunc op)

检查[beg1, end1)区间内的元素是否为beg2起始之区间元素的一个排列组合（a permutation）；即“顺序无所谓”的情况下两区间的元素是否相等，beg2起始区间的元素有可能多于[beg1, end1)也会相等。

第2形式使用binary predicate op(elem1, elem2)比较元素，应该在“elem1等于elem2”时返回true。

注意：所有iterator必须具有相同的value type（即所指向的元素的类型）

例：程序stl\_test67

// is\_permutation

*vector*<int> coll3 = {1, 1, 2, 3, 4, 5, 6, 7, 8, 9};

*list*<int> coll4 = {1, 9, 8, 7, 6, 5, 4, 3, 2, 1, 10};

*deque*<int> coll5 = {11, 12, 13, 19, 18, 17, 16, 15, 14, 11};

PrintElements(coll3, "coll3: ");

PrintElements(coll4, "coll4: ");

PrintElements(coll5, "coll5: ");

if (*is\_permutation*(coll3.*cbegin*(), coll3.*cend*(), // first range

coll4.*cbegin*())) // second range

{

*cout* << "coll3 and coll4 have equal elements" << *endl*;

}

else

{

*cout* << "coll3 and coll4 don't have equal elements" << *endl*;

}

if (*is\_permutation*(coll3.*cbegin*(), coll3.*cend*(),

coll5.*cbegin*(), BothEvenOrOdd))

{

*cout* << "numbers of even and odd elements match" << *endl*;

}

else

{

*cout* << "numbers of even and odd elements don't match" << *endl*;

}

输出为：

coll3: 1 1 2 3 4 5 6 7 8 9

coll4: 1 9 8 7 6 5 4 3 2 1 10

coll5: 11 12 13 19 18 17 16 15 14 11

coll3 and coll4 have equal elements

numbers of even and odd elements match

1. 查找第1处不同：mismatch，线性复杂度

pair<Iterator1, Iterator2> mismatch(Iterator1 beg, Iterator1 end, Iterator2 cmpBeg)

pair<Iterator1, Iterator2> mismatch(Iterator1 beg, Iterator1 end, Iterator2 cmpBeg, BinaryPredicate op)

第1形式返回[beg, end)区间和“以cmpBeg起始的区间”内第1组两两相异的对应元素。

第2形式返回[beg, end)区间和“以cmpBeg起始的区间”内第1组“造成以下binary predicate结果为false”的对应元素：

op(elem, cmpElem)

注意：如果没有相异点，返回“以end和第二序列对应的元素组成”的pair，这并不意味着两序列相等，有可能第二序列含有较多元素。

注意：以cmpBeg起始的区间元素个数必须大于等于[beg,end)的元素个数，否则会抛出异常。

例：程序stl\_test67

// mismatch

*vector*<int> coll6 = {1, 2, 3, 4, 5, 6};

*list*<int> coll7 = {1, 2, 4, 8, 16, 3};

PrintElements(coll6, "coll6: ");

PrintElements(coll7, "coll7: ");

auto values = *mismatch*(coll6.*cbegin*(), coll6.*cend*(), coll7.*cbegin*());

if (values.*first* == coll6.*end*())

{

*cout* << "no mismatch" << *endl*;

}

else

{

*cout* << "first mismatch: "

<< \*values.*first* << " and "

<< \*values.*second* << *endl*;

}

values = *mismatch*(coll6.*cbegin*(), coll6.*cend*(), coll7.*cbegin*(), *less\_equal*<int>());

if (values.*first* == coll6.*end*())

{

*cout* << "always less-or-equal" << *endl*;

}

else

{

*cout* << "not less-or-equal: "

<< \*values.*first* << " and "

<< \*values.*second* << *endl*;

}

输出为：

coll6: 1 2 3 4 5 6

coll7: 1 2 4 8 16 3

first mismatch: 3 and 4

not less-or-equal: 6 and 3

1. 以字典序检验小于：lexicographical\_compare，线性复杂度

bool lexicographical\_compare(Iterator1 beg1, Iterator1 end1, Iterator2 beg2, Iterator2 end2)

bool lexicographical\_compare(Iterator1 beg1, Iterator1 end1, Iterator2 beg2, Iterator2 end2, CompFunc op)

判断[beg1, end1)区间内的元素是否小于[beg2, end2)的元素。所谓“小于”指“字典（lexicographical）次序”意义而言。

第1形式以operator <比较元素。

第2形式以binary predicate

op(elem1, elem2)

如果elem1小于elem2， 返回true。

字典次序：

1. 如果两元素不相等，则这两个元素的比较结果就是整个两序列的比较结果；
2. 如果两序列的元素数量不同，则元素较少的序列小于另一个序列；如果第1序列的元素数量较少，比较结果为true；
3. 如果两序列都没有更多的元素进行比较，则这两个序列相等，比较结果返回false。

例：程序stl\_test68

void PrintList(const list<int>& list)

{

PrintElements(list);

}

bool LessForCollection(const list<int>& list1, const list<int>& list2)

{

return *lexicographical\_compare*(list1.*cbegin*(), list1.*cend*(),

list2.*cbegin*(), list2.*cend*());

}

int main(int argc, char\* argv[])

{

list<int> c1, c2, c3, c4;

InsertElements(c1, 1, 5);

c4 = c3 = c2 = c1;

c1.*push\_back*(7);

c3.*push\_back*(2);

c3.*push\_back*(0);

c4.*push\_back*(2);

*vector*<list<int>> cc;

cc.*insert*(cc.*begin*(), {c1, c2, c3, c4, c3, c1, c4, c2});

*cout* << "original: " << *endl*;

*for\_each*(cc.*cbegin*(), cc.*cend*(), PrintList);

*cout* << *endl*;

*cout* << "sort: " << *endl*;

*sort*(cc.*begin*(), cc.*end*(), LessForCollection);

*for\_each*(cc.*cbegin*(), cc.*cend*(), PrintList);

return 0;

}

输出为：

original:

1 2 3 4 5 7

1 2 3 4 5

1 2 3 4 5 2 0

1 2 3 4 5 2

1 2 3 4 5 2 0

1 2 3 4 5 7

1 2 3 4 5 2

1 2 3 4 5

sort:

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5 2

1 2 3 4 5 2

1 2 3 4 5 2 0

1 2 3 4 5 2 0

1 2 3 4 5 7