2-迭代器-iterator

----------比特科技整理----------

【预习：《STL源码剖析》-- 第3章 迭代器概念与traits编程技巧】

- 迭代器

迭代器模式--提供一种方法，使之能够依序寻访某个聚合物（容器）所含的各个元素，而又无需暴露该聚合物的内部表达方式。

STL的中心思想在于：将数据容器和算法分开，彼此独立设计，最后再以一贴胶合剂（iterator）将它们撮合在一起。

STL的迭代器是一个可遍历STL容器全部或者部分数据。

http://www.cplusplus.com/reference/iterator/

- 迭代器的使用

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

void Test1 ()
{
    vector<int > v1;
    v1.push_back (5);
    v1.push_back (4);
    v1.push_back (3);
    v1.push_back (2);
    v1.push_back (1);

    // 迭代器遍历顺序表
    vector<int >::iterator it = v1 .begin();
    for (; it != v1. end(); ++it )
    {
        cout<<*it <<" ";
    }
    cout<<endl ;

    // STL的排序算法
    sort(v1 .begin(), v1.end ());

    it = v1 .begin();
    for (; it != v1. end(); ++it )
    {
        cout<<*it <<" ";
    }
    cout<<endl ;
}

void Test2 ()
{
    list<string > l1;
    l1.push_back ("xjh");
    l1.push_back ("zpw");
    l1.push_back ("yb");
    l1.push_back ("whb");

    // 迭代器遍历链表
    list<string >::iterator it = l1 .begin();
    for (; it != l1. end(); ++it )
    {
        cout<<*it <<" ";
    }
    cout<<endl ;
```

```
    // STL的替换算法
    replace(l1.begin(), l1.end(), "xjh", "lcf");

    it = l1.begin();
    for (; it != l1.end(); ++it)
    {
        cout<<*it<<" ";
    }
    cout<<endl;
}

void Test3()
{
    list<string> l1;
    l1.push_back("xjh");
    l1.push_back("zpw");
    l1.push_back("yb");
    l1.push_back("whb");

    //迭代器遍历链表
    list<string>::iterator it = l1.begin();
    for (; it != l1.end(); ++it)
    {
        cout<<*it<<" ";
    }
    cout<<endl;

    // STL的find算法查找迭代器区间的数据，并返回找到节点的迭代器
    it = find(l1.begin(), l1.end(), "yb");
    if (it != l1.end())
    {
        cout<<"find success:"<<*it<<endl;

        //通过迭代器修改节点数据
       *it = "yls";
    }

    it = find(l1.begin(), l1.end(), "yb");
    if (it == l1.end())
    {
        cout<<"find failed"<<endl;
    }
}
```

- 什么是迭代器失效

```
void PrintVector(vector<int>& v)
{
    vector<int>::iterator it = v.begin();
    for (; it != v.end(); ++it)
    {
        cout<<*it<<" ";
    }
    cout<<endl;
}

void Test1()
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(3);
    v1.push_back(4);
    v1.push_back(5);
```

```
    v1.push_back(6);
    v1.push_back(7);
    v1.push_back(8);

    PrintVector(v1);

    //迭代器失效
    vector<int>::iterator it=v1.begin();
    while(it!=v1.end())
    {
        if(*it% 2 == 0)
            it=v1.erase(it);
        else
            ++it;
    }
    PrintVector(v1);
}

void PrintList(list<int>& l1)
{
    list<int>::iterator it=l1.begin();
    for(;it!=l1.end(); ++it)
    {
        cout<<*it<<" ";
    }
    cout<<endl;
}

void Test2()
{
    list<int> l1;
    l1.push_back(1);
    l1.push_back(2);
    l1.push_back(3);
    l1.push_back(4);
    l1.push_back(5);
    l1.push_back(6);
    l1.push_back(7);
    l1.push_back(8);
    PrintList(l1);

    //迭代器失效
    list<int>::iterator it=l1.begin();
    while(it!=l1.end())
    {
        if(*it% 2 == 0)
            it=l1.erase(it);
        else
            ++it;
    }
```

```
    PrintList(l1);
}
```

- 深度剖析迭代器原理

迭代器是一种行为类似智能指针的对象，而指针最常见的行为就是内容提领和成员访问。因此迭代器最重要的行为就是对operator*和operator->进行重载。

【模拟实现简化版List迭代器】

```
#pragma once
#include <assert.h>

template <class T>
struct __ListNode
{
    __ListNode()
        : _next(NULL)
        , _prev(NULL)
    {}

    __ListNode(const T& x)
        : _data(x)
        , _next(NULL)
        , _prev(NULL)
    {}

    T _data;                    // 结点数据
    __ListNode<T>* _prev;       // 指向前一个结点的指针
    __ListNode<T>* _next;       // 指向后一个结点的指针
};

// List 的迭代器
template <class T>
class __ListIterator
{
public:
    typedef __ListIterator<T> Iterator;

    typedef T ValueType;
    typedef T* Pointer;
    typedef T& Reference;
    typedef __ListNode<T>* LinkType;

    // 指向节点的指针
    LinkType _node;

    __ListIterator(LinkType node = NULL)
        : _node(node)
    {}

    bool operator==(const __ListIterator<T>& x) const
    {
        return _node == x._node;
    }
    bool operator!=(const __ListIterator<T>& x) const
    {
        return _node != x._node;
    }

    Reference operator*() const
    {
        return _node->_data;
    }

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

    __ListIterator<T>& operator++()
    {
        _node = _node->_next;
```

```
        return *this ;
    }

    __ListIterator<T > operator++( int)
    {
        __ListIterator<T > tmp( _node);
        _node = _node ->_next;
        return tmp ;
    }

    __ListIterator<T >& operator--()
    {
        _node = _node ->_prev;
        return *this ;
    }

    __ListIterator<T >& operator--( int)
    {
        __ListIterator<T > tmp( _node);
        _node = _node ->_prev;
        return tmp ;
    }
};

//
// 设计为双向循环链表
//
template <class T>
class List
{
public :
    typedef __ListIterator <T> Iterator;

    typedef T ValueType;
    typedef __ListNode <T>* LinkType;

    List()
    {
        _head._prev = &_head;
        _head._next = &_head;
    }

    ~List()
    {
        Clear();
    }

public :
    // 在pos 前插入一个节点
    void Insert (Iterator pos, const ValueType& x)
    {
        LinkType tmp = new __ListNode<T >(x);
        LinkType prev = pos. _node->_prev ;
        LinkType cur = pos. _node;

        prev->_next = tmp;
        tmp->_prev = prev;

        tmp->_next = cur;
        cur->_prev = tmp;
    }

    void PushBack (const T& x )
    {
        Insert(End (), x);
    }

    void PushFront (const T& x )
    {
        Insert(Begin (), x);
    }

    // 删除pos 指向的节点，返回 pos之后的一个节点
    Iterator Erase (Iterator pos)
    {
```

```
        LinkType prev = pos._node->_prev;
        LinkType next = pos._node->_next;

        prev->_next = next;
        next->_prev = prev;

        delete pos._node;

        return Iterator(next);
    }

    void PopBack()
    {
        Erase(--End());
    }

    void PopFront()
    {
        Erase(Begin());
    }

    Iterator Begin()
    {
        return _head._next;
    }

    Iterator End()
    {
        return &_head;
    }

    void Clear()
    {
        Iterator begin = Begin();
        while(begin != End())
        {
            LinkType tmp = begin._node;
            ++begin;
            delete tmp;
        }
    }

private:
    //
    // 哨兵位的头结点，方便作为迭代器的 end
    //
    __ListNode<T> _head;
};
```

- 【模拟实现简化版Vector迭代器】

```
#pragma once
#include <assert.h>

template <class T>
class Vector
{
public:
    typedef T ValueType;
    typedef ValueType* Pointer;
    typedef ValueType* Iterator;
    typedef ValueType& Reference;

    Iterator Begin() { return _start; }
    Iterator End() { return _finish; }
    size_t Size()
    {
        return _finish - _start;
    }

    size_t Capacity()
    {
        return _endOfStorage - _start;
```

```
    }

    Vector()
        : _start(NULL)
        , _finish(NULL)
        , _endOfStorage(NULL)
    {}

    void _CheckExpand()
    {
        if (_finish == _endOfStorage)
        {
            size_t size = Size();
            size_t capacity = size*2 + 3;
            T* tmp = new T[capacity];
            if (_start)
                memcpy(tmp, _start, sizeof(T)*size);

            _start = tmp;
            _finish = _start + size;
            _endOfStorage = _start + capacity;
        }
    }

    void PushBack(const T& x)
    {
        _CheckExpand();

        assert(_finish != _endOfStorage);

        *_finish = x;
        ++_finish;
    }

    void PopBack()
    {
        --_finish;
    }

    // 返回删除数据的下一个数据
    Iterator Erase(Iterator pos)
    {
        // 拷贝数据
        Iterator begin = pos + 1;
        while(begin != _finish)
        {
            *(begin - 1) = *begin;
            ++begin;
        }

        --_finish;
        return pos;
    }

private:
    Iterator _start;          // 指向数据块的开始
    Iterator _finish;         // 指向有效数据的尾
    Iterator _endOfStorage;// 指向存储容量的尾
};
```

- 什么是迭代器失效？

```
void PrintList(List<int>& l1)
{
    List<int>::Iterator it = l1.Begin();
    for(; it != l1.End(); ++it)
    {
        cout<<*it<<" ";
    }
    cout<<endl;
}

void Test1()
```

```
{
    List<int> l1;
    l1.PushBack(1);
    l1.PushBack(2);
    l1.PushBack(3);
    l1.PushBack(4);
    l1.PushBack(5);
    l1.PushBack(6);
    l1.PushBack(7);
    l1.PushBack(8);
    PrintList(l1);

    //迭代器失效
    List<int>::Iterator it = l1.Begin();
    while(it != l1.End())
    {
        if (*it % 2 == 0)
            it = l1.Erase(it);
        else
            ++it;
    }
    PrintList(l1);
}

void PrintVector(Vector<int>& v)
{
    Vector<int>::Iterator it = v.Begin();
    for (; it != v.End(); ++it)
    {
        cout<<*it<<" ";
    }
    cout<<endl;
}

void Test2()
{
    Vector<int> v1;
    v1.PushBack(1);
    v1.PushBack(2);
    v1.PushBack(3);
    v1.PushBack(4);
    v1.PushBack(5);
    v1.PushBack(6);
    v1.PushBack(7);
    v1.PushBack(8);

    PrintVector(v1);

    //迭代器失效
    Vector<int>::Iterator it = v1.Begin();
    while(it != v1.End())
    {
        if (*it % 2 == 0)
            it = v1.Erase(it);
        else
            ++it;
    }
    PrintVector(v1);
}
```

- 【迭代器的型别&类型萃取技术】

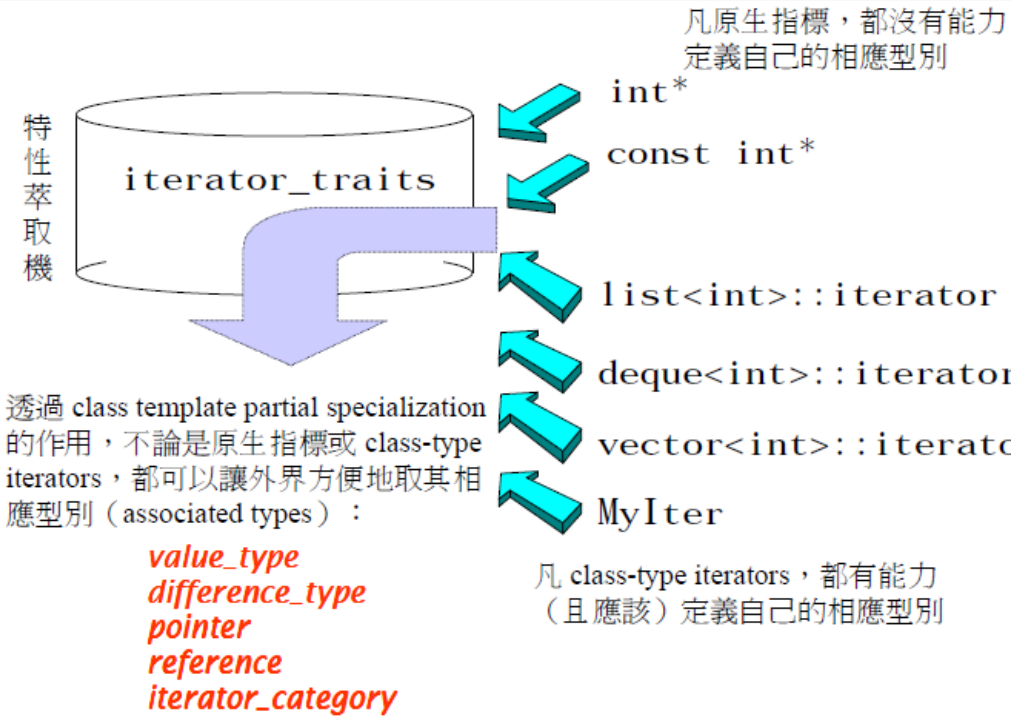


圖 3-1 **traits** 就像一台「特性萃取機」，擷取各個迭代器的特性（相應型別）。

3.4.1 迭代器相應型別之一：*value type*

所謂 *value type*，是指迭代器所指物件的型別。任何一個打算與 STL 演算法有完美搭配的 class，都應該定義自己的 *value type* 巢狀型別，作法就像上節所述。

3.4.2 迭代器相應型別之二：*difference type*

difference type 用來表示兩個迭代器之間的距離，也因此，它可以用來表示一個容器的最大容量，因爲對於連續空間的容器而言，頭尾之間的距離就是其最大容量。如果一個泛型演算法提供計數功能，例如 STL 的 `count()`，其傳回值就必須使用迭代器的 *difference type*：

```
template <class I, class T>
typename iterator_traits<I>::difference_type // 這一整行是函式回返型別
count(I first, I last, const T& value) {
  typename iterator_traits<I>::difference_type n = 0;
  for ( ; first != last; ++first)
    if (*first == value)
      ++n;
  return n;
}
```

針對相應型別 *difference type*，**traits** 的兩個（針對原生指標而寫的）特化版本如下，以 C++ 內建的 `ptrdiff_t`（定義於 `<cstddef>` 表頭檔）做爲原生指標的 *difference type*：

```
template <class I>
struct iterator_traits {
  ...
  typedef typename I::difference_type   difference_type;
};

// 針對原生指標而設計的「偏特化（partial specialization）」版
template <class T>
struct iterator_traits<T*> {
  ...
  typedef ptrdiff_t    difference_type;
};

// 針對原生的 pointer-to-const 而設計的「偏特化（partial specialization）」版
template <class T>
struct iterator_traits<const T*> {
  typedef ptrdiff_t    difference_type;
};
```

3.4.3 迭代器相應型別之三：*reference type*

從「迭代器所指之物的內容是否允許改變」的角度觀之，迭代器分爲兩種：不允許改變「所指物件之內容」者，稱爲 constant iterators，例如 `const int* pic`；允許改變「所指物件之內容」者，稱爲 mutable iterators，例如 `int* pi`。當我們對一個 mutable iterators 做提領動作時，獲得的不應該是個右值（rvalue），應該是個左值（lvalue），因爲右值不允許賦值動作（assignment），左值才允許：

```
int* pi = new int(5);
const int* pci = new int(9);
*pi = 7;   // 對 mutable iterator 做提領動作時，獲得的應該是個左值，允許賦值。
*pci = 1;  // 這個動作不允許，因爲 pci 是個 constant iterator，
           // 提領 pci 所得結果，是個右值，不允許被賦值。
```

在 C++ 中，函式如果要傳回左值，都是以 by reference 的方式進行，所以當 `p` 是個 mutable iterators 時，如果其 *value type* 是 `T`，那麼 `*p` 的型別不應該是 `T`，應該是 `T&`。將此道理擴充，如果 `p` 是一個 constant iterators，其 *value type* 是 `T`，那麼 `*p` 的型別不應該是 `const T`，而應該是 `const T&`。這裡所討論的 `*p` 的型別，即所謂的 *reference type*。實作細節將在下一小節一併展示。

3.4.4 迭代器相應型別之四：*pointer type*

pointers 和 references 在 C++ 中有非常密切的關連。如果「傳回一個左值，令它代表 `p` 所指之物」是可能的，那麼「傳回一個左值，令它代表 `p` 所指之物的位址」也一定可以。也就是說我們能夠傳回一個 pointer，指向迭代器所指之物。

這些相應型別已在先前的 `ListIter` class 中出現過：

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

`Item&` 便是 `ListIter` 的 *reference type* 而 `Item*` 便是其 *pointer type*。

現在我們把 *reference type* 和 *pointer type* 這兩個相應型別加入 **traits** 內：

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

// 針對原生指標而設計的「偏特化版（partial specialization）」
template <class T>
struct iterator_traits<T*> {
  ...
  typedef T*      pointer;
  typedef T&      reference;
};

// 針對原生的 pointer-to-const 而設計的「偏特化版（partial specialization）」
template <class T>
struct iterator_traits<const T*> {
  ...
  typedef const T*     pointer;
  typedef const T&     reference;
};
```

3.4.5 迭代器相應型別之五：*iterator_category*

最後一個（第五個）迭代器相應型別會引發較大規模的寫碼工程。在那之前，我必須先討論迭代器的分類。

根據移動特性與施行動作，迭代器被分爲五類：

- *Input Iterator*：這種迭代器所指物件，不允許外界改變。唯讀（read only）。
- *Output Iterator*：唯寫（write only）。
- *Forward Iterator*：允許「寫入型」演算法（例如 `replace()`）在此種迭代器所形成的區間上做讀寫動作。
- *Bidirectional Iterator*：可雙向移動。某些演算法需要逆向走訪某個迭代器區間（例如逆向拷貝某範圍內的元素），就可以使用 *Bidirectional Iterators*。
- *Random Access Iterator*：前四種迭代器都只供應一部份指標算術能力（前三種支援 `operator++`，第四種再加上 `operator--`），第五種則涵蓋所有指標算術能力，包括 `p+n`, `p-n`, `p[n]`, `p1-p2`, `p1<p2`。

這些迭代器的分類與從屬關係，可以圖 3-2 表示。直線與箭頭代表的並非 C++ 的繼承關係，而是所謂 concept（概念）與 refinement（強化）的關係[2]。

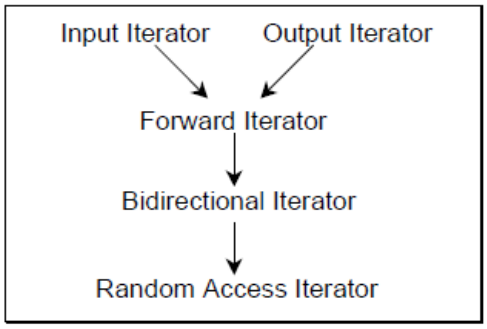
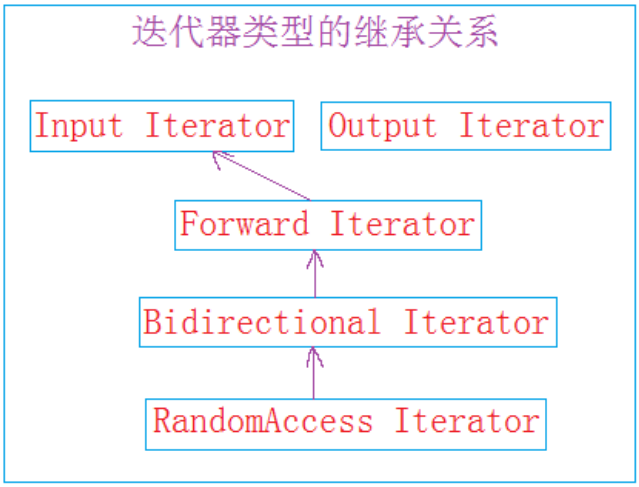


圖 3-2 迭代器的分類與從屬關係

設計演算法時，如果可能，我們儘量針對圖 3-2 中的某種迭代器提供一個明確定義，並針對更強化的某種迭代器提供另一種定義，這樣才能在不同情況下提供最大效率。研究 STL 的過程中，每一分每一秒我們都要念茲在茲，效率是個重要課題。假設有個演算法可接受 *Forward Iterator*，你以 *Random Access Iterator* 餵給它，它當然也會接受，因爲一個 *Random Access Iterator* 必然是一個 *Forward Iterator*（見圖 3-2）。但是可用並不代表最佳！

Itput Iterator: 只读迭代器。
Output Iterator: 只写迭代器。
Forward Iterator: 前向迭代器。
Bidirectional Iterator: 双向迭代器。
RandomAccess Iterator：随机访问迭代器。



3.5 std::iterator 的保證

爲了符合規範，任何迭代器都應該提供五個巢狀相應型別，以利 **traits** 萃取，否則便是自外於整個 STL 架構，可能無法與其他 STL 組件順利搭配。然而寫碼難免掛一漏萬，誰也不能保證不會有粗心大意的時候。如果能夠將事情簡化，就好多了。STL 提供了一個 iterators class 如下，如果每個新設計的迭代器都繼承自它，就保證符合 STL 所需之規範：

```
template <class Category,
          class T,
          class Distance = ptrdiff_t,
          class Pointer = T*,
          class Reference = T&>
struct iterator {
  typedef Category     iterator_category;
  typedef T            value_type;
  typedef Distance     difference_type;
  typedef Pointer      pointer;
  typedef Reference    reference;
};
```

總結

設計適當的相應型別（associated types），是迭代器的責任。設計適當的迭代器，則是容器的責任。唯容器本身，才知道該設計出怎樣的迭代器來走訪自己，並執行迭代器該有的各種行爲（前進、後退、取值、取用成員…）。至於演算法，完全可以獨立於容器和迭代器之外自行發展，只要設計時以迭代器爲對外介面就行。

traits 編程技法，大量運用於 STL 實作品中。它利用「巢狀型別」的寫碼技巧與編譯器的 template 引數推導功能，補強 C++ 未能提供的關於型別認證方面的能力，補強 C++ 不爲強型（strong typed）語言的遺憾。瞭解 **traits** 編程技法，就像獲得「芝麻開門」口訣一樣，從此得以一窺 STL 源碼堂奧。

```
//
// 迭代器的型别
//
struct InputIteratorTag {};
struct OutputIteratorTag {};
struct ForwardIteratorTag : public InputIteratorTag {};
struct BidirectionalIteratorTag : public ForwardIteratorTag {};
struct RandomAccessIteratorTag : public BidirectionalIteratorTag {};

//
// 迭代器内嵌包含的种相应的型别
// Iterator Category 、Value Type、 Difference Type、Pointer、 Reference
// 这种内嵌的型别定义，确保了能够更方便的跟 STL融合。
// 且方便Iterator Traits的类型萃取
//
template <class Category, class T, class Distance = ptrdiff_t ,
class Pointer = T*, class Reference = T&>
struct Iterator
{
    typedef Category   IteratorCategory;      // 迭代器类型
    typedef T          ValueType;             // 迭代器所指对象类型
    typedef Distance   DifferenceType;        // 两个迭代器之间的距离
    typedef Pointer    Pointer;               // 迭代器所指对象类型的指针
    typedef Reference Reference;              // 迭代器所指对象类型的引用
};

//
```

```
// Traits 就像一台“特性萃取机”，榨取各个迭代器的特性（对应的型别）
//
template <class Iterator>
struct IteratorTraits
{
     typedef typename Iterator:: IteratorCategory IteratorCategory ;
     typedef typename Iterator:: ValueType        ValueType;
     typedef typename Iterator:: DifferenceType   DifferenceType ;
     typedef typename Iterator:: Pointer           Pointer;
     typedef typename Iterator:: Reference         Reference;
};

//
// 偏特化原生指针类型
//
template <class T>
struct IteratorTraits< T*>
{
     typedef RandomAccessIteratorTag IteratorCategory ;
     typedef T                            ValueType;
     typedef ptrdiff_t                    DifferenceType ;
     typedef T *                          Pointer;
     typedef T &                          Reference;
};

//
// 偏特化const原生指针类型
//
template <class T>
struct IteratorTraits< const T *>
{
     typedef RandomAccessIteratorTag IteratorCategory ;
     typedef T                            ValueType;
     typedef ptrdiff_t                    DifferenceType ;
     typedef const T*                    Pointer;
     typedef const T&                    Reference;
};

////////////////////////////////////////////////////////
// Distance 的实现

template <class InputIterator>
inline typename IteratorTraits <InputIterator>:: DifferenceType
__Distance (InputIterator first, InputIterator last, InputIteratorTag )
{
     IteratorTraits<InputIterator >::DifferenceType n = 0;
     while (first != last) {
         ++ first; ++n ;
    }
     return n ;
}

template <class RandomAccessIterator>
inline typename IteratorTraits <RandomAccessIterator>:: DifferenceType
__Distance (RandomAccessIterator first, RandomAccessIterator last,
           RandomAccessIteratorTag)
{
     return last - first;
}

template <class InputIterator>
inline typename IteratorTraits <InputIterator>:: DifferenceType
Distance (InputIterator first, InputIterator last)
{
     return __Distance (first, last, IteratorTraits <InputIterator>:: IteratorCategory());
}

////////////////////////////////////////////////////////
// Advance 的实现

template <class InputIterator, class Distance>
inline void __Advance(InputIterator & i, Distance n, InputIteratorTag )
{
```

```
    while (n--) ++i;
}

template <class BidirectionalIterator, class Distance>
inline void __Advance(BidirectionalIterator & i, Distance n,
                      BidirectionalIteratorTag )
{
    if (n >= 0)
        while (n--) ++i;
    else
        while (n++) --i;
}

template <class RandomAccessIterator, class Distance>
inline void __Advance(RandomAccessIterator & i, Distance n,
                      RandomAccessIteratorTag)
{
    i += n ;
}

template <class InputIterator, class Distance>
inline void Advance(InputIterator & i, Distance n)
{
    __Advance(i , n, IteratorTraits <InputIterator>:: IteratorCategory());
}

// 测试Distance 算法
void Test3 ()
{
    List<int > l1;
    l1.PushBack (1);
    l1.PushBack (2);
    l1.PushBack (3);
    l1.PushBack (4);

    cout<<"List Distance: "<< Distance(l1 .Begin(), l1.End ())<<endl;

    Vector<int > v1;
    v1.PushBack (1);
    v1.PushBack (2);
    v1.PushBack (3);
    v1.PushBack (4);
    v1.PushBack (5);

    cout<<"Vector Distance: "<< Distance(v1 .Begin(), v1.End ())<<endl;
}

// 测试Advance 算法
void Test4 ()
{
    List<int > l1;
    l1.PushBack (1);
    l1.PushBack (2);
    l1.PushBack (3);
    l1.PushBack (4);

    List<int >::Iterator listIt = l1 .Begin();
    Advance(listIt , 3);
    cout<<"List Advance ? 3 : " <<*listIt<< endl;

    Vector<int > v1;
    v1.PushBack (1);
    v1.PushBack (2);
    v1.PushBack (3);
    v1.PushBack (4);
    v1.PushBack (5);

    Vector<int >::Iterator vecIt = v1 .Begin() + 1;
    Advance(vecIt , 3);
    cout<<"Vector Advance ? 5 : " <<*vecIt<< endl;
}
```